Pós-Graduação em Ciência da Computação

“X-PRO (Extreme Software Process): Um Framework para Desenvolvimento Eficiente de Software Baseado em Metodologias Ágeis”

Por

Carlos Diego Cavalcanti Pereira

Dissertação de Mestrado Profissional

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, 2014

Universidade Federal de Pernambuco

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Carlos Diego Cavalcanti Pereira

X-PRO (EXTREME SOFTWARE PROCESS): UM FRAMEWORK PARA DESENVOLVIMENTO EFICIENTE DE SOFTWARE BASEADO EM METODOLOGIAS ÁGEIS

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.

ORIENTADOR(A): ***Prof. Dr. Alexandre Marcos Lins de Vasconcelos***

RECIFE, 2014

Catalogação na fonte
Bibliotecária Joana D'Arc L. Salvador, CRB 4-572

Pereira, Carlos Diego Cavalcanti.
X-PRO (Extreme software process): um framework para desenvolvimento eficiente de software baseado em metodologias ágeis / Carlos Diego Cavalcanti Pereira. – Recife: O Autor, 2014.
165 f.: fig., tab.

Orientador: Alexandre Marcos Lins de Vasconcelos.
Dissertação (Mestrado Profissional) - Universidade Federal de Pernambuco. CIN. Ciência da Computação, 2014.
Inclui referências e apêndices.

1. Engenharia de software. 2. Desenvolvimento ágil de software. 3. Scrum (Desenvolvimento de software). I. Vasconcelos, Alexandre Marcos Lins de (orientador). II. Título.

005.1 (22. ed.) MEI 2014-95

Dissertação de Mestrado Profissional apresentada por **Carlos Diego Cavalcanti Pereira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título, "**X-PRO: Um Framework para Desenvolvimento Eficiente de Software Baseado em Metodologias Ágeis**", orientada pelo Professor Alexandre Marcos Lins de Vasconcelos e aprovada pela Banca Examinadora formada pelos professores:

_____

Prof. Hermano Perrelli de Moura
Centro de Informática / UFPE

_____

Prof. Cristine Martins Gomes de Gusmão
Universidade Federal de Pernambuco / CTG

_____

Prof. Alexandre Marcos Lins de Vasconcelos
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 18 de março de 2014.

_____

Profª. EDNA NATIVIDADE DA SILVA BARROS
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Dedico esse trabalho a minha filha, Maya; minha esposa, Laís; minha mãe, Esther; meu pai, Carlos; meu irmão, Thiago; meu avô, Cleto; e a minha avó, Ivanise. Sem vocês minha vida não seria completa.

AGRADECIMENTOS

Dedico esse trabalho primeiramente a minha filha, Maya, por cultivar em mim o amor pleno e absoluto. Esse trabalho é para você, minha filha.

A minha esposa, Laís, por ser minha companheira, amiga e inspiração para a luta do dia a dia.

A minha mãe, Esther, por nunca ter medido esforços para me oferecer uma educação adequada, além do seu empenho e amor incondicional.

A meu pai, que junto a Deus olha por mim e guia meu caminho.

Ao meu avô, Cleto, por ser meu maior ídolo, exemplo de vida e referência de como ser um homem de bem.

A minha avó, Ivanise, por sua sapiência e amor dedicado, os quais forjaram meu coração.

Ao meu irmão, Thiago, por ser meu amigo e parte fundamental da minha história.

Aos meus sogros, Valmir e Fátima, por serem como pais para mim.

A meus familiares e amigos, que nesse momento reservo-me a distinção de nomes, pois todos são meus maiores tesouros de vida.

A Fiabesa, representada pelos senhores Paulo Maranhão, Arthur Maranhão e Gerson Maranhão, pela oportunidade profissional e apoio acadêmico que me proporcionaram, sem os quais essa importante etapa da minha vida não seria possível.

A Visão Mundial, por me permitir a oportunidade de desempenhar o meu trabalho e colaborar para construção de um mundo melhor.

Aos amigos de departamento de Tecnologia da Informação com quem tive e tenho a oportunidade de trabalhar, pela compreensão e apoio em minhas ausências devido ao mestrado.

Aos amigos que comigo dividiram esse mestrado: Felipe Barbalho, Luiz Vieira, Paulo Rodrigues e Ubiracy Junior. Realizar os trabalhos com pessoas como vocês foi enriquecedor!

Ao Prof. Dr. Hemano Moura e a Profa. Dra. Cristine Gusmão, pelas valiosas contribuições a este trabalho.

E ao meu orientador, Prof. Dr. Alexandre Vasconcelos, pela paciência, apoio e amizade dividida ao longo de todo esse trabalho.

"*A resposta para a vida, o universo e tudo mais é: 42.*"

Douglas Adams

X-PRO (Extreme Software Process): Um Framework para Desenvolvimento Eficiente de Software Baseado em Metodologias Ágeis

Resumo

As metodologias ágeis de desenvolvimento de software emergiram como alternativa para processos prescritivos, os quais são classificados como excessivamente formais. Com objetivo de viabilizar entregas frequentes, processos adaptáveis, flexíveis e focados no cliente, as metodologias ágeis se tornaram referência sobre como produzir software de forma produtiva. Contudo, com objetivo de tornar o processo mais simples, essas abordagens invariavelmente incorrem em problemas como: modelos pouco generalistas para cobrir todo o ciclo de vida da Engenharia de Software; foco em disciplinas específicas do processo de software; subestimação de aspectos relevantes em iniciativas de desenvolvimento, como esforços de arquitetura, design e documentação; e indicação para equipes e projetos de pequena escala. A proposta desta dissertação de mestrado é propor um framework de processo eficiente de desenvolvimento de software, o qual se baseia nos ideários das metodologias ágeis, porém cobre todo o ciclo de vida de projeto desde a concepção, gestão, implementação e entrega. O framework inclui ferramentas como princípios, valores, atividades, artefatos e práticas específicas, incluindo esforços como arquitetura, design e documentação, concebendo um processo ágil, eficiente e extremamente produtivo para projetos de concepção e manutenção software, independentemente da sua escala.

Palavras-chave: Engenharia de Software. Metodologias Ágeis. Modelagem Ágil de Software. Scrum. Extreme Programming. Crystal. Dynamic Systems Development Method. Adaptive Software Development. Feature-Driven Development. Behavior-Driven Development. Test-Driven Development.

X-PRO (Extreme Software Process): A Framework for Efficient Software Development Based on Agile Methodologies

Abstract

Agile software development methodologies have emerged as an alternative to prescriptive processes, which are classified as excessively formal. Aiming to facilitate frequent deliveries, adaptable, flexible and customer-focused processes, agile methodologies have become a reference on how to produce software productively. However, in order to make the process easier, these approaches invariably incur in problems like: generalist models to cover the entire life cycle of software engineering; focusing on specific disciplines in the software process; underestimation of relevant development efforts as architecture, design and documentation, and indication for small teams and scale projects. The purpose of this dissertation is to propose a framework for efficient software development process, which is based on the ideals of agile methodologies, but covers the entire project life cycle from design, management, implementation and delivery. The framework includes tools such as principles, values, activities, artifacts and specific practices, including efforts such as architecture, design and documentation, designing an agile, efficient and highly productive process for projects of software development and maintenance, regardless of its scale.

Keywords: Software Engineering. Agile Methodologies. Agile Modeling. Scrum. Extreme Programming. Crystal. Dynamic Systems Development Method. Adaptive Software Development. Feature-Driven Development. Behavior-Driven Development. Test-Driven Development.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

Neste capítulo é apresentado o contexto de realização do trabalho, os aspectos que motivaram a pesquisa, seus objetivos e estrutura do restante do documento.

1.1. Contextualização

Pesquisas realizadas pelo Gartner Group em 2010 previam que até 2012, cerca de 80% dos projetos de desenvolvimento de software utilizariam alguma metodologia ágil como modelo para seus processos.[1] Conforme a mesma pesquisa, as organizações ampliaram sua utilização de metodologias ágeis buscando alcançar maior produtividade dos times de desenvolvimento especialmente para responder a mudanças dos requisitos, bem como viabilizar ganhos de produtividade total dos projetos.

As metodologias ágeis de desenvolvimento de software surgiram como alternativa aos modelos prescritivos, os quais implementam e propõem um escopo de desenvolvimento orientado a fluxos pré-estabelecidos e que invariavelmente devem ser seguidos para produção de produtos de software (SOMMERVILLE, 2009). Esses modelos prescritivos surgiram como resposta a um cenário anterior em que não existiam processos formais para desenvolvimento de software, o que leva à conclusão de que há uma tendência de transformação e evolução nos processos de sofware (LARMAN ET AL., 2001).

É notório que as discussões sobre a adoção de processos de software sempre permearam as organizações e equipes de desenvolvimento. Parte dessas discussões recai sobre o fato de que sem processos, os projetos eventualmente são levados ao insucesso (PRESSMAN, 2004). E esse insucesso de iniciativas de desenvolvimento de software tem como principal causador a dificuldade em responder às inerentes mudanças dos contextos que buscavam atender, bem como na improdutividade gerada pela ausência ou mesmo sobrecarga de processos (SOMMERVILLE, 2009). Conforme Sommerville (2009, pág. 57, Tradução do Autor), a conjuntura atual das organizações trouxe a discussão sobre como as iniciativas de software poderiam responder aos movimentos dos mercados aos quais os negócios estão inseridos:

> "*Os negócios agora operam em um ambiente global e de rápidas mudanças. Eles precisam responder a novas oportunidades e mercados, mudanças de condições econômicas, e a emergente competição de produtos e serviços. Software é parte de praticamente todas as operações e negócios, então um novo software é desenvolvido rapidamente para viabilizar vantagem e novas oportunidades, além de responder a pressão competitiva. Atualmente, desenvolvimento e entregas rápidas são portanto o mais crítico requisito para sistemas de softwares. De fato, muitos negócios estão dispostos a trocar a qualidade e compromisso com os requisitos para alcançar a rápida implantação do software que eles precisam.*"

Esse cenário de subjulgamento da qualidade e formalidade aos requisitos, conforme apresentado por Sommerville (2009), tornou-se um dos principais fatores impulsionadores pela adoção de metodologias ágeis de desenvolvimento de software. Como colocado pelo

[1] Pesquisa "*Predicts 2010: Agile and Cloud Impact Application Development Directions*" publicada pelo Gartner Group em 3 de Dezembro de 2009 disponível em https://www.gartner.com/doc/1244514, acessada em 10 de Dezembro de 2013 no site http://www.infoq.com/news/2012/12/gartner-agile-2012.

autor, para os clientes dos projetos, o propósito de se desenvolver software é puramente o próprio software. Contudo, as organizações durante muito tempo lutaram pelo estabelecimento de processos para atender um objetivo: responder bem às mudanças, o que por consequência gera processos auxilires à atividade de desenvolvimento, como forma de viabilizar um fluxo que suporte melhor as mudanças. As mudanças sempre causaram uma grande preocupação em gestores de processos de software (LARMAN ET AL., 2001). Parte disto está relacionado ao fato de que por muito tempo se pensou em desenvolvimento de software como um processo em cascata: inicialmente compreende-se o contexto do negócio, em seguida identificam-se os requisitos, para então estabelecer-se a arquitetura, seguir com o desenvolvimento e assim sucessivamente (LARMAN ET AL., 2001). Em um contexto como esse, as mudanças são tidas como problemas aos processo, uma vez que nesses cenários se faz necessário retornar ao ponto inicial do projeto para novamente compreender o contexto do negócio, implementar eventuais mudanças para só então retomar ao ponto onde se estava antes da mudança. Conforme descreve Ambler (2002), um dos principais valores das metodologias ágeis é que as mudanças devem ser abraçadas.

> "*Desenvolvedores ágeis sabem que seu trabalho será afetado por mudanças; eles agem ativamente para se comunicar com seus clientes, para coletar seu feedback, para que então eles identifiquem eventuais mudanças e então possam agir adequadamente. Eles não culpam seus clientes pela mudança; em vez disso, eles trabalham ativamente com eles para entender e comunicar as implicações das mudanças e habilitarem seus clientes a tomarem decisão como se, como e quando a mudança será suportada por seus esforços de desenvolvimento. Além disso, desenvolvedores ágeis entendem que seus modelos são apenas modelos, que desenvolvedores irão destruí-los e reconstruí-los em algo melhor; eles aceitam que seu trabalho será melhorado por outros.*" (AMBLER, 2002, pág. 30, Tradução do Autor)

Outros aspectos foram abordados pelas metodologias ágeis como valores a serem seguidos em projetos de desenvolvimento de software. Esses ideários, inclusive, compõem o "*Manifesto Ágil*"[2], que consiste em uma declaração de princípios fundamentais para o desenvolvimento ágil de software, conforme elencado a seguir:

- Indivíduos e interações mais que processos e ferramentas;
- Software em funcionamento mais que documentação abrangente;
- Colaboração com o cliente mais que negociação de contratos;
- Responder a mudanças mais que seguir um plano.

As definições atuais de desenvolvimento ágil de software emergiram a partir da década de 1990, como uma resposta aos modelos prescritivos, os quais conforme descrito anteriormente eram caracterizados por restrições quanto ao modelo de desenvolvimento. As metodologias ágeis sugeriam um retorno às práticas de desenvolvimento típicas do início da história do desenvolvimento de software (BOEHM ET AL., 2003). Anteriormente, as metodologias ágeis eram conhecidas como "métodos leves", até a proposta do "método ágil", conforme propôs o Manifesto Ágil. As metodologias iniciais incluíam Scrum (1986), Crystal, Extreme

[2] Manifesto Ágil, publicado em 2001 por Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas. Acessado em http://agilemanifesto.org/iso/ptbr/ em 10 de Dezembro de 2013.

Programming (1996), Adaptive Software Development, Feature-Driven Development e Dynamic Systems Development Method (1995).

Ainda que as metodologias ágeis tenham emergido como alternativa para os problemas previamente abordados nos modelos prescritivos, ao longo do tempo algumas questões relevantes e contrárias a processos ágeis passaram a ser levantadas. Conforme pesquisa apresentada por Begel e Nagappan (2007), diversos desenvolvedores percebem problemas com processos ágeis de desenvolvimento. Um desses problemas está relacionado ao fato de por serem extremamente simplificadas, as metodologias ágeis não poderiam ser escaladas para projetos grandes. Outros pontos identificados na pesquisa apontam: falta de visão do contexto amplo do negócio; ausência de design prévio ou mesmo design ruim; integração entre desenvolvimento e teste é difícil; entre outros pontos. Como apresentado pelos autores, percebe-se que cada metodologia aparentemente resolve individualmente apenas uma parte dos problemas oriundos de modelos prescritivos de processos.

Considerando o contexto apresentado, essa dissertação de mestrado objetiva propor um framework para desenvolvimento eficiente de software baseado em metodologias ágeis. O modelo, intitulado X-PRO (Extreme Software Process, do português Processo de Software Extremo), objetiva propor um framework de processo que contemple os principais benefícios das diferentes metodologias ágeis disponíveis na literatura, buscando igualmente resolver alguns dos problemas habitualmente encontrados nas metodologias ágeis atuais, conforme levantaram Begel e Negappan (2007).

1.2. Problema de Pesquisa

No que se refere aos problemas da pesquisa que objetivam-se ser respondidos ante a proposta do modelo X-PRO, consideram-se:

1. Cada metodologia ágil disponível na literatura objetiva resolver aspectos específicos de iniciativas de desenvolvimento de software, como por exemplo Test-Driven Development e o XP (Extreme Programming), as quais estão mais focadas aos esforços de codificação do que do projeto de software como um todo (BEGEL & NEGAPPAN, 2007). **Espera-se que o framework X-PRO defina diretrizes e seja aplicável tanto para os esforços de gestão de projeto quanto de codificação**;
2. Buscando maior agilidade nos projetos de software, as metodologias ágeis tendem a negligenciar aspectos relevantes em Engenharia de Software, tais como o Design e Documentação (AKIF & MAJEED, 2012). **O framework X-PRO abordará em seu escopo as práticas sugeridas por modelagem ágil (*Agile Modeling*), bem como considera em sua estrutura a produção e manutenção de artefatos de documentação de software**;
3. Metodologias Ágeis habitualmente são compreendidas como aplicáveis a projetos de software de pequena e média escala, sendo inclusive desaconselhadas como referência única de processo em projetos de larga escala, com times acima de 40 pessoas (LINDVALL ET AL., 2002). **X-PRO poderá ser aplicável para qualquer escala de projeto de software**.

1.3. Objetivos da Pesquisa

Propor um framework de processo de software baseado em metodologias ágeis. O modelo, initulado X-PRO (Extreme Software Process, do português Processo de Software Extremo),

objetiva propor um framework de processo que contemple os principais benefícios das diferentes metodologias ágeis disponíveis na literatura, buscando igualmente resolver alguns dos problemas habitualmente encontrados nas metodologias ágeis atuais

A fim de estabelecer a viabilidade na utilização do X-PRO, o mesmo foi aplicado em um projeto de desenvolvimento de uma fábrica de software brasileira com sede em Recife/PE:

a) desenvolvimento de uma plataforma (software) de integração entre os múltiplos sistemas de informação de um cliente da fábrica de software.

A principal colaboração deste trabalho para a literatura está na definição de um modelo que auxilie nos projetos de desenvolvimento de software, independentemente do seu tamanho ou característica contextual, seja um processo ágil de desenvolvimento porém contemple igualmente conhecimento e documentação relevantes ao produto de software. Quanto aos objetivos específicos deste trabalho, consideram-se:

- Apresentação de conceitos, estruturas e arquitetura de modelos de processos de software e metodologias ágeis de desenvolvimento;
- Realizar uma pesquisa quanto a adoção de práticas e metodologias ágeis, fundamentando por meio da sua realização quais os aspectos fundamentais em uma abordagem ágil de desenvolvimento de software, sob a ótica de profissionais diretamente envolvidos com os esforços de desenvolvimento;
- Realizar estudo de caso aplicado do X-PRO, instanciando-o em um projeto de desenvolvimento de uma fábrica de software brasileira;
- Avaliar os resultados obtidos na aplicação do modelo X-PRO.

1.4. Metodologia da Pesquisa

A presente pesquisa está estruturada fundamentalmente em quatro blocos, sendo o primeiro a análise e revisão da literatura especializada sobre os temas e conceitos objetos dessa pesquisa; o segundo realiza uma pesquisa quanto à adoção de práticas e metodologias ágeis, fundamentando por meio da sua realização e quais os aspectos fundamentais em uma abordagem ágil de desenvolvimento de software, sob a ótica de profissionais diretamente envolvidos com os esforços de desenvolvimento; o terceiro propõe um framework de processo de software baseado em metodologias ágeis; enquanto o quarto considera a realização de um estudo empírico de adoção do modelo proposto, análise dos resultados encontrados e conclusões finais. As etapas da pesquisa são detalhadas a seguir:

- **ETAPA 1**: É realizado um estudo de revisão bibliográfica sobre os conceitos de processos de software, principalmente aqueles relacionados a metodologias ágeis de desenvolvimento de software;

- **ETAPA 2**: É realizada uma pesquisa quanto a adoção de práticas e metodologias ágeis, cujo objetivo é identificar quais os aspectos fundamentais em uma abordagem ágil de desenvolvimento de software, sob a ótica de profissionais diretamente envolvidos com os esforços de desenvolvimento;

- **ETAPA 3**: A partir da revisão bibliográfica realizada com objetivo de identificar as abordagens atualmente previstas na literatura sobre metodologias ágeis, foi definido o modelo arquitetural do framework X-PRO (Extreme Software Process);

- **ETAPA 4**: Dada a conclusão da revisão bibliográfica e definição da arquitetura do modelo, essa pesquisa avançou no sentido de realizar um estudo empírico de adoção do X-PRO em um projeto de desenvolvimento de uma fábrica de software brasileira com sede em Recife/PE. O objetivo foi identificar se os conceitos e resultados propostos pelo modelo seriam efetivamente alcançados. Concluído o estudo, foram realizadas as análises dos resultados alcançados, sendo por fim, apresentadas as conclusões da pesquisa.

1.5. Estrutura da Dissertação

A estrutura definida os capítulos desta dissertação é a seguinte:

- **Capítulo 1 – Introdução**: Apresenta a introdução e contextualização ao trabalho, motivação para o tema da pesquisa, seus objetivos e estrutura;
- **Capítulo 2 – Visão Geral de Processos de Software**: Apresenta os conceitos relacionados a modelos de processos software, abordando frameworks relacionados à Engenharia de Software;
- **Capítulo 3 – Metodologias Ágeis de Software**: Este capítulo apresenta os conceitos relacionados a metodologias ágeis de desenvolvimento de software, como: Scrum; Extreme Programming; Crystal; Dynamic Systems Development Method; Adaptive Software Development; Feature-Driven Development; Behavior Driven Development; Test-Driven Development e Agile Modeling;
- **Capítulo 4 – Análise comparativa entre Metodologias Ágeis de Software**: Este capítulo apresenta o resultado de uma pesquisa aplicada quanto a adoção de práticas e metodologias ágeis, cujo objetivo é identificar quais os aspectos fundamentais em uma abordagem ágil de desenvolvimento de software, sob a ótica de profissionais diretamente envolvidos com os esforços de desenvolvimento;
- **Capítulo 5 – Pesquisa sobre a Adoção de Práticas e Metodologias Ágeis**: Este capítulo apresenta o resultado de uma pesquisa aplicada, fundamentando por meio da sua realização quais os aspectos fundamentais em uma abordagem ágil de desenvolvimento de software, sob a ótica de profissionais diretamente envolvidos com os esforços de desenvolvimento;
- **Capítulo 6 – X-PRO (Extreme Software Process)**: Este capítulo apresenta a estrutura do framework de processo de software X-PRO, objeto dessa dissertação;
- **Capítulo 7 – Execução do Modelo**: Apresenta o estudo empírico do uso do X-PRO;
- **Capítulo 8 – Conclusões**: Apresenta as considerações finais e propostas de trabalhos futuros que poderão ser realizados a partir deste trabalho.

2. VISÃO GERAL DE PROCESSOS DE SOFTWARE

Neste capítulo é apresentada uma visão geral sobre processos de software. São abordados os conceitos relacionados à processos em desenvolvimento de software, considerando as práticas, modelos de ciclo de vida e de capacidade de processos de Software.

2.1. Introdução

A indústria de software têm buscado formas de melhorar a qualidade de seus produtos e serviços visando atingir um diferencial no mercado e atender a clientes cada vez mais exigentes. Essa busca pela melhor qualidade está relacionada diretamente a melhoria dos seus processos. Grande parte dos projetos de software, por exemplo, é entregue, porém invariavelmente fora dos custos e prazos previamente estabelecidos e pior: não atendendo as expectativas e requisitos instituídos pelo cliente, o que gera retrabalho em análise e codificação das soluções (AMBLER, 2002).

Segundo Reis (2003), estudos mostram que a indústria nacional de software ainda adota em muitos casos práticas artesanais durante o processo de desenvolvimento. Para Moitra apud Reis et al. (2002), um dos grandes obstáculos para favorecer o crescimento de uma indústria de software em países em desenvolvimento é a disponibilização de infraestrutura necessária à gerência dos processos implantados nas organizações. Essa infraestrutura, além de prover a gestão dos processos já existentes, fornece o suporte necessário para a análise de pontos de melhoria. Casos de sucesso na indústria de software em países como a Índia, evidenciam os diversos benefícios provenientes da maturidade dos processos. Segundo Reis (2003), esses benefícios variam desde a simples solução de dificuldades específicas através da melhoria contínua, até a certificação a partir de avaliações formais, onde as empresas participantes atingem um reconhecimento internacional da qualidade dos seus produtos e processos.

Para Pfleeger apud Rocha (2005), o desenvolvimento das soluções de software e apoio às necessidades dos clientes devem, entre outros fatores, ser mais produtivas e diminuir o custo dos projetos. Em virtude dessas necessidades, as empresas estão adotando práticas de reengenharia de processos para aumentar seu nível de maturidade. Decorrente desse contexto, diversos modelos de referência e normas internacionais de qualidade de processos foram definidos para atender as necessidades das empresas em melhoria de processos (ROCHA ET AL., 2005). Esses modelos servem não só para a melhoria dos processos de uma empresa, como também servem para titulação perante o mercado, dando aos clientes uma forma de avaliar seus possíveis fornecedores.

> "*Tem sido de grande importância a definição de modelos de avaliação do processo de engenharia de software em organizações. As organizações de desenvolvimento de software têm procurado atender aos requisitos de tais modelos, a fim de terem seu processo de desenvolvimento avaliado de forma positiva frente ao mercado.*" (REIS, 2003, p. 18)

Melhorar os processos de uma organização implica em fazer ajustes, implantar novos processos e muitas vezes executar mudanças radicais na forma de trabalho de uma empresa. Em pesquisa realizada por Rocha et al. (2005), foram levantados os principais fatores determinantes do sucesso ou insucesso da implantação de processos nas organizações, esses

fatores vão desde aspectos motivacionais até ao nível de competências dos envolvidos. Os principais fatores de sucesso são:

- Comprometimento dos colaboradores da organização e da alta gerência;
- Motivação da equipe da empresa;
- Disponibilidade de tempo para acompanhamento pela equipe implementadora;
- Grau de experiência da equipe de implementadores;
- Alinhamento dos processos com as estratégias de negócio da empresa;
- Relacionamento dos resultados da melhoria com a melhora dos resultados de negócios da empresa.

Sobre os principais fatores de insucesso, foram elencados:

- Competência da equipe da empresa em engenharia de software;
- Mudança da cultura organizacional;
- Dificuldades encontradas a respeito da estratégia de implementação;
- Falta de comprometimento da alta gerência;
- Ausência de ferramentas de apoio à execução dos processos.

As empresas têm enfrentado grandes desafios na execução de suas melhorias. Os estudos publicados por Rocha et al. (2005) e Reis (2003) deixam claras as dificuldades encontradas na implantação de novos processos, dificuldades essas que vão desde problemas de infraestrutura (ferramentas) a questões de resistência e metodologia adotada para conduzir a estratégia de implementação. De acordo com o SWEBOK (*Software Engineering Body of Knowledge*, 2004), o processo de mudança constitui um caso de mudança organizacional, e os mais bem-sucedidos esforços de mudança organizacional tratam a mudança como um projeto, com planos adequados, monitoramento e revisão. A melhoria de processos de software é uma ação contínua, realizadas através de ações para alterar os processos já existentes para que os mesmos atendam de forma mais eficiente a necessidades de negócio da organização. Por serem contínuas e envolverem desafios nos mais diversos níveis, essas mudanças devem ser incorporadas através de motivação, esforço e colaboração de todos os envolvidos no processo.

Qualquer processo, seja ele de tecnologia ou não, é criado com objetivo principal de entregar um resultado (ROCHA ET AL., 2005). Naturalmente a composição desse resultado pode – e provavelmente irá – variar com o tempo, dado ao dinamismo dos negócios. Em se tratando de processos de software, a busca por melhores práticas é constante devido aos resultados insatisfatórios alcançados pela indústria de desenvolvimento de sistemas, como descreve Ambler (2002, pág. 3, Tradução do Autor):

> "*A situação atual do desenvolvimento de software é aquém do ideal. Os sistemas normalmente não atendem as necessidades dos nossos clientes e nos precisamos desenvolver várias e várias vezes. Nossos clientes estão insatisfeitos por conta desses problemas e não estão dispostos a confiar em nós porque eles têm sido várias vezes decepcionados.*"

Um dos principais pontos relacionados aos problemas quanto ao processo de software recai sobre a cultura de que o desenvolvimento em si nunca é o problema. É comum que a

dificuldade em entregar de software de qualidade seja atribuída a aspectos como gestão, demanda excessiva por documentação, o entendimento do processo de negócio pelo cliente, infraestrutura, entre outros.

2.2. Processos de Software

A grande maioria do esforço de desenvolvimento de software está relacionada a uma atividade onde softwares são desenvolvidos para propósitos específicos de negócios (SOMMERVILLE, 2009). O contexto de desenvolvimento considera que para a produção de um produto de software, se faz necessária a execução de atividades específicas, que congregadas terão como resultado um produto. Nesse sentido, o conceito de Engenharia de Software é adotado como uma disciplina de engenharia que está relacionada a todos os aspectos para produção de software (SOMMERVILLE, 2009). Entende-se que para que o produto final seja concebido, todo um ciclo de processo de desenvolvimento precisa ser executado, já que os processos são a fundação da engenharia de software (PRESSMAN, 2004).

Quando abordada a temática relacionada ao processo de Engenharia de Software, é fundamental diferenciar as suas diversas abordagens e classificações de processo, de tal forma a alinhar as expectativas e exigências para cada um desses referenciais de processo. Habitualmente a literatura aborda três grandes grupos de abordagens de processo: Modelos, Frameworks e Metodologias (TOMHAVE, 2005). Conforme conceituado por Tomhave (pág. 10, 2005, Tradução do Autor), esses referenciais de processo são descritos como:

- **Modelo:** Trata-se de um resumo, uma construção conceitual que representa processos, variáveis e relacionamentos, sem prover orientações específicas ou práticas para sua implementação e aplicação;
- **Framework:** Trata-se de um construto fundamental que define pressupostos, conceitos, valores e práticas, e que inclui orientações para a sua execução propriamente dita;
- **Metodologia:** Trata-se de um construto orientado que define práticas, procedimentos e regras para aplicação de uma tarefa ou função específica, sendo prescritivo quanto aos passos para sua execução.

Os processos adotados em engenharia de software proveem uma visão técnica sobre como e o que fazer para construir um software. Para que o produto seja construído de forma racional, no prazo e atendendo as necessidades dos clientes, é fundamental que haja uma estrutura de processo que suporte esse ambiente. Atrelado a isso está o fato de que deve ser estabelecido um framework que suporte e seja referência para todo o ciclo de vida de produção de um produto de software, tal como afirma Pressman (2004, pág. 54, Tradução do Autor):

> "*O processo define que um framework deve ser estabelecido para a entrega efetiva da tecnologia de engenharia de software. O processo de software forma a base para gestão e controle dos projetos de software e estabelece o contexto onde métodos são aplicados, produtos de trabalho (modelos, documentos, dados, relatórios, formulários, etc.) são produzidos, marcos são estabelecidos, a qualidade é assegurada e mudanças são apropriadamente geridas.*"

Essa definição quanto a um modelo de referência para guiar o processo de desenvolvimento de software, provê diversas vantagens para a organização, conforme afirma Pereira (2005):

- Redução dos problemas relacionados a treinamento, revisões e suporte de ferramentas;
- Experiências adquiridas em cada projeto podem ser incorporadas ao processo padrão, contribuindo para sua melhoria;
- Maior facilidade em medições de processo e qualidade;
- Maior facilidade de comunicação entre os membros da equipe;
- Melhor adequação de novos membros na equipe de projeto;
- Melhor desempenho, previsibilidade e confiabilidade dos processos de trabalho.

É igualmente significativo que os esforços estabelecidos pela organização em definir um processo padrão para desenvolvimento de software são habilitadores e facilitadores para projetos de implantação de modelos de qualidade como as normas ISO/IEC 15504, ISO/IEC 12207 e CMMI (PEREIRA, 2005). Em se tratando de abordagens de processos de software, existem diversas abordagens e frameworks para suportar esse fim. Esses arcabouços, entretanto, são instâncias de modelos de ciclo de vida de processo de software, os quais propõem abordagens diferentes de como tratar o ciclo de desenvolvimento de um produto de software, sendo estes extensões de modelos mais genéricos (SOMMERVILLE, 2009). Comumente os processos de software são categorizados como *plan-driven* (do português "baseado em plano"), onde todas as atividades de processo são planejadas antecipadamente e são mensuradas frente aos planos; e os processos *agile-driven* (do português "baseado em agilidade"), onde os planos são incrementais e mais fáceis de mudar conforme os requisitos dos clientes (SOMMERVILLE, 2009). Com base nessas categorias, Sommerville (2009) descreve os principais modelos de processos de software como:

- **Modelo Cascata**: Prevê atividades fundamentais de processo, como especificação, desenvolvimento, validação e evolução, representando-as como fases de processos separadas, tais como especificação de requisitos, design, implementação, testes e outros. Conforme apresentado na Figura 1, esse modelo prevê que cada fase seja concluída para que a seguinte seja iniciada, caracterizando a referência a uma cascata, como o nome sugere. O modelo em cascata é um exemplo de representação de processos baseados em plano, onde no princípio deve-se planejar e agendar todas as atividades de processo antes de iniciar os trabalhos. A princípio o modelo cascata só deveria ser utilizado quando os requisitos são bem compreendidos e praticamente impossíveis de serem alterados durante o desenvolvimento do sistema. Entretanto, o modelo em cascata reflete o tipo de processo usado em outros modelos de engenharia de projetos. Como é mais fácil utilizar um modelo comum de gestão para todo o projeto, o modelo em cascata ainda é comumente utilizado.

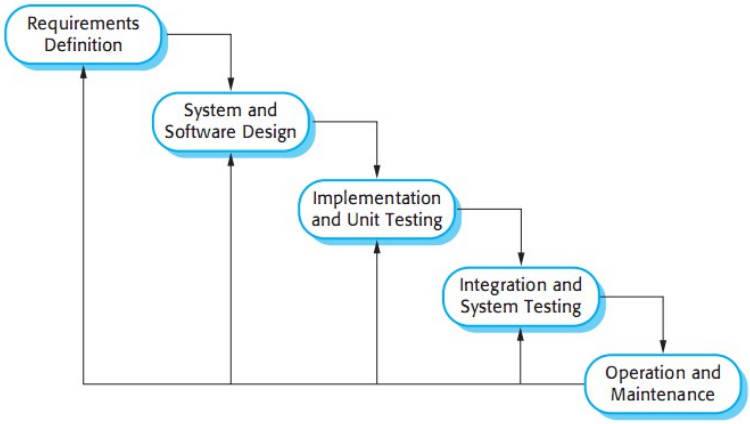


Figura 1: Modelo de ciclo de vida em Cascata de desenvolvimento de software (SOMMERVILLE, 2009).

- **Modelo Incremental**: prevê intercalações das atividades de especificação, desenvolvimento e validação. Como ilustrado a seguir, o sistema é desenvolvido em uma série de versões (incrementos), onde cada versão adiciona uma funcionalidade na versão anterior. O desenvolvimento incremental reflete a maneira como habitualmente se resolve problemas. Desenvolvendo software de forma incremental, torna-se mais barato e fácil promover mudanças enquanto esse é desenvolvido. Esse modelo provê três importantes benefícios, se comparado ao modelo em cascata: 1) O custo para acomodar mudanças de requisitos é reduzido, haja vista que os requisitos podem ser adicionados a cada ciclo; 2) É mais fácil obter retorno sobre o trabalho de desenvolvimento enquanto esse está sendo realizado, uma vez que é possível a concepção de versões intermediárias do sistema; 3) Entrega e implantação mais rápida de software utilizável pelo cliente, mesmo que nem todas as funcionalidades tenham sido incluídas. Da perspectiva de gestão, a abordagem incremental possui dois problemas: 1) O processo não é explicitamente visível, haja vista que os gestores medem o progresso do desenvolvimento pelos entregáveis que são liberados em cada incremento; 2) A estrutura do sistema tende a se degradar a cada novo incremento. A menos que tempo e dinheiro sejam investidos em *refactoring*[3], mudanças regulares tendem a corromper a estrutura. Incorporar mudanças torna-se gradativamente difícil e custoso.

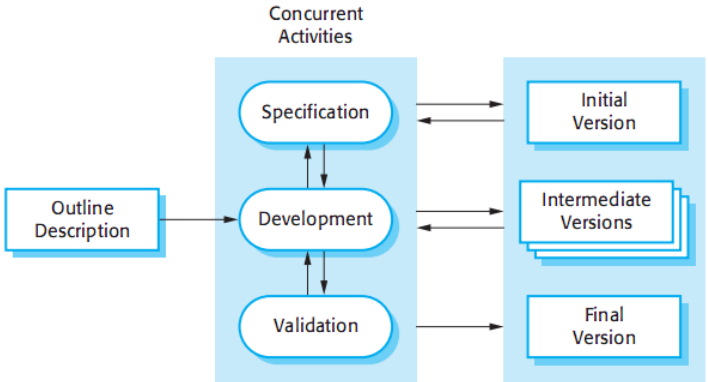


Figura 2: Modelo de ciclo de vida Incremental de desenvolvimento de software (SOMMERVILLE, 2009).

[3] Refactoring é uma técnica de reestruturação de código, onde a ideia básica é que sejam realizadas pequenas alterações no código, chamadas de *refactorings*, para suportar novos requisitos e manter o desing o mais simples possível. Um importante aspecto do *refactoring* é que para cada mudança no código, esse se deve manter semanticamente igual a como estava antes das alterações. (AMBLER, 2002).

- **Modelo Espiral**: Nesse modelo, o processo de software é representado como um espiral, em vez de uma sequência de atividades, com algum retrocesso de uma atividade para outra. Cada loop na espiral representa uma fase do processo de software. Assim, o laço mais interior pode estar preocupado com a viabilidade do sistema, o ciclo seguinte com a definição dos requisitos, o próximo ciclo com desenho do sistema, e assim por diante. O modelo espiral combina a tentativa de evitar mudanças com tolerância a mudanças. Assume-se que as mudanças são o resultado de riscos do projeto e incluem-se as atividades de gerenciamento de risco explícitas para reduzir esses riscos.

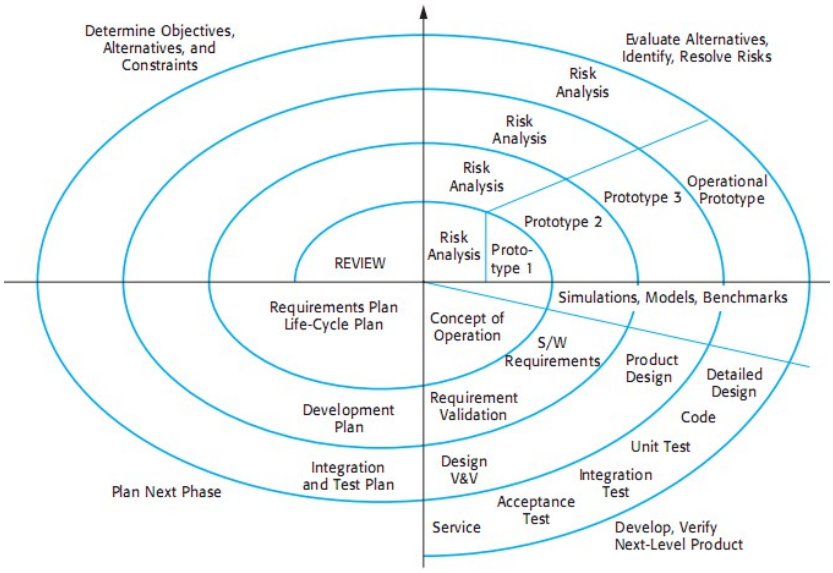


Figura 3: Modelo de ciclo de vida Espiral de desenvolvimento de software (SOMMERVILLE, 2009).

- **Modelo Ágil**: Abordagens ágeis para desenvolvimento de software, como apresenta a Figura 4, consideram o design e implementação as atividades centrais no processo de software. Porém podem também incorporar outras atividades, tais como elicitação de requisitos e testes, concepção e implementação. Em contraste, a abordagem orientada a planos identifica fases distintas no processo de software com saídas associadas a cada uma das fases. As saídas de uma fase são utilizadas como base para o planejamento da atividade seguinte do processo. Esses ciclos de entradas e saídas temporais e com produtos entregáveis e de valor ao cliente são comumente nomeados como iterações (*sprints*, no caso da metodologia Scrum). Em modelos de processo de software ágeis, os requisitos são priorizados com base no maior valor agregado para o proprietário do produto (cliente ou representante deste).

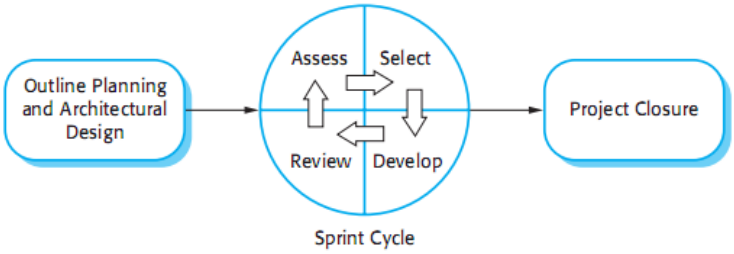


Figura 4: Modelo de ciclo de vida Ágil de desenvolvimento de software (SOMMERVILLE, 2009).

Ainda que a literatura preveja, conforme apresentado, uma série de modelos de processo de software, o mercado tende a adotar padrões que consequentemente tornam-se tendência em

um determinado segmento. No cenário de Engenharia de Software, em meados da década de 1990, a concepção de frameworks de processos baseados no modelo Iterativo Incremental cresceu exponencialmente (SHUJA & KREBS, 2008). Diferentes abordagens surgiram para atender a indústria de software, o que acarretou em muitos modelos distintos, mas poucos sendo utilizadas em larga escala. Em 1999, Ivar Jacobson, Grady Booch e James Rumbaugh publicaram o livro The Unified Software Development Process, o qual fazia a primeira referência ao conceito de Processo Unificado, que utiliza como ciclo de vida o modelo incremental para desenvolvimento de software. Posteriormente a Rational Software, empresa que viria a se fundir com a IBM, concebeu o RUP (*Rational Unified Process*), que é a mais difundida variante de Processo Unificado do mercado (WESTERHEIM, 2005). Pelo fato do RUP ter se tornado um dos mais utilizados e conhecidos frameworks para engenharia de software nas organizações, esse trabalho detalhará os conceitos presentes em tal modelo.

2.3. Engenharia de Software e o Processo Unificado

O processo unificado (UP, *Unified Process*) de desenvolvimento de software consiste no conjunto de atividades demandadas para transformar requisitos em um produto de software. O processo unificado de desenvolvimento combina ciclos iterativos e incrementais para a produção de softwares. Para o processo unificado, é fundamental que o avanço de um projeto deve estar baseado na construção de artefatos de software, e não apenas na codificação ou documentação (PRESSMAN, 2004).
Essa abordagem de desenvolvimento está centrada em algumas características específicas:

- **Iterativo e Incremental –** Cada iteração resulta em um incremento no produto, que é uma versão do software contendo funcionalidades adicionais e melhoradas ante a versão anterior;
- **Dirigidos por Casos de Uso –** Os casos de uso capturam os requisitos dos clientes e refinam o conteúdo das iterações, as quais contemplam um conjunto de requisitos;
- **Centrado na Arquitetura –** O processo unificado aborda que a arquitetura do software deve estar no foco central da equipe do projeto;
- **Focado no Risco –** O processo unificado determina que a equipe do projeto esteja focada no enfrentamento dos riscos inerentes do projeto já no início do ciclo de vida do projeto;
- **Baseado em Componentes –** O processo unificado é baseado em componentes, o que sugere que o sistema deve ser concebido a partir de componentes de software interconectados por meio interfaces bem definidas. O processo unificado define a utilização da Linguagem de Modelagem Unificada (UML, *Unified Modeling Language*) na concepção dos artefatos dos sistemas.

Como principal modelo baseado no processo unificado (SHUJA & KREBS, 2008), o *Rational Unified Process* (RUP) consiste em um framework de processos para desenvolvimento iterativo e incremental de software (SHUJA & KREBS, 2008). Trata-se de um exemplo de um modelo moderno de processo, o qual originalmente foi derivado dos trabalhos da definição do UML (*Unified Modeling Language*) (SOMMERVILLE, 2009). Conforme descreve Sommerville (2009), o RUP reconhece que modelos de processos convencionais apresentam uma única visão do processo. Em contrapartida, o RUP é normalmente descrito por três perspectivas:

- Uma perspectiva dinâmica, a qual considera as fases do modelo ao longo do tempo;
- Uma perspectiva estática, a qual considera as atividades que são determinadas;

- Uma perspectiva prática, a qual sugere boas práticas para serem usadas ao longo do processo.

O RUP também sugere seis conceitos chave em Engenharia de Software, os quais em versões anteriores do modelo eram tratados como boas práticas. Esses princípios devem guiar todas as iniciativas que objetivem utilizar ou implantar um processo baseado no RUP. Esses princípios, associados as seis primeiras letras do alfabeto, constituem a fundação do que é o RUP, conforme explanado por Shuja e Krebs (2008):

- **A –** Adaptar o processo;
- **B –** Balancear as prioridades dos *stakeholders*;
- **C –** Colaboração entre times de desenvolvimento;
- **D –** Demonstrar valor iterativamente;
- **E –** Elevar o nível de abstração;
- **F –** Focar continuamente na qualidade.

Os princípios chave não são sequenciais, no entanto um reforça a aplicação do outro. Outro aspecto relevante abordado pelo RUP refere-se ao reuso de processos e da customização dos mesmos para aplicação em cada contexto (PEREIRA, 2005). Como um framework de processo, ele pode ser definido como uma estrutura de suporte incompleta na qual outro processo pode ser organizado e desenvolvido (SHUJA & KREBS, 2008). Ou seja, é preciso adaptar o framework originalmente proposto pelo modelo antes de aplica-lo, de forma a atender as necessidades específicas dos projetos de software da organização.

2.3.1. Arquitetura do Processo Unificado

O RUP propõe um modelo em que o processo de software considera quatro fases discretas, as quais tratam a sua perspectiva dinâmica. Entretanto, conforme afirma Sommerville (2009), as fases no RUP estão mais relacionadas ao negócio do que a aspectos técnicos. As fases propostas pelo RUP são ilustradas e explanadas a seguir:

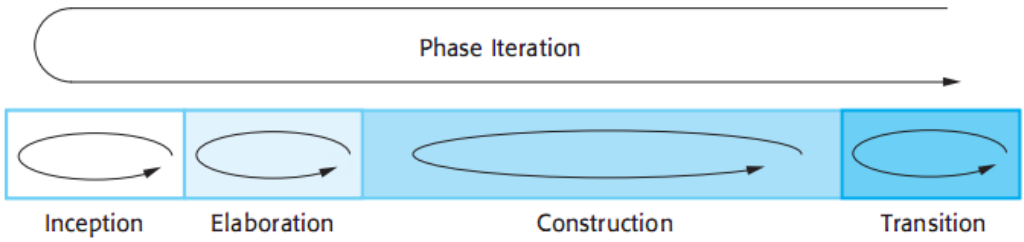


Figura 5: Fases do Rational Unified Process (RUP) (SOMMERVILLE, 2009).

- **Concepção**: O objetivo da fase de concepção é estabelecer as necessidades do negócio para o sistema. Nela identificam-se todas as entidades externas (pessoas e sistemas) as quais irão interagir com o sistema e devem-se definir essas interações. Nessa fase também se identifica como o sistema pode contribuir com mudanças e melhorias no negócio. Essa fase também prevê que uma análise seja realizada quanto à contribuição do novo sistema para o negócio, onde se essa contribuição for pequena, o projeto pode ser cancelado após essa fase (SOMMERVILLE, 2009);
- **Elaboração**: A fase de elaboração objetiva estabelecer uma linha de base da arquitetura do sistema para prover uma base estável de design para a implementação prevista na fase de Construção. A arquitetura envolve os requisitos mais significativos

e a avaliação dos riscos (SHUJA & KREBS, 2008). Para avaliar a estabilidade da arquitetura, conforme sugere Shuja & Krebs (2008), um ou mais protótipos de arquitetura podem ser desenvolvidos;

- **Construção**: A fase de construção objetiva esclarecer os requisitos remanescentes e complementar o desenvolvimento do sistema baseado no modelo arquitetural definido na fase de elaboração. A fase de construção é concluída com o marco de Capacidade Operacional Inicial (do inglês, *Initial Operation Capability*), o qual determina se o produto está pronto para ser implantado em um ambiente de teste (SHUJA & KREBS, 2008);
- **Transição**: A fase de transição consiste em mover o sistema do ambiente de desenvolvimento para o ambiente do usuário, fazendo-o operar em um contexto real. Conforme afirma Sommerville (2009), o esforço de transição é algo ignorado pela maioria dos modelos de processos de software, ainda que seja de fato uma atividade custosa e por vezes problemática. Em complemento a essa fase, está prevista a documentação do sistema assim como a confirmação de que o mesmo está operando corretamente no ambiente conforme previsto.

Conforme apresentado na Figura 5, as iterações no RUP são suportadas de duas maneiras: iterações que contemplem a execução de todas as atividades em uma única fase; ou iterações que considerem tanto as atividades quanto todas as fases do processo, conforme exibido pela ilustração da seta em *loop* da fase de Transição a Concepção. (SOMMERVILLE, 2009).

Já em sua perspectiva dinâmica, o RUP foca em um conjunto de atividades para o processo de desenvolvimento, chamados de *workflows*. Como o modelo foi desenvolvido em conjunto com o UML (SHUJA & KREBS, 2008), as descrições dos *workflows* é feita através de diagramas UML, tais como Diagrama de Sequência, Diagramas de Atividades, modelos de objetos, entre outros. O RUP prevê seis atividades primárias em sua estrutura de processo, as quais são complementadas por três atividades de suporte (PRESSMAN, 2004). A seguir são apresentadas as atividades previstas na arquitetura do RUP.

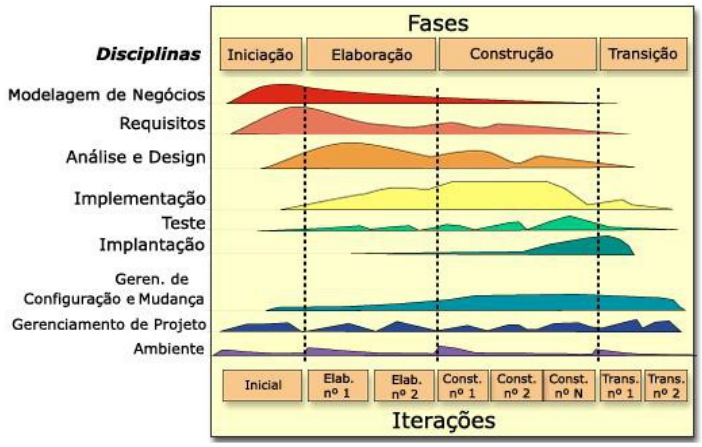


Figura 6: Fluxo de trabalho do Rational Unified Process (RUP), adaptado de SHUJA & KREBS (2008).

A seguir são descritas as atividades primárias previstas pelo RUP – destacadas em tom escuro – e as atividades de suporte do framework – destacadas em branco.

Workflow	Descrição
Modelagem de Negócios	Os processos de negócios são modelados utilizando casos de uso.
Requisitos	Atores que interagem com o sistema são identificados e casos de uso são desenvolvidos para modelar os requisitos do sistema.
Análise e Design	Um modelo de design é criado e documentado utilizado diagramas arquiteturais, diagramas de componentes, diagrama de objetos e diagrama de sequência.
Implementação	Os componentes no sistema são implementados e estruturados em implementações de subsistemas (módulos). Geração automática de código a partir de diagramas de design auxiliam a acelerar esse processo (SOMMERVILLE, 2009).
Teste	Os testes consistem em um processo iterativo o qual e seguido em conjunto com a implementação. Os testes de sistema seguem como complemento da implementação.
Implantação	Um lançamento (release) do produto é criado, distribuído aos usuários e instalado em suas áreas de trabalho.
Gerência de Configuração e Mudanças	Trata-se de uma atividade de suporte para gerir e controlar as alterações e mudanças ao sistema.
Gerenciamento de Projetos	Trata-se de uma atividade de suporte que prevê o ciclo de gestão do projeto de desenvolvimento de software.
Ambiente	Trata-se de uma atividade de suporte com objetivo de prover uma infraestrutura para o ambiente de desenvolvimento.

Tabela 1: Atividades do RUP (Rational Unified Process) (SHUJA & KREBS, 2008).

Em sua perspectiva prática, o RUP propõe boas práticas de Engenharia de Software as quais devem ser adotadas em desenvolvimento de sistemas. Conforme descreve Sommerville (2009), as seis boas práticas fundamentais sugeridas pelo RUP são:

1. ***Desenvolver software iterativamente*:** Devem-se planejar incrementos ao sistema baseado nas prioridades dos clientes, desenvolvendo a princípio os requisitos de alta prioridade;
2. ***Gerenciar requisitos*:** Documentar explicitamente os requisitos do cliente e manter o acompanhamento das mudanças a esses requisitos. Deve-se também analisar o impacto de quaisquer eventuais mudanças ao sistema antes de aceita-las;
3. ***Utilizar arquitetura baseada em componentes*:** Estruturar a arquitetura do sistema em componentes, promovendo melhor manutenabilidade para o sistema;
4. ***Modelar visualmente o software***: Utilizar diagramas UML para modelar o sistema, provendo as visões estáticas e dinâmicas do mesmo;
5. ***Verificar a qualidade do software***: Garantir que o software atende aos padrões de qualidade estabelecidos pelo cliente;
6. ***Controlar mudanças***: Devem-se gerir as mudanças ao sistema utilizando gerenciamento de mudança, procedimentos e ferramentas de gerência de configuração.

Como qualquer framework de processos, o RUP não é perfeitamente aplicável para todos os tipos de desenvolvimento de software (SOMMERVILLE, 2009). Como um framework, o RUP deve ser adaptado para as necessidades e circunstâncias de cada projeto. (LARMAN ET AL., 2001).

2.3.2. Análise Geral do Processo Unificado

Em se tratando dos frameworks baseados no Processo Unificado (*Unified Process*), o RUP é a mais conhecida e utilizada variação comercial na área de Engenharia de Software. Como

um framework, o RUP pode oferecer um conjunto de elementos mais ou menos completos para projetos de desenvolvimento de software. Entretanto, o RUP prevê que seus elementos precisam ser customizados para cada projeto específico. Diversos aspectos devem ser considerados quando se aborda a adequação do modelo proposto pelo RUP para cada característica de projeto.

> "*Uma customização do RUP deve ser baseada na tecnologia utilizada no projeto, no tipo de cliente e no domínio no qual o software irá rodar, assim como outras características de design e desenvolvimento da aplicação.*" (WESTERHEIM ET AL., 2005, p. 1, Tradução do Autor)

É fato que customizar o processo do RUP é importante, contudo o resultado dessa customização também pode ser um problema (WESTERHEIM ET AL., 2005). Alguns padrões de falha na adoção do RUP foram levantados por Larman et al. (2001) e que comumente são encontrados em organizações que utilizam o framework. Os padrões identificados para falhar na adoção do RUP conforme Larman et al. (2001) são:

1. **Impor o pensamento em Cascata**: É fundamental que haja a compreensão de que o RUP propõe um modelo iterativo e incremental e não um modelo cascata. Um processo de software alinhado com os preceitos do RUP não pode se caracterizar por: a) tentar definir e estabilizar a maior parte dos requisitos; b) detalhar o design nas primeiras iterações; c) implementar baseado no design completo; d) integrar, testar e implantar simultaneamente. Desse cenário, Larman et al. (2001) apontam que o entendimento mais equivocado desse contexto é caracterizado pela compreensão errada de que a maior parte dos requisitos precisa ser definido na primeira fase do projeto. Como a Figura 6 apresenta, existe um esforço maior na fase inicial, porém esse não deve ser integral. Uma vez que as iterações ocorrerão, novos entendimentos dos requisitos bem como do negócio poderão surgir.

   > "*Por conseguinte, no RUP, o desenvolvimento prossegue através de uma série de iterações, cada uma das quais são encapsuladas com "timeboxing" para uma duração fixa (como, por exemplo, exatamente quatro semanas), e que termina numa libertação estável de um subconjunto do sistema final. Timeboxing é um conceito-chave em desenvolvimento iterativo: significa fixar a data final da iteração, e normalmente não permitir que essa data seja mudada. Se todos os objetivos não podem ser cumpridos, algumas exigências são removidas da iteração, em vez de expandir a duração da iteração. Dentro de uma iteração, há algo como uma "mini cascata". Um pequeno conjunto de requisitos é escolhidos e mais plenamente analisados (talvez priorizada pelo alto risco ou o valor do negócio).*" (LARMAN ET AL., 2001, p. 3, Tradução do Autor)

2. **Aplicar o RUP como um processo pesado e preditivo**: O RUP foi pensado e encorajado para ser aplicado como leve, ágil e adaptativo. Deve ser utilizado o mínimo de atividades e artefatos possível, sendo apenas as que realmente agregam valor. Na medida em que o projeto evolua, caso algum artefato ou atividade não estiver agregando valor, deve ser descartada; bem como se houver alguma incompletude no processo, deve ser acrescentada a atividade ou artefato

complementar no contexto instanciado para atender a alguma deficiência do projeto. Outro ponto abordado é que não precisa haver o planejamento detalhado para todas as iterações. Elabora-se o cenário geral, com previsões de data para início e fim de cada iteração, contudo, seu detalhamento deve ser gradativo. Como um processo incremental, a especificação da iteração seguinte deve ser feita após a conclusão da iteração anterior;

3. **Evitar o conhecimento específico de ferramentas em detrimento da Orientação a Objetos**: O RUP foi concebido com o foco no desenvolvimento de software orientado a objetos, ainda que possa ser aplicado a outras abordagens de desenvolvimento. O que se observa é que projetos de sistemas orientados a objetos falham ou encaram sérios problemas por não ter pessoas que realmente conheçam o pensamento orientado a objetos, *object design* e padrões de projeto (*design patterns*). Se não houver desenvolvedores realmente capacitados em orientação a objetos, não será o processo que irá salvar o projeto;

4. **Subvalorizar o desenvolvimento iterativo e adaptativo**: Existem diversas formas de ignorar as implicações do desenvolvimento iterativo. Algumas são as imposições de atitudes baseadas no pensamento "Cascata" - uma etapa do processo executada por vez - outras com diferentes ênfases, tais como: adotar atitudes rígidas e preditivas; perverter a prática do desenvolvimento iterativo; não educar os *stakeholders* quanto às implicações do desenvolvimento iterativo; criar diagramas em excesso e artefatos de baixo valor agregado para a manutenção do processo e do projeto;

5. **Evitar mentores que pareçam entender sobre desenvolvimento iterativo, mas não o conhecem**: Devem-se procurar pessoas que deixem de lado o pensamento e atitudes "Cascata". É importante também garantir que toda a equipe esteja alinhada quanto ao entendimento dos conceitos e valores propostos pelo RUP;

6. **Adotar o RUP localmente nos projetos, porém, ignorar a necessidade de apresentá-lo a toda a organização**: É fundamental que a organização compreenda como se dá o fluxo do processo de desenvolvimento. Esse entendimento evita com que os clientes não compreendam, por exemplo, que ao final de cada iteração é possível se obter uma parte completa do sistema;

7. **Obter conselhos de fontes mal informadas**: Pode ser observada na literatura uma série de afirmações incorretas sobre o RUP, tais como: "*No RUP é importante se obter 100% dos requisitos definidos antes de iniciar o design."; "Uma boa e típica iteração no RUP deve durar cerca de seis meses"*.

Conforme os fatores apresentados, os processos podem ser estruturados considerando o desenvolvimento iterativo de forma que as mudanças possam ser feitas sem comprometer o sistema como um todo, assim como o processo de desenvolvimento seja gradativo e evolutivo (SOMMERVILLE, 2009). É fato, porém, que processos em excesso podem comprometer o bom andamento do projeto, assim como a falta deles pode trazer riscos (POLLICE, 2001).

2.4. Considerações finais

Compreende-se que diante da conjuntura atual da indústria de software, é fundamental a adoção de processos, não só para garantir qualidade aos produtos desenvolvidos, como

principalmente, para prover a gestão adequada do ciclo de desenvolvimento (ROCHA ET AL., 2005). O processo de software forma a base para gestão e controle dos projetos de software e estabelece o contexto onde métodos são aplicados, produtos de trabalho (modelos, documentos, dados, relatórios, formulários, etc.) são produzidos, marcos são estabelecidos, a qualidade é assegurada e mudanças são apropriadamente geridas. (PRESSMAN, 2004). Ainda que atualmente a literatura preveja a relevância dos processos de software, essa realidade nem sempre se fez evidente na história evolutiva da indústria de software.

Pôde-se verificar que a indústria de software viveu três momentos específicos no que diz respeito as práticas de processos de software: a primeira fase, onde poucos ou nenhum método formal era adotado para produção de software, sendo essa principalmente focada em iniciativas *ad-hoc*, focadas exclusivamente na codificação; a segunda fase, na qual emergiram modelos prescritivos, baseados em ciclos de vida de desenvolvimento de software, os quais serviram como meta-modelos e defiram diretrizes gerais, como desenvolvimento em cascata, incremental, estrela, entre outros. Nessa fase, os modelos baseado em ciclo de vida incrementais se destacaram, em especial o Processo Unificado (*Unified Process, UP)*, em específico o RUP (*Rational Unified Process*), framework de processo de software iterativo e incremental de propriedade da empresa IBM/Rational Software; por fim, a terceira fase, na qual as metodologias ágeis surgiram como alternativa às prescrição e formalidade em modelos baseados no Processo Unificado. As metodologias ágeis objetivaram focar em valores específicos, como: Indivíduos e interações mais que processos e ferramentas; Software em funcionamento mais que documentação abrangente; Colaboração com o cliente mais que negociação de contratos; Responder a mudanças mais que seguir um plano.

Conclui-se, igualmente, que algumas interpretações equivocadas das organizações que adotaram modelos de processo unificado como o RUP foram cabais para a redução na adoção desses, conforme identificou Larman et al. (2001). Essas divergências entre a proposta do modelo e a execução prática das organizações está identificada em aspectos como (LARMAN ET AL., 2001):

- Impor o pensamento em Cascata;
- Aplicar o RUP como um processo pesado e preditivo;
- Evitar o conhecimento específico de ferramentas em detrimento da Orientação a Objetos;
- Subvalorizar o desenvolvimento iterativo e adaptativo;
- Evitar mentores que pareçam entender sobre desenvolvimento iterativo, mas não o conhecem ;
- Adotar o RUP localmente nos projetos, porém, ignorar a necessidade de apresentá-lo a toda a organização ;
- Obter conselhos de fontes mal informadas.

No próximo capítulo serão analisadas as principais metodologias ágeis disponíveis na literatura, avaliando aspectos gerais desses modelos, tais como: conceitos, princípios, arquitetura e aplicabilidade destes.

3. METODOLOGIAS ÁGEIS DE SOFTWARE

Nesse capítulo são apresentados os conceitos e diversas abordagens previstas na literatura a cerca de Metodologias Ágeis de Desenvolvimento de Software. Serão abordados os principais modelos de processos ágeis, bem como a abordagem de Modelagem Ágil (Agile Modeling) de software.

3.1. Introdução

A Engenharia de Software considera os passos necessários para o desenvolvimento de um produto de software de alta qualidade (AKIF & MAJEED, 2012). A qualidade de produtos de software está diretamente relacionada a como esse foi desenhado pelos engenheiros, de forma que esse desenho melhore os esforços e qualidade do desenvolvimento. Ante esse cenário, diferentes abordagens de desenvolvimento de software são providas na literatura, as quais contam com propostas de políticas, procedimentos e processos, sendo chamados de Metodologias de Desenvolvimento de Software (SDM, do inglês *Software Development Methodologies*) (AKIF & MAJEED, 2012).

Conforme mencionado no capítulo 2, essas diferentes metodologias são baseadas em modelos de ciclo de vida (SDLC, *Software Development Life Cycle*). Quando se propõe a adaptação de qualquer modelo de ciclo de vida para o desenvolvimento de software, busca-se adaptar as políticas, procedimentos e processos de forma a prover qualidade e segurança à solução desenvolvida. Como igualmente explanado no capítulo anterior, existem diversos modelos de ciclo de vida, como Cascata, espiral e Ágil, e esta pesquisa focará na conceituação e análise das metodologias baseadas no ciclo de vida Ágil.

3.2. Visão Geral das Metodologias Ágeis de Software

Compreende-se que em geral, o desenvolvimento de software trata-se de um processo imprevisível e complexo. Deve-se reconhecê-lo igualmente como processo empírico, que aceita imprevisibilidade e por meio de mecanismos específicos toma ações corretivas para responder às imprevisões. Em se tratando das metodologias ágeis de desenvolvimento de software, essas possuem a característica de serem adaptativas em vez de preditivas.

As metodologias ágeis de desenvolvimento de software emergiram para direcionar os diversos desafios encarados pelos desenvolvedores a cerca de seus processos. Uma das principais dificuldades foi considerar abordagens de processos que priorizassem o próprio desenvolvedor, haja visto que modelos prescritivos de software tendem a favorecer a gestão, tal qual como afirma Ambler (2002, pág. 6, Tradução do Autor):

> "*O interessante em processos prescritivos é que eles são atrativos para a gestão, mas não para os desenvolvedores. Processos prescritivos são tipicamente baseados em um paradigma de controle e comando, o qual coloca a gestão no controle das coisas, ou pelo menos, faz com que eles sintam que estão no controle das coisas.*"

Ante ao então cenário de formalismo e prescrição dos modelos de processos de software, um grupo de 17 metodologistas constituíram a *Agile Software Development Alliance*[4], cujo propósito foi discutir os modelos convencionais de desenvolvimento de software, a fim de conceber técnicas que favorecem a inerente necessidade de responder rapidamente às mudanças, aspecto característico da área de desenvolvimento de software. Um aspecto interessante desse grupo é que todos os participantes vinham de diferentes experiências e áreas de especialização na área desenvolvimento de software, e ainda assim conseguiram concordar com os princípios que iriam então compor o Manifesto Ágil.

O Manifesto Ágil é definido por quatro princípios de valor. Uma boa forma de avaliar quanto ao que é o manifesto é que ele define preferências e não alternativas, encorajando em focar áreas específicas e não simplesmente eliminar a outras quanto às práticas de engenharia e desenvolvimento de software. Os valores previstos no Manifesto Ágil são (AMBLER, 2002):

- **Indivíduos e interações mais que processos e ferramentas:** Times de pessoas desenvolvem sistemas de software, e para fazerem isso eles precisam trabalhar efetivamente juntos com programadores, testadores, gerentes de projeto, analistas e clientes;
- **Software em funcionamento mais que documentação abrangente:** Usuários irão compreender qualquer software produzido mais facilmente do que diagramas e documentações técnicas descrevendo a operação interna de um sistema. Documentação tem seu lugar, mas o objetivo principal em desenvolvimento de software é criar softwares e não documentos;
- **Colaboração com o cliente mais que negociação de contratos:** Apenas o cliente pode dizer o que ele deseja. Certamente eles podem não ter as habilidades necessárias para especificar um sistema, mas são detentores da necessidade. Desenvolvedores de sucesso trabalham próximos aos seus clientes, investindo esforço para descobrir suas necessidades e educá-los durante o caminho do projeto;
- **Responder a mudanças mais que seguir um plano:** Pessoas mudam suas prioridades por diversos motivos. À medida que o trabalho evolui no projeto, os envolvidos passam a entender melhor o domínio do problema. Os ambientes de negócios mudam e, a tecnologia muda com o tempo, mesmo que não seja para melhor. A mudança é uma realidade no desenvolvimento de software e essa realidade precisa ser refletida no processo de software adotado.

Para auxiliar as pessoas a compreenderem melhor sobre em que se fundamenta o desenvolvimento ágil de software, os membros da *Agile Alliance* refinaram as filosofias capturadas em seu manifesto em uma coleção de doze princípios, os quais deveriam ser seguidos por qualquer metodologia ágil proposta para o processo desenvolvimento de software (AMBLER, 2002):

1. A prioridade máxima é satisfazer o cliente através de breves e contínuas entregas de software de valor;
2. Mudanças nos requisitos são bem vindas, mesmo tardias ao longo do desenvolvimento. Processos ágeis consideram as mudanças como vantagens competitivas para o cliente;
3. Entrega de software funcionando constantemente, de preferência em intervalos de duas semanas a dois meses;

[4] *Agile Software Development Alliance*, www.agilealliance.org.

4. Pessoas de negócio e desenvolvedores devem trabalhar juntas diariamente ao longo do projeto;
5. Construa projetos em torno de indivíduos motivados. Dê-lhes o ambiente e apoio de que necessitam, e confie neles para fazer o trabalho;
6. O método mais eficiente e eficaz de transmitir informações para e dentro uma equipe de desenvolvimento é conversando "cara-a-cara";
7. Software funcionando é a principal medida de progresso;
8. Processos ágeis promovem o desenvolvimento sustentável. Os patrocinadores, desenvolvedores e os usuários devem ser capazes de manter um ritmo constante indefinidamente;
9. Atenção contínua à excelência técnica e ao bom design aumenta a agilidade;
10. Simplicidade - a arte de maximizar a quantidade de trabalho não feito - é essencial;
11. As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizáveis;
12. Em intervalos regulares, a equipe deve refletir sobre como se tornar mais eficaz e então sintonizar e ajustar seu comportamento.

Com base nos princípios analisados pela *Agile Alliance*, são definidos os valores e atributos gerais de uma metodologia ágil. Para Levine (2005), porém, existem quatro macro atributos característicos em uma metodologia ágil, sendo eles:

- **Rapidez:** O processo precisa ser rápido;
- **Agilidade:** Deve permitir improvisação, usando padrões criativamente para construir novas soluções durante a execução, ou seja, ser flexível;
- **Adaptável:** O processo deve ser responsivo, dinâmico e interativo, de forma a responder aos clientes e a evolução das circunstâncias;
- **Engenhoso:** Reflexivo e que exiba alguma disciplina, percebendo, porém, que não deve utilizar aabordagem tradicional de "comando e controle" e modelos prescritivos.

Lindvall et al. (2002) observou igualmente outros atributos relevantes a processos ágeis de desenvolvimento de software, tais como: ser iterativo; incremental; auto organizável, onde o time deve ter autonomia para se organizar da melhor forma para completar os itens de trabalho; e emergente, onde tecnologia e requisitos são permitidos a emergir através do ciclo de desenvolvimento do produto.

Agilidade, em suma, está relacionada a como retirar ao máximo, na medida do possível, o peso comumente associado com os tradicionais métodos prescritivos de desenvolvimento de software, com objetivo de responder às mudanças em pequenas iterações, tornando mais fácil de perceber problemas em estágios iniciais do projeto (SHRIVASTAVA & DATE, 2010). Nesse sentido, a literatura prevê uma série de metodologias que habilitam diferentes benefícios em diferentes frentes de um projeto de desenvolvimento de software. Como parte dessa pesquisa, serão analisadas as principais metodologias ágeis previstas na literatura, as quais terão seus conceitos e práticas explanadas.

3.3. Scrum

O Scrum trata-se de um processo ágil iterativo incremental para gestão de projetos e desenvolvimento de software (SOMMERVILE, 2009). Seu foco é centrado na premissa de ser "*uma estratégia de desenvolvimento flexível e holística, onde o time de desenvolvimento*

trabalha como uma unidade para alcançar um objetivo comum", a qual se contrapõem a abodagens tradicionais e em cascata (SCHWABER & SUTHERLAND, 2013).

O objetivo do Scrum é possibilitar a criação de times auto-organizáveis, encorajando-os a trabalhar de forma compartilhada, buscando principalmente uma comunicação verbal entre todos os envolvidos nas disciplinas do projeto (SCHWABER & SUTHERLAND, 2013). Um dos princípios básicos do Scrum é o reconhecimento de que durante um projeto, os clientes podem mudar de ideia sobre o que eles desejam e precisam, ou seja, mudar os requisitos (JANOFF & RISING, 2000). Esse desafio imprevisível não pode ser facilmente atendido em uma abordagem tradicional preditiva ou planejada. O Scrum adota uma abordagem empírica – aceitando que os requisitos não podem ser totalmente compreendidos e definidos – o que cria um foco, porém, em maximizar a habilidade do time para entregar rapidamente e responder às mudanças dos requisitos (SCHWABER & SUTHERLAND, 2013).

O Scrum foi desenvolvido nas décadas de 80 e 90 por Ken Schwaber, Jeff Sutherland e Mike Beedle (SATO, 2007). Sua proposta foi concentrar principalmente nos aspectos gerenciais do desenvolvimento de software, baseando-se em conceitos do desenvolvimento iterativo, propondo ciclos de entregas entre duas semanas a 30 dias. Com objetivo de facilitar o entendimento entre o time, propôs-se um acompanhamento do projeto por meio de reuniões rápidas em pé, as chamadas *Stand-up Meetings*. Por dar menos enfoque em aspectos técnicos do processo de desenvolvimento de software, requer a utilização de outras práticas ágeis, tais como as de XP, para complementar o processo de desenvolvimento (SATO, 2007).

3.3.1. Visão Geral do Processo

O ciclo de vida do Scrum inicia-se com o estabelecimento do *Product Backlog*, do português "*Histórico do Produto*", o qual consiste em uma lista de requisitos priorizados para desenvolvimento. Esses requisitos são estabelecidos junto aos *stakeholders* do projeto e têm em sua característica um baixo nível de detalhamento, sendo sugerido que apenas os aspectos principais do requisitos sejam elicitados. A partir do estabelecimento do *Product Backlog*, inicia-se o processo de desenvolvimento, o qual será executado iterativamente por meio de *Sprints* (SCHWABER & SUTHERLAND, 2013).

O *Sprint* é a unidade básica de desenvolvimento no Scrum. É um esforço com tempo estabelecido, isto é, é restrito a uma duração específica. A duração é fixada com antecedência para cada *sprint* e dura normalmente entre uma semana a um mês, apesar de duas semanas ser o período comumente estabelecido (SATO, 2007). Cada *sprint* é iniciado por uma reunião de planejamento, a *Sprint Planning Meeting*, onde as tarefas para o *sprint* são identificadas – conforme constam no *Product Backlog* – e um compromisso estimado para o objetivo do *sprint* é estabelecido. Esse compromisso de itens para o *sprint* é chamado de *Sprint Backlog*, o qual nada mais é do que os itens do *Product Backlog* que farão parte do referido *sprint*. Ao longo do desenvolvimento de um *sprint*, são realizadas reuniões diárias de acompanhamento, chamadas de *Daily Scrum Meeting*. A realização desse encontro é embasada em alguns princípios, tais como: a reunião começa na hora marcada mesmo sem a presença de algum membro do time; a reunião precisa ser realizada no mesmo local e hora todos os dias; a reunião deve durar no máximo 15 (quinze) minutos; todos são bem vindos, porém apenas os papéis principais falam.

Ao final de cada ciclo de execução de um *sprint*, são realizadas duas reuniões: a *Sprint Review Meeting*, cujo objetivo é revisar o trabalho concluído e quais não foram concluídos; e

a *Sprint Retrospective*, cujo objetivo é promover a reflexão dos membros do time sobre o que ocorreu no último *sprint* e quais melhorias de processo podem ser realizadas para o próximo *sprint*. Ao final de cada *sprint*, é esperado que entregas de produtos funcionando sejam feitas ao cliente. O Scrum enfatiza, que entregas de software consideram que o sistema foi integrado, totalmente testado e a documentação para o usuário foi produzida (JANOFF & RISING, 2000).

Figura 7: Estrutura e ciclo de vida do Scrum (Ilustração do Autor).

Terminologias

O Scrum utiliza-se de uma série de conceitos e terminologias importantes para seu entendimento e adequada adoção. Esses termos são apresentados na tabela 2:

Abnormal Termination	Do português "*Encerramento anormal*", significa que o Product Owner pode cancelar um Sprint, se necessário. O Product Owner pode fazer isso com o incentivo do time, Scrum Master ou da gestão. Por exemplo, a gestão deseja cancelar um Sprint se circunstâncias externas negarem o valor do objetivo do Sprint. Se um Sprint é encerrado de forma anormal, o próximo passo é conduzir uma nova reunião de planejamento de Sprint, onde as razões do encerramento serão revistas.
Definition of Done (DoD)	Do português "*Definição de Feito*", é um critério de saída para determinar se um item do Product Backlog está completo. Em muitos casos, DoD requer que todos os testes de regressão[5] tenham sido bem sucedidos.
Development Team	Do português "*Time de Desenvolvimento*", consiste em um grupo multifuncional de pessoas responsável por disponibilizar incrementos entregáveis do Produto ao final de cada Sprint.
Impediment	Do português "*Impedimento*", consiste em qualquer coisa que impeça um membro do time de executar o trabalho da forma mais eficiente possível.
Product Backlog (PBL)	Do português "*Histórico do Produto*", consiste em uma lista de requisitos priorizados para desenvolvimento.
Product Owner	Do português "*Dono do Produto*", é a pessoa responsável por manter o

[5] Teste de regressão é uma técnica do teste de software que objetiva garantir que não surgiram novos defeitos em componentes já analisados. Se, ao juntar o novo componente ou as suas alterações com os componentes restantes do sistema surgirem novos defeitos em componentes inalterados, então considera-se que o sistema regrediu. Muitas vezes são usadas ferramentas específicas para o teste de regressão, chamadas de ferramentas de automação. Elas conseguem um resultado mais exato do teste executando exatamente os passos seguidos para o teste das primeiras versões já que elas permitem a gravação do teste. (IEEE, 2004)

	Product Backlog, representando os interesses dos *stakeholders* e garantindo o valor do trabalho que o time de desenvolvimento executa.
Release Burndown Chart	Do português "*Gráfico de Manejo de Entrega*", consiste em um gráfico do percentual de itens concluídos do Product Backlog em nível de Sprint
Sashimi	Relatório de que algo está completo. A definição de completo pode variar de time para time, mas precisa ser consistente dentro de um time. Utilizado dentro da realização de um sprint.
Scrum But	É uma exceção ao conceito puro da metodologia Scrum, onde o time muda customiza o Scrum para atender as suas próprias necessidades.
Scrum Master	Do português "*Mestre Scrum*", é a pessoa responsável por manter o processo Scrum, garantindo que ele é utilizado corretamente e seus benefícios são maximizados.
Scrum Team	Do português "*Time Scrum*", é o grupo formado por Product Owner, Scrum Master e Development Team.
Spike	Período fixo de tempo utilizado para pesquisar um conceito e/ou criar um protótipo simples. Spikes podem ser planejados para ocorrer entre Sprints ou, para as equipes maiores, um Spike poderia ser aceito como um dos muitos objetivos de entrega do Sprint. Spikes são frequentemente lançados antes da entrega dos itens do Product Backlog grandes ou complexos, a fim de garantir orçamento, ampliar o conhecimento e / ou produzir uma prova de conceito. A duração e objetivo de um Spike serão acordados entre o Product Owner e o Time de Desenvolvimento antes do início. Ao contrário de compromissos de Sprint, Spikes podem ou não entregar funcionalidades tangíveis, funcionais. Por exemplo, o objetivo de um Spike pode ser alcançar com sucesso uma decisão sobre um curso de ação.
Sprint	Um período de tempo (tipicamente entre 1 a 4 semanas) onde o desenvolvimento ocorre e um grupo de itens do Product Backlog é desenvolvido conforme compromisso do Time de Desenvolvimento.
Sprint Backlog (SBL)	Lista de tarefas priorizadas para serem completadas dentro de um Sprint.
Sprint Burndown Chart	Do português " *Gráfico de Manejo do Sprint*", consiste em um gráfico com o progresso diário ante o que foi planejado para esse Sprint.
Tasks	Do português "*Tarefas*", são itens adicionados ao Sprint Backlog no começo de um Sprint e são quebradas em horas. Cada tarefa não pode exceder 12 horas (ou dois dias).
Velocity	O esforço total que uma equipe é capaz de executar em um sprint. O número é obtido pela avaliação do trabalho (normalmente em pontos de Estórias de Usuário) concluído a partir de itens do backlog do último Sprint. A coleta de dados históricos da velocidade é uma diretriz para ajudar a equipe a compreender o quanto de trabalho que eles podem fazer em um Sprint.

Tabela 2: Terminologias do Scrum (SCHWABER & SUTHERLAND, 2013, Tradução do Autor).

3.3.2. Papéis

O Scrum apresenta três funções principais – *core* roles – e outras funções auxiliares – *auxiliary roles* (SCHWABER & SUTHERLAND, 2013). Embora outros papéis possam ser encontrados em projetos reais, o Scrum foca sua abordagem nos papéis principais, haja visto que são eles que produzem o produto (SOMMERVILLE, 2009). Os papéis do Scrum são apresentados a seguir:

- **Product Owner (Proprietário do Produto):** O *Product Owner* é responsável por maximizar o valor do produto e do trabalho da equipe de desenvolvimento

(SCHWABER & SUTHERLAND, 2013). O *Product Owner* é o único encarregado por gerir o *Product Backlog*. Essa gestão está relacionada a: a) claramente expressar os itens do *Product Backlog*; b) ordenar os itens do *Product Backlog* de forma a melhor atingir os objetivos e missões do projeto; c) otimizar o valor do trabalho executado pelo time de desenvolvimento; d) garantir que o *Product Backlog* é compreendido por todos os membros do time e apresenta com clareza no que o time Scrum irá trabalhar; e) garantir que o time de desenvolvimento entende os itens do *Product Backlog* no nível necessário.

- **Development Team (Time de Desenvolvimento):** O time de desenvolvimento é responsável pela entrega de incrementos de produtos potencialmente utilizável no final de cada *Sprint* (SCHWABER & SUTHERLAND, 2013). O Scrum sugere times compostos de 3 a 9 indivíduos com habilidades multifuncionais para execução do trabalho (analisar, projetar, desenvolver, testar, documentar, entre outros) (SOMMERVILLE, 2009).
- **Scrum Master (Mestre Scrum):** Scrum prevê um papel garantidor aos conceitos do seu ciclo de vida, chamado de *Scrum Master.* Ele é responsável pela remoção de impedimentos à capacidade da equipe para entregar as metas de produtos e entregas (JANOFF & RISING, 2000). O *Scrum Master* não é um líder de equipe ou gerente de projeto tradicional, e sim um facilitador para o time e principalmente, um analista do processo Scrum (SOMMERVILLE, 2009). O *Scrum Master* garante que o processo Scrum é usado como pretendido, pela aplicação das regras do Scrum, presidindo as reuniões-chave, e desafiando a equipe a melhorar.

3.3.3. Práticas e Execução do Processo

O Scrum prevê a realização de eventos específicos ao logo do ciclo de vida de execução do seu processo. Esses eventos são listados na tabela 3:

Sprint Planning Meeting (Reunião de Planejamento de *Sprint*)	No início de um ciclo de *sprint* (cada 7 a 30 dias), uma reunião de planejamento de *sprint* é realizada. Os principais direcionamentos são: • Selecionar o trabalho a realizar; • Preparar o *Sprint Backlog* o qual detalha o tempo de cada trabalho a ser realizado pelo time; • Identificar e comunicar quanto do trabalho será provavelmente finalizado durante o *sprint* atual; • Deve ter o limite de oito horas, onde: as primeiras quatro horas contam com todo o time e o objetivo é dialogar para priorizar o *Product Backlog*; as quatro horas finais contam com o time de desenvolvimento, definindo o planejamento para o *sprint* e assim estabelecendo o *Sprint Backlog*.
Daily Scrum Meeting (Reunião Diária de Scrum)	Durante a execução de um *sprint*, diariamente é realizada uma reunião de comunicação entre o time do projeto. Os principais direcionamentos são: • Todos os membros da equipe de desenvolvimento devem estar preparados com os reportes das suas atividades para a reunião; • A reunião começa precisamente na hora marcada mesmo que alguns membros da equipe de desenvolvimento não estejam presentes; • A reunião deve acontecer no mesmo local e hora todos os dias; • A duração da reunião deve ser de 15 minutos. • Todos são bem vindos, mas normalmente apenas as funções principais falam.

End Meetings (Reuniões de Fechamento)	Ao final de cada ciclo de execução de um *sprint*, são realizadas duas reuniões: a *Sprint Review Meeting*, cujo objetivo é revisar o trabalho concluído e quais não foram concluídos; e a *Sprint Retrospective*, cujo objetivo é promover a reflexão dos membros do time sobre o que ocorreu e quais melhorias de processo podem ser realizadas para o próximo *sprint*.
Backlog Refinement / Grooming (Reuniões de Refinamento)	O *Backlog Refinement* (Refinamento do *Backlog*) é a revisão de itens do *Product Backlog* para verificação se eles estão devidamente priorizados e elicitados de uma forma compreensível para as equipes.

Tabela 3: Eventos do Scrum (SCHWABER & SUTHERLAND, 2013).

3.3.4. Artefatos

Os artefatos do Scrum representam o trabalho ou valor para prover transparência e oportunidades para inspeção e adaptação (SCHWABER & SUTHERLAND, 2013). Artefatos do Scrum são definidos para maximizar a transparência de informações chave, de forma que todos tenham o mesmo entendimento sobre o artefato.

- **Product Backlog:** O *Product Backlog* é uma lista ordenada de todos os requisitos do produto, sendo a única fonte para as alterações a serem feitas ao produto. O *Product Owner* é o responsável pelo *Product Backlog* no que diz respeito, seu conteúdo, disponibilidade e pedidos de mudança. É um artefato que nunca está completo (SCHWABER & SUTHERLAND, 2013). Inicialmente são estabelecidos apenas os requisitos conhecidos e melhor entendidos. Na medida em que o produto e o ambiente em que ele será usado evoluem, o *Product Backlog* é incrementado. Trata-se de um artefato dinâmico, que muda constantemente para identificar o que o produto precisa para ser apropriado, competitivo e útil. Enquanto um produto existe, o seu *Product Backlog* existe. O *Product Backlog* lista todos os recursos, funções, exigências, melhorias e correções que constituem as mudanças a serem feitas no produto em versões futuras. Itens do *Product Backlog* têm como seus atributos uma descrição, a ordem, estimativa e valor (JANOFF & RISING, 2000).
- **Sprint Backlog:** O *Sprint Backlog* consiste na lista de itens de trabalho a serem executados pelo time de desenvolvimento durante o *sprint*. Essa lista é uma derivação do *Product Backlog*, e é estabelecida a cada planejamento de *sprint*. O *Sprint Backlog* é de propriedade do time de desenvolvimento, haja vista se tratar dos seus objetivos de entrega para o *sprint*. Habitualmente, o *Sprint Backlog* é acompanhado através de um *Task Board* (Painel de Atividades), onde são listados em colunas: as estórias de usuários (requisitos); atividades a executar; atividades em execução; e por fim atividades concluídas. Na Figura 8 é exibido um exemplo de *Task Board* para acompanhamento do *Sprint Backlog*.

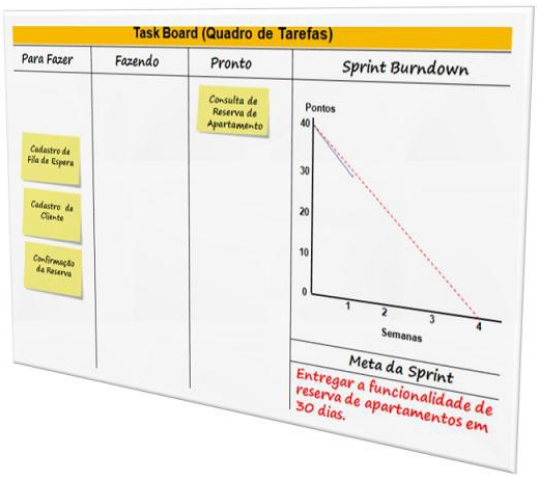


Figura 8: Quadro de Tarefas (*Story Board*) do Scrum. (Ilustração disponível em http://www.rildosan.com/2011/02/para-dar-transparencia-ao-Scrum-use-o.html acessada em 21/12/2013).

- **Increment:** O *Increment* (Incremento), consiste na sumarização de todos os itens completados do *Product Backlog* durante um *sprint* e todos os *sprints* anteriormente executados. Ao final de cada *sprint*, o *Increment* precisa ser realizado conforme o critério estabelecido pelo time Scrum quanto ao *Definition of Done (DoD*, do português "Definição de Feito"). O *Increment* precisa estar utilizável independentemente do *Product Owner* decidir por liberá-lo.
- **Burndown Chart:** O *Burndown Chart* é um gráfico exibido publicamente mostrando o trabalho restante no S*print Backlog*. Atualizado todos os dias, dá uma visão simples do progresso do *sprint*. Ele também fornece visualizações rápidas para referência. Existem também outros tipos de aplicações, como por exemplo, o *burndown chart* que mostra a quantidade de trabalho para completar o compromisso para um lançamento do produto (normalmente abrangendo várias iterações).

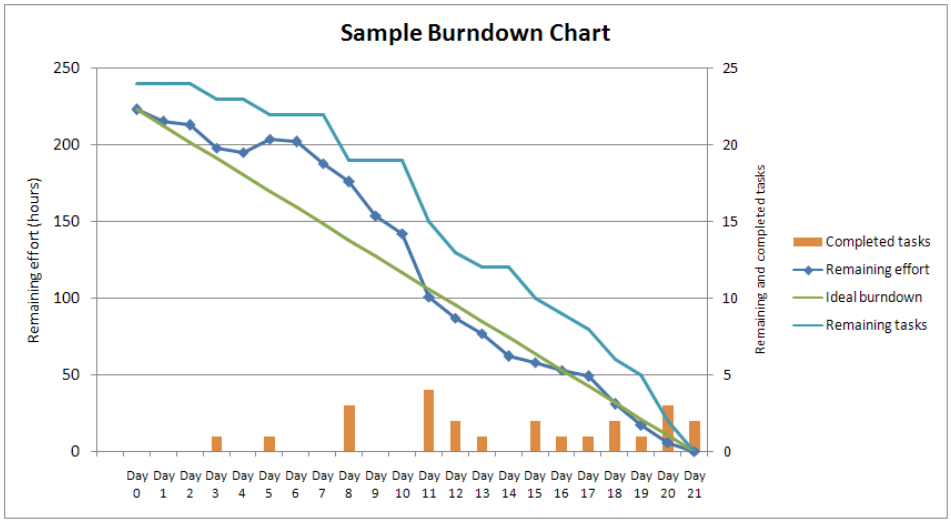


Figura 9: Exemplo de *Burndown Chart*. (Ilustração disponível em http://www.dainf.ct.utfpr.edu.br/~adolfo/etc/slides/metodos_ageis_iniciantes/metodos_ageis_iniciantes.html acessada em 21/12/2013).

3.3.5. Considerações Finais

O Scrum apresenta-se como uma metodologia ágil para gestão de projetos de software (SCHWABER & SUTHERLAND, 2013). A estrutura do processo prevê papéis principais os

quais são responsáveis por gerir e executar todo o ciclo de vida do projeto, através de artefatos dinâmicos e concisos, sendo o foco principal do processo o trabalho em equipe e o apoderamento dos times de desenvolvimento (JANOFF & RISING, 2000). Compreende-se, porém, que é comum a adoção híbrida entre o Scrum e outros processos de software, haja vista que o Scrum não cobre todo o ciclo de vida de desenvolvimento de um produto de software – tais como o detalhamento de aspectos técnicos do desenvolvimento (BEGEL & NAGAPPAN, 2007). Assim, é possível que alguns projetos precisem adotar processos complementares para criar uma implementação mais compreensiva, tal como: processo de levantamento de requisitos de software; design da aplicação; orçamento do projeto e previsão de cronograma (AKIF & MAJEED, 2012).

3.4. Extreme Programming

Extreme Programming (XP, do português Programação Extrema) é uma metodologia ágil de desenvolvimento de software que objetiva melhorar a qualidade do software e a capacidade de resposta à evolução das necessidades dos clientes (BECK, 2000). Como sugere o desenvolvimento ágil de software, o XP propõe *releases* (entregas) frequentes em ciclos curtos de desenvolvimento, objetivando melhorar a produtividade e introduzir pontos de verificação em que podem ser adotados novos requisitos dos clientes (WAKE, 2002).

O XP incorpora em sua metodologia uma série de elementos e práticas, tais como: programação em pares; revisões de código; desenvolvimento orientado a testes; simplicidade e clareza no código, esperando mudanças nos requisitos do cliente; comunicação frequente com o cliente e entre os programadores (SOMMERVILLE, 2009). A metodologia propõe a ideia de que os elementos benéficos de práticas de engenharia de software tradicionais sejam levados para níveis "extremos" (BECK, 2000). Como exemplo, revisões de código são consideradas uma prática benéfica, levada ao extremo, o código pode ser revisto de forma contínua, através da prática de programação em pares (WAKE, 2002).

Extreme Programming foi criado por Kent Beck durante seu trabalho no projeto de folha de pagamento da Chrysler. Beck tornou-se o líder do projeto e começou a refinar a metodologia de desenvolvimento utilizada, vindo a escrever um livro esta (BECK, 2000). Embora o conceito de Extreme Programming seja relativamente novo, muitas de suas ideias já eram utilizadas como “boas práticas”, tais como o uso de testes unitários em todo o código e a prática de programação em pares (BECK, 2000).

A adoção de XP gerou interesse nas comunidades de software no final de 1990 e início de 2000, principalmente com a emergente adoção de metodologias ágeis de desenvolvimento de software. A alta disciplina exigida pelas práticas originais muitas vezes inviabilizava a adoção de processos, fazendo com que algumas dessas práticas não fossem adotadas (LINDVALL ET AL., 2002). Atualmente o uso de XP está evoluindo, com mais lições de experiências na prática disponíveis para *benchmarks*. Nas seções a seguir serão explorados os conceitos e práticas do Extreme Programming.

3.4.1. Visão Geral do Processo

Conforme descreve Beck (2000), XP (Extreme Programming) é uma forma leve, eficiente, de baixo risco, flexível, previsível, científica e divertida de se produzir software. A proposta do XP é aplicar uma série de práticas consolidadas de desenvolvimento sob uma mesma perspectiva, garantindo que essas sejam aplicadas o mais minuciosamente possível e que

todas essas práticas suportem umas as outras em um nível mais grandioso possível (BECK, 2000).

O XP compreende que o problema básico no desenvolvimento de software é o risco (BECK, 2000; WAKE, 2002; LAYMAN ET AL., 2004). Riscos como: falhas de cronograma, cancelamento de projetos, má compreensão do negócio, rotatividade da equipe, entre outros. Para cada um dos possíveis riscos relatados anteriormente, o XP propõe abordá-los da seguinte forma (BECK, 2000):

- **Falhas de cronograma:** XP exige ciclos curtos liberações (releases), de alguns meses no máximo, de modo que o impacto de qualquer falha é limitado. Dentro de uma liberação (release), o XP usa iterações de uma a quatro semanas para atender a requisitos solicitados pelo usuário, facilitando dessa forma o entendimento do progresso do desenvolvimento;
- **Cancelamento do projeto:** XP sugere ao cliente a escolher a menor liberação (release) que faz mais sentido ao negócio, de forma que há menos a dar errado e menos em produção antes do produto ter um alto valor;
- **Má compreensão do negócio:** XP requisita que o cliente seja uma parte integral do time. A especificação é continuamente refinada durante o desenvolvimento, garantindo o atendimento aos requisitos de negócio;
- **Rotatividade da equipe:** XP incentiva o contato humano entre a equipe, reduzindo a solidão que muitas vezes é o foco da insatisfação no trabalho. XP incorpora um modelo explícito de rotatividade de pessoal. Novos membros da equipe são encorajados a aceitar gradualmente mais e mais responsabilidades, e são assistidos ao longo do caminho.

Para ser bem sucedido, o XP propõe uma série de valores básicos que precisam ser buscados para atender tanto as necessidades humanas quanto comerciais em um projeto de software (WAKE, 2002). São quatro os valores propostos pelo XP (BECK, 2000):

- **Comunicação:** É fundamental que a comunicação seja intensa, tanto entre a equipe de desenvolvimento quanto com o cliente;
- **Simplicidade:** Sempre se deve ter em mente a pergunta "*Qual a ação mais simples que poderia funcionar?*" Essa abordagem leva a equipe a sempre pensar em formas simples de resolver os desafios que surgem ao longo do projeto;
- **Feedback:** Um retorno adequado a respeito do estado atual do sistema é absolutamente fundamental. O time precisa compreender que feedback exige diferentes frequências – horas, dias, semanas, meses – e que independentemente de sua periodicidade, precisam ser previstos e acima de tudo, realizados;
- **Coragem:** Complementarmente aos três valores anteriores, a coragem será o fator determinante para suportar os demais.

A partir dos valores previstos pelo XP, são derivados princípios básicos que deverão guiar o time de desenvolvimento. Estes princípios auxiliam quando é necessário escolher entre alternativas, no caso de uma alternativa que atender mais plenamente aos princípios do que outra. São princípios fundamentais do XP (BECK, 2000; WAKE, 2002):

- **Feedback rápido:** Um dos princípios é obter feedback, interpretá-lo e implementar no sistema as observações recebidas no feedback o mais rápido possível;

- **Assumir simplicidade:** Tratar cada problema como se ele pudesse ser resolvido de uma forma muito simples. Conforme afirma Beck (2000), este é o princípio mais difícil para os programadores de compreender. Tradicionalmente programadores planejam o futuro, projetam reutilização de código e etc. Em vez disso, XP diz para fazer um bom trabalho (testes, refatoração, comunicação) hoje e confiar em sua capacidade de adicionar complexidade no futuro, quando você precisar dela;
- **Mudanças incrementais:** Muitas e grandes mudanças sendo feitas simultaneamente não funcionam. Qualquer problema é resolvido com uma série de pequenas mudanças que fazem a diferença;
- **Abraçar mudanças:** As mudanças precisam ser compreendidas como fundamentais para o software, do contrário será produzido um sistema que não atende às necessidades do cliente;
- **Trabalho de qualidade:** O XP é extremamente exigente quanto a esse princípio. Em um projeto de quatro variáveis – escopo, custo, tempo e qualidade – qualidade não é uma variável livre. Para o XP, os únicos valores possíveis são "excelentes" e "insanamente excelente" (BECK, 2000).

Existem outros princípios não centrais do XP e que podem auxiliar sobre o que fazer em determinadas situações. São eles: aprenda ensinando; investimento inicial pequeno; jogue para ganhar; experimentos concretos; comunicação aberta e honesta; trabalhe com os instintos das pessoas, não contra eles; aceite responsabilidades; adaptação local; viagens leves; métricas honestas (BECK, 2000; WAKE, 2002).

O XP baseia-se no conceito de que precisa realizar todo seu processo de uma forma estável e previsível para o desenvolvimento de software (SOMMERVILLE, 2009). Considerando essa premissa, o XP descreve quatro atividades específicas no seu processo de desenvolvimento (BECK, 2000):

- **Codificação (*Coding*):** O XP afirma que o produto verdadeiramente importante do processo de desenvolvimento do sistema é o código. Sem código, não há produto. Um programador de lidar com um problema de programação complexa, ou que encontre dificuldades para explicar a solução para colegas programadores, pode codificá-lo de uma forma simplificada e usar o código para demonstrar o que quer dizer. Código deve ser sempre claro e conciso e não pode ser interpretado de mais de uma maneira. Outros programadores podem fazer comentários sobre este código, também codificando seus pensamentos;
- **Testes (*Testing*):** A abordagem do XP é que se um pequeno teste pode eliminar algumas falhas, uma série de testes pode eliminar muitas falhas. Nesse sentido, o XP propõe dois tipos de testes: 1) Testes de Unidade (*Unity Tests*), os quais determinam se um determinado recurso funciona como pretendido (SOMMERVILLE, 2009). O XP advoga que o programador deve escrever quantos testes automatizados ele pensar que poderia "quebrar" o código; se todos os testes forem executados com êxito, a codificação é concluída. Cada pedaço de código que é escrito é testado antes de passar para a próxima implementação; 2) Testes de Aceitação (*Acceptance Tests*), os quais verificam se os requisitos compreendidos pelos programadores realmente atendem aos requisitos do cliente. Testes de integração completa do sistema eram inicialmente encorajados pelo XP, como atividades a serem realizadas no final do dia, como forma de identificar previamente problemas de interfaces. Contudo, testes de integração completa têm sido reduzidos a atividades semanais ou ainda menos frequentes, dependendo da estabilidade geral das interfaces do sistema (WAKE, 2002);

- **Escutar (*Listening*):** Os programadores devem ouvir o que os clientes precisam que o sistema execute, qual "lógica de negócio" é necessária. Eles devem entender essas necessidades bem o suficiente para dar o feedback ao cliente sobre os aspectos técnicos de como o problema pode ou não ser resolvido;
- **Projetar (*Designing*):** Do ponto de vista da simplicidade, pode-se dizer que o desenvolvimento não precisa mais do que codificar, testar e executar. Assume-se que se essas atividades têm um bom desempenho, o resultado deve ser sempre um sistema que funciona. Na prática, porém, isso não funciona. Pode-se percorrer um longo caminho em um projeto sem projetar, mas em um determinado momento isso vai se tornar um problema (AMBLER, 2002). O sistema torna-se muito complexo e as dependências dentro dele deixam de ser claras. Um bom projeto (design) evita dependências dentro de um sistema, permite a visão geral de impacto e dependência interna entre componentes do sistema, entre outros aspectos.

Compreende-se que o foco XP é concentrar-se na codificação, uma vez que para ele, "*sem codificação não se tem nada*" (BECK, 2000; SHRIVASTAVA ET AL., 2010). Adicionalmente, as atividades do XP fazem uma correlação com seus próprios princípios e valores:

> *"Então você codifica, pois se não codificar, você não fez nada. Você testa, porque se não testar, você não sabe quando você sua codificação foi terminada. Você ouve, porque se você não ouvir, você não sabe o que codificar ou o que testar. E você projeta para que possa manter a codificação e testes e ouvir indefinidamente. É isso aí. Essas são as atividades que temos para ajudar a estrutura: Codificar, Testar, Escutar e Projetar." (BECK, 2000, p. 44, Tradução do Autor)*

Considerados os conceitos, valores, princípios e atividades previstas em Extreme Programming (XP), se faz necessária a compreensão de como esses aspectos são abordados em um cenário prático de um projeto de desenvolvimento de software que utiliza XP como processo, o qual será apresentado na seção a seguir.

3.4.2. Papéis

Assim como em qualquer contexto onde existe um time, cada membro desse time desempenha um papel importante e que precisa ser previamente definido (WAKE, 2002). Em XP, os papéis são estabelecidos com foco no objetivo central do processo que é a codificação (SHRIVASTAVA ET AL., 2010). São papéis no XP:

- **Programador (*Programmer*):** É definido como o "coração" do XP (BECK, 2000). O Programador é encarregado não só por codificar, mas por tomar decisões, definir alternativas, balancear prioridades de curto e longo prazo, entre outras demandas relevantes a um projeto XP;
- **Cliente (*Customer*):** O Cliente é outra parte essencial em um projeto XP. Baseado na prática de "*On Site Customer*", é fundamental que o cliente esteja disponível para participar do processo de desenvolvimento como figura ativa;
- **Testador (*Tester*):** Uma vez que boa parte das responsabilidades de testes recaem sobre os Programadores, o papel do Testador no XP é centrado no cliente. A responsabilidade do Testador é auxiliar o cliente a escolher e escrever testes funcionais de aceitação;

- **Rastreador (*Tracker*):** O XP define o Rastreador como a "consciência do time" (BECK, 2000). O papel do Rastreador está diretamente relacionado a estimativas, catalogação de dados de testes, apoio ao time no caso de mudanças, entre outros aspectos relacionados a atividades de suporte ao processo XP;
- **Treinador (*Coach*):** O papel de Treinador é responsável pelo processo XP. Seu papel é garantir que o time está se mantendo dentro dos padrões, práticas, valores e princípios estabelecidos por XP;
- **Consultor (*Consultant*):** O Consultor é um papel variante cujo objetivo é suportar ao time no caso deste eventualmente precisar de algum apoio de conhecimento técnico. Pode ser requerido no caso de uso de novas tecnologias, tecnologias proprietárias, entre outros;
- **Chefe (*Big Boss*):** O papel do Chefe está centrado principalmente na liderança. Um Chefe em XP é alguém em que o time encontre uma referência de coragem e confiança. O Chefe utiliza-se da comunicação com o time para direcionar decisões, apoiar em escolhas no projeto, motivar o time, entre outras ações que ojetivem atingir o propósito do projeto.

3.4.3. Práticas e Execução do Processo

O Extreme Programming propõe um processo de software onde quatro valores – comunicação, simplicidade, feedback e coragem – habilitam princípios - feedback rápido, assumir simplicidade, mudanças incrementais, abraçar mudanças, trabalho de qualidade – na execução de quatro atividades básicas – codificar, testar, escutar e projetar (BECK, 2000; WAKE, 2002). Essa estrutura é respaldada em uma série de práticas, as quais refletem os princípios de metodologias ágeis (SOMMERVILLE, 2009). Essas práticas, conforme apresentado na tabela 4, são fundamentais na execução do XP e devem ser seguidas fielmente para a efetiva adoção do Extreme Programming em sua essência,

Planning Game (Planejamento do Jogo)	Deve ser determinado rapidamente o escopo da próxima versão, combinando prioridades de negócios e estimativas técnicas. Quando a realidade superar o plano, o plano deve ser atualizado.
Small Releases (Entregas Curtas)	Um sistema simples deve ser colocado em produção rapidamente, em seguida, novas versões devem ser lançadas em um ciclo mais curto.
Metaphor	Todo o desenvolvimento deve ser guiado com uma história compartilhada simples de como todo o sistema funciona.
Simple Design (Design Simples)	O sistema deve ser projetado da forma mais simples possível. Complexidade extra deve ser removida assim que for descoberta.
Testing (Testes)	Programadores implementam testes de unidade continuamente, que devem ser executados sem erros para o desenvolvimento continuar. Os clientes descrevem testes demonstrando que os requisitos foram atendidos (testes de aceitação).
Refactoring (Refatoração)	Programadores reestruturam o sistema sem alterar seu comportamento para remover duplicação, melhorar a comunicação, simplificar ou adicionar flexibilidade.
Pair Programming (Programação em Pares)	Todo o código de produção é escrito por dois programadores em uma máquina.
Collective Ownership (Propriedade Coletiva)	Qualquer um pode alterar qualquer código em qualquer lugar no sistema a qualquer momento.
Continuous Integration (Integração Contínua)	Construir e integrar o sistema muitas vezes por dia, cada vez que uma tarefa ou um requisito seja concluído.
Sustainable Pace	Como regra, o trabalho não deve durar mais que 40 horas por semana. Nunca se

(Ritmo Sustentável)	deve fazer hora extra por duas semanas consecutivas.
On-Site Customer (Cliente no Local)	Incluir, um cliente na equipe, disponível em tempo integral para responder perguntas.
Coding Standards (Padrões de Codificação)	Programadores escrevem todo o código de acordo com regras que enfatizam comunicação através do código.

Tabela 4: Soluções propostas por Extreme Programming (BECK, 2000, Tradução do Autor).

Além do estabelecimento de práticas específicas para a execução do processo, o XP define um ciclo de vida padrão para seus projetos, bem como o estabelecimento de papéis específicos a serem executados.

Ciclo de Vida de um projeto XP

Conforme afirma Beck (2000), "*um projeto XP ideal passa por um pequeno período inicial de desenvolvimento, seguido por anos de produção simultânea, suporte e refinamento, e finalmente uma aposentadoria tranquila quando o projeto não mais fizer sentido*". Essa assertiva embasa que o XP, apesar de ser um processo centrado no desenvolvimento e codificação (WAKE, 2002; SHRIVASTAVA ET AL., 2010), possui igualmente um ciclo de vida o qual é transpassado ao longo da execução de um projeto de software. O ciclo de um projeto XP considera:

- **Exploração (*Exploration*):** A fase de Exploração é o lugar onde todos os conceitos do XP são reunidos. A Exploração termina quando o cliente está confiante de que há mais do que material suficiente descritos em cartões de estórias de usuário para fazer uma boa primeira versão do sistema e os programadores estão confiantes de que não é possível estimar melhor sem realmente implementar o sistema;
- **Planejamento (*Planning*):** O objetivo da fase de Planejamento é que os clientes e os programadores concordem com confiança sobre a data em que o menor e o mais valioso conjunto de estórias será produzido. A prática de *Planning Game* (Jogo de Planejamento) auxilia na execução dessa fase. O Planejamento (produção do cronograma de compromissos) deve demorar entre um ou dois dias;
- **Iteração para Primeira Entrega (*Iterations to first Release*):** A Iteração para Primeira Entrega foca na entrega da arquitetura. Sugere-se a escolha de estórias para a primeira iteração que forcem a criar "todo o sistema", mesmo que seja em forma de esqueleto;
- **Produção (*Productionizing*):** Durante a Produção, aumenta-se o envolvimento no desenvolvimento do sistema, baseando-se principalmente no conceito de "*make it run, make it right, make it fast*" (do português "faça rodar, faça certo, faça rápido";
- **Manutenção (*Maintenance*):** A fase de Manutenção é considerada o "estado normal" em um projeto XP. Faz-se necessário produzir novas funcionalidades, manter os sistemas existentes rodando e incorporar novas pessoas ao time.
- **Finalização (*Death*):** O time precisa compreender que os sistemas eventualmente deixam de ser úteis. Nesse sentido, é preciso haver um alinhamento entre o cliente, gestores e demais envolvidos de que o sistema não entrega mais o que é necessário.

3.4.4. Artefatos

Considerando seu valor de simplicidade, o XP considera uma quantidade de artefatos bastante reduzida (BECK, 2000). De fato, em XP não se aborda artefatos como em processos preditivos. Com exceção do código fonte, todos os demais artefatos produzidos pelo processo

implicam em trabalho que é convergente à meta final, ou seja, um produto executável (WAKE, 2002). Todos os demais artefatos são considerados como opcionais. Entretanto, ainda que opcionais, existe um conjunto de artefatos comumente usado em projetos XP. Como regra, todavia, deve se considerar a limitação da quantidade de esforço colocado em qualquer trabalho convergente, a menos que esse seja comprovadamente um habilitador para melhoria do processo e que essa melhoria seja de alguma forma demonstrável.

Documento de Visão	Define o ponto de vista dos stakeholders sobre o produto a ser desenvolvido, especificado em termos das principais funcionalidades e necessidades dos stakeholders.
Estórias de Usuário	Uma descrição resumida de alguma funcionalidade fornecida pelo sistema do ponto de vista de um usuário desse sistema.
Testes de Cliente	É executado no sistema para verificar se uma funcionalidade foi implementada corretamente.
Plano de Liberação	Uma lista de estórias de usuário priorizadas que serão implementadas nas próximas liberações.
Metáfora (Sistema de Nomes)	Um vocabulário comumente entendido que descreve as partes importantes do sistema e seus respectivos relacionamentos.
Plano de Iteração	Essencialmente uma lista de estórias de usuário e tarefas de engenharia que serão trabalhadas na iteração atual.
Padrão de Codificação	Descreve as convenções a serem usadas ao se trabalhar com a linguagem de programação.
Testes Unitários	Usado para garantir que uma determinada funcionalidade de um componente do sistema esteja funcionando corretamente.
Código Fonte	A especificação executável do sistema que está sendo construído.
Construção	O resultado do processo de integração de código e criação da construção.

Tabela 5: Artefatos de Extreme Programming (BECK, 2000, Tradução do Autor).

3.4.5. Considerações Finais

O objetivo principal do XP é estabelecer um processo de software que privilegie e enfoque os esforços na codificação (SHRIVASTAVA ET AL., 2010). Ainda que seja reconhecida como uma metodologia ágil, e como tal deve privilegiar a adaptação do seu processo às necessidades dos times, observou-se que a não adoção de algumas das suas características – tais como princípios e práticas – caracteriza que o processo utilizado não é XP, sendo a metodologia taxativa nesse sentido (BECK, 2000). Outro aspecto identificado é que a adoção do XP é indicada a equipes entre dois a dez programadores, sendo grupos maiores ou menores que a faixa estabelecida não recomendada, devido às particularidades previstas nas práticas do XP (BECK, 2000; WAKE, 2002).

3.5. Crystal

Crystal consiste em uma metodologia ágil de desenvolvimento de software concebida por Cockburn (2004). Descrita como uma família de metodologias, intitulada *Crystal Family* (Família Crystal) com estrutura comum, enfatiza a entrega frequente, comunicação próxima entre os envolvidos e melhoria contínua (COCKBURN, 2004). Existem diferentes metodologias Crystal para diferentes tipos de projetos. Cada projeto ou organização utiliza a estrutura da família para formar novos membros – customizações de processo. A abordagem do Crystal possui um enfoque voltado à Gestão de Pessoas. Seus princípios são customizados

para cada cenário específico de projeto, utilizando como referência sua complexidade (DYBA & DINGSOYR, 2008). A partir de um conjunto de políticas e convenções adequadas para cada cenário, Crystal é deliberadamente pouco definida, estando a qualidade do projeto totalmente relacionada a fatores humanos.

Ainda que seja pouco definida, Crystal considera princípios básicos em sua abordagem: as entregas frequentes, reduzindo assim a necessidade de produtos intermediários. O time deve possuir integração e relacionamento íntimo no projeto, entretanto, com especialidades distintas. A comunicação eficaz é fundamental para que haja um bom fluxo de informação. Os projetos que utilizam Crystal possuem ciclos incrementais de desenvolvimento, com liberações de versões em um período que varia de um a quatro meses. Uma vez concluídas as iterações, é proposta uma reflexão por parte do time sobre possíveis melhorias no processo (COCKBURN, 2004). Em Crystal é possível que cada organização avalie o contexto do seu projeto por duas perspectivas distintas: número de pessoas e repercussão/consequência dos erros (COCKBURN, 2004). Baseando-se na análise das perspectivas, Crystal é definida graficamente por cores indicando o "peso" – complexidade e detalhamento – da metodologia para o referido projeto, conforme apresentado na ilustração a seguir.

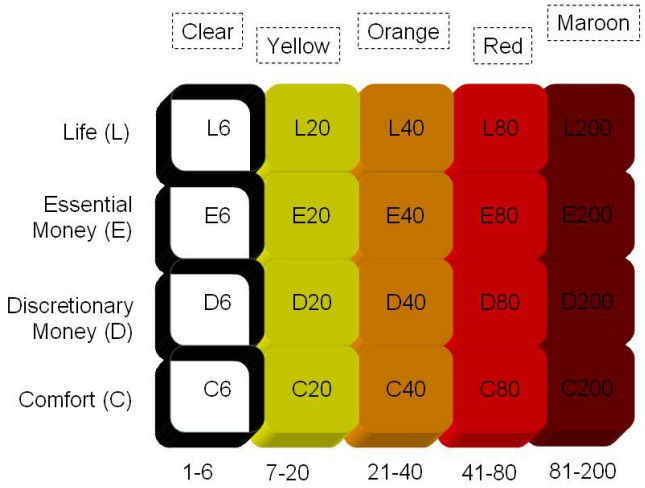


Figura 10: Quadro da Família Crystal (Ilustração disponível em: http://www.devx.com/architect/Article/32836/0/page/2, acessada em 30 de Dezembro de 2013)

Caso o projeto seja pequeno e de complexidade técnica rotineira, uma metodologia como Crystal Clear, Crystal Orange ou Crystal Yellow pode ser usada. Caso o projeto seja de missão crítica, onde a vida humana pode estar em perigo, os métodos Crystal Diamond ou Crystal Sapphire seriam usados (COCKBURN, 2004). Nas seções a seguir serão apresentadas as propriedades, estratégias, técnicas, processo, produtos e papéis relacionados à estrutura das metodologias Crystal para desenvolvimento ágil de software.

3.5.1. Visão Geral do Processo

As metodologias ágeis de desenvolvimento de software possuem em comum propriedades e princípios, os quais possuem como arcabouço os valores descritos no Manifesto Ágil (DYBA & DINGSOYR, 2008). A família Crystal, por sua vez, está estruturada baseada em um "código genético" que considera (COCKBURN, 2004):

- Um modelo de jogos econômico cooperativo;
- Prioridades selecionadas;

- Propriedades selecionadas;
- Princípios selecionados;
- Técnicas exemplo selecionadas;
- Projetos exemplo.

O modelo de jogos cooperativos dita que o desenvolvimento de software é uma série de "jogos", cujos movimentos consistem em nada além de inventar e comunicar, ações essa que são naturalmente limitadas por recursos (COCKBURN, 2004). Cada jogo na série possui dois objetivos que competem por recursos: entregar o software nesse jogo e preparar para o próximo jogo na série. O jogo nunca se repete, então cada projeto demanda uma estratégia ligeiramente diferente de todos os jogos anteriores. O modelo de jogos econômico cooperativo direciona as pessoas de um projeto a pensarem em seu trabalho de uma forma bastante específica, focada e efetiva. Essa forma está baseada em prioridades, as quais para o Crystal são:

- *Segurança*, quanto aos resultados do projeto;
- *Eficiência*, quanto ao desenvolvimento;
- *Viabilidade*, quanto as convenções (os desenvolvedores conseguem trabalhar seguindo as convenções).

O time de um projeto Crystal é guiado por sete propriedades de segurança, onde as três primeiras propriedades são o núcleo do Crystal (COCKBURN, 2004). As demais podem ser adicionadas em qualquer ordem para ampliar a margem de segurança. As propriedades são:

- Entregas frequentes;
- Comunicação próxima;
- Melhoria refletiva;
- Segurança pessoal;
- Foco;
- Acesso fácil a usuários experientes;
- Ambiente técnico com testes automatizados, gerência de configuração e integração frequente.

Conforme descrito por Dyba & Dingsoyr (2008), as metodologias ágeis são guiadas por princípios. Os princípios do Crystal são baseados em algumas ideias centrais:

- A quantidade de detalhes necessários nos documentos de requisitos, projeto e planejamento varia de acordo com as circunstâncias do projeto, especificamente: pelo tamanho do estrago que pode ser causado por defeitos não detectados; pela frequência de colaboração pessoal compartilhada pelo time;
- Pode não ser possível eliminar todos os produtos de trabalho intermediários, como os requisitos, documentos de projeto e planos de projeto, mas eles podem ser reduzidos na medida em que: caminhos de comunicação curtos, ricos e informais estão disponíveis ao time; software funcional e testado é entregue com brevidade e frequentemente;
- O time com frequência ajusta convenções para adequar: a personalidade individual dos membros do time; ambiente atual de trabalho; peculiaridades do tipo de atividade que foi atribuída.

3.5.2. Papéis

Crystal prevê igualmente uma estrutura de papéis e produtos de trabalho, os quais podem ser adotados em projetos Crystal. Salienta-se, porém, que tanto os papéis quanto os produtos de trabalho são facultados ao nível do projeto, o qual, conforme descrito anteriormente, é definido baseado em duas perspectivas: número de pessoas e repercussão/consequência dos erros (COCKBURN, 2004).

Patrocinador (*Sponsor*)	• Declaração de Missão com Negociação de Prioridades (*Mission Statement with Tradeoff Priorities*)
Time como Grupo (*Team as a Group*)	• Estrutura do Time e Convenções (*Team Structure and Conventions*); • Resultados de Workshops de Reflexão (*Reflection Workshop Results*).
Coordenador (*Coordinator*)	• Mapa do Projeto (*Project Map*); • Plano de Entrega (*Release Plan*); • Status de Projeto (*Project Status*); • Lista de Riscos (*Risk List*); • Plano de Iteração e Status (*Iteration Plan & Status*); • Cronograma de Visualização (*Viewing Schedule*).
Especialista de Negócio & Usuário Embaixador (*Business Expert & Ambassador User*)	• Lista de Objetivos por Ator (*Actor-Goal List*); • Casos de Uso & Requisitos (*Use Cases & Requirements File*); • Modelo de Papéis de Usuários (*User Role Model*).
Líder Designer (*Lead Designer*)	• Descrição de Arquitetura (*Architecture Description*).
Programador Designer (*Designer-Programmers*)	• Rascunhos de Telas (*Screen Drafts*); • Modelo de Domínio Comum (*Commom Domain Model*); • Modelo de Design e Notas (*Design Sketches & Notes*); • Código Fonte (*Source Code*); • Código de Migração (*Migration Code*); • Testes (*Tests*); • Pacote de Sistema (*Package System*).
Testador (*Tester*)	• Relatório de Erros (*Bug Reports*).
Escritor (*Writer*)	• Texto de Ajuda ao Usuário (*User Help Text*).

Tabela 6: Papéis e Artefatos do Crystak (COCKBURN, 2004).

3.5.1. Práticas e Execução do Processo

Diversas estratégias e técnicas bastante uteis para o contexto de metodologias ágeis foram propostas, como, por exemplo, o “Planning Game” do Extreme Programming (BECK, 2000). Existem, porém, algumas estratégias e técnicas não relatadas em outras metodologias que podem ser uteis a um time Crystal, especialmente para guiar o caminho do time no período inicial do projeto e para as primeiras entregas (COCKBURN, 2004). As estratégias sugeridas pelo Crystal consideram:

- **Exploração 360º (*Exploratory 360º*):** No início de um novo projeto, geralmente durante atividades de abertura, a equipe precisa estabelecer que o projeto é significativo e que o mesmo pode ser entregue usando a tecnologia pretendida. Eles olham ao redor em todas as direções, definindo os seguintes aspectos do projeto: a) o valor do negócio, b) requisitos; c) modelo de domínio; d) planos de tecnologia, e) plano de projeto; f) a composição da equipe; g) processo ou metodologia. Toda Exploração 360º de um projeto Crystal leva de alguns dias até uma semana ou duas,

se alguma tecnologia nova e peculiar for usada. Com base no que o time aprende, ele decide se faz sentido continuar ou não;

- **Vitória Antecipada (*Early Victory*):** Em projetos de software, a Vitórica Antecipada busca o primeiro pedaço do código em execução e testado. Este é geralmente o Esqueleto Ambulante (pequeno pedaço de função do sistema utilizável, por exemplo, não muito mais do que adicionar um item no banco de dados do sistema). Embora isto possa não parecer muito, os membros da equipe aprendem com esta pequena vitória quanto aos estilos de trabalho uns dos outros, além dos usuários terem uma visão inicial do sistema, e os patrocinadores poderem ver o time entregar;
- **Esqueleto Ambulante (*Walking Skeleton*):** O Esqueleto Ambulante é uma pequena implementação do sistema que realiza uma pequena função *end-to-end*. Essa função não precisa utilizar a arquitetura final, mas deve interligar os principais componentes da arquitetura. A arquitetura e a funcionalidade podem evoluir em paralelo;
- **Rearquitetura Incremental (*Incremental Rearchitecture*):** A arquitetura do sistema terá que evoluir a partir do Esqueleto Ambulante, e também lidar com as mudanças de tecnologias e requisitos de negócio ao longo do tempo. Raramente é eficaz interromper o desenvolvimento para realizar uma revisão na arquitetura. Dessa forma, a equipe evolui a arquitetura em estágios, mantendo o sistema funcionando à medida que as mudanças são feitas. A equipe aplica a ideia de desenvolvimento incremental para a revisão da infraestrutura ou arquitetura, bem como as funcionalidades finais do sistema;
- **Radiadores de Informação (*Information Radiators*):** Um Radiador de Informação é um painel de exibição exposto em algum lugar onde as pessoas possam ver enquanto trabalham ou andam pelo local de trabalho. Esse painel mostra aos leitores informações que eles se preocupam, sem precisar solicitar a alguém. Isso significa mais comunicação com menos interrupções.

Quanto às técnicas sugeridas por Crystal, consideram-se (COCKBURN, 2004):

- **Modelagem de Metodologia (*Methodology Shaping*):** Coleta de informações sobre experiências anteriores e uso para avançar com as convenções iniciais do projeto;
- **Workshop de Reflexão (*Reflection Workshop*):** Formato de workshop especial para melhoria reflexiva;
- **Blitz de Planejamento (*Blitz Planning*):** Também chamada de planejamento "*jam session*" (como no jazz), para enfatizar a natureza colaborativa, projeto rápido e colaboração da técnica de planejamento,
- **Estimativa Delphi (*Delphi Estimation*)**: Uma maneira de chegar a uma estimativa inicial para o projeto total;
- **Stand-ups Diários (*Daily Stand-ups*):** Uma maneira rápida e eficiente para passar informações à equipe, com periodicidade diária;
- **Design de Interação Ágil (*Agile Interation Design*):** Versão mais rápida de Design Centrado no Uso[6];
- **Miniatura de Processo (*Process Miniature*):** Técnica de aprendizagem, cujo objetivo é apresentar à equipe a metodologia de processo utilizada no projeto;

[6] Design Centrado no Uso (*Usage-Centered Design*) é uma abordagem para o design de interface de usuário com base em um enfoque sobre as intenções do usuário e padrões de uso. Ele analisa os usuários em termos dos papéis que desempenham em relação aos sistemas e emprega casos abstratos de uso para a análise de tarefas. Deriva design visual e interação a partir de protótipos abstratos baseados na compreensão das funções de usuário e casos de tarefas. Design centrado no uso foi introduzido por Larry Constantine e Lucy Lockwood. *Software for Use: A Practical Guide to the Essential Models and Methods of Usage-Centered Design.* Addison-Wesley, 1999.

- **Programação Lado a Lado (*Side-by-Side Programming*):** Alternativa menos intensa para programação em pares *(Pair Programming*, BECK, 2000)*;*
- **Gráficos de Evolução (*Burn Charts*):** Uma maneira eficiente de planejamento e relatórios de progresso, particularmente apropriada para uso por Radiadores de Informação.

Estas estratégias e técnicas oferecem um bom ponto de partida para o time Crystal na reflexão quanto a como adotar a metodologia (NERUR ET AL., 2005). Crystal utiliza processos cíclicos aninhados de vários tamanhos. O que o time faz em qualquer momento do projeto depende de onde eles estão em cada um dos ciclos. Para o Crystal, um projeto não possui fases específicas que precisam ser transpostas, como em processos prescritivos ou mesmo em cascata, mas sim, descreve ciclos que são executados e variam quanto ao tempo de duração desses ciclos (COCKBURN, 2004; DYBA & DINGSOYR, 2008). Crystal reforça que seu processo requer múltiplas entregas por projeto, mas não múltiplas iterações por entrega, o que reforça o conceito de que os ciclos no Crystal referem-se ao tempo de execução e não o que se objetiva com esse tempo (entregas). Os principais ciclos de desenvolvimento identificados em projetos Crystal são ilustrados na Figura 11:

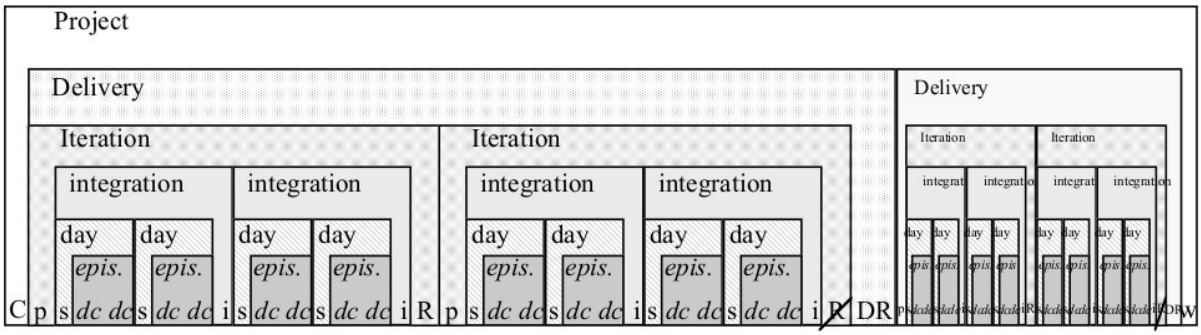


Figura 11: Ciclos do processo Crystal detalhados de forma a exibir atividades diárias (COCKBURN, 2004).

- **Ciclo de Projeto (*The Project Cycle*):** Unidade relacionada a todo o projeto e que poderia ter qualquer duração;
- **Ciclo de Entrega (*The Delivery Cycle*):** Unidade de entrega, entre uma semana até três meses;
- **Ciclo de Iteração (*The Iteration Cycle*):** Unidade de estimativa, desenvolvimento e celebração, de uma semana a três meses;
- **Semana de Trabalho (*Work Week*):** Unidade relacionada ao período de uma semana de trabalho – normalmente 5 dias úteis;
- **Ciclo de Integração (*The Integration Cycle*):** Unidade de desenvolvimento, integração e testes do sistema, 30 minutos a três dias;
- **Dia de Trabalho (*Work Day*):** Unidade relacionada ao período de um dia de trabalho;
- **Episódio de Desenvolvimento (*The Development Episode*):** Desenvolvimento e verificação em uma seção do código, durando de poucos minutos a poucas horas;

Cada ciclo do processo Crystal divide-se em partes a serem executadas com propósito específico, conforme apresentado na tabela 7.

Ciclo de Projeto (*The Project Cycle*)	Embora cada projeto seja fundamentado como uma atividade única, ele é tipicamente seguido por outro projeto em um ciclo que se repete. Um ciclo de projeto em Crystal possui três partes: • Uma atividade de abertura (*Chartering Activity*), que objetiva conhecer o time, realizar uma Exploração 360º, definir convenções da metodologia e elaborar o plano de projeto inicial; • Uma série de dois ou mais ciclos de entregas; • Um ritual de conclusão (*Project Wrap-up*).
Ciclo de Entrega (*The Delivery Cycle*)	O Ciclo de Entrega tem três ou quatro partes: • A recalibração do plano de liberação; • Uma série de uma ou mais repetições, cada uma, resultando em códigos integrados e testados; • Entrega para os usuários reais; • Um ritual de conclusão, incluindo a reflexão tanto sobre o produto que está sendo criado como sobre as convenções utilizadas.
Ciclo de Iteração (*The Iteration Cycle*)	A duração das iterações varia de acordo com as equipes. Uma iteração tem três partes: • Planejamento de iteração; • Atividades diárias e Ciclo de Integração; • Ritual de conclusão (Workshop de Reflexão).
Ciclo de Integração (*The Integration Period*)	Um ciclo de integração pode ser executado com duração que vai de meia hora a vários dias, dependendo da prática da equipe. Algumas equipes têm uma máquina autônoma que executa testes continuamente. Outros ainda integram uma vez por dia ou três vezes por semana. Embora quanto mais curto melhor, Crystal não determina a duração do tempo de um ciclo de integração.
Dia e Semana de Trabalho (*Work Day and Week*)	Dos vários ciclos, somente os ciclos diários e semanais são ritmos de calendário. Muitas atividades de grupo ocorrem em uma base semanal. Estas podem incluir reuniões, discussão técnica, entre outras. A jornada de trabalho também tem o seu próprio ritmo.
Episódio de Desenvolvimento (*The Development Episode*)	Durante um episódio, a pessoa pega alguma atribuição pequena do projeto, codifica até a conclusão (de preferência com testes unitários), e os submete para o sistema de gerenciamento de configuração. Isso pode demorar entre 15 minutos e vários dias, dependendo do programador e as convenções do projeto. Manter cada episódio com duração de menos de um dia geralmente funciona melhor.

Tabela 7: Detalhamento das partes executadas em cada ciclo Crystal (COCKBURN, 2004, Tradução do Autor).

O "código genético" do Crystal, conforme descreveu Cockburn (2004), ainda prevê técnicas e projetos exemplo, os quais servem como referência para melhor adequar e definir a perspectiva de adoção da metodologia. Esses, porém, fazem parte de um material complementar ao Crystal e não compõem o cerne da estrutura do processo.

3.5.2. Artefatos

Conforme descrito na Seção 3.5.2., os artefatos do Crystal estão diretamente relacionados aos papéis atribuídos no processo Crystal adotado para o projeto (COCKBURN, 2004). Reitera-se que nenhum artefato é completamente requerido nem tampouco totalmente opcional. De acordo com as atribuições do projeto, serão definidos os artefatos necessários.

3.5.3. Considerações Finais

Dyba & Dingsoyr (2008) descreveram que as metodologias ágeis são guiadas por princípios. Nesse sentido, a metodologia Crystal se vale dessa premissa para propor não um método formal de processo, mas para descrever um ecossistema amplo e adaptativo de processo de desenvolvimento de software. A proposta de que em Crystal é possível que cada organização avalie o contexto do seu projeto por duas perspectivas distintas – número de pessoas e repercussão/consequência dos erros – favorece a seus adeptos a flexibilidade que invariavelmente se requer em qualquer processo ágil de software (NERUR ET AL., 2005).

Um dos principais diferenciais propostos por Crystal, como sendo uma metodologia ágil completa para qualquer tipo e tamanho de projeto, trata-se justamente dos diferentes níveis de amplitude do processo. Um dos habilitadores da viabilidade das metodologias ágeis reside no fato que os processos precisam ser adaptativos e não podem refletir sequencias e etapas, mas sim, atividades a serem desempenhadas, independentemente de quando e como, tal como descreve o processo ágil Crystal (COCKBURN, 2004).

3.6. Dynamic Systems Development Method

O Dynamic Systems Development Method (DSDM, do português Método Dinâmico de Desenvolvimento de Sistemas) é uma metodologia ágil de projeto e desenvolvimento de software (COLEMAN & VERBRUGGEN, 1998). Lançada em 1994, o DSDM foi originalmente concebido para prover disciplina ao método RAD (*Rapid Application Development*, do português Desenvolvimento Rápido de Aplicações)[7]. Atualmente o DSDM é uma abordagem iterativa e incremental que adota os princípios de desenvolvimento ágil, incluindo o contínuo envolvimento dos usuários e clientes (ABRAHAMSSON ET AL., 2002). O DSDM é mantido pelo *DSDM Consortium*[8] e é o principal processo adotado para desenvolvimento baseado em RAD (ABRAHAMSSON ET AL., 2002). Habitualmente as abordagens de desenvolvimento consideram a ideia de estabelecer o conjunto de requisitos e em seguida definir o tempo e recursos disponíveis para o desenvolvimento (LEVINE, 2005). Em se tratando do DSDM, a ideia é que sejam fixados o tempo e recursos para só então se definir os requisitos passíveis de serem desenvolvidos (ABRAHAMSSON ET AL., 2002).

3.6.1. Visão Geral do Processo

A adoção do DSDM como processo de software está fortemente amparada em princípios, os quais direcionam o time quanto à atitude que eles precisam ter para entregar soluções de forma consistente (COLEMAN & VERBRUGGEN, 1998). Os princípios que direcionam o Dynamic Systems Development Method são:

- Foco nas necessidades do negócio;
- Entregar no prazo;
- Colaboração entre o time;
- Nunca comprometer a qualidade;

---

[7] Desenvolvimento Rápido de Aplicações (RAD, *Rapid Application Development*) é uma metodologia de desenvolvimento de software que usa um mínimo de planejamento em favor de prototipagem rápida. O planejamento de software desenvolvido usando RAD é intercalado com a escrita do próprio software. A falta de um extenso pré-planejamento geralmente permite que o software seja escrito muito mais rápido, e torna mais fácil para alterar os requisitos. *Kerr, James M.; Hunter, Richard. Inside RAD: How to Build a Fully Functional System in 90 Days or Less. McGraw-Hill*, 1993.

[8] DSDM Consortium, www.dsdm.org.

- Desenvolver iterativamente;
- Comunicar e esclarecer os envolvidos continuamente;
- Demonstrar controle do projeto.

O DSDM consiste em um processo de software dividido em cinco fases específicas: Estudo de Viabilidade (*Feasibility Study*); Estudo de Negócio (*Business Study*); Iteração de Modelo Funcional (*Functional Model Iteration*); Iteração de Desenho e Construção (*Design and Build Iteration*) e Implementação (*Implementation*) (ABRAHAMSSON ET AL., 2002).

As duas primeiras fases são executadas de forma sequencial e apenas uma vez durante o projeto. As demais fases estão relacionadas à construção do produto e são executadas de forma iterativa e incremental, considerando iterações com período de tempo (*timeboxing*) e objetivo de entrega pré-estabelecidos. A seguir é apresentado na Figura 12 o fluxo de execução do processo DSDM, bem como a tabela 8 descrevendo as fases e atividades correspondentes (STAPLETON, 1997).

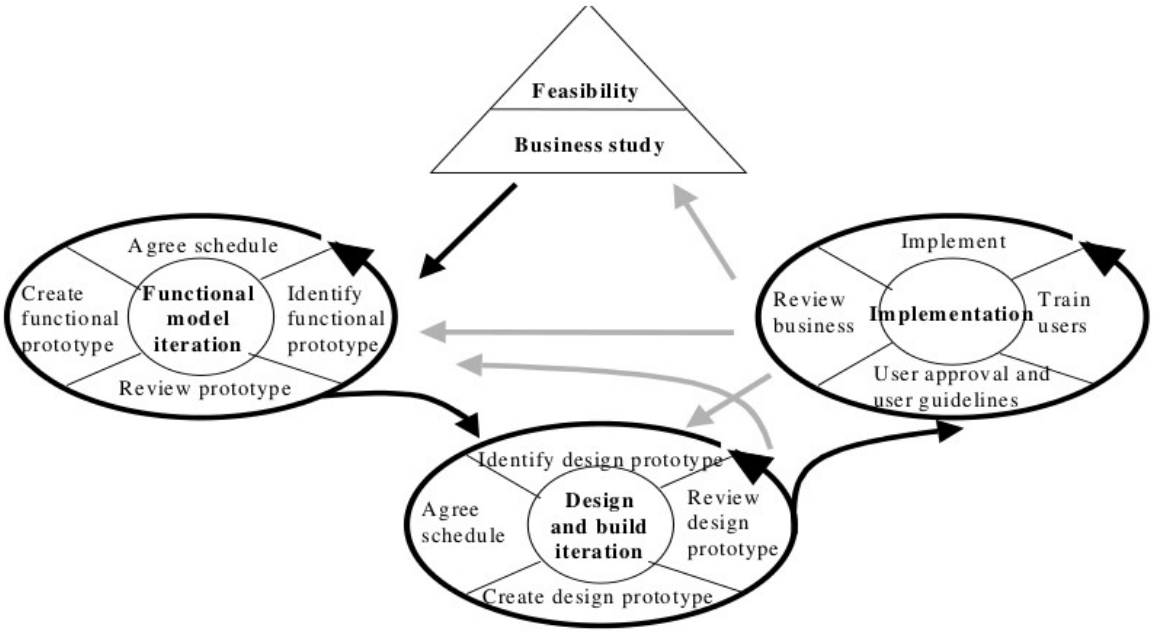


Figura 12: Estrutura do processo Dynamic Development System Method (STAPLETON, 1997).

Estudo de Viabilidade (*Feasibility Study*)		Avaliação quanto a adoção do DSDM, utilizando como referência o projeto, problemas da organização e pessoas, é verificado se o DSDM será adotado. São gerados os artefatos de **Relatório de Viabilidade**, **Protótipo de Viabilidade**, e um **Plano de Detalhamento Global** que inclui o **Plano de Desenvolvimento** e **Controle de Risco**.
Estudo de Negócio (*Business Study*)		Analisa as características essenciais do negócio e tecnologias a serem empregadas.
Iteração de Modelo Funcional (*Functional Model Iteration*)	Identificar protótipo funcional	Determinação das funcionalidades que serão implementadas. Um Modelo Funcional é desenvolvido de acordo com o resultado do artefato resultante do estágio de Estudo do Negócio.
	Acordar cronograma	Definir quando e como as funcionalidades serão implantadas, estabelecendo o **Cronograma**.
	Criar protótipo funcional	Desenvolver um **Protótipo Funcional**, de acordo com o Cronograma e o **Modelo Funcional**.

	Revisar protótipo funcional	Efetuar correções do protótipo desenvolvido. Isto pode ser feito através de testes com usuários finais ou por análise da documentação. O artefato gerado é o **Documento de Revisão do Protótipo Funcional**.
Iteração de Desenho e Construção (*Design and Build Iteration*)	Identificar o modelo do Desenho	Identificação do requisitos funcionais e não-funcionais. Baseado nestas identificações, uma **Estratégia de Implantação** é gerada e caso haja **Evidências de Testes** de iterações anteriores, estas serão utilizadas para criação desta estratégia.
	Acordar cronograma	Como e quando serão realizados estes requisitos, sendo o **Cronograma** atualizado.
	Criar protótipo do Desenho	Criar um **Protótipo de Sistema** que pode ser manipulado pelos usuários finais no uso diário, também para razões de teste.
	Revisar protótipo do Desenho	Efetuar correções no sistema, novamente testando e revisando. Uma **Documentação para Usuário** e **Evidência de Testes** serão gerados.
Implementação (*Implementation*)	Guia e aprovação de usuários	Usuários finais aprovam o sistema testado pela implantação e orientação fornecida pelo sistema criado. Um **Termo de Aprovação** é gerado.
	Treinar usuários	Treinar futuros usuários finais no uso do sistema. **Usuários Treinados** é o artefato entregue neste sub-estágio.
	Implementar	Implantar o sistema testado e liberar aos usuários finais.
	Revisar negócio	Rever o impacto que o sistema implantado causa sobre o negócio, pode-se utilizar o cruzamento dos objetivos iniciais com a análise atual como termômetro. Esta revisão será documentada através do **Documento de Revisão do Projeto**.

Tabela 8: Fases e atividades do Dynamic Systems Development Method (STAPLETON, 1997, Tradução do Autor).

3.6.2. Papéis

O Dynamic Systems Development Method estabelece papéis específicos em sua abordagem. Deve-se, porém, definir a atribuição dos papéis aos membros do time de forma prévia ao início do projeto. Os papéis previstos em DSDM são (STAPLETON, 1997):

- **Patrocinador Executivo (Executive Sponsor)**: Papel encarregado da ultima palavra na tomada de decisões do projeto;
- **Visionário (*Visionary*):** Papel responsável por iniciar o projeto certificando que os requisitos essenciais foram definidos;
- **Usuário Embaixador (*Ambassador User*):** Papel que traz o conhecimento de outras áreas e certifica que os desenvolvedores receberam feedback suficiente dos usuários;
- **Usuário Consultor (*Advisor User*):** Qualquer usuário que represente um importante ponto de vista e traga constantemente conhecimento ao projeto;
- **Gerente de Projeto (*Project Manager*):** Papel responsável pela gestão do projeto;
- **Coordenador Técnico (*Technical Coordinator*):** Papel responsável pelo desenho da arquitetura do sistema e controle da qualidade técnica do projeto;
- **Líder de Time (*Team Leader*):** Papel de liderança do time e mantenedor da harmonia do projeto e trabalho em grupo;
- **Desenvolvedor (*Developer*):** Papel que interpreta e desenvolve os requisitos do sistema, além de artefatos e protótipos;
- **Testador (*Tester*):** Papel que confere o funcionamento da parte técnica através da execução de testes;

- **Escrivão (*Escriber)*:** Papel responsável por escrever requisitos, acordos e decisões tomadas entre todos os grupos de trabalho;
- **Facilitador (*Facilitator*):** Papel que gerencia o progresso dos grupos de trabalho, age como agente de preparação dos grupos e gestor da comunicação;
- **Papéis Especialistas (*Specialist Roles*):** Arquiteto de negócios, Gestor de Qualidade, Integrador de Sistema, etc.

3.6.3. Práticas e Execução do Processo

A aplicação prática do Dynamic Systems Development Method está associada a adoção de nove princípios os quais para o DSDM, são fundamentais para a execução adequada de um projeto de software adotante da metodologia (STAPLETON, 1997; ABRAHAMSSON ET AL., 2002). Os princípios do DSDM são listados a seguir:

Envolvimento ativo do usuário é imperativo.	Alguns usuários experientes precisam estar presentes durante todo o desenvolvimento do sistema para garantir uma resposta antecipada e rigorosa.
Os times DSDM precisam ser incentivados a tomarem decisões.	Longos processos de tomada de decisão não podem ser tolerado em ciclos de desenvolvimento rápidos. Os envolvidos no desenvolvimento têm o conhecimento para definir o direcionamento do sistema.
Foco na frequente entrega de produtos.	Decisões erradas podem ser corrigidas se o ciclo de entrega é curto e os usuários podem fornecer feedback preciso.
Adequação ao propósito do negócio é um critério essencial para aceitação de entregáveis.	*"Construa o produto certo antes de construí-lo direito*" (STAPLETON, 1997). Deve-se buscar as principais necessidades do negócio estejam satisfeitas, a excelência técnica em áreas menos importantes deve ser buscada posteriormente.
Desenvolvimento iterativo e incremental é necessário para convergir com a precisão das soluções de negócio.	Requisitos de sistema raramente permanecem intactos a partir do início de um projeto até seu fechamento. Ao permitir que os sistemas evoluam através do desenvolvimento iterativo, os erros podem ser encontrado e corrigidos cedo.
Todas as mudanças durante o desenvolvimento são reversíveis.	No percurso do desenvolvimento, um caminho errado pode facilmente ser tomado. Usando iterações curtas e garantindo que os estados anteriores do desenvolvimento possam ser revertidos, o caminho errado pode ser seguramente corrigido.
Linha de base de requisitos é definida em alto nível.	Congelar os requisitos essenciais deve ser feito apenas em um alto nível, para permitir aos requisitos detalhados as alterações necessárias. Isso garante que os requisitos essenciais são capturados em uma fase inicial, mas o desenvolvimento é permitido começar antes que cada requisito seja congelado. À medida que o desenvolvimento progride, mais requisitos devem ser congelados, pois ficaram claros e estão acordados.
Testes são integrados ao ciclo de vida.	Com restrição de tempo, o teste tende a ser negligenciado se deixado para o fim do projeto. Portanto, cada componente do sistema deve ser testado pelos desenvolvedores e usuários na medida em que são desenvolvidos. Os testes também são incrementais, e os testes construídos tornam-se mais abrangentes ao longo do projeto. Teste de regressão é particularmente enfatizado por conta do estilo de desenvolvimento evolutivo.

É essencial uma abordagem colaborativa e cooperativa entre os stakeholders.	Para que o DSDM funcione, a organização deve se comprometer com o processo, e as suas áreas de negócios e TI cooperarem. A escolha do que é entregue no sistema e o que é deixado de fora requer um comum acordo. Em menor escala, as responsabilidades do sistema são compartilhadas, por isso a colaboração do usuário / programador também deve funcionar perfeitamente.

Tabela 9: Princípios para aplicação prática do Dynamic Systems Development Method (Adaptado de ABRAHAMSSON ET AL., 2002, Tradução do Autor).

Complementarmente aos princípios para sua aplicação, o DSDM aborda outros aspectos sugestivos para sua adoção, tais como (STAPLETON, 1997): estabelecimento de prioridades; respeito aos prazos; finalização suficiente de uma etapa para iniciar a próxima; maleabilidade quanto a alterações de design; técnicas de gerenciamento de projetos e desenvolvimento de software; utilização do DSDM tanto para novos projetos como para manutenção; objetividade na gerencia dos riscos; foco nas entregas constantes e estimativas que se baseiem nas funcionalidades e não em linhas de código.

3.6.4. Artefatos

O Dynamic Systems Development Method considera um fluxo integrado entre suas entregas e os artefatos gerados pelo processo (STAPLETON, 1997). Nesse sentido, a produção dos artefatos propostos no processo é fundamental para efetiva adoção do DSDM como processo de software (ABRAHAMSSON ET AL., 2002). Os artefatos previstos no DSDM são listados na Tabela 10.

Controle de Riscos	Lista dos riscos identificados.
Lista de Priorização de Requisitos	Lista de requisitos baseadas em suas prioridades. Sugere-se a priorização destes é baseado no modelo MoSCoW[9].
Lista de Requisitos Não Funcionais	Lista dos requisitos não funcionais, tratados em estágios posteriores.
Requisitos Funcionais	Lista dos requisitos funcionais para construir o protótipo.
Modelo Funcional	Modelo baseado em requisitos funcionais, utilizado de acordo com o desenvolvimento do protótipo Funcional.
Protótipo	Produção de protótipo para testes de aspectos de design, ilustração de ideias e funcionalidades, coletando o feedback do usuário.
Divisão de Tempo	Lista de tempo necessário para certas atividades de forma a realizar o planejamento de acordo com o cronograma.
Plano de Prototipagem	Plano de atividades do processo de prototipagem.
Cronograma	Definição do plano de atividades e tempo necessário acordados.
Protótipo Funcional	Protótipo de funções que o sistema deve executar e como deve ser feito.
Plano de Implantação	Preparação de atividades necessárias à implantação do protótipo funcional.

[9] O modelo MoSCoW é um método de priorização de requisitos. Seu nome é composto pelas iniciais de cada tipo de priorização, em inglês: Must, Should, Could e Would (ou Won't como citam muitos autores). Os "os" no meio da palavra foram adicionados para facilitar sua pronúncia. Este método dá ao grupo uma visão clara do que é essencial para o projeto, e do que pode esperar. Suas divisões são: Must have (tem que ter); Should have (deve ter); Could have (poderia ter); Won't have (não vai ter, mas poderia). *A Guide to the Business Analysis Body of Knowledge, International Institute of Business Analysis, 2009*.

Função Refinada	Função do protótipo que está sendo revisada na iteração atual.
Função Combinada	Função do protótipo que é combinada com outros protótipos funcionais de iterações anteriores.
Registro de Testes	Conjunto de evidências de teste onde o script, procedimento, e resultados dos testes são incluídos.
Documento de Revisão de Prototipagem Funcional	Coleta comentários gerais de usuários sobre o incremento atual.

Tabela 10: Artefatos do Dynamic Systems Development Method (Adaptado de ABRAHAMSSON ET AL., 2002, Tradução do Autor).

3.6.5. Considerações Finais

O Dynamic Systems Development Method é uma metodologia ágil de projeto e desenvolvimento de software (COLEMAN & VERBRUGGEN, 1998) e que foi originalmente concebida para prover disciplina ao método RAD (Rapid Application Development, do português Desenvolvimento Rápido de Aplicações). O DSDM, contudo, propõe uma mudança na perspectiva e análise de um projeto de software. Enquanto abordagens tradicionais de desenvolvimento consideram o estabelecimento do conjunto de requisitos para então se definir o tempo e recursos disponíveis (LEVINE, 2005), o DSDM sugere que sejam fixados o tempo e recursos para só então se definir os requisitos a serem desenvolvidos (ABRAHAMSSON ET AL., 2002), o que é interessante do ponto de vista de que a abordagem torna-se mais realista quanto à efetiva disponibilidade de recursos – pessoas e tempo – dos times de desenvolvimento.

3.7. Adaptive Software Development

Adaptive Software Development (ASD, do português Desenvolvimento de Software Adaptativo) é um processo de desenvolvimento de software, o qual emergiu a partir da abordagem RAD (*Rapid Application Development*, do português Desenvolvimento Rápido de Aplicações) e foca no princípio de adaptação contínua do processo (HIGHSMITH, 2000).

O propósito da abordagem do ASD é sobrescrever a visão tradicional de um processo em cascata – preditivo – com uma série de ciclos que encorajem o aprendizado contínuo e a adaptação do projeto (HIGHSMITH, 1997). É uma das precursoras das metodologias ágeis de desenvolvimento de software, e é caracterizada por enfatizar: desenvolvimento focado na missão; focado nos recursos do produto; iterativo com ciclo de tempo pré-definido; dirigido a riscos e tolerante a mudanças.

O foco do ASD é centrado em problemas relacionados a desenvolvimentos complexos, de grandes sistemas (ABRAHAMSSON ET AL., 2002). A metodologia encoraja o desenvolvimento iterativo e incremental com constante prototipação. Outra característica do ASD é que ele provê um framework para prevenir que projetos entrem em estados de caos, porém, sem suprimir o senso de emergência e criatividade, aspecto fundamental em projetos de desenvolvimento de software (ABRAHAMSSON ET AL., 2002).

3.7.1. Visão Geral do Processo

O ciclo de vida de processo do ASD propõe uma visão diferente de ciclos de vida em cascata ou em espiral. No caso do ciclo de vida em cascata, previa-se uma linearidade e predição na execução do processo, conforme ilustrado na Figura 13.

Figura 13: Ciclo de vida em cascata (HIGHSMITH, 1997).

Já no ciclo de vida espiral, as fases do processo de desenvolvimento são vistas em ciclos, conforme ilustrado a seguir:

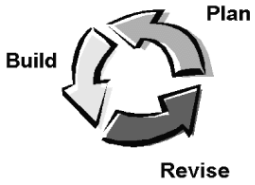


Figura 14: Ciclo de vida em espiral (HIGHSMITH, 1997).

Enquanto no modelo espiral os nomes das fases refletem o contexto imprevisível cada vez mais complexos dos sistemas, o desenvolvimento adaptativo vai mais longe do que a herança evolutiva dos dois modelos principais – cascata e espiral (ABRAHAMSSON ET AL., 2002). Primeiramente, ele substitui explicitamente o determinismo com a rápida resposta a novo cenários. Em segundo lugar, ele vai além de uma mudança no ciclo de vida para uma mudança mais profunda no estilo de gestão (HIGHSMITH, 1997). Por mais sutil que seja a diferença, ela pode ser exemplificada: em um ambiente que muda, os adotantes de um modelo determinístico buscariam um novo conjunto de regras de causa e efeito, enquanto que aqueles que utilizam o modelo adaptativo, não precisam buscar tais regras (ABRAHAMSSON ET AL., 2002). Para tal, o ASD se ampara em ciclos de três fases específicas: Especulação (*Speculate*), Colaboração (*Colaborate*) e Aprendizado (*Learn*), conforme ilustrado a seguir.

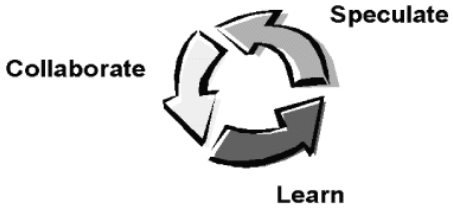


Figura 15: Fases do ciclo do Adaptive Software Developemnt (HIGHSMITH, 1997).

As fases do ASD são nomeadas de forma a enfatizar o papel das mudanças no processo (ABRAHAMSSON ET AL., 2002). A fase de Especulação faz um paradoxo com planejamento – assume-se que os envolvidos no projeto estão errados sobre certos aspectos da missão do projeto, em vez de tentar defini-la de imediato. A fase de Colaboração refere-se aos esforços de balancear o trabalho em grupo como maneira de viabilizar o desenvolvimento de sistemas com grande volume de mudanças. Já a fase de Aprendizado enfatiza a necessidade de reconhecimento e reação a equívocos, bem como ao fato de que os requisitos mudarão durante o desenvolvimento (ABRAHAMSSON ET AL., 2002).

3.7.2. Papéis

O processo ASD não descreve em detalhes a estrutura de um time para projetos que adotem a metodologia (ABRAHAMSSON ET AL., 2002). Entretanto, por se tratar de uma abordagem originada da cultura organizacional e de gestão, considera-se a importância de papéis colaborativos nos projetos (HIGHSMITH, 2000). Ainda assim, alguns papéis são descritos na literatura quanto a papéis previstos em um projeto ASD, tais como: Patrocinador Executivo (*Executive Sponsor*), Facilitador (*Facilitator*), Gerente de Projeto (*Project Manager*), entre outros papéis igualmente encontrados na metodologia Dynamic Systems Development Method (COLEMAN & VERBRUGGEN, 1998) e que foram previamente descritos na Seção 3.6.2.

3.7.3. Práticas e Execução do Processo

Conforme ilustrado na Figura 16, a execução do ciclo de vida de processo baseado em ASD inicia-se com a fase de Especulação, com a atividade de Iniciação de Projeto (*Project Initiation*), cujo objetivo é definir a missão do projeto, sendo um dos principais aspectos dessa atividade definir quais são as informações necessárias seguir com o projeto (HIGHSMITH, 2000). É nessa atividade onde são definidos os três principais artefatos do ASD: a Escritura de Visão do Projeto; Planilha de Dados do Projeto e Esboço de Especificação do Produto. Esses artefatos serão detalhados na seção a seguir. A atividade de Iniciação do Projeto estabelece o cronograma geral e objetivos para os ciclos de desenvolvimento (*Adaptive Cycle Planning*). Os ciclos tipicamente duram de quatro a oito semanas (ABRAHAMSSON ET AL., 2002).

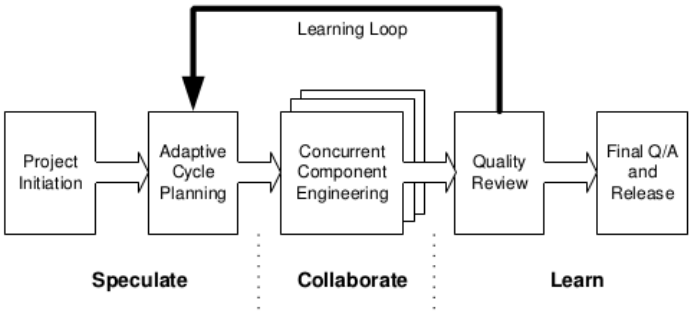


Figura 16: Detalhamento das fases do Adaptive Software Developemnt (HIGHSMITH, 2000).

O foco do ASD está mais centrado nos resultados e na qualidade destes do que na execução de tarefas, o que o configura como um processo orientado a componentes (*component-oriented*) em vez de orientado a tarefas (*task-oriented*) (ABRAHAMSSON ET AL., 2002). Nesse sentido, a fase de Colaboração possui uma função fundamental no núcleo do ASD, conforme descreve Abrahamsson et al. (2002, p. 70, Tradução do Autor):

> *"A maneira como ASD aborda este ponto de vista é através de ciclos de desenvolvimento adaptativos contidos na fase Colaboração, onde diversos componentes podem estar em desenvolvimento concorrentemente. O planejamento dos ciclos é parte do processo iterativo, assim como as definições dos componentes são continuamente modificadas para incorporar novas informações, e para atender a mudanças nos requisitos desses componentes."*

A base para os demais ciclos é obtida a partir de repetidas revisões de qualidade (*Quality Review / Learning Loop*). Um fator importante na realização das avaliações é a presença do cliente, como um grupo de peritos. No entanto, uma vez que as avaliações de qualidade são bastante escassas (ocorrem apenas no final de cada ciclo), a presença do cliente em ASD é apoiada por seções de Desenvolvimento Conjunto de Aplicações (JAD, *Joint Application Development*[10]).

A fase final de um projeto ASD é o *Final Q/A and Release* (Avaliação de Qualidade Final e Entrega). O ASD não considera como uma fase a ser realizada, contud ressalta a importância de capturar as lições aprendidas. Avaliações como essa são tidas como extremamente importantes em projetos alta velocidade e de alta mudança, onde alguma metodologia ágil é adotada (ABRAHAMSSON ET AL., 2002).

3.7.4. Artefatos

Assim como nos papéis e responsabilidades, o processo ASD propõe a produção de praticamente todos os artefatos propostos pelo processo Dynamic Systems Development Method, conforme apresentado na Seção 3.6.4. Entretanto, o ASD propõe a produção de três documentos específicos, intitulados "Artefatos de Missão" (*Mission Artifacts*), os quais são considerados como fundamentais para entendimento do contexto geral do projeto (HIGHSMITH, 2000).

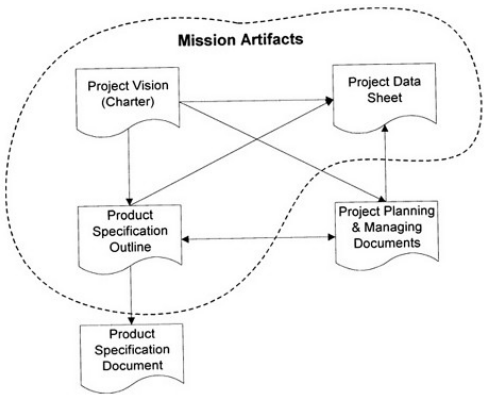


Figura 17: Artefatos de Missão do processo Adaptive Software Development (HIGHSMITH, 2000).

- **Documento de Visão do Projeto (*Project Vision – Charter*):** Provê uma descrição curta, com no máximo duas páginas, definindo os objetivos chave do negócio, especificações do produto e posicionamento de mercado;
- **Planilha de Dados do Projeto (Project Data Sheet):** Declaração de uma página contendo os principais benefícios de negócio do produto, especificação do produto e informações de gestão de projeto. Essa planilha serve como documento para auxiliar o foco do time de projeto, gestão e clientes;

[10] Joint Application Development (JAD) é um método usado no ciclo de vida de prototipação na metodologia Dynamic Systems Development Method para coletar requisitos de negócio enquanto um sistema de informação é desenvolvido. JAD também inclui uma abordagem para potencializar a participação dos usuários, acelear o desenvolvimento e melhorar a qualidade das especificações de requisitos. Consiste na realização de um workshop onde usuários com conhecimento do negócio e especialistas de TI se encontram, por vezes durante vários dias, para definir e revisar requisitos de negócio para o sistema. *Yatco, M. C. Joint Application Design/Development. University of Missouri-St. Louis, 1999.*

- **Esboço de Especificação do Produto (*Product Specification Outline*):** Prevê o inventário dos recursos e requisitos do sistema, funções, objetos, dados, desempenho, operações e outras especificações relevantes de alto nível a cerca do produto.

3.7.5. Considerações Finais

Considerando os modelos de processo precursores de metodologias ágeis, o Adaptive Software Development é o que propõe o menor conjunto de novos paradigmas ante aos modelos prescritivos de processo de software, tendo, inclusive, boa parte de sua estrutura amparada em outros modelos já existentes. Tal assertiva se sustenta por aspectos como:

- A fundação do ASD está amparada em uma sutil releitura de um processo baseado no ciclo de vida espiral (ABRAHAMSSON ET AL., 2002);
- A metodologia não faz menção a princípios e valores específicos, exceto por citar características como: ser um processo iterativo com ciclo de tempo pré-definido; dirigido a riscos e tolerante a mudanças (HIGHSMITH, 2000);
- Ausência de menção a princípios, papéis e artefatos específicos, sugerindo informalmente a adoção de alguns desses previstos em outra metodologia ágil, no caso, a Dynamic Systems Development Method (STAPLETON, 1997), como por exemplo o uso de sessões JAD (*Joint Application Development*, do português Desenvolvimento Conjunto de Aplicações).

Identificam-se, porém, algumas características diferenciais do ASD, como por exemplo: a adaptabilidade do processo e a abordagem cíclica que favorece o aprendizado do time na atividade de Revisão de Qualidade, a qual é positiva tanto para adaptações no processo, como também, no produto.

Conforme identificado por Abrahamsson et al. (2002), não existem pesquisas significativas quanto a adoção do ASD. O fato de a metodologia permitir muita adaptação – seja pela não definição de princípios, papéis ou mesmo artefatos específicos – ou mesmo pelos diferenciais propostos ante a outras metodologias ágeis – especificamente a Dynamic Systems Development Method – configura-se como um possível motivador para a sua baixa adoção e referências disponíveis na literatura.

3.8. Feature-Driven Development

Feature-Driven Development (FDD, do português Desenvolvimento Orientado por Funcionalidade) é uma metodologia ágil, adaptativa, iterativa e incremental de desenvolvimento de software (PALMER & FELSING, 2002). FDD incorpora uma série de melhores práticas ágeis de desenvolvimento de software, todas direcionadas à perspectiva de funcionalidade com valor agregado ao cliente (PALMER & FELSING, 2002). Conforme afirmam Palmer e Felsing (2002), FDD não cobre todo o processo de desenvolvimento, focando especificamente nas atividades de Design e Desenvolvimento, ainda assim, não especifica que se faz necessário integrá-lo a outro processo.

O conceito de funcionalidade proposto por FDD corresponde a expressões granulares que representem algum valor para o cliente (AMBLER., 2005). Uma funcionalidade é nomeada por meio da estrutura ação-resultado-objeto, por exemplo: “calcular o desconto de uma venda”, sendo o termo “calcular desconto” a funcionalidade e o termo “de uma venda” o resultado/objeto (PALMER & FELSING, 2002). Funcionalidades estão para o FDD como

Casos de Uso estão para o Rational Unified Process (RUP) e Estórias de Usuários estão para o XP, sendo a fonte primária de requisitos e entradas para o planejamento de esforços de desenvolvimento (AMBLER, 2005).

3.8.1. Visão Geral do Processo

FDD é um processo de iteração curta, dirigido a modelos e que contempla cinco atividades básicas. Diferentemente de outras metodologias ágeis, FDD é definida como aderente ao desenvolvimento de sistemas críticos (ABRAHAMSSON ET AL., 2002). A estrutura do processo FDD inclui objetivos, papéis, artefatos e uma visão estruturada de marcos lógicos para acompanhamento da gestão (PALMER & FELSING, 2002). A metodologia provê relatórios de estado para controle do projeto, e por meio dos marcos lógicos, é possível identificar o progresso feito em cada funcionalidade (PANG & BLAIR, 2004).

Conforme ilustrado na Figura 18 e detalhado na tabela 11, o FDD contempla cinco processos sequenciais. Um projeto FDD começa executando as três primeiras etapas, com o objetivo de identificar o escopo do esforço, a arquitetura inicial e o plano inicial de alto nível (AMBLER, 2005). Os esforços de Construção ocorrem em iterações que duram duas semanas (ou menos), com a equipe trabalhando de forma iterativa através de todas as cinco etapas. FDD sugere que uma funcionalidade deve possuir uma granularidade menor que o tamanho da sua iteração, por exemplo: caso tenha sido definido que o projeto terá iterações de duas semanas, uma funcionalidade não deve ultrapassar esse tempo. Caso isso ocorra, a funcionalidade deve ser quebrada em partes menores (PALMER & FELSING, 2002). Tal como acontece em outros processos de desenvolvimento ágil de software, os sistemas são entregues de forma incremental (PALMER & FELSING, 2002).

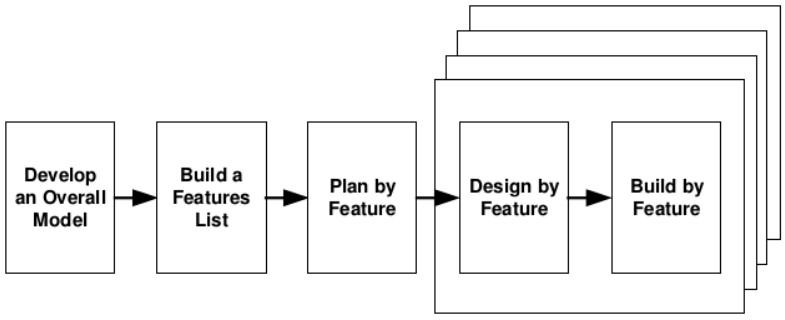


Figura 18: Processos do Feature-Driven Development (PALMER & FELSING, 2000).

Desenvolver um Modelo Geral (*Develop an Overall Model)*	O projeto inicia por um passo a passo de alto nível para entendimento do escopo do sistema e seu contexto. Em seguida, são detalhadas as orientações de domínio para cada área de modelagem. Como suporte para cada domínio, modelos de passo a passo são produzidos por grupos pequenos, os quais são apresentados e discutidos pelo grupo. Uma das propostas, ou mesmo a junção de várias delas, é escolhida e definida com modelo geral.
Construir Lista de Funcionalidades (*BuidFeatures List*)	O conhecimento do que foi reunido na modelagem inicial é usado para construir a lista de funcionalidades. Isso é realizado através da decomposição do domínio em funcionalidades por áreas de assunto (*subject areas*). As áreas de assunto contempla atividades de negócio, os passos para cada atividade de negócio, formando assim uma lista categorizada de funcionalidades. As funcionalidades são expressões de valor ao cliente compostas pela estrutura <ação> < resultado> <objeto>. Funcionalidades não devem ser maiores que o tamanho da iteração (se necessário, podem ser divididas em funcionalidades menores).
Planejar por	Após a construção da lista de funcionalidades, é conduzido o planejamento do

Funcionalidade (*Plan by Feature*)	desenvolvimento, onde o conjunto de funcionalidades é ordenado de acordo com a prioridade as quais são atribuídas ao Programador Chefe (vide Seção 3.8.2.). Além disso, cada Classe identificada no Modelo Geral deve ser atribuída individualmente a um Programador (vide Seção 3.8.2.), haja vista que o FDD propõe o conceito de Proprietário de Classe (*class ownership*), onde cada classe é de responsabilidade de um único programador. Nesse processo também é definido o cronograma.
Desenhar por Funcionalidade (*Design by Feature*)	Um pacote de design é produzido para cada funcionalidade. Um Programador Chefe seleciona um pequeno grupo de funcionalidades que serão desenvolvidas dentro de uma iteração. Juntamente com os proprietários de classes correspondentes, o programador chefe trabalha no detalhamento de Diagramas de Sequência para cada funcionalidade e refina o modelo geral. Em seguida, as assinaturas das classes e métodos são escritas e, finalmente, uma inspeção de design é realizada.
Construir por Funcionalidade (*Buid by Feature*)	Após uma inspeção de design por funcionalidade bem sucedida, uma função de valor ao cliente é produzida. Os proprietários de classes desenvolvem o código das suas classes. Após a execução de um Teste de Unidade e uma inspeção de código bem sucedida, a funcionalidade completa é promovida para a compilação principal.

Tabela 11: Processos do Feature-Driven Development (Adaptado de ABRAHAMSSON ET AL., 2002, Tradução do Autor).

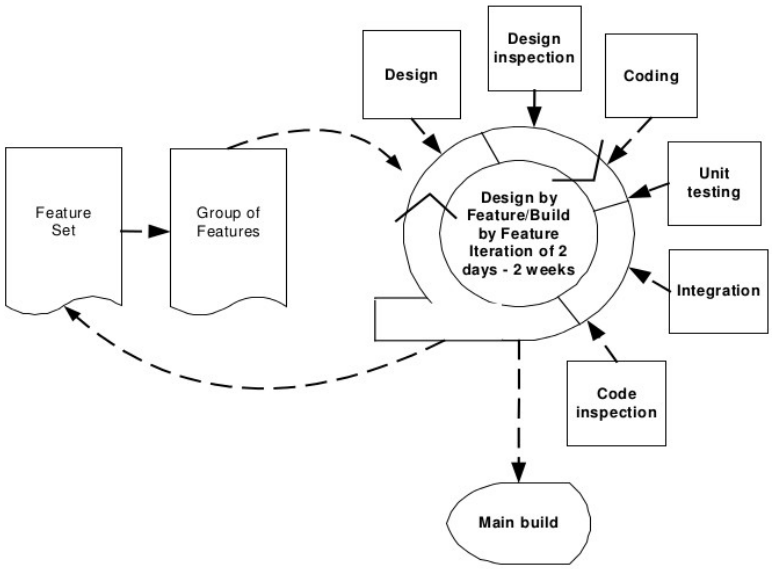


Figura 19: Processos de Desenho por Funcionalidade (*Design by Feature*) e Construção por Funcionalidade (*Build by Feature*) do FDD (ABRAHAMSSON ET AL., 2002).

3.8.2. Papéis

Conforme Palmer e Felsing (2002), o FDD classifica os papéis executores do processo em três categorias distintas: Papéis Principais, Papéis de Suporte e Papéis Adicionais. Na tabela 12 são apresentados os papéis previsto por FDD e uma breve descrição das suas responsabilidades e atribuições.

Gerente de Projeto (*Project Manager*)	Líder administrativo e financeiro do projeto. Possui atribuições relacionadas a manter o time focado, bem como prover os recursos necessários para tal.
Arquiteto Chefe (*Chief Architect*)	Responsável pelo design geral do sistema, bem como por conduzir sessões de design com o time.
Gerente de Desenvolvimento (*Development Manager*)	Lidera atividades diárias relacionadas ao desenvolvimento, soluciona conflitos e gere o time.

Programador Chefe (*Chief Programmer*)	Desenvolvedor experiente, envolvido com análise de requisitos e design dos projetos.
Proprietário de Classe (*Class Owner*)	Trabalham sob a orientação de Programadores Chefe nas tarefas de projeto, codificação, testes e documentação. Responsável pelo desenvolvimento da classe que lhe foi atribuída como proprietário. Para cada iteração os proprietários de classe estão envolvidos de acordo com as classe que estão incluídas nas características selecionadas à iteração seguinte do desenvolvimento.
Especialista de Domínio (*Domain Expert*)	Usuário, cliente, patrocinador, analista de negócio ou alguém que conheça como diferentes requisitos do sistema devem se comportar.
Gerente de Domínio (*Domain Manager*)	O Gerente de Domínio lidera Especialistas de Domínio e resolve diferentes opiniões a cerca de um determinado requisito do sistema.
Gerente de Liberação (*Release Manager*)	Controla o progresso do projeto através dos relatórios providos pelos Programadores Chefe, reportando ao Gerente do Projeto.
Advogado/Guru de Linguagem (*Language Lawyer/Guru*)	Membro do time responsável por possuir conhecimento específico de uma linguagem de programação ou tecnologia.
Engenheiro de Compilação (*Build Engineer*)	Responsável por preparar, manter e executar processos de compilação, incluindo tarefas de controle de versão e publicação de documentos.
Especialista de Ferramentas (*Toolsmith*)	Responsável por construir pequenas ferramentas para o desenvolvimento, tests e conversão de dados. Mantém bases de dados e websites do projeto.
Administrador de Sistema (*System Administrator*)	Responsável por configurar, gerir e resolver problemas com servidores, redes, estações de trabalho, ambientes de testes e desenvolvimento.
Testador (*Tester*)	Responsável por verificar se o sistema atender aos requisitos.
Implantador (*Deployer*)	Responsável por converter dados existentes ao formato requerido pelo novo sistema e implantar novas liberações (*releases*) do sistema.
Escritor Técnico (*Technical Writer*)	Responsável por preparar documentação técnica.

Tabela 12: Papéis do Feature-Driven Development (Adaptado de PALMER &FELSING, 2002, Tradução do Autor).

3.8.3. Práticas e Execução do Processo

FDD foi desenvolvido através da compilação de uma série de boas práticas derivadas da Engenharia de Software (ABRAHAMSSON ET AL., 2002). Todas essas práticas estão direcionadas a prover funcionalidades de valor ao cliente, sendo a compilação dessas práticas uma das características que fornecem valor ao FDD. As principais práticas do FDD são listadas a seguir:

- **Modelagem de Objetos de Domínio (*Domain Object Modeling*):** Consiste na exploração e explicaçãodo domínio do problema a ser resolvido. O resultado do Modelo de Domínio é prover um modelo de trabalho geral das funcionalidades;
- **Desenvolver por Funcionalidade (*Developing by Feature*):** Qualquer função é complexa demais para ser implementada em uma iteração. Nesse sentido, deve-se decompor a função em funções menores, até que cada problema possa ser chamado de funcionalidade;
- **Propriedade Individual de Classes (*Individual Class Ownership*):** Significa que pedaços distintos ou grupos de códigos são atribuídos a um único proprietário;
- **Times de Funcionalidades (*Feature Teams*):** Time pequeno e dinâmico formado para desenvolver uma pequena atividade;

- **Inspeções (*Inspections*):** Inspeções são realizadas para garantir a boa qualidade do design e da codificação;
- **Gerência de Configuração (*Configuration Management*):** Apoia na identificação de código fonte para todas as funcionalidades, bem como mantém o histórico de mudanças das classes;
- **Compilações Regulares (*Regular Builds*):** Garantia de que há sempre um sistema atualizado para apresentar ao cliente;
- **Visibilidade de Progresso e Resultados (*Visibility of Progress and Results*):** Reporte frequente, apropriado e acurado do progresso do projeto.

Uma característica importante da prática do FDD é que uma vez que as funcionalidades possuem um tamanho pequeno, desenvolvê-las é um trabalho relativamente pequeno (PALMER & FELSING, 2002). Para que haja um efetivo reporte do estado e evolução do projeto, o FDD define seis marcos por funcionalidade os quais precisam ser concluídos sequencialmente. Os primeiros três marcos são obtidos durante a atividade de Desenho por Funcionalidade (*Design by Feature*), e os demais na atividade de Construção por Funcionalidade (*Build by Feature*). Para auxiliar no monitoramento do progresso, um percentual de evolução é atribuído para cada marco, conforme exemplificado a seguir.

Análise do Domínio (*Domain Walkthrough*)	**Desenho (*Design*)**	**Inspeção de Desenho (*Design Inspection*)**	**Codificação (*Code*)**	**Inspeção de Codificação (*Code Inspection*)**	**Promover para Compilação Principal (*Promote to Build*)**
1%	40%	3%	45%	10%	1%

Tabela 13: Exemplo de Acompanhamento de Progresso dos Marcos por Funcionalidade (PALMER & FELSING, 2002).

3.8.4. Artefatos

Por se tratar de uma metodologia centrada no Design e Desenvolvimento, FDD menciona um conjunto reduzido de artefatos essenciais ao seu processo (PALMER & FELSING, 2002). Um dos fatores que reforçam essa característica é o fato de que FDD pressupõe que ao entrar no desenvolvimento do projeto, já se tenha em mãos os requisitos, sejam eles Casos de Uso (*Use Cases*), Estórias de Usuários (*User Stories*) ou qualquer outro detalhamento de requisitos (PANG & BLAIR, 2004). Ainda assim, alguns artefatos são fundamentais ao processo descrito por FDD, conforme listados a seguir.

- **Modelo Geral (*Overall Model*):** Modelo geral do sistema, definido de forma incremental e que representa tecnicamente o entendimento do domínio;
- **Lista de Funcionalidades (*Features List*):** Listagem das funcionalidades derivadas dos requisitos, com a respectiva categorização e atribuições de Proprietários de Classes;
- **Plano de Desenvolvimento (*Development Plan*):** Sequencia de desenvolvimento de funcionalidades, considerando as prioridades dos clientes;
- **Diagramas de Sequência por Funcionalidade (*Sequence Diagram per Feature*):** Diagrama de Sequência em UML de cada Funcionalidade prevista;
- **Relatório de Marcos (*Milestone Reports*):** Tabela de evolução do progresso do desenvolvimento por Funcionalidade.

3.8.5. Considerações Finais

O entendimento de que os requisitos do sistema precisam ser decompostos em partes pequenas – funcionalidades – é uma abordagem interessante que permite suportar o princípio ágil de entregas constantes. Isso porque, quanto menor o tamanho do componente, mais rápido ele pode ser entregue e apresentado ao cliente (PALMER & FELSING, 2002). Outra característica interessante do FDD diz respeito à atribuição do papel de Proprietário de Classe (*Class Owner*). Conforme apresentado, o conceito de Proprietário de Classe é um diferencial do FDD ante as outras metodologias ágeis, tais como o XP (Extreme Programming). O XP inclui uma prática chamada Propriedade Coletiva (*Collective Ownership*), onde a ideia é que qualquer desenvolvedor possa atualizar qualquer artefato, incluindo o código fonte (AMBLER, 2005). Já o FDD propõe uma abordagem diferente, uma vez que cada classe é atribuída a desenvolvedores individuais, e por isso, caso uma característica exija mudanças em várias classes, os donos dessas classes devem trabalhar juntos como uma equipe de funcionalidades para implementá-la (AMBLER, 2005).

Conforme apresentado por Abrahamsson et al. (2002), não há na literatura relatos significativos quanto à aplicação bem sucedida ou mesmo má sucedida do FDD. Provavelmente por se tratar de uma metodologia recente, ainda não existem evidências suficientes na literatura que respaldem sua aplicação. Entretanto, FDD apresenta uma visão diferenciada a cerca do processo de desenvolvimento.

3.9. Test-Driven Development

Test-Driven Development (TDD, do português Desenvolvimento Orientado a Testes) consiste em um método ágil de desenvolvimento de software, o qual propõe uma abordagem baseada na repetição de ciclos curtos de desenvolvimento (BECK, 2003). O ciclo básico de uma abordagem baseada em TDD sugere três etapas: inicialmente o desenvolvedor escreve um caso de teste automatizado, o qual precisa falhar para assim definir a demanda de melhoria ou desenvolvimento da nova função; em seguida o desenvolvedor escreve a mínima quantidade de código suficiente para que o teste passe; por fim, o desenvolvedor refatora o código para padrões aceitáveis de codificação (SHORE & WARDER, 2008). A abordagem proposta por TDD encoraja o design simples e implementa na prática o conceito de "testar primeiro" sugerido por Extreme Programming (MADEYSKI & SZALA, 2007). Os conceito propostos por TDD também auxiliam a melhoria e correção de erros em códigos legados desenvolvidos com técnicas antigas (BECK, 2003).

3.9.1. Visão Geral do Processo

O TDD descreve um fluxo baseado em três macro atividades: Escrever Teste; Escrever código de Produção; Refatorar (GEORGE & WILLIAMS, 2003). Esse fluxo é repetido sequencialmente até que a condição do caso de teste esteja satisfeita pelo código de produção construído (BUFFARDI & EDWARDS, 2012). A sequência do fluxo do processo de Test-Driven Development é apresentada na Figura 20 e descrita a seguir.

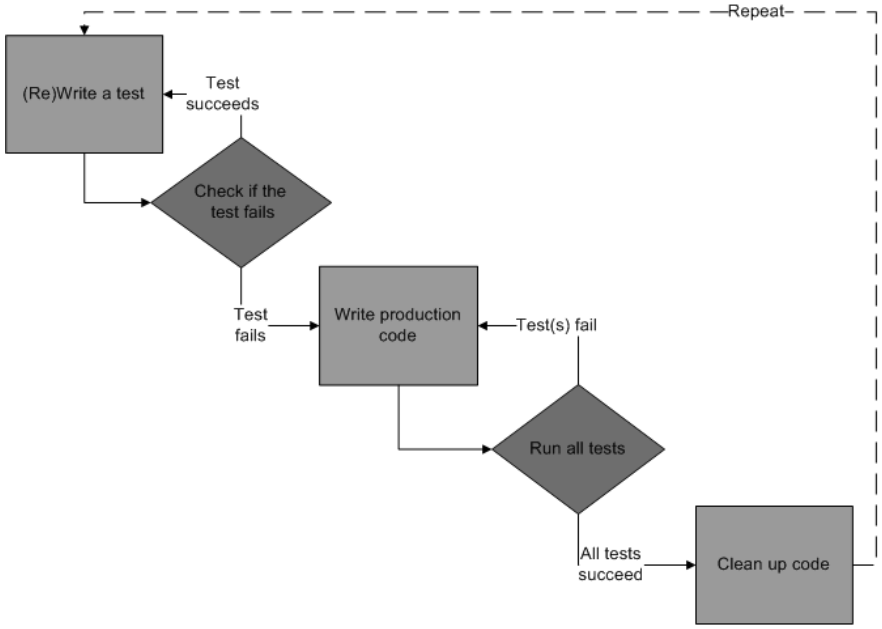


Figura 20: Ilustração baseada no fluxo de desenvolvimento de Test-Driven Development (BECK, 2003).

- **Escrever um Teste (*Write a Test*):** Em TDD, a implementação de cada nova funcionalidade começa com a escrita de um teste. Este teste deve inevitavelmente falhar uma vez que é escrito antes do recurso ser implementado. É fundamental que o teste falhe, do contrário, ou o recurso "novo" proposto já existe ou o teste está com defeito. Para escrever um teste, o desenvolvedor deve entender claramente a especificação e requisitos da função. O desenvolvedor pode fazer isso por meio de Casos de Uso ou Estórias de Usuários para cobrir os requisitos e as condições de exceção, e pode escrever o teste em qualquer framework de teste apropriado para o ambiente de software. O referido teste também pode ser uma modificação de um teste existente. Essa característica é o grande diferencial do Desenvolvimento Orientado a Testes, ante a prática comum de escrever testes de unidade após o código estar escrito: o desenvolvedor foca nos requisitos antes de escrever o código, uma diferença sutil, mas importante (BECK, 2003);
- **Verificar se o teste falha (*Check if the test fails*):** Isso confirma que o teste está funcionando corretamente e que o novo teste não passa sem a necessidade de um novo código. Esta etapa também testa o próprio teste em si: exclui a possibilidade de que o novo teste sempre passa, e, portanto, é inútil;
- **Escrever Código de Produção (*Write Production Code*):** O próximo passo é escrever um código que faça com que o teste passe. O código escrito nesta fase não é perfeito, e pode, por exemplo, passar no teste de uma maneira deselegante. Isso é aceitável, pois as próximas etapas irão melhorar e aprimorar o código. Neste momento, a única finalidade do código é passar no teste. Nenhuma nova (e, portanto, não testada) funcionalidade deve ser prevista e permitida nesta fase (BECK, 2003);
- **Executar todos os testes (*Run all tests*):** Se todos os casos de teste passarem, o desenvolvedor pode ter a certeza de que o código preenche todos os requisitos testados. Este é um bom ponto de partida para começar a etapa final (BECK, 2003);
- **Limpar o código e Refatorar (*Clean up code and Refactor*):** Nesse momento o código deve ser limpo, se necessário. Deve-se mover o código de onde era conveniente para passar no teste para onde ele pertence logicamente. São removidas quaisquer duplicações, certificar-se quanto aos nomes de variáveis, bem como se métodos representam o seu uso. É importante utilizar a regra de Beck (2003) de

Design Simples[11] quanto à clareza do código. Ao re-executar os casos de teste, o desenvolvedor deve ter certeza de que o código refatorado não é prejudicial à outra funcionalidade existente. O conceito de eliminação de duplicações é importante em qualquer projeto de software. Neste caso, no entanto, também se aplica à remoção de qualquer duplicação entre o código de teste e código de produção (BECK, 2003);

- **Repetir (*Repeat*):** O processo deve ser rexecutado com um novo teste, repetindo o ciclo para avançar com no desenvolvimento das funcionalidades. O tamanho dos passos entre escrita de teste, falha, codificação e refatoração devem ser sempre pequenos, algo entre 1 a 10 edições para cada execução de teste. Se o novo código não satisfaz rapidamente um novo teste, ou outros testes falham inesperadamente, o desenvolvedor deve desfazer ou reverter a alteração do código, em vez de debugar o código. Para isso, é fundamental a execução de integração contínua[12] entre as funcionalidades do sistema, fornecendo pontos de verificação – *checkpoints* – reversíveis (BECK, 2003).

Características do Desenvolvimento baseado em Test-Driven Development

Existem vários aspectos característicos no uso de TDD, tidos como princípios da metodologia. Os princípios de "*Keep It Simple Stupid*" (KISS, do português "Mantenha Isto Simples Estúpido") e "*You Ain't Gonna Need It*" (YAGNI, do português "Você Não Irá Precisar Disso"). Ao focar em escrever apenas o código necessário para passar nos testes, o design tende a ser mais limpo e claro do que o alcançado por outros métodos. Beck (2003) também sugere o princípio "*Fake it till you make it*" (Finja até você fazer). Esse princípio consiste na sugestão de que métodos retornem uma constante e que essas constantes sejam gradativamente substituídas por variáveis até o código final ser produzido.

A escrita dos testes primeiro dirige um entendimento mais profundo e mais cedo dos requisitos do produto, que além de assegurar a eficácia do código de teste, mantém um foco contínuo na qualidade do produto. Em uma abordagem tradicional, ao se escrever o código do recurso antes do teste, há uma tendência dos desenvolvedores e das organizações em forçar o desenvolvimento para a próxima funcionalidade antes da execução completa dos testes (GEORGE & WILLIAMS, 2004). Escrevendo os testes antecipadamente, o primeiro teste pode até não compilar, uma vez que nem todas as classes e métodos existem. No entanto, à medida em que o desenvolvimento evolui, evolui também a concepção dos componentes do sistema.

Desenvolvimento Orientado a Testes constantemente repete os passos de adição de casos de teste que falham, passando-os, e refatorando-os. Receber os resultados dos testes previstos

---

[11] A regra de Design Simples de Beck (2003) propõe que deve-se codificar apenas o necessário para passar nos testes e remover todas as duplicações de código. Através disso, se obtém automaticamente um design que é perfeitamente adaptável aos requisitos atuais e igualmente preparado para futuras Estórias de Usuário. A mentalidade de que se está procurando apenas o design suficiente para atender a arquitetura do sistema, faz com que a escrita dos testes seja mais fácil (BECK, 2003).

[12] Integração Contínua é uma pratica de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um *build* automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente (FOWLER, 2006, Tradução do Autor, disponível em: http://martinfowler.com/articles/continuousIntegration.html).

em cada etapa reforça o modelo mental do desenvolvedor, além de aumentar a confiança e a produtividade (GEORGE & WILLIAMS, 2004).

3.9.2. Papéis

O TDD não descreve em detalhes a estrutura de papéis e responsabilidades, haja visto que o método descrito por TDD é claramente centrado e focado nos esforços de desenvolvimento, não tratando em seu escopo outros aspectos da Engenharia de Software, tal como a gestão do projeto (BECK, 2003). Ainda assim, alguns papéis implícitos às atividades do TDD são descritos na literatura sobre a metodologia, tais como: Analista (*Analyst*), Desenvolvedor (*Developer*) e Usuário (*User*) (BECK, 2003).

3.9.3. Práticas e Execução do Processo

Um dos princípios fundamentais do TDD roga que é preciso manter as unidades pequenas (BECK, 2003). Para TDD, uma unidade é mais comumente definida como uma classe ou grupo de funções relacionadas, muitas vezes chamadas de um módulo. Manter unidades relativamente pequenas é proposto para proporcionar benefícios como:

- **Redução do esforço de depuração de código:** Unidades menores facilitam o rastreamento de erros quando falhas nos testes são detectadas;
- **Testes auto-documentados:** Casos de Testes pequenos melhoraram legibilidade a rápida compreensão do código.

Práticas avançadas de TDD podem levar ao desenvolvimento de Acceptance Test-Driven Development (ATDD, do português “Desenvolvimento Orientado a Testes de Aceitação), onde os critérios especificados pelo cliente são automatizados para testes de aceitação, os quais em seguida conduzem ao processo tradicional de TDD (CHRISTENSEN, 2010). O processo de ATDD garante que o cliente tem um mecanismo automatizado para decidir se o software atende aos seus requisitos. Com ATDD, a equipe de desenvolvimento passa a ter um alvo específico para satisfazer, os Testes de Aceitação, o que mantém os desenvolvedores continuamente focados no que o cliente realmente quer do referido requisito (KOUDELIA, 2001). Os conceitos de ATDD são detalhados na Seção 3.12.3. dessa pesquisa. TDD sugere a adoção de boas práticas específicas, ou mesmo, práticas que devem ser evitadas para que a abordagem seja utilizada conforme prevista. Essas práticas, conforme previsto por Beck (2003) são explanadas a seguir:

- **Estrutura de Teste:** A estrutura padrão de aplicação de um Caso de Teste consiste em: 1) Setup, colocar a Unidade sob Teste (*Unit Under Test*, UUT) ou todo o sistema de teste no estado necessário para executar os testes; 2) Execução, disparar a UUT para executar o comportamento desejado, coletando toda a saída, desde valores a parâmetros; 3) Validação, garantir que o resultado do teste está correto; 4) Limpeza, restaurar o UUT ou todo o sistema ao estado inicial antes dos testes;
- **Dependência do Estado do Sistema:** Não é indicado se ter Casos de Testes dependendo do estado manipulado do sistema na execução anterior;
- **Dependência de Casos de Testes:** Deve-se evitar a dependência entre Casos de Testes. A ordem de execução deve ser determinada e constante. Refatoração dos casos de teste pode acarretar em um crescente aumento do impacto em testes associados;
- **Tests Independentes:** Casos de Testes dependentes causam resultados falso negativos em cascata, uma vez que uma falha no caso de teste inicial causam uma falha em um

caso de teste seguinte, mesmo que não exista nenhuma falha no UUT, aumentando a análise de defeitos e esforços de debugging.

Test-Driven Development na Prática

Conforme observaram Buffardi e Edwards (2012), por vezes é difícil para desenvolvedores com pouca experiência compreender inicialmente os conceitos propostos por TDD. Nesse sentido, a aplicação prática da metodologia torna-se um forte aliado para a adequada compreensão dos conceitos e práticas de TDD (BUFFARDI & EDWARDS, 2012). Será explanada a seguir a aplicação típica de um modelo completamente guiado por TDD. O propósito é compreender o ritmo do desenvolvimento orientado por testes.

Para aplicação prática dos conceitos de TDD, será usado um exemplo baseado em Kata de Programação. Nas artes maciais, um kata é um conjunto preciso de movimentos que simula um combate. A prática do kata objetiva a perfeição do executor, quanto à execução perfeita de movimentos de forma automática (HUNT & THOMAS, 1999). O Kata de Programação é um conceito proposto por Hunt e Thomas (1999), cujo objetivo é capacitar o programador a atingir a excelência através da prática. O kata de programação é bastante utilizado para o aprendizado de TDD, uma vez que solucionar um kata de programação é muito mais fácil utilizando TDD do que uma abordagem tradicional (HUNT & THOMAS, 1999). No exemplo a seguir, a proposta é resolver um Kata de Calculadora de String[13], onde essa calculadora receberá números de uma string e deverá retornar a soma desses números. A seguir são elucidados os requisitos completos dessa calculadora:

1. Criar uma calculadora de strings com o método **int add(string numbers)**, onde esse método pode receber os números 0, 1 ou 2 e deverá retornar a soma destes números (no caso de string vazia, deverá retornar 0), por exemplo: " ", ou "1" ou "1,2";
2. Permitir ao método **int add(string numbers)** qualquer quantidade de números;
3. Permitir ao método **int add(string numbers)** aceitar novas linhas, onda a seguine entrada será válida: "1\n2,3", sendo a soma igual a 6 e sendo a entrada "1,\n" inválida.

A proposta é que esse kata seja solucionado com a utilização de TDD, onde em vez de buscar a codificação da solução completa, deverá se pensar na calculadora como pequenos problemas a serem resolvidos, criando-se assim pequenos casos de testes para cada um dos pequenos problemas da calculadora. A primeira condição de problema seria a criação de um caso de teste para uma String vazia:

```
package calculator;
import junit.framework.TestCase;
public class StringCalculatorTest extends TestCase {
   private StringCalculator sCalc;
   protected void setUp() throws Exception {
        sCalc = new StringCalculator();
    }
   public void testStringCalculatorEmptyString() throws Exception {
        assertEquals(0, sCalc.add(""));
    }
}
```

No passo anterior, foi criado o caso de teste, porém não foi criada a classe **StringCalculator**, o que por consequência não permitiria a compilação. Essa é exatamente a abordagem

[13] Problema Kata de Calculadora de String proposto em http://osherove.com/tdd-kata-1/, acessado em 8 de Janeiro de 2014.

proposta por TDD, uma vez que esse teste falhou, para passar, deverá ser criada a classe **StringCalculator**, implementando o método **add(String numbers)**.

```
package calculator;
import java.util.StringTokenizer;
public class StringCalculator {
   public int add(String numbers) {
         return 0;
    }
}
```

Após a implementação da classe **StringCalculator** e do método **add(String numbers)**, não só o caso de teste compila como passa sem erros. É fato, porém, que a implementação é insuficiente, uma vez que o retorno do método é 0. Contudo, como a prerrogativa de TDD é fazer o teste passar sem erros e continuar com a implementação, a ideia é seguir com o desenvolvimento dos casos de testes e quando satisfeita a condição de execução, refatorar o código para um padrão aceitável de codificação. No passo a seguir dessa implementação, será adicionado um novo caso de teste, o qual testará uma string com um número.

```
package calculator;
import junit.framework.TestCase;
public class StringCalculatorTest extends TestCase {
   private StringCalculator sCalc;
   protected void setUp() throws Exception {
         sCalc = new StringCalculator();
    }
   public void testStringCalculatorEmptyString() throws Exception {
         assertEquals(0, sCalc.add(""));
    }
   public void testStringCalculatorZero() throws Exception {
         assertEquals(0, sCalc.add("0"));
    }
}
```

No trecho de código acima foi implementado do caso de teste **testStringCalculatorZero()**, o qual testa o método **add(String numbers)** com o valor 0. Uma vez executado esse teste, ele irá passar, haja vista que 0 é esperado e o método retorna zero. Seguindo com as condições apontadas no requisito levantado inicialmente, será implementado o teste para o número 1.

```
package calculator;
import junit.framework.TestCase;
public class StringCalculatorTest extends TestCase {
    private StringCalculator sCalc;
    protected void setUp() throws Exception {
         sCalc = new StringCalculator();
    }
    public void testStringCalculatorEmptyString() throws Exception {
         assertEquals(0, sCalc.add(""));
    }
    public void testStringCalculatorZero() throws Exception {
         assertEquals(0, sCalc.add("0"));
    }
    public void testStringCalculatorOne() throws Exception {
         assertEquals(1, sCalc.add("1"));
    }
}
```

No código anterior, foi implementado o teste **testStringCalculatorOne()**, o qual testa o método **add(String numbers)** com o número 1. Com a execução deste teste, passando o valor 1 como parâmetro, ele falha uma vez que o retorno esperado era 1, porém o retorno foi zero. Por fim, o código será refatorado para atender aos padrões de codificação.

```
package calculator;
import java.util.StringTokenizer;
public class StringCalculator {
    public int add(String numbers) {
        if(numbers.isEmpty()) {
            return 0;
        }
        int number = Integer.valueOf(numbers);

        if(number == 1) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

Com a refatoração, todas as condições são testadas para garantir que essas são atendidas: testa-se a String vazia, com retorno de 0; testa-se o valor da String em um inteiro, com o retorno de 1, senão, de 0. Com a execução positiva de todas as condições, é garantida a implementação do requisito inicial, bem como de todos os testes para a sua execução adequada. Naturalmente a implementação pode não ser a mais adequada, porém TDD possibilita criar o código de forma incremental e refatorá-lo a cada execução de rodada de testes. Salienta-se igualmente que TDD possibilita a validação de que ações de refatoração não afetam o comportamento esperado do código.

Test-Driven Development Mantra

Conforme afirma Beck (2003), em TDD o objetivo é alcançar um design simples e uma codificação mais simples ainda. Para Beck, em TDD deve-se: escrever um novo código apenas se inicialmente a execução de um teste automático falhou ou para e a eliminar duplicação. Essas regras básicas geram não só um comportamento individual complexo no desenvolvedor, como também em todo o grupo. Nesse sentido, algumas implicações técnicas são consideradas (BECK, 2003):

- É preciso focar no design, com código executando e provendo o feedback necessário entre decisões;
- O próprio desenvolvedor deve escrever seus testes;
- O ambiente de desenvolvimento deve prover uma rápida resposta a mudanças pequenas;
- O design deve consistir em componentes coesos, livres de dependências excessivas, de forma a tornar a atividade de testes simples.

Os princípios abordados anteriormente são definidos por um conceito chamado de Mantra do TDD: Vermelho-Verde-Refatorar (*Red-Green-Refactor*) (BECK, 2003). Esse conceito está associado a uma sequencia lógica de desenvolvimento, conforme a seguir:

- **Vermelho (*Red*):** Escreve-se um teste simples que não funciona, talvez nem mesmo compile inicialmente;
- **Verde (*Green*):** Implementa-se rápido para que o teste funcione, mesmo que de uma forma não tradicional de codificação;
- **Refatorar (*Refactor*):** Elimina-se toda a duplicação criada apenas para fazer o teste funcionar. Os padrões de codificação são considerados, desde que não alterem o comportamento do código.

Benefícios da aplicação de Test-Driven Development

Um dos principais questionamentos de quem não conhece a prática de TDD está relacionado ao fato de que, por igualmente dedicar esforços em codificar a lógica do software, bem como seus casos de testes, o desenvolvimento pode mais lento, dada a necessidade de se escrever mais código (BUFFARDI & EDWARDS, 2012). Entretanto, uma pesquisa realizada por Erdogmus et al. (2005) concluiu que o uso de TDD de fato produz mais código, porém, os desenvolvedores que escrevem mais testes tornam-se mais produtivos. A pesquisa revelou que desenvolvedores que utilizam puramente TDD reportaram que raramente precisam realizar *debug* de código, onde quando um teste falha inesperadamente, reverter o código a sua última versão funcional é mais produtivo do que realizar *debug*.

Outro ponto identificado é que o uso de TDD resulta em um desenvolvimento mais modularizado, flexível e extensível (ERDOGMUS ET AL., 2005). Essa consequência provavelmente está relacionada ao fato de que em TDD, o desenvolvedor sempre precisa pensar em pequenas unidades que podem ser escritas e testadas de forma independente. Madeyski (2010) identificou através de evidência empírica (experimentos em laboratório com mais de 200 desenvolvedores) a superioridade de TDD ante a uma abordagem tradicional de teste após a codificação, onde o baixo acoplamento entre objetos (*Lower Coupling Between Objects, CBO*) sugere uma melhor modularização (ou seja, um design mais modular) e uma adoção mais fácil de reuso de código (MADEYSKI, 2010; JANZEN & SAIEDIAN, 2006).

3.9.4. Artefatos

Por se tratar de uma metodologia dedicada aos esforços de codificação e desenvolvimento, não considerando assim outras disciplinas normalmente abordadas em Engenharia de Software, TDD não trata em específico a adoção de artefatos em seu método (BECK, 2003). Contudo, identificou-se na literatura sobre o tema a menção a documentos e componentes específicos e diretamente relacionados à prática e adoção de TDD.

- **Estória de Usuário (*User Story*):** Por tratar-se de um método ágil, a literatura comumente refere-se aos requisitos do cliente como Estórias de Usuário (*User Stories*), ainda que menções a Casos de Uso também tenham sido identificados (MADEYSKI, 2010);
- **Teste de Unidade (*Unit Test*):** Os Testes de Unidade consideram a implementação de um teste automático o qual valida uma única unidade de código. Conforme afirma Beck (2003), um módulo complexo pode conter milhares de testes de unidade e um módulo simples pode contar apenas dez, por exemplo. Os testes de unidade são o principal componente na adoção da prática de TDD;
- **Falso, Modelo (*Fake, Mock*):** Métodos de modelos falsos de objetos podem ser configurados em modos de falha predefinidos, para que as rotinas de tratamento de erros possam ser desenvolvidas e testadas de forma confiável. Em um modo de falha, um método pode retornar uma resposta inválida, incompleta ou nula, ou pode levantar uma exceção (BECK, 2003);
- **Dublê de Testes (*Test Double*):** Um dublê de testes é um recurso de teste que substitui uma funcionalidade do sistema, normalmente uma classe ou função, que uma UUT (*Unit Under Test*, do português Unidade em Testes) depende. Existem duas situações em que dublês de teste podem ser utilizados em um sistema: ligação e execução. Substituição do tempo de ligação é quando um dublê de teste é compilado para o módulo de carga, no qual é executado para validar o teste. Esta abordagem é

normalmente usada quando executada em um ambiente diferente do ambiente de destino, e que requer dublês para o código de nível de hardware para a compilação. Quanto a execução, refere-se a substituição de uma funcionalidade real durante a execução de um dos casos de teste. Esta substituição é normalmente feita através da redesignação de ponteiros de função conhecidos ou substituição de objetos (BECK, 2003).

3.9.5. Considerações Finais

A prática de desenvolvimento dirigido a testes foi inicialmente proposta em Extreme Programmig, como um dos muitos conceitos da metodologia cuja a abordagem foca em tornar mais ágil o processo de desenvolvimento de software (BECK, 2000). Entretanto, dado o paradigma ante as práticas tradicionais de engenharia de software, onde os testes são realizados após o desenvolvimento, a inversão dessa perspectiva é algo que ainda complexo de se compreender no primeiro contato com a metodologia, principalmente por desenvolvedores menos experientes (BUFFARDI & EDWARDS, 2012).

TDD, porém, não resume-se a "testar antes de desenvolver" (GRENYER, 2013). A prática do desenvolvimento dirigido a testes está centrada na busca por alguns objetivos específicos, tais como: 1) viabilizar um design simples; 2) modularização do software; 3) maior entendimento dos requisitos pelo desenvolvedor; 4) menor incidência de erros e defeitos no software; 5) maior produtividade pela redução de esforços com *debug*. A assertiva de que TDD não se resume a testar antes do desenvolvimento foi observada por Grenyer (2013, p. 27, Tradução do Autor):

> *"[...] Há muitas vantagens para o uso de TDD. As duas principais são de baixo acoplamento de código e que é fácil de se testar (obviamente). Na verdade, fazendo seu código ser fácil de testar torna baixo o acoplamento e fácil de realizar mudanças e esse é o ponto. [...] Escrever os testes primeiro obriga a tornar o código testável, mas não é o único caminho. Se os testes são automatizados e a cobertura de medição é automatizada ou melhor ainda, o sistema de integração contínua executa os testes e mede a cobertura de código, você é forçado a fazer o seu código testável."*

Desenvolvimento orientado a testes é um processo de desenvolvimento de software que se concentra na codificação confiável e de fácil manutenção: código limpo que funciona (CHRISTENSEN, 2010). TDD baseia-se nos valores de simplicidade, manter o foco e dar pequenos passos. Simplicidade sobre como manter as coisas simples e evitando a implementação de um código que não é realmente necessário (SHORE & WARDEN, 2008).

3.10. Behavior-Driven Development

Behavior-Driven Development (BDD, do português Desenvolvimento Dirigido a Comportamento) é uma metodologia ágil de desenvolvimento de software, cuja fundamentação está baseada em Test-Driven Development (HELLENSOY & WYNNE, 2012). BDD agrega técnicas e princípios do TDD com práticas derivadas do Domain-Driven

Design (DDD, do português Design Orientado a Domínio)[14] e análise e design orientado a objetos (HELLENSOY & WYNNE, 2012).

A motivação para a concepção do BDD se deu a partir do uso de Test-Driven Development, onde, considerando os mais diversos ambientes de projeto, é possível haver a confusão quanto ao que testar, o que não testar, quando testar e como compreender que um teste efetivamente falhou (NORTH, 2006). Conforme abordado por North (2006), a prática de TDD implica em algumas perguntas que precisam de respostas e que antes do estabelecimento da prática do BDD, não tinham resposta:

- Onde iniciar o processo de testes?
- O que testar e o que não testar?
- Quanto deve-se testar para um objetivo específico?
- Qual a nomenclatura dos testes?
- Como entender porque um teste falhou?

O núcleo do BDD é repensar a abordagem para testes de unidade e testes de aceitação. Propõe-se que os nomes de teste de unidade sejam frases inteiras que começam com a palavra "deve" e que devem ser escritos em ordem de valor do negócio (NORTH, 2006). Testes de aceitação devem ser escrito usando uma estrutura padrão ágil de uma estória de usuário: "Como o [papel], eu desejo [recurso], para que [benefício] (do inglês *As a [role], I want [feature], só that [benefit])*". Os critérios de aceitação devem ser escritos em termos de cenários e implementados como classes: "Dado que [contexto inicial], quando [ocorre evento], então[garantir alguns resultados] (*Given [initial context], When [event occurs], Then [ensure some outcomes])*" (NORTH, 2006).

3.11. Visão Geral do Processo

A proposta de Test-Driven Development aborda uma metodologia onde software é construído essencialmente em forma de unidades, onde o desenvolvedor deve: a) definir um teste para cada unidade do software; b) implementar a unidade; c) verificar se a implementação da unidade faz com que o teste seja bem sucedido (BECK, 2003). Essa definição, porém não especifica como devem ser estabelecidos os testes, nem em nível de requisitos de usuário, nem tampouco quanto a aspectos técnicos (SOLIS & WANG, 2011).

Com base nessa falta de especificação quanto a escrita dos testes, North (2006) propôs a metodologia Behavior-Driven Development, a qual define que testes de unidade devem ser especificados em termos de comportamentos desejados para a unidade (NORTH, 2006; AGILE ALLIANCE, 2013). Nesse sentido BDD, como metodologia ágil, aborda o comportamento desejado como requisitos estabelecidos pelo negócio, onde esses representam o valor agregado da unidade em construção (NORTH, 2006). Essa visão de requisitos como comportamentos é referenciada em BDD como um pensamento *outside-in* (do português, "De dentro para fora"), por priorizar o resultado e valor dos requisitos (NORTH, 2007).

[14] Domain-Driven Design (DDD, do português Design Orientado a Domínio) é uma abordagem para desenvolvimento de softwares para necessidades complexas, através da conexão entre implementação e um modelo evolutivo. A premissa do DDD considera: direcionar o foco primário do projeto no domínio núcleo e na lógica do domínio; basear designs complexos em um modelo de domínio; e uma colaboração criativa entre os técnicos e especialistas de domínio para iterativamente refinar o modelo conceitual para direcionar um domínio de problema em específico. *EVANS, E. Domain-Driven Design – Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.*

BDD descreve que um comportamento desejado deve ser especificado. Para tal, adota o uso de um formato semi-formal para especificação dos comportamentos, os quais são derivados das Estórias de Usuário, bem como da análise e design orientado a objetos (NORTH, 2006; NORTH, 2006). A estrutura de um comportamento em BDD é a seguinte:

Título: A estória de usuário deve conter um título claro e objetivo.

Uma breve introdução que especifique:

- **Qual efeito o envolvido deseja que a estória possua;**
- **Quem (papel)** é o envolvido primário na estória (o ator que entrega benefício ao negócio a partir da estória;
- **Qual valor de negócio** o envolvido irá entregar através desse efeito.

Critérios de aceitação e cenários, contemplando uma descrição para cada caso da narrativa:

- **Especificação do cenário;**
- **Definição de gatilhos** que disparam a execução do cenário;
- **Resultado e saída** do comportamento.

Considerando a estrutura abordada acima, BDD propõe o seguinte padrão estrutural:

Estória (*Story*): Título da História;
Dado que (*In order to*) …
Como (*As a*) …
Eu desejo que (*I want to*) ...
Cenário 1 (*Scenario 1*) ...
Dado que (*Given*) …
E (*And*) …
Quando (*When*) …
Então (*Then*) ...

BDD utiliza essa abordagem de descrição dos comportamentos do conceito de *linguagem ubíqua*, originalmente estabelecido em Domain-Driven Design (Design Dirigido a Domínios), onde uma linguagem ubíqua consiste em uma linguagem semi-formal que é compartilhada por todos os membros do time de desenvolvimento de software, tanto desenvolvedores como pessoal não técnico (SOLIS & WANG, 2011). Nesse sentido, BDD usa a especificação do comportamento desejado como uma linguagem ubíqua para os membros do time de projeto.

3.11.1. Papéis

Conforme propôs North (2006), a metodologia Behavior-Driven Development foi descrita para tratar aspectos até então não cobertos por Test-Driven Development, os quais conforme exposto anteriormente, estão relacionados às condições e especificidades para a escrita de testes de unidade. Bem como TDD, Behavior-Driven Development não descreve em detalhes uma estrutura de papéis e responsabilidades, haja vista que o método propõe uma evolução para o TDD, o qual é centrado nos esforços de desenvolvimento, não tratando em seu escopo outros aspectos da Engenharia de Software, tal como a gestão do projeto (BECK, 2003;

NORTH, 2006). Os papéis mencionados na literatura sobre BDD são os mesmos considerados em TDD, e esses estão igualmente implícitos nas atividades do TDD descritas na literatura sobre a metodologia, tais como: Analista (*Analyst*), Desenvolvedor (*Developer*) e Usuário (*User*) (BECK, 2003).

3.11.2. Práticas e Execução do Processo

Assim como em TDD, Behavior-Driven Development sugere o uso de ferramentas para suportar o processo de BDD (HELLENSOY & WYNNE, 2012). O princípio de uma ferramenta de suporte ao BDD consiste em um framework de teste, assim como em TDD. Entretanto, enquanto ferramentas de TDD tendem a ser de formato geral para especificação dos testes, ferramentas de BDD estão alinhadas as definições da linguagem ubíqua proposta pela metodologia (HELLENSOY & WYNNE, 2012). Nesse sentido, o processo geral do uso de uma ferramenta de suporte ao BDD considera:

- A ferramenta lê a especificação em linguagem ubíqua;
- A ferramenta compreende diretamente a linguagem formal – tal como a palavra-chave *Given* (Dado que). Baseado nisso, a ferramenta quebra cada cenário em cláusulas;
- Cada cláusula individual em um cenário é transformado em um parâmetro para testar a estória do usuário. Nessa parte, é requerida a intevervenção do desenvolvedor;
- O framework executa os testes para cada cenário, com os respectivos parâmetros.

Uma série de frameworks foram estabelecidos para o uso de BDD, tais como JBehave (para desenvolvimento em Java) e RBehave (para desenvolvimento em Ruby), onde a operação é baseada no modelo de linguagem sugerido para descrever as estórias de usuário (NORTH, 2007).

Aplicação prática de ferramentas de suporte a BDD

Existem diversas ferramentas de suporte a BDD, bem como exemplos clássicos para apresentação da aplicação prática dos conceitos propostos pela metodologia (HELLENSOY & WYNNE, 2012). Possivelmente o framework mais difundido é o JBehave[15], o qual foi desenvolvido por Dan North, autor da proposta de BDD (NORTH, 2007). Para exemplificar a aplicação de BDD com o JBehave, será descrito um caso prático da sua execução, dividindo-a em cinco etapas: 1) Escrever a estória; 2) Mapear os passos em Java; 3) Configurar as estórias; 4) Executar as estórias; 5) Visualizar os relatórios. A título de didática, estória será traduzida para o português, entretanto, as palavras-chave da linguagem ubíqua de BDD serão mantidas em inglês.

1ª Etapa: Escrever a Estória

Cenário: Um vendedor é alertado quanto a um status
Scenario: A trader is alerted of a status
Dado que o estoque e o limite de 15.0
Given a stock and threshold of 15.0
Quando o estoque atingir 5.0
When stock is traded at 5.0
Então o alerta de status deve estar setado como OFF
Then the alert status should be OFF

[15] JBehave: www.jbehave.org.

Quando o estoque atingir 16.0
When stock is traded at 16.0
Então o alerta de status deve estar setado como ON
Then the alert status should be ON

2ª Etapa: Mapear os passos em Java

```
public class TraderSteps {
  private TraddingService service;
  private Stock stock;

  @Given("a stock and a threshold of $threshold")
  public void aStock(double threshold) {
     stock = service.newStock("STK", threshold);
  }

  @When("the stock is traded at price $price")
  public void theStockIsTraded(double price) {
     stock = tradeAt(price)
  }

  @Then("the alert status is $status")
  public void theAlertStatusIs(String status) {
     assertThat(stock.getStatus().name(), equalTo(status));
  }
}
```

3ª Etapa: Configurar as Estórias

```
public class TraderStories extends JUnitStories {

  public Configuration configuration() {
    return new MostUsefulConfiguration()
       .useStoryLoader(new LoadFromClassPath(this.getClass()))
       .useStoryReporterBuilder(new StoryReporterBuilder()
          .withCodeLocation(codeLocationFromClass(this.getClass()))
          .withFormats(CONSOLE, TXT, HTML, XML));
  }

  public List<CadidateSteps> candidateSteps() {
    return new InstanceStepsFactory(configuration(),
       new TraderSteps(new TradingService())).createCandidateSteps();
  }

  protected List<String> storyPaths() {
    return new StoryFinder().findPaths(codeLocationFromClass(this.getClass()),"**/*.story");
  }

}
```

4ª e 5ª Etapas: Executar as estórias e Visualizar os relatórios

Considerando-se a execução das três primeiras etapas – escrita das estórias, mapeamento dessas em Java pelo JBehave e configuração da execução das estórias – as mesmas são executadas no ambiente de desenvolvimento escolhido – o qual pode ser o Eclipse, Netbeans, JUnit, Maven, entre outros – e então visualizar a saída do relatório em HTML, o qual deve conter o formato a seguir.

Scenario: A trader is alerted of a status

Given a stock and threshold of 15.0
When stock is traded at 5.0

```
Then the alert status should be OFF
When stock is traded at 16.0
Then the alert status should be ON
```

Estando a saída do relatório de acordo com a descrição do comportamento escrito em linguagem ubíqua, pode-se considerar o dado comportamento como implementado e funcional (NORTH, 2006).

3.11.3. Artefatos

Assim como não aborda explicitamente a atribuição de papéis para a adoção da metodologia, a literatura sobre BDD não menciona a concepção e manutenção de artefatos específicos particulares às práticas de Behavior-Driven Development. Entretanto, conforme apresentado nas seções anteriores, BDD tem em seu princípio básico a especificação das Estórias de Usuários sob a perspectiva de quais comportamentos essas devem possuir, de forma a melhor definir a concepção dos testes de unidade (NORTH, 2006; AGILE ALLIANCE, 2013).

Ainda que BDD não possua um requisito formal sobre como essas Estórias de Usuário devem ser escritas, a metodologia sugere que cada time usando BDD estabeleça uma forma simples e padronizada para escrever suas Estórias de Usuário, considerando a estrutura mencionada nas seções anteriores – *Given, When, Then* (do português, Dado que, Quando, Então). North (2007) sugeriu um modelo padrão para um formato textual de escrita de comportamentos, conforme exemplo a seguir.

```
Estória: Retorno ao estoque

De forma a manter o controle de estoque
Como proprietário de loja
Eu desejo adicionar itens de volta ao estoque quando eles são devolvidos

Cenário 1: Itens reembolsados devem retornar ao estoque
Dado que um cliente previamente comprou um casaco preto a mim
E eu atualmente tenho três casacos pretos restantes no estoque
Quando ele devolver o casaco para um reembolso
Então eu preciso ter quatro casacos pretos no estoque

Cenário 2: Substituição de itens devem ser retornadas ao estoque
Dado que um cliente compre uma camisa azul
E eu tenha duas camisas azuis no estoque
E três camisas pretas no estoque
Quando ele devolver a camisa azul para substituir pela preta
Então eu preciso ter três camisas azuis no estoque
E duas camisas pretas no estoque
```

Os cenários são descritos de forma declarativa em vez de forma imperativa, utilizando uma linguagem de negócio, sem nenhuma referência a aspectos técnicos ou de interface (NORTH, 2007). Esse formato de escrita, inclusive, possui uma sintaxe semelhante a usada por ferramentas de suporte ao desenvolvimento orientado a comportamento (HELLENSOY & WINNE, 2012).

3.11.4. Considerações Finais

A abordagem proposta por Behavior-Driven Development caracteriza-se como uma evolução ao Test-Driven Development, haja visto que TDD não especifica como, onde e quando os testes de unidade devem ser escritos (NORTH, 2006). Nesse sentido, North (2006) identificou a necessidade de se estabelecer um método para não só cobrir essa lacuna, como também, identificar pontos de caracterização das estórias de usuário, o que foi chamado por ele de comportamento (NORTH, 2007).

Compreende-se que BDD é uma combinação de diversas abordagens, tais como linguagem ubíqua e Test-Driven Development (SOLIS & WANG, 2012). Ainda que se trate de uma metodologia ágil de desenvolvimento de software, BDD não cobre todo o ciclo de vida de um projeto de software, contemplando apenas os aspectos relacionados aos esforços de desenvolvimento, conforme afirmam Solis e Wang (2012, p. 5, Tradução do Autor):

> *"Também identificamos que BDD é voltado principalmente para a fase de implementação de um projeto de software e provê suporte limitado para a fase de análise, e nenhum para a fase de planejamento."*

Outro ponto exposto por BDD está relacionada à relevância tanto da escrita dos testes de unidade como dos testes de aceitação (HELLENSOY & WYNNE, 2012). Enquanto os testes de unidade estão guiados a direcionar os desenvolvedores a verificar o design do software, os testes de aceitação está mais relacionado aos objetivos de negócio. Conforme exposto por Hellensoy e Wynne (2012), os testes de unidade garantem que "a coisa seja construída da melhor forma" enquanto os testes de aceitação garantem que "seja construída a coisa certa".

3.12. Outros Conceitos de Desenvolvimento Ágil de Software

Esta pesquisa avaliou até aqui as principais metodologias ágeis de desenvolvimento de software. Objetivou-se dar o enfoque nos métodos originados do ideário apresentado pelo Manifesto Ágil. As metodologias foram analisadas sob um quadro geral e equivalente, de forma a analisar os respectivos pontos em cada uma delas (Visão Geral do Processo, Papéis, Práticas e Execução do Processo e Artefatos). Contudo, de forma a embasar os conceitos propostos pelo modelo objeto dessa pesquisa, se faz necessário explorar algumas ideias adicionais relacionadas às práticas do desenvolvimento ágil de software.

3.12.1. Agile Modeling

Agile Modeling (do português Modelagem Ágil), consiste em uma abordagem para execução de atividades de modelagem, tendo em seu foco principal o estabelecimento de princípios e práticas pré-estabelecidos (AMBLER, 2002). A ideia central de Agile Modeling é encorajar os desenvolvedores a produzir modelos suficientes para suportar a compreensão de problemas de design e propósitos de documentação, porém, mantendo a quantidade de modelos produzidos a mais baixa possível (ABRAHAMSSON ET AL., 2002).

Ainda que não tenha como propósito principal ser um conjunto de práticas e princípios, o foco de Agile Modeling é garantir que as atividades de Análise e Design de um projeto de software sejam garantidas – haja vista que algumas metodologias ágeis não preveem essas atividades em seus processos (ABRAHAMSSON ET AL., 2002) – e executadas de uma

forma alinhada aos princípios e valores do desenvolvimento ágil de software. Assim, Agile Modeling divide sua abordagem em: Valores, Princípios e Práticas.

Os valores de Agile Modeling basicamente são os mesmos propostos por Extreme Programming: Comunicação, Simplicidade, Feedback e Coragem, além incluir o valor de Humildade, o qual roga que as pessoas do time possuem diferentes especialidades e que essas devem ser encorajadas a cooperarem (AMBLER, 2002). Em se tratando dos princípios, Agile Modeling aborda princípios como: software é o objetivo principal; assumir simplicidade; abraçar mudanças; mudanças incrementais; modelo com um propósito; múltiplos modelos; trabalho de qualidade; maximizar envolvimento dos stakeholders.

Agile Modeling também aborda princípios complementares: o conteúdo é mais importante que a representação gráfica de um modelo; qualquer um pode aprender com todo mundo; conheça seus modelos; adaptação local de modelos; comunicação aberta e honesta; trabalhar com os instintos das pessoas; e se beneficiar dos instintos das pessoas (AMBLER, 2002). Conforme apresentado anteriormente, o propósito de Agile Modeling é prover um método ágil para os esforços relacionados à Análise e Design, o que incorre diretamente a atividades de Modelagem e Documentação (AMBLER, 2004). Nesse sentido, Ambler (2002, 2004) descreve práticas específicas tanto para esses esforços:

- **Modelagem**

    1. ***Just Barely Good Enough (JBGE) Artifacts* (do português Artefatos Apenas Suficientemente Bons):** Um modelo ou documento deve ser suficiente para lidar com uma situação;
    2. **Visualizar a Arquitetura:** No começo de um projeto ágil, um alto nível de modelagem da arquitetura é feito para identificar a estratégia técnica;
    3. **Riscos:** A modelagem antecipada é usada para identificar e reduzir riscos;
    4. **Múltiplos modelos:** Diversos modelos podem ser utilizados, desde que tragam efetividade ao desenvolvimento;
    5. **Participação ativa dos stakeholders:** É fundamental envolver os stakeholders para compartilhamento de informações e decisões de projeto;
    6. **Requisitos priorizados:** Os requisitos são implementados em ordem de prioridade;
    7. **Modelagem Iterativa:** No início de cada iteração, um pequeno trabalho de modelagem é realizado;
    8. **Tempestade de Modelagem (*Model Storming*):** Durante uma iteração, uma seção de *brainstorming* pode ser realizada, de forma a refletir sobre detalhes de design e arquitetura.

- **Documentação**

    1. **Documentar Continuamente:** A documentação é produzida durante o ciclo de vida de desenvolvimento, em paralelo com as demais atividades;
    2. **Documentação Tardia:** A documentação é produzida mais tardiamente possível, evitando especulação e estabilização de informação;
    3. **Especificações Executáveis:** Requisitos são especificados em forma de testes executáveis, com práticas como Test-Driven Development;
    4. **Fonte Única de Informação:** As informações do projeto são armazenadas em um único local, evitando confusão quanto à versão e a validade da informação.

Para esforços de modelagem, Agile Modeling propõe que sejam usadas o máximo de notações ou abordagens necessárias para se entregar uma ideia de design (AMBLER, 2002). É fato, porém, que Agile Modeling reconhece que UML (*Unified Modeling Language*, do português Linguagem Unificada de Modelagem) seja o padrão da indústria de software e a principal abordagem para modelagem (AMBLER, 2002). Agile Modeling igualmente roga que não seja restrita a utilização de ferramentas para o exercício das práticas propostas por Agile Modeling, podendo até mesmo ser adotada a modelagem manual de artefatos, conforme ilustração a seguir.

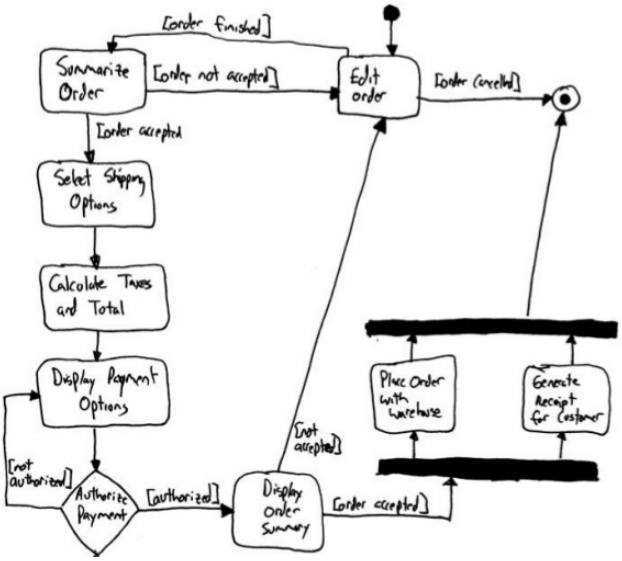


Figura 21: Exemplo de Diagrama de Atividades em UML, modelado manualmente (BECK, 2002).

Para Agile Modeling, ainda que UML não seja suficiente para representar todas as possíveis necessidades de modelagem, é o padrão referencial de notação para modelos de design e arquitetura (AMBLER, 2002).

3.13. Considerações finais

Conforme observado por Abrahamsson et al. (2002), estudos mostram que abordagens tradicionais orientadas a planos para desenvolvimento de software não são tão utilizadas na prática. A literatura mostra que o excesso de mecanismos, práticas imperativas e prescritivas tendem a reprimir a adoção de processos. As metodologias ágeis de desenvolvimento de software buscaram resolver esse gargalo, com uma mudança no paradigma do processo de um projeto de software. Os princípios abordados no Manifesto Ágil direcionaram os esforços na definição de diversas de metodologias ágeis, conforme apresentadas nesse capítulo.

Identifica-se, porém, que as metodologias ágeis propostas na literatura buscaram sempre resolver uma lacuna específica dos esforços de desenvolvimento, o que por consequência, criou métodos focados em partes específicas do ciclo de vida de um projeto de software, em vez de cobrir todo o ciclo. Ainda assim, a adoção de práticas ágeis notoriamente viabilizam benefícios e ganhos aos times e projetos de software, tais como: desenvolvimento iterativo; entregas constantes; receptividade às mudanças; autonomia dos times de desenvolvimento; multidisciplinaridade do time; entre outros.

4. ANÁLISE E AVALIAÇÃO COMPARATIVA ENTRE METODOLOGIAS ÁGEIS DE SOFTWARE

Neste capítulo é apresentada uma análise comparativa entre as metodologias ágeis de software abordadas nessa pesquisa. São abordados aspectos críticos e controversos das metodologias ágeis, bem como é realizada uma análise comparativa das metodologias.

4.1. Introdução

A comparação entre metodologias tende a se basear em experiências subjetivas de praticantes, bem como na compreensão dos autores (SONG & OSTERWEIL, 1991). Conforme exposto por Song e Osterweil (1992), existem duas abordagens específicas para realizar comparações entre metodologias: abordagem informal e a abordagem quasinformal. A abordagem informal baseia-se na percepção de um agente analítico, como um autor específico ou um grupo de autores realizadores da análise. A abordagem quasinformal busca cobrir as limitações da abordagem informal, por meio de cinco meios específicos:

1. Descrever um método idealizado e avaliá-lo ante outros métodos;
2. Distinguir um conjunto de aspectos gerais e compará-los entre os métodos;
3. Formular uma hipótese inicial quanto aos requisitos dos métodos e derivar um modelo de trabalho a partir de evidências empíricas em diversos métodos;
4. Definir uma metalinguagem como um veículo de comunicação e quadro de referência contra o qual serão descritos os diversos métodos.

Song e Osterweil (1992) expõem que a segunda e quarta abordagens são as mais próximas ao método científico clássico adotado para comparação de metodologias. Com base nisso, essa pesquisa irá adotar a segunda prática de Song e Osterweil (1992) para realizar a análise comparativa entre as metodologias ágeis de desenvolvimento de software exploradas nesse trabalho. Essa comparação será baseada na compreensão dos conceitos identificados na revisão da literatura, bem como na avaliação de aspectos críticos e controversos de cada uma das metodologias e por fim, através da realização de uma pesquisa conduzida entre desenvolvedores para identificar padrões de adoções de práticas e metodologias ágeis de desenvolvimento de software. Os resultados e conclusões dessas frentes de avaliação serão utilizados como motivadores para a proposta de um framework de processo de software baseado em metodologias ágeis.

4.2. Aspectos Críticos e Controversos de Metodologias Ágeis de Software na Literatura

Ainda que metodologias ágeis de desenvolvimento de software possuam notáveis benefícios e ganhos ante o paradigma prescritivo de processos, é fato, porém, que tais metodologias são costumeiramente criticadas na literatura. Conforme descreve Turk et al. (2000), tanto os autores quanto os críticos de metodologias ágeis habitualmente enfatizam que esses processos focam prioritariamente na codificação, sendo outras atividades importantes do ciclo de vida de um processo de software, deixadas em segundo plano.

"Proponentes costumeiramente argumentam que o código é o único entregável que importa, e marginalizam o papel de análise e

> *design de modelos e a documentação na criação e evolução. Os críticos de processos ágeis apontam quanto a ênfase na codificação apenas direciona para a perda de memória corporativa uma vez que há uma ênfase pequena em produzir boa documentação e modelos para suportar a criação e a evolução de sistemas grandes e complexos." (TURK ET AL., 2000, p. 1, Tradução do Autor).*

A pesquisa realizada por Turk et al. (2000) objetivou avaliar as limitações das metodologias ágeis com base na revisão da literatura especializada, com o propósito de identificar aspectos críticos apontados tanto pelos próprios autores das respectivas metodologias, como por meio de avaliação de resultados de estudos empíricos. A primeira revelação do estudo é que efetivamente nenhuma metodologia ágil de desenvolvimento de software pode ser vista como "bala de prata", ou seja, pode ser utilizada para todo e qualquer cenário de projeto. Cada metodologia, ou mesmo um conjunto delas, pode ser aplicado a diferentes cenários de projetos. A mesma pesquisa revelou um conjunto de seis aspectos comumente tratados como limitações nas atuais metodologias vigentes.

1. **Suporte limitado para ambiente de desenvolvimento distribuído:** A ênfase em colocação entre desenvolvedores e clientes, conforme proposto como prática de todas as metodologias ágeis citadas nesse trabalho, apresenta-se como um dificultador se considerado um ambiente onde desenvolvedores não estejam fisicamente no mesmo local, ainda que uso de tecnologias como videoconferência sejam propostas, não sumprem totalmente a necessidade de compartilhar o trabalho sendo realizado;
2. **Suporte limitado para subcontratação:** Na realização de *outsourcing* (terceirização) de desenvolvimento, a relação entre desenvolvedores e contratante é regida por um contrato formal. Esse mesmo arranjo tende a estabelecer marcos, etapas e entregáveis formais, o que torna o processo preditivo, haja vista que determinados fluxos de trabalho precisam ser seguidos;
3. **Suporte limitado para construir artefatos reutilizáveis:** Processos ágeis tais como Extreme Programming objetivam a produção de software para a solução de um problema específico, consequentemente não produzindo soluções generalistas. Nesse sentido, o reuso é favorecido em projetos onde o próprio reuso é um dos objetivos, considerando que as soluções devem considerar o eventual reaproveitamento;
4. **Suporte limitado para desenvolvimento envolvendo grandes times:** Os processos ágeis encorajam práticas que exigem a proximidade entre os membros do time. No caso de grandes equipes, as linhas de comunicação podem ser difíceis de serem mantidas, tais como discussões individuais com membros do time, haja vista a característica de grandeza do próprio time;
5. **Suporte limitado para desenvolvimento de software crítico para segurança:** Os projetos críticos para segurança consideram que uma falha no sistema pode resultar prejuízo para pessoas. Os mecanismos de controle de qualidade de metodologias ágeis (tais como Programação em Pares ou Revisões Informais) não se mostraram suficientemente eficientes para tratar esse tema.

Partindo das limitações gerais colocadas por Turk et al. (2000) a cerca das metodologias ágeis, esse trabalho identificou as principais limitações encontradas na literatura para cada uma das metodologias que foram abordadas nesse trabalho. Essas limitações tratam principalmente citações diretas dos autores. Assim, foram compiladas as principais críticas e aspectos controversos identificados na literatura quanto às metodologias abordadas nesse trabalho, sendo essas apresentadas a seguir.

Scrum

A principal crítica quanto ao Scrum está relacionada à necessidade da utilização em conjunto com outros métodos, haja vista que o Scrum não cobre todo o ciclo de vida de desenvolvimento do produto, estando focado principalmente na gestão do projeto (SOMMERVILLE, 2009; SCHWABER & SUTHERLAND, 2013). Em pesquisa conduzida por Akif e Majeed (2012), uma série de críticas foram identificadas, tanto no resultado da aplicação de um questionário como por meio de sessões de entrevistas. As críticas relacionam-se desde a implementação do Scrum até os seus processos de gestão, desenvolvimento e entrega. Os principais pontos abordados na pesquisa são dispostos a seguir:

- **Integração de Módulos:** Com entregas constantes, por vezes ocorre de não ser possível executar todos os testes necessários para garantir que a integração entre os módulos está funcionando adequadamente;
- **Qualidade do Código:** Com a obrigação de apresentar resultados em períodos curtos, eventualmente os desenvolvedores entregam componentes com baixa qualidade técnica de implementação;
- **Duração dos Sprints:** A pesquisa identificou que a duração dos sprints afeta diretamente na quantidade de problemas do produto desenvolvido com Scrum. Sprints de uma ou três semanas apresentaram um volume maior de *bugs*, enquanto sprints de duas ou quatro semanas apresentaram menos ocorrências de *bugs*;
- **Falta de Práticas Técnicas:** Apesar de possuir uma boa cobertura para gestão de projetos, o Srum não aborda nenhuma prática especificamente relacionada aos aspectos técnicos de um projeto de software;
- **Múltiplos Times:** Identificou-se que trabalhar com Scrum com múltiplos projetos pode ser uma tarefa árdua, haja vista que a metodologia não prevê esse tipo de contexto;
- **Métricas:** O Scrum usa Burndown Charts para acompanhamento de métricas, porém, a pesquisa revelou que esse indicador é pouco utilizado e não cobre plenamente o projeto;
- **Documentação:** Assim como boa parte dos métodos ágeis, Scrum não prevê a produção de documentação, o que na prática torna-se um problema principalmente no que diz respeito a gestão do conhecimento do projeto. Na pesquisa realizada por Akif e Majeed (2012), idenficaram-se casos onde requisitos eram detalhados por e-mail e por vezes não constavam no Product Backlog do projeto;
- **Demasiadamente Idealista:** Scrum assume que os times são auto gerenciáveis e que o papel do Scrum Master é de apenas remover os empecilhos que se apresentam no caminho. Contudo, na prática, identificou-se que esse quesito é pouco aplicável, principalmente pelo fato de que majoritariamente, as organizações operam em uma estrutura hierárquica. Nesse sentido, a ausência de um líder pode enfraquecer o grupo;
- **Excesso de Cerimônias/Encontros:** A realização excessiva de reuniões foi apontada como um empecilho para a concentração e efetividade dos times. Como o tempo de realização de reuniões é consumido do sprint, assume-se que o tempo útil de desenvolvimento é consequentemente reduzido.

A pesquisa conduzida por Akif e Majeed (2012), além de apresentar os problemas e críticas quanto as metologias ágeis, também propôs soluções para estas. Contudo, essas soluções basicamente se basearam em duas grandes frentes: a primeira está relacionada à necessidade de treinamento e capacitação de todo o time Scrum, haja vista que como toda metodologia ágil, boa parte da sua aplicação está relacionada a compreensão e execução prática dos princípios da metodologia (SCHWABER & SUTHERLAND, 2013); a segunda proposta de solução está relacionada à utilização de outra metodologia ágil para complementar as atividades técnicas de um projeto de software, principalmente os esforços relacionados a codificação.

Extreme Programming

Conforme descreve Beck (2000), os limites exatos da aplicação de Extreme Programming ainda não estão claros. Contudo, ainda assim, existem limitadores previamente conhecidos na estrutura da metodologia, os quais não só impedem o processo de funcionar, como também são apresentados como restrições para adoção da metodologia (BECK, 2000).

A primeira restrição e limitação da adoção de Extreme Programming está relacionada a esforços de Modelagem e Documentação. Ainda que a metodologia descreva e cite a relevância dessas duas frentes para um projeto XP, existe a indicação de que não sejam dedicados grandes esforços nessas atividades, devendo-se focar prioritariamente na codificação em detrimento dessas frentes.

> *"Modelagem e documentação são aspectos importantes da XP, assim como eles são importantes aspectos de qualquer outra metodologia de desenvolvimento de software. No entanto, explicitamente XP aconselha a minimizar o esforço que você investe nessas atividades a ser apenas o suficiente." (AMBLER, 2002, p. 183) (Tradução do Autor)*

Outro limitador citado para a adoção do XP está relacionado ao tamanho das equipes. Há uma indicação explícita de que os times possuam um número limitado de membros, sendo inclusive pontualmente citado um limitador de membros entre 2 a 10 programadores, conforme descrito a seguir.

> *"XP foi projetado para trabalhar com projetos que podem ser construídos por equipes de 2 a 10 programadores, que não estão fortemente limitados pelo ambiente computacional existente, e onde um trabalho razoável de execução de testes pode ser feito em uma fração de um dia." (BECK, 2000, p. 8) (Tradução do Autor)*

Entretanto, a mais significativa das restrições do uso de Extreme Programming está relacionada ao fato de que a metodologia foi concebida com a clara vocação para atender as atividades e esforços relacionados a codificação.

> *"Provavelmente, os métodos ágeis mais utilizados são Scrum e Extreme Programming, ou XP. Eles podem e muitas vezes são usados em conjunto, uma vez que o Scrum é focado em técnicas de gerenciamento de projetos e o XP é mais focado no trabalho de desenvolvimento real." (SHRIVASTAVA ET AL., 2010, p. 3, Tradução do Autor)*

Durante a realização desse trabalho, porém, foram identificadas algumas contradições quanto aos princípios de Extreme Programming ante as suas próprias indicações de adoção prática. Conforme descrito por Beck (2000), *"o XP é uma forma leve, eficiente, de baixo risco, flexível, previsível, científica e divertida de se produzir software*" (BECK, 2000, p. 7). A contradição reside no fato que por definição, metodologias ágeis são adaptativas e flexíveis (SOMMERVILLE, 2009), entrentanto, em sua definição sobre Extreme Programming, Beck (2000) afirma:

> *"XP é uma disciplina de desenvolvimento de software. É uma disciplina, porque há certas coisas que você tem que fazer para executar o XP. Você não pode escolher se vai ou não escrever testes. Se não o fizer, você não é extremo: fim da discussão." (BECK, 2000, p. 8, Tradução do Autor).*

Conforme definição anterior, entende-se que existe no XP o contexto de prescrição em parte do seu processo, o que vai de encontro com os princípios propostos pelas metodologias ágeis. Ademais, é possível compreender que cada cenário de projeto pode demandar a aplicação de uma abordagem adaptada, característica comum de metodologias ágeis, na qual o XP se qualifica.

Crystal

Conforme analisado por Turk et al. (2000), ainda que a metodologia proposta por Crystal tenha o propósito de cobrir toda e qualquer escala de projeto, à medida em que o projeto torna-se mais complexo, o nível de práticas do Crystal se amplia. Em Crystal Blue, por exemplo, haverá uma boa amplitude no processo para cobrir todos os aspectos de um projeto amplo e complexo, porém, ele será certamente menos ágil do que Crystal Clear (TURK ET AL., 2000).

Percebe-se que algumas características do Crystal elevam sua complexidade e tornam o processo de difícil aplicabilidade, como o próprio Cockburn (2004) versa a cerca do modelo baseado em ciclos de tempo. Igualmente a esse aspecto, a amplitude de combinações de prioridades, propriedades, princípios, estratégias e técnicas, remetem à necessidade de adequação de um processo Crystal para cada projeto específico, o que pode tornar-se custoso e principalmente, levar tempo – tanto no conhecimento do ecosistema Crystal como no nível de experiência em diferentes cenários de projetos, para então se definir a configuração adequada. As técnicas e projetos exemplo apresentados pelo Crystal facilitam nesse trabalho, porém, não cobrem uma ampla variedade de cenários distintos.

Por fim, os diferentes produtos de trabalho propostos por Crystal remetem a uma característica comum em processos preditivos, os quais possuem ampla demanda de artefatos que por vezes têm seu valor agregado questionado nos projetos (POLLICE, 2001). Ratifica-se que mesmo a característica de facultabilidade na adoção de tais produtos é igualmente encontrada em modelos preditivos de processos de software (KRUTCHEN, 1999).

Dynamic Systems Development Method

Apesar de Dynamic Systems Development Method ser considerado um processo ágil, por considerar em sua aplicação prática princípios básicos de metodologias ágeis, tais

como entregas frequentes, desenvolvimento iterativo e enfoque no cliente, algumas das suas práticas remetem a características igualmente encontradas em modelos prescritivos de processos, principalmente considerando a quantidade de artefatos, atividades, práticas e premissas entre as atividades do processo.

Conforme define Abrahamsson et al. (2002), o DSDM sugere que em algum momento do desenvolvimento seja feito o congelamento dos requisitos, o que de certa forma diverge do princípio das metodologias ágeis de abraçar as mudanças (AMBLER, 2002; LEVINE, 2005). Outro aspecto divergente dos princípios das metodologias ágeis encontrado em DSDM refere-se ao fato de que das cinco fases previstas no ciclo de vida do processo, as duas primeiras são dependentes uma da outra, não havendo entregas de produto durante essas fases, ficando restrita ao Estudo de Viabilidade e Estudo de Negócios (STAPLETON, 1997).

Outro aspecto restritivo quanto a adoção de DSDM está relacionado à rigidez quanto a necessidade de aderência plena aos seus princípios: Foco nas necessidades do negócio; Entregar no prazo; Colaboração entre o time; Nunca comprometer a qualidade; Desenvolver iterativamente; Comunicar e esclarecer os envolvidos continuamente; Demonstrar controle do projeto (STAPLETON, 1997). Em função dessa rigidez, DSDM tende a se tornar uma metodologia complexa de se aderir ante outras metodologias ágeis de desenvolvimento de software.

Adaptive Software Development

Conforme Highsmith (1997), o processo descrito por Adaptive Software Development é uma proposta sutilmente evoluída de processos baseado no ciclo de vida espiral. Outro aspecto controverso quanto ao modelo proposto por ASD diz respeito à ausência de aspectos estruturais importantes em qualquer processo. Em ASD, há a ausência de menção a princípios, papéis e artefatos específicos, sugerindo informalmente a adoção de alguns desses previstos em outra metodologia ágil, no caso, a Dynamic Systems Development Method (STAPLETON, 1997), como o uso de sessões JAD (*Joint Application Development*, do português Desenvolvimento Conjunto de Aplicações) (HIGHSMITH, 2000). Em suma, ASD propõe uma abordagem mais direcionada a conceitos do que a práticas.

Feature-Driven Development

Feature-Driven Development tem o enfoque dirigido apenas a atividades de Design e Implementação, o que por sua vez requer obrigatoriamente a utilização de outros métodos para complementar o ciclo de vida de um projeto de software (PALMER & FELSING, 2002). Isso pode levar a uma situação onde o software torna-se mais difícil de ser modificado ao longo do tempo, requerendo eventuais retrabalhos. Por fim, existem desvantagens associadas a característica de Propriedade de Classe (*Class Ownership*) proposta por FDD. Uma vez que há a relação de propriedade de uma classe para com um desenvolvedor específico, pode haver problemas relacionados a uma grande espera para edição de uma classe, além do risco inerente de desconhecimento da estrutura da classe por parte de outros desenvolvedores, haja vista que até então apenas um desenvolvedor matinha a respectiva classe (PALMER & FELSING, 2002).

Test-Driven Development

A primeira restrição relacionada à Test-Driven Development está relacionada com o fato de que trata-se de uma metodologia especificamente dirigida a design e implementação (BECK, 2002). Originalmente proposta como uma das práticas do Extreme Programming, TDD derivou para uma visão mais ampla, cobrindo efetivamente todo o processo de implementação de um projeto de software. Contudo, não prevê em seu escopo atividades, conceitos ou práticas relacionadas a outras disciplinas relacionadas ao ciclo de vida de um projeto de software. Outra restrição marcante em TDD está relacionada a como, quando e onde devem ser concebidos os testes de unidade, o que torna a implementação de testes de unidade uma atividade subjetiva e dependente do entendimento do desenvolvedor quanto aos requisitos. Outras restrições de Test-Driven Development são: TDD não cobre situações onde testes funcionais completos são requeridos, tais como em interfaces de usuário, programas que trabalham com bancos de dados e casos onde há dependência de configurações de rede; com um amplo número de testes sendo bem sucedidos ao longo do desenvolvimento, pode haver o negligenciamento de outras atividades de testes, tais como testes de integração e testes de conformidade.

Behavior-Driven Development

Por se tratar de uma evolução de Test-Driven Development, Behavior-Driven Development foi concebido com objetivo de responder a questão sobre quando, onde e como devem ser escritos os testes de unidade ao longo do processo de desenvolvimento. Através da abordagem baseada em comportamentos, BDD resolve essa deficiência de TDD. Entretanto, as demais restrições existentes em TDD são equivalentes em BDD: foco exclusivo em design e implementação; cobertura quanto a testes funcionais completos; excesso de segurança com testes bem sucedidos, negligenciando testes de integração e conformidade.

4.2.1. Considerações Gerais sobre as Limitações de Metodologias Ágeis de Software

Em suma, a literatura prevê diversas abordagens diferentes para desenvolvimento ágil de software e todas possuem suas limitações, principalmente quanto a amplitude de cobertura das atividades previstas no ciclo de vida de desenvolvimento de software (IEEE, 2004). Para Turk et al. (2000), projetos longos e complexos de desenvolvimento de software provavelmente não teriam a indicação de adoção de um método ágil de desenvolvimento, conforme atualmente disponíveis na literatura. Considerando essa perspectiva, um processo de software prático pode ser classificado como um espectro entre processos puramente ágeis e processos preditivos (TURK ET AL., 2000).

4.3. Análise Comparativa de Metodologias Ágeis de Software

Conforme analisado, as metodologias ágeis de desenvolvimento de software convergem em muitas das suas práticas, princípios e abordagens, haja vista que foram construídas com base na mesma perspectiva de valor: os ideários do Manifesto Ágil (BEGEL & NAGAPPAN, 2007). Ainda assim, cada método avaliado nesse trabalho objetiva tratar um ou vários desafios de um projeto de software sob um diferente panorama prático. Nesse sentido, todas metodologias abordadas nesse trabalho tiveram suas características compiladas e revisadas, com base na análise realizada originalmente por Abrahamsson et al. (2002), ainda que não

para todas metodologias analisadas nesse trabalho. Entretanto, por propor uma análise sucinta e objetiva, optou-se por adaptar o referido modelo proposto por Abrahamsson et al. (2002).

Metodologia	**Pontos-Chave**	**Principais Recursos Especiais**	**Pontos Negativos Identificados**
Scrum	- Processo simples; - Times de desenvolvimento auto-organizáveis; - Ciclos de entrega de 30 dias.	- Reforça a mudança de paradigma de "definido e repetível" para "orientado ao cliente".	- Requer outras abordagens para complementar o ciclo de vida do processo de software; - Focado na gestão do projeto de software; - Ausência de práticas e técnicas de software.
Extreme Programming	- Desenvolvimento dirigido ao cliente; - Times pequenos; - Pacotes diários.	- Refatoração; - Desenvolvimento orientado a testes; - Programação em Pares.	- Poucas práticas de gestão estabelecidas; - Negligenciamento a Design e Documentação; - Focada principalmente aos esforços de codificação; - Indicada a times pequenos; - Exigência à aderência nos princípios.
Crystal	- Família de métodos; - Cada método possui os mesmos valores e princípios centrais; - Técnicas, papéis e ferramentas variam.	- O método estabelece princípios; - Possui a característica de ter um método mais indicado baseado no tamanho e criticidade do projeto.	- Algumas características de processos preditivos; - Amplitude de recursos passíveis de adoção pode tornar o processo complexo, por excesso de atividades ou artefatos.
Dynamic Systems Development Method	- Aplicação de controles para Desenvolvimento Rápido de Aplicações (RAD); - Tempo estabelecido de iteração (*timeboxing*); - Autonomia dos times; - Consórcio ativo para manter a metodologia.	- Primeira metodologia realmente ágil de desenvolvimento de software; - Utiliza prototipação; - Diveros papéis de usuário.	- Sugere a estabilização dos requisitos; - Rigidez à aderência aos princípios.
Adaptive Software Development	- Cultura adaptativa; - Colaboração; - Baseada em componentes; - Desenvolvimento iterativo.	- As organizações são vistas como sistemas adaptativos, criando uma ordem a partir de uma rede de indivíduos.	- ASD está mais ligada a conceitos e cultura do que a práticas de software.
Feature-Driven Development	- Processo dividido em cinco passos; - Desenvolvimento baseado em componentes orietados a objetos; - Pequenas iterações.	- Método simplista; - Design e implementação de sistemas por funcionalidade; - Modelagem de objetos.	- Foco exclusivo em Design e Implementação; - Requer outras abordagens para complementar o ciclo de vida do processo de software.
Test-Driven Development	- Redução de defeitos por meio da atividade conjunta de implementação e testes; - Maior entendimento dos requisitos por parte do desenvolvedor; - Maior produtividade nos esforços de codificação.	- Refatoração; - Desenvolvimento orientado a testes.	- Foco exclusivo em Design e Implementação; - Requer outras abordagens para complementar o ciclo de vida do processo de software; - Não estabelece critérios específicos sobre como e quando escrever testes.
Behavior-	- Redução de defeitos por	- Refatoração;	- Foco exclusivo em Design e

Driven Development	meio da atividade conjunta de implementação e testes; - Maior entendimento dos requisitos por parte do desenvolvedor; - Maior produtividade nos esforços de codificação.	- Desenvolvimento orientado a testes; - Definição de critérios de necessidade de testes baseada em comportamentos.	Implementação; - Requer outras abordagens para complementar o ciclo de vida do processo de software.

Tabela 14: Recursos gerais de metodologias ágeis de desenvolvimento de software (Adaptado de ABRAHAMSSON ET AL., 2012).

Complementarmente à comparação dos aspectos gerais das metodologias ágeis apresentada na Tabela 14, foi realizada uma avaliação das mesmas sob a ótica das atividades e fases previstas no ciclo de vida de um projeto de desenvolvimento de software (IEEE, 2004). Cada método foi avaliado quanto: a) cobertura para as atividades tradicionalmente previstas em um ciclo de vida de projeto de software; b) nível de aderência aos princípios ágeis propostos no Manifesto Ágil; c) conclusões apresentadas na Seção 4.2, a cerca dos aspectos de crítica e controvérsias de cada uma das metodologias apresentadas.

SCR	A	-	A	-	-	-	-	A	-	A	/
XPR	-	A	A	PA	A	A	A	A	-	A	-
CRY	-	-	-	A	A	A	A	A	-	A	-
DSD	A	A	A	A	A	A	A	A	A	PA	PA
ASD	-	-	PA	PA	PA	PA	PA	PA	PA	A	-
FDD	-	-	A	A	A	A	A	A	-	A	-
TDD	-	-	-	-	-	A	A	A	A	A	-
BDD	-	-	-	-	-	A	A	A	A	A	-
	Gestão de Projeto	Análise Negócios	Requisitos	Design	Protótipo	Codific.	Teste de Unidade	Teste de Integração	Teste de Aceitação	Aderência Princípios Ágeis	Aplicável a Grandes Times

Legendas: SCR – Scrum; XPR – Extreme Programming; CRY – Crystal; DSD – Dynamic Systems Development Method; ASD – Adaptive Software Development; FDD – Feature-Driven Development; TDD – Test-Driven Development; BDD – Behavior-Driven Development.
Escalas: "A" Aderente; "-" Não Aderente; "PA" Parcialmente Aderente.

Tabela 15: Análise comparativa das metodologias ágeis analisadas nessa pesquisa quanto a aspectos comuns em projetos de desenvolvimento de software (Adaptado de ABRAHAMSSON ET AL., 2012).

Com base na análise individual dos aspectos críticos e controversos de cada uma das metodologias abordadas nesse trabalho conforme apresentado na Seção 4.2, bem como nas avaliações comparativas apresentadas nas tabelas 14 e 15, é possível estabelecer algumas conclusões importantes quanto as características e recursos providos pelas principais metodologias ágeis de desenvolvimento de software. Essas conclusões também baseiam-se na revisão da literatura, a qual considera a aplicação de estudos empíricos dessas metodologias.

Considerando a adaptação da avaliação comparativa de Abrahamsson et al. (2012) apresentadas nas tabelas 14 e 15, é possível concluir que nenhuma das metodologias cobriu plenamente todos os critérios de avaliação. Entretanto, a metodologia Dynamic Systems Development Method foi a única a cobrir todos os critérios, ainda que não plenamente, devido ao seu atendimento parcial ao critério "Aderência aos Princípios Ágeis". Essa atribuição se dá pelo fato de que a amplitude de recursos da metodologia pode ser classificada como um ponto negativo, dada a eventual complexidade em utilizá-los e até mesmo características prescritivas em seu processo, como a sugestão de que os requisitos

devem ser estabilizados em algum momento do projeto, conforme apresentado na Seção 4.2. Essa sugestão contradiz o princípio de "abraçar as mudanças" proposto pelo Manifesto Ágil e adotado por todas as metodologias ágeis de desenvolvimento de software.

A avaliação revelou igualmente uma clara distinção entre dois grupos de metodologias: um grupo mais focado em práticas de gestão de projetos, aplicação de valores e princípios ágeis (Scrum, Crystal, Dynamic Systems Development Method, Adaptive Software Development, Feature-Driven Development); e outro grupo dirigido a práticas e esforços relacionados às atividades de desenvolvimento e codificação (Extreme Programming, Test-Driven Development e Behavior-Driven Development). Essa segregação entre as metodologias valida o primeiro problema de pesquisa abordado por esse trabalho, conforme apresentado na Seção 1.2: "*1) Cada metodologia ágil disponível na literatura objetiva resolver aspectos específicos de iniciativas de desenvolvimento de software*".

Identificou-se que há igualmente o negligenciamento dos esforços de Design e Documentação nas metodologias ágeis de software. A avaliação identificou que as metodologias avaliadas variam entre dois extremos: desde a concepção excessiva de documentação e modelos (Crystal, Dynamic Systems Development Method e Feature-Driven Development) à desconsideração da relevância desses esforços (Scrum, Extreme Programming, Adaptive Software Development, Test-Driven Development, Behavior-Driven Development). Essa conclusão valida o segundo problema de pesquisa abordado por esse trabalho, apresentado na Seção 1.2: "*2) Buscando maior agilidade nos projetos de software, as metodologias ágeis tendem a negligenciar aspectos relevantes em Engenharia de Software, tais como o Design e Documentação*".

Outro ponto identificado na análise e avaliação comparativa das metodologias de desenvolvimento de software abordadas nessa pesquisa, está relacionado à aderência a projetos com grandes times (acima de 40 pessoas). Foi possível identificar na literatura que há menções diretas a não adoção das metodologias em grandes times, ainda que algumas delas peguem ser possível adaptá-las para atender a esse contexto (conforme descrito por Scrum e Dynamic Systems Development Method). O fato, porém, é que essa característica valida o terceiro problema de pesquisa abordado nesse trabalho, conforme apresentado na Seção 1.2: "*3) Metodologias Ágeis habitualmente são compreendidas como aplicáveis a projetos de software de pequena e média escala, sendo inclusive desaconselhadas como referência única de processo em projetos de larga escala, com times acima de 40 pessoas.*"

Contudo, analisando sob a ótica de atendimento aos princípios ágeis propostos pelo Manifesto Ágil, bem como pela amplitude de estabelecimento de práticas claras de agilidade no processo de desenvolvimento, é possível concluir que Extreme Programming destaca-se como sendo a mais aderente e completa do ponto de vista de esforços de desenvolvimento. Esse resultado é uma consequência natural, haja vista que grande parte das metodologias ágeis de desenvolvimento de software estenderam conceitos, práticas e princípios do Extreme Programming (BECK, 2000).

4.4. Considerações Finais

Baseado nos resultados da análise e avaliação comparativa das metodologias ágeis de desenvolvimento de software apresentadas nas seções 4.2 e 4.3 desse trabalho, conclui-se que por mais aderentes que essas estejam aos princípios propostos pelo Manifesto Ágil, há uma

lacuna a cerca da relação do nível de amplitude da metodologia versus o seu nível de complexidade.

Identificou-se que algumas metodologias ágeis, na tentativa de cobrir todas as atividades previstas no ciclo de vida de um projeto de software, tornam-se excessivamente complexas e burocráticas, o que remete a modelos prescritivos de processos de software, os quais o próprio paradigma ágil objetivou resolver. Por outro lado, foi possível identificar que há uma clara vocação nas metodologias ágeis em cobrir principalmente os esforços relacionados à codificação e desenvolvimento. Esse direcionamento, todavia, gera a consequente falta de atendimento a esforços igualmente importantes em projetos de software, tal como esforços relacionados à gestão do próprio projeto.

5. PESQUISA SOBRE A ADOÇÃO DE PRÁTICAS E METODOLOGIAS ÁGEIS

Neste capítulo são apresentados os resultados da pesquisa conduzida neste trabalho, sobre a adoção de práticas e metodologias ágeis de desenvolvimento de software. Os resultados da pesquisa serão utilizados para definição do framework X-PRO (Extreme Software Process), objeto desse trabalho.

5.1. Introdução

Como parte desse trabalho, foi realizada uma revisão da literatura a cerca das principais metodologias ágeis de desenvolvimento de software. Igualmente, conforme apresentado na Seção 4.3, as metodologias foram avaliadas quanto a seus aspectos críticos e controversos, bem como foi realizada uma análise comparativa entre elas, com objetivo de identificar suas características, benesses e vulnerabilidades.

Conforme apresentado no capítulo 1, o objetivo deste trabalho é propor um framework para o desenvolvimento eficiente de software baseado em metodologias ágeis, onde este deve resolver os problemas das atuais abordagens, os quais conforme descritos nesse trabalho consistem em: 1) Cada metodologia ágil disponível na literatura objetiva resolver aspectos específicos de iniciativas de desenvolvimento de software, como por exemplo Test-Driven Development e o XP (Extreme Programming), as quais estão mais focadas aos esforços de codificação do que do projeto de software como um todo (BEGEL & NEGAPPAN, 2007); 2) Buscando maior agilidade nos projetos de software, as metodologias ágeis tendem a negligenciar aspectos relevantes em Engenharia de Software, tais como o Design e Documentação (AKIF & MAJEED, 2012); 3) Metodologias Ágeis habitualmente são compreendidas como aplicáveis a projetos de software de pequena e média escala, sendo inclusive desaconselhadas como referência única de processo em projetos de larga escala, com times acima de 40 pessoas (LINDVALL ET AL., 2002).

É possível afirmar que as conclusões abordadas na Seção 4.3 a partir da revisão da literatura, respaldam a procedência dos problemas de pesquisa abordados por este trabalho. Todavia, parte das deficiências das metodologias ágeis identificadas até o momento são de conhecimento comum, tanto pelos autores quanto por praticantes (BEGEL & NAGAPPAN, 2007). Nesse sentido, como o objetivo geral desse trabalho é propor um framework que resolva os problemas identificados, se fez necessário identificar a percepção de desenvolvedores e membros de times de desenvolvimento, a cerca da adoção e características de processos de desenvolvimento de software e especificamente, o uso de metodologias ágeis. Nesse sentido foi realizada uma pesquisa que objetivou identificar essa percepção por parte de profissionais envolvidos com a prática de desenvolvimento de software.

5.2. Visão Geral da Pesquisa

A pesquisa foi realizada entre o período de 10 de Setembro de 2013 a 22 de Outubro de 2013, teve uma duração total de 43 dias. Para sua condução, foi utilizado um questionário disponibilizado através da web. Um convite foi enviado por e-mail para cerca 800 profissionais diretamente envolvidos com a área de desenvolvimento de software – Engenheiros de Software, Arquitetos, Desenvolvedores, Testadores, Gerentes, entre outros,

em empresas públicas e privadas, em diferentes países. Foram recebidas 130 respostas, que corresponde a uma taxa percentual geral de respostas de 16,25%. Os participantes poderiam se identificar, se desejassem, bem como a organização em que trabalhavam.

Foram dispostas um total de 17 questões, divididas em três seções: 1) Visão geral sobre sua organização; 2) Adoção de práticas e metodologias ágeis na sua organização; 3) Experiência pessoal. As duas primeiras seções objetivaram identificar o perfil das organizações e como elas adotam metodologias ágeis de desenvolvimento, com objetivo de respaldar a relevância do estudo do tema para a conjuntura atual do mercado de software. A terceira seção objetivou identificar qual a visão dos profissionais envolvidos com o processo de desenvolvimento quanto ao que é fundamental em um processo de software, ou seja, suas percepções a cerca de quais características devem ser consideradas por qualquer metodologia ou framework de processo de software. Como referência das atividades padrão que devem constar em um processo de desenvolvimento de software, considerou-se para essa pesquisa a estrutura proposta pelo SWEBOK (*Software Engineering Body of Knowledge*), o qual prevê as seguintes atividades: Modelagem de Negócios, Requisitos, Análise & Design, Implementação, Testes, Implantação, Gerência de Configuração, Gerência de Projeto e Gerência de Ambiente. Os principais resultados dessa pesquisa serão apresentados a seguir, estando a pesquisa completa disponível como anexo deste trabalho.

5.3. Resultados Encontrados

A pesquisa iniciou com a análise geral sobre as organizações do participantes, tendo o percentual de 90% atuantes em fábricas de software, onde desses 40% em fábricas de múltiplos produtos, 20% em fábricas de produto único e 30% de fábricas de terceirização. As organizações dos participantes estão distribuídas globalmente, estando, porém, 70% dos respondentes atuando em empresas com sede no Brasil, 13% no Canadá, 7% nos Estados Unidos, 7% em Portugal e 3% em outros países. Considera-se, com isso, que o cenário avaliado configura uma contexto nacional da indústria de software. Ainda sobre as organizações em que os participantes atuam, é possível identificar que essas são de médio e grande porte, dado que 53% possuem entre 51 e 500 profissionais diretamente envolvidos com esforços de desenvolvimento. Um dado relevante identificado na pesquisa é que 53% das organizações possuem alguma certificação do processo de desenvolvimento de software – sendo 24% CMMI e 29% MPS-BR – ante a 47% que não possuem certificação. No que diz respeito a aspectos técnicos do processo de desenvolvimento, 60% dos participantes responderam que não existe esforços de Análise e Design, sendo nos casos em que são realizados esses esforços, a maioria está relacionada à modelagem de dados (Diagrama de Modelo de Dados). Existe, porém, a adoção em 98% dos participantes de algum padrão arquitetural (*Architectural Pattterns*) e padrão de projeto (*Design Pattern*). Esse dado é relevante no sentido que como 47% das organizações participantes possuem certificações de processo de desenvolvimento, esforços de design são prescritivos e devem ser previstos. Nesse sentido, esse dado pode levar ao entendimento que os esforços de design que existem nessas organizações poderiam estar relacionados mais a atendimento aos requisitos das certificações do que a efetiva aplicação das práticas de design.

Na seção seguinte da pesquisa, foram elaboradas questões relacionadas à adoção de práticas e metodologias ágeis nas organizações dos participantes. Percebeu-se um bom volume de praticantes de alguma metodologia ágil, correspondendo ao percentual de 86%, ainda que desses 49% utilizem o método Scrum de gerenciamento de projetos de software. Apenas 9% dos participantes afirmaram que utilizam um processo baseado no Processo Unificado

(*Unified Process*), o que revela uma mudança quanto ao paradigma de processo de desenvolvimento, anteriormente mais centrado no uso de processos prescritivos (NERUR ET AL., 2005). Quanto ao uso de práticas ágeis, apenas 6% respondeu não utilizar nenhuma prática, mostrando que mesmo aqueles que utilizam outro paradigma de processo de software que não o paradigma ágil, utiliza-se de alguma das suas práticas – 12% adotam times multifuncionais (*Cross-Functional Teams*), 18% desenvolvimento iterativo e incremental e 16% iterações com tempo pré-estabelecido (*Timeboxed Iterations*).

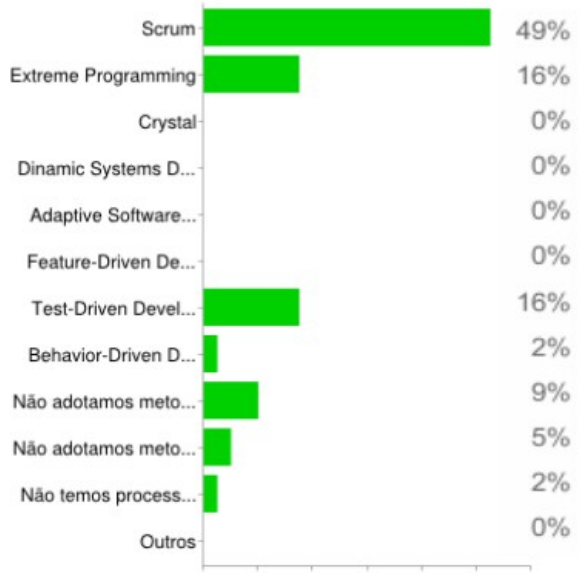


Figura 22: Metodologias ágeis adotadas pelas organizações participantes da pesquisa.

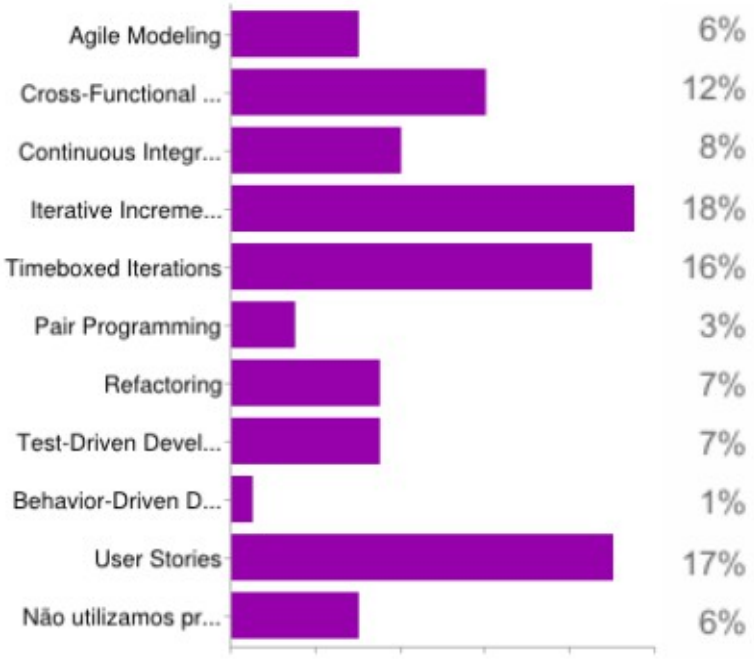


Figura 23: Práticas ágeis adotadas pelas organizações participantes da pesquisa.

Por fim, a pesquisa objetivou identificar a experiência pessoal de cada participante, independentemente da sua realidade atual nas organizações em que atuam. O objetivo dessa seção foi identificar as percepções dos profissionais quanto à aderência desses às tendências atuais, bem como se essas condiziam com uma realidade almejada por eles quanto a

metodologias e práticas ágeis de desenvolvimento de software. Nesse caso, os resultados revelam números interessantes quanto ao alinhamento e concordância dos profissionais envolvidos com esforços de desenvolvimento, ante aos direcionamentos dados pelas suas organizações quanto aos seus processos de software. Quando questionados a cerca de quais atividades consideram fundamentais em qualquer ciclo de vida de projeto de software, 68% responderam que nenhum projeto de software deveria deixar de possuir esforços de Requisitos, Análise e Design, Implementação e Testes. Surpreendentemente, considerou-se que a atividade de Levantamento de Requisitos (19%) seja mais relevante do que Implementação (18%), a qual teve sua relevância equiparada com Análise e Design (18%), procedida pela atividade de Testes (13%). Essa percepção dos profissionais contradiz a tendências duas organizações, as quais conforme apresentado anteriormente, em 60% não possui esforços de Análise e Design e onde esforços de Análise e Design não considerados padronizados e estabelecidos – apenas em 8% destas.

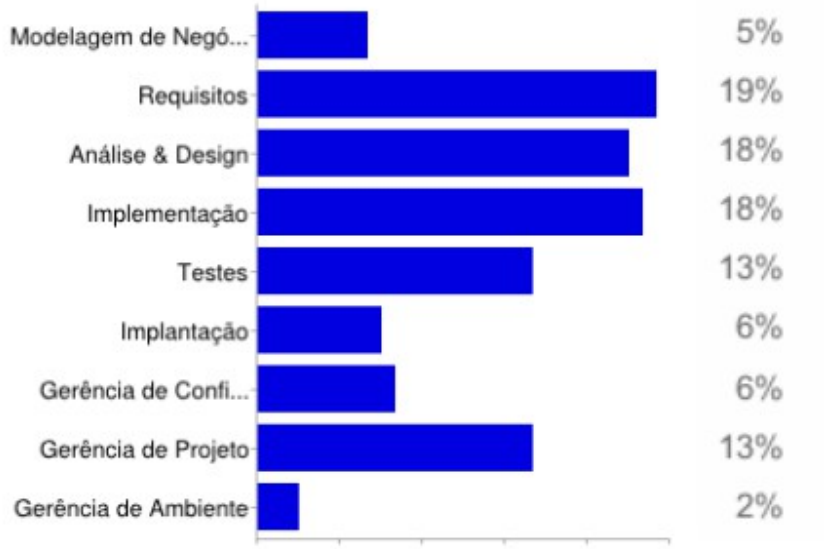


Figura 24: Atividades fundamentais em um ciclo de vida de projeto de software sob a ótica dos participantes da pesquisa.

Outro resultado contraditório quanto às práticas adotadas pelas organizações e aquelas que os profissionais diretamente relacionados com desenvolvimento consideram importantes, diz respeito aos modelos concebidos nas atividades de Análise e Design. Das poucas organizações que efetivamente produzem modelos nas atividades de Design (23% não produzem), identificou-se que 27% desses modelos correspondem a Diagrama de Modelos de Dados (ER), para concepção dos bancos de dados das aplicações. Em contrapartida, os participantes relevaram que consideram importante a produção de outros modelos, principalmente com o objetivo de conhecer e compreender a estrutura da aplicação, característica claramente relacionada à manutenção do produto após a sua construção. Desses, 21% consideram a necessidade de produzir o Diagrama de Classes, 18% o Diagrama de Implantação, 12% o Mockup de Interface e 22% o Diagrama de Modelo de Dados (ER).

Mostrando como as práticas ágeis passaram a ter uma maior relevância e adoção, 58% dos participantes revelaram preferirem produzir e utilizar Estórias de Usuários do que Casos de Uso, para entender o detalhamento dos requisitos do software.

Por fim de 87% dos participantes consideraram que práticas e metodologias ágeis podem sim ser utilizadas em times acima de 40 pessoas.

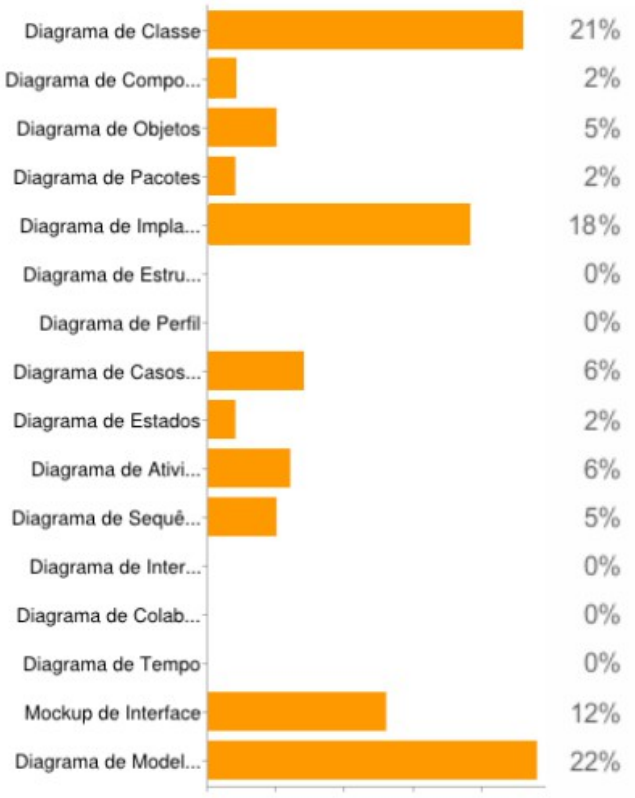


Figura 25: Modelos fundamentais para auxiliar no conhecimento da aplicação sob a ótica dos participantes da pesquisa.

Em suma, os resultados encontrados nessa pesquisa reforçam a relevância e importância de se resolver os problemas de pesquisa apontados por esse trabalho.

5.4. Considerações Finais

A pesquisa conduzida nesse trabalho e apresentada neste capítulo mostra que há um entendimento padrão por parte dos participantes quanto a quais atividades e esforços não podem deixar de existir em qualquer projeto de software. Essa conclusão é importante para direcionar o estabelecimento do framework X-PRO (*Extreme Software Process*), objeto desse trabalho, o qual deve ser aderente aos princípios ágeis de desenvolvimento de software, mas deve igualmente cobrir todas as atividades fundamentais que devem ser consideradas em projetos de software, conforme identificado na pesquisa apresentada nesse capítulo.

6. X-PRO (EXTREME SOFTWARE PROCESS)

Neste capítulo é apresentado o modelo X-PRO (Extreme Software Process), o qual consiste em um framework para desenvolvimento eficiente de software baseado em Metodologias Ágeis. É apresentada a visão geral do modelo, sua arquitetura, princípios, práticas, artefatos e benefícios da sua adoção.

6.1. Introdução

As metodologias ágeis de desenvolvimento de software, conforme apresentado nos capítulos anteriores, em sua maioria buscaram atender objetivos específicos do ciclo de vida de um projeto de software, sendo em muitos casos, excessivamente especializadas. Essa característica não as torna aderentes a todo o ciclo de vida de desenvolvimento (IEEE, 2004). Na busca por resolver essa dificuldade, algumas metodologias ágeis propostas na literatura tornaram-se excessivamente complexas e quase que prescritivas em seu processo, tais como Crystal ou Dynamic Systems Development Method.

Partindo dessa premissa, este trabalho identificou problemas específicos a serem tratados e solucionados com a proposta de framework objeto dessa dissertação. Os problemas, conforme apresentado na Seção 1.2, são os seguintes:

1. Cada metodologia ágil disponível na literatura objetiva resolver aspectos específicos de iniciativas de desenvolvimento de software, como por exemplo Test-Driven Development e o XP (Extreme Programming), as quais estão mais focadas aos esforços de codificação do que do projeto de software como um todo (BEGEL & NEGAPPAN, 2007);
2. Buscando maior agilidade nos projetos de software, as metodologias ágeis tendem a negligenciar aspectos relevantes em Engenharia de Software, tais como o Design e Documentação (AKIF & MAJEED, 2012);
3. Metodologias Ágeis habitualmente são compreendidas como aplicáveis a projetos de software de pequena e média escala, sendo inclusive desaconselhadas como referência única de processo em projetos de larga escala, com times acima de 40 pessoas (LINDVALL ET AL., 2002).

Complementarmente, pode-se considerar o cenário de organizações que por ventura não possuam um processo de software definido. Nesses casos, tanto a adoção de modelos baseados em um ciclo de vida "cascata", como a adoção de princípios ou práticas ágeis tendem a serem realizadas de forma *ad-hoc* (LARMAN ET AL., 2001). Nesse caso, uma proposta de solução para estabelecimento de um processo de software precisa considerar os notórios benefícios trazidos pela adoção de princípios e práticas ágeis, bem como viabilizar um modelo viável, compreensível e principalmente eficiente para organizações ou times de desenvolvimento que não possuam um processo de software estabelecido.

Para tratar os aspectos e problemas relatados nesse trabalho, propõe-se o estabelecimento do X-PRO (Extreme Software Process, do português Processo de Software Extremo). O X-PRO tem como propósito ser um framework eficiente de processo de desenvolvimento de software. Sua definição como framework se dá pelo fato de que o X-PRO define pressupostos, conceitos, valores e práticas, incluindo também orientações sobre como

executá-lo, podendo porém, ser adequado a cada cenário de projeto, não sendo prescritivo nesse sentido.

6.2. Visão Geral do X-PRO

O X-PRO tem como propósito atender a projetos e times de desenvolvimento que busquem um framework de processo ágil que cubra todo o ciclo de vida de um projeto de software. Nesse sentido, X-PRO é definido como um framework eficiente de processo de software. Quanto a ênfase da definição do X-PRO como framework "eficiente", refere-se ao fato de que seu processo busca como premissa fundamental a agilidade. Por definição, eficiência refere-se à relação entre os resultados obtidos ante aos recursos empregados, ou de forma objetiva, alcançar produtividade, atindingo-se o objetivo com o mínimo de recursos necessários[16]. A eficiência é o conceito e objetivo fundamental buscado por X-PRO, de forma que todas suas características e definições buscam como premissa básica a eficiência do time de desenvolvimento e o resultado do projeto. Naturalmente a busca pela eficiência não deve negligenciar a eficácia, ou seja, realizar o escopo conforme seu objetivo e fazer as coisas certas.

A eficiência também pode ser compreendida como uma extensão do conceito ágil. Como elucidado anteriormente, a maioria das metodologias ágeis são bastante específicas para tipos diferentes de cenários de projetos e organizações. Algumas são demasiadamente relacionadas à gestão de projetos, outras bastante específicas e focadas no desenvolvimento e codificação, enquanto outras não cobrem esforços de modelagem ou de alguma forma negligenciam a documentação. Nesse sentido, X-PRO propõe um framework, práticas, princípios, atividades e artefatos com o objetivo de entregar um processo de desenvolvimento eficiente e integrado.

Figura 26: Framework X-PRO (Extreme Software Process).

[16] MEGGINSON, Leon C.; MOSLEY, Donald C.; PIETRI, Paul H. Jr. Administração: Conceitos e Aplicações. Tradução de Maria Isabel Hopp. 4 ed. São Paulo: Harbra, 1998.

A estrutura do X-PRO foi projetada com o propósito de buscar o máximo de eficiência do time de desenvolvimento. Por meio da revisão dos conceitos e práticas atualmente vigentes em metodologias ágeis, bem como através dos resultados identificados na pesquisa apresentada no capítulo 5 desse trabalho, foram identificadas uma série de variáveis estruturais que devem compor o modelo de trabalho – *framework* – do X-PRO. O X-PRO utiliza em sua composição conceitos de metodologias ágeis como: Extreme Programming, Test-Driven Development, Behavior-Driven Development e Scrum. Além dessas metodologias e outras práticas ágeis, as quais serão explanadas posteriormente nesse trabalho, o X-PRO utiliza o UML (*Unified Modeling Language*) como linguagem para modelagem e produção dos diagramas de design previstos no framework. Conforme apresentado na Figura 26, a estrutura do X-PRO consiste em:

- **Valores e Princípios**

  Os valores e princípios guiam a adoção e toda a execução do X-PRO. Por tratar-se de um framework de processo baseado em metodologias ágeis, os valores e princípios são fundamentais para a compreensão e principalmente direcionamento quanto às decisões tomadas em um projeto baseado em X-PRO. Pequenos objetivos individuais invariavelmente são conflitantes com grandes objetivos coletivos (BECK, 2000). Conforme Beck (2000) explicou, as sociedades aprenderam a lidar com os problemas desenvolvendo uma série de valores, onde sem esses, os pequenos objetivos individuais poderiam se sobrepor aos grandes objetivos coletivos. Nesse sentido, para a efetiva e consistente adoção e execução de um processo de software baseado em X-PRO, a compreensão dos seus valores e princípios por parte de todo o time de desenvolvimento é fundamental. Em situações de conflitos internos entre o time, ou mesmo por definições específicas a cerca do projeto, os valores e princípios servem como balizadores das decisões do time. É de fundamental importância que mesmo antes de compreender os aspectos técnicos e de execução do X-PRO, o time tenha plena compreensão dos seus valores e princípios, do contrário a adoção do framework será comprometida. Para X-PRO, os valores e princípios são mantras, ideários sem os quais a compreensão do modelo torna-se inviável.

- **Atividades**

  As atividades previstas no framework do X-PRO foram identificadas com base na revisão da literatura a cerca de metodologias ágeis de desenvolvimento de software, bem como por meio de avaliação de estudos empíricos de processos de software. Além de prever atividades específicas ao seu próprio processo, X-PRO implementa a execução de atividades que com base nos estudos previamente citados, foram identificadas como fundamentais em qualquer ciclo de vida de projeto de software. Sempre buscando o princípio da eficiência, as atividades previstas no X-PRO foram definidas por serem fundamentais em qualquer ciclo de vida de software, ainda que possam ser nomeadas de formas diferente em outros modelos de processo.

- **Práticas**

  As práticas do X-PRO definem como o ciclo de vida do processo e suas atividades são executadas. O conceito de prática está relacionado aos métodos pelos quais são conduzidas as ações dentro de um processo X-PRO. Além de compilar as melhores práticas de metodologias ágeis de desenvolvimento de software, as práticas do X-PRO

implementam funcionalidades que apoiam e maximizam os resultados alcançados pela adoção do framework. Ainda que para outros modelos de processos ágeis as práticas sejam apontadas como sugestivas, em X-PRO as práticas são definidas como "habilitadores de resultados". Para X-PRO, a não adoção das suas práticas pode efetivamente comprometer os resultados esperados pela implementação e adoção do processo por parte do time e do projeto como um todo.

- **Papéis e Responsabilidades**

  O X-PRO estabelece papéis e responsabilidades para a execução do processo e consequentemente, um projeto de software baseado em X-PRO. Para o framework, não são considerados cargos ou funções formais, conforme padrões organizacionais. Para X-PRO, um papel é um estado de um ator, ou seja, em um determinado projeto um ator pode ser um Desenvolvedor, em outro projeto ele pode vir a ser um Analista de Negócios. É fato, porém, que assim como toda a estrutura do X-PRO busca a eficiência, o framework estabelece papéis fundamentais em um processo de software. Nesse caso, é importante que todos os papéis previstos no framework do X-PRO sejam atribuídos dentro do time de desenvolvimento, mesmo que algum ator acumule mais de um papel.

- **Artefatos**

  Boa parte das metodologias ágeis de desenvolvimento de software negligenciam a produção de artefatos dentro dos seus processos (AKIF & MAJEED, 2012). Para X-PRO, os artefatos são de fundamental importância, por conterem conhecimento organizacional, estrutural, operacional e técnico sobre o produto de software. É tecnicamente inviável para um novo membro em um time de desenvolvimento compreender a estrutura de um software sem o apoio de artefatos que suportem esse fim. Nesse sentido, X-PRO estabelece artefatos fundamentais e essenciais para a devida compreensão da estrutura lógica de uma aplicação.

A estrutura do X-PRO prevê em seu framework todos os recursos necessários para se obter o máximo de eficiência no processo de desenvolvimento de software. Por tratar-se do conceito e premissa fundamental do X-PRO, a eficiência consiste na única variável que pode ser utilizada para questionar ou justificar a adaptação de alguma das características do processo X-PRO. Por exemplo, em um projeto específico, algumas das práticas do framework – conforme poderá ser visto na Seção 6.7 – ou mesmo outra determinada característica do X-PRO, podem comprometer a eficiência do projeto. Sob essas condições, a premissa fundamental do X-PRO de eficiência pode ser acionada como gatilho para alguma adaptação ou especialização do processo. Entretanto, essas adaptações não podem, sob nenhuma condição, contrariarem os valores e princípios previstos por X-PRO. Nesse sentido, qualquer adaptação ou especialização do X-PRO deve se basear nos valores e princípios do framework, do contrário o processo não pode ser considerado X-PRO.

6.3. Valores e Princípios do X-PRO

Para X-PRO, seus valores e princípios são considerados as "pedras fundamentais" que sustentam o conceito associado à eficiência, premissa objetivada pelo X-PRO. Do ponto de vista filosófico, a definição de valor tem sido analisada e conceituada em diferentes áreas do conhecimento. Para a filosofia, o valor é descrito como algo que não é nem plenamente

subjetivo tampouco objetivo, porém, é definido como algo determinado pela interação entre o sujeito e o objeto[17]. Já os princípios, são conceituados como leis ou regras que precisam – ou desejavelmente seriam – seguidas. Os princípios também podem ser definidos como a consequência inevitável de algo, tal como as leis observadas na natureza ou a maneira que um processo é construído. Os princípios são compreendidos pelos seus usuários como as características essenciais do processo, ou a reflexão da sua finalidade[18].

A investigação da literatura a cerca de metodologias ágeis de desenvolvimento de software, conforme apresentado no Capítulo 4 desse trabalho, concluiu que é comum haverem divergências nas metodologias quanto à aderência e conformidade dessas aos seus próprios valores e princípios. Entretanto, em X-PRO essa divergência é verificada como um indício para uma adoção e até mesmo adaptação equivocada, haja vista que se realizada de forma contraditória aos seus valores e princípios, invariavelmente comprometeria a premissa básica de eficiência do X-PRO. Ainda que não seja um processo prescritivo, visto que permite adaptação em sua estrutura, o X-PRO estabelece que os seus valores e princípios devem ser analisados como as "regras do jogo", termo esse igualmente alcunhado em Extreme Programming (BECK, 2000).

Valores

- **Autonomia**

  O time precisa ter autonomia – essa definição será melhor explorada na Seção 6.4 – para conduzir as ações e direcionamentos necessários para entregar o acordado em um plano. Uma vez estabelecidos os macro objetivos e definidos prazos para estes o time precisa ter autonomia para conduzir o desenvolvimento conforme achar necessário. Por exemplo: suponha uma situação em que dentro de uma lista de recursos que foram levantados junto ao cliente, o time de desenvolvimento tenha concordado que dentro do prazo de 15 dias, cinco de dez recursos identificados serão entregues para o cliente. Dentro desse prazo, o time deve possuir autonomia para conduzir os esforços, estabelecer seus planejamentos individuais ou definir qualquer direcionamento relacionado. Considerando que o acordo entre o time e o cliente – ou mesmo a gestão do projeto – será cumprido, os desdobramentos dentro da janela estabelecida são de total liberdade de condução do time.

- **Unidade**

  É fundamental que haja uma união plena do time. É notório que times coesos e integrados tendem a potencializar características de qualidade individuais de cada membro. Por exemplo: em esportes coletivos como o Futebol, é comum serem identificados jogadores que apresentam um melhor desempenho quando colocados com um companheiro de equipe – principalmente se da mesma posição – de qualidade destacada. Além de fatores motivacionais, que potencializam as qualidades individuais mesmo de membros do time que não tenham um alto nível de habilidade, as qualidades adicionais do companheiro de time permitem que haja um balanceamento entre as habilidades gerais do time, o que por consequência exerce

[17] Burnham, Douglas; Buckingham, Will. O Livro da Filosofia, Editora Globo, 2011.

[18] Alpa, Guido (1994) "General Principles of Law," Annual Survey of International & Comparative Law: Vol. 1: Is. 1, Article 2.

influência sobre o resultado final. Desta forma, é de extrema importância que o valor de unidade no time seja constantemente evidenciado, reforçado e incentivado.

- **Foco**

  Deve-se focar na solicitação do cliente, conforme especificado por ele, procurando evitar a prática de *Gold Plating*[19]. Por vezes é comum que times julguem que suas convicções pessoais, ou mesmo entendimento do processo para o qual o software será desenvolvido, devem ser priorizadas em detrimento à solicitação inicial do cliente. Em X-PRO, o foco deve ser a requisição do cliente, independentemente de outras variáveis – sejam elas subjetivas ou não. Para exemplificar esse caso, existem situações de projeto, quando requisitos são levantados junto ao cliente para serem desenvolvidos, em que o time identifica alguma melhoria ou mesmo um recurso adicional que poderia ser concebido a partir da requisição do cliente. Por exemplo: foi solicitado que sempre que uma necessidade de compra seja inserida no sistema, essas sejam enviadas a todos os fornecedores cadastrados no mesmo. Partindo desse exemplo, o time pensa que poderia ser desenvolvido um portal onde as necessidades de compras fossem disponibilizadas aos fornecedores cadastrados, para que esses pudessem inserir suas propostas de venda. Sem dúvida o recurso pensado pelo time vai além do requisitado pelo cliente e possui uma característica de valor para o processo. O time então sugere ao cliente a implementação do recurso adicional idealizado. Porém, ainda que o cliente identifique valor na solução proposta e aceite a sugestão, requisita que a solicitação inicial seja atendida prioritariamente. Nesse caso, o time precisa focar para que o requisito inicial seja entregue, e só então – e se possível for dentro dos recursos, tempo e demais variáveis de projeto – o recurso adicional seja considerado para implementação.

- **Respaldo**

  X-PRO tem como valor que o respaldo é fundamental na sua execução. Todo o acompanhamento e interações entre o cliente e o time precisam ser documentados, com dois propósitos específicos: reter conhecimento técnico e operacional do software e documentar processos decisórios durante o desenvolvimento do produto. É comum que haja pouco conhecimento quanto ao porque de determinadas decisões a cerca das características ou regras de negócio de um software. Esse desconhecimento abre espaço para mudanças sem o devido entendimento do porque do requisito original ter sido projetado de tal forma. É importante, então, que sejam previstos mecanismos de respaldo para o processo. X-PRO não determina ou estabelece artefatos para isso, estando o time à vontade para definir a melhor forma de gerir o processo de interação com o cliente.

---

[19] *Gold Plating*, em Engenharia de Software, refere-se à inclusão de funcionalidades adicionais a um sistema que não foram solicitadas pelos usuários (clientes), motivada pelo entendimento do desenvolvedor de que o sistema fica melhor com as novas funcionalidades. A intenção do desenvolvedor nesse caso é agregar valor ao sistema. Entretanto, novas funcionalidades podem não agregar valor, na perspectiva do usuário, como pode haver desperdício de tempo pela perda de foco dos desenvolvedores. Além disso, o *Gold Plating* potencializa riscos adicionais de falhas no sistema. *WIEGERS, K. E. Software Requirements, 2nd Edition, Microsoft Press, 2003.*

- **Antecipação**

  O princípio da antecipação está relacionado ao feedback constante. X-PRO estabelece que o time deve sempre se antecipar ao cliente quanto ao compartilhamento de alguma informação que possa ser do seu interesse. Em situações de problemas técnicos, por exemplo, invariavelmente alguns times optam por não compartilharem com o cliente sobre um determinado problema. Para X-PRO, é importante que o time sempre mantenha o cliente informado e tenha com ele uma relação aberta. Essa antecipação em sempre manter o cliente a par do que está acontecendo no projeto transparece credibilidade, confiança e acima de tudo, a crença por parte do cliente de que todas as ações possíveis estão sendo tomadas para que o acordo inicialmente estabelecido seja cumprido.

Princípios

- **Ágil, mas com ordem**

  Uma das características mais marcantes das metodologias ágeis é que não há prescrição quanto à execução do processo. Em X-PRO, essa característica de não prescrição é igualmente presente. Entretanto, para que o processo proposto por X-PRO possa ser eficiente, acredita-se que deva haver ordenamento em sua adoção e execução. Para X-PRO, o ordenamento mencionado se dá quanto à forma em que as eventuais adaptações ao processo são realizadas. Ainda que permitidas, essas especializações ou customizações devem respeitar os valores e princípios declarados pelo framework.

- **As mudanças são parte do negócio**

  Uma das poucas certezas que se pode ter de cenários de projetos de software é que o contexto do negócio irá mudar ao longo do tempo. A dinamicidade é uma característica intensa em qualquer contexto de negócio, o que consequentemente leva a mudanças nos planos e nos processos. Em um cenário como esse, um processo de software que não esteja pronto para lidar com mudanças será fadado ao fracasso. Para X-PRO, o bom processo é aquele que se adéqua e se adapta para melhor reagir ao contexto de mudança. X-PRO aceita plenamente essa característica de mudança e busca estabelecer em seu framework as diretrizes ideais para lidar com isso.

- **Design e documentação das regras de negócio são ativos de conhecimento**

  Alguns métodos ágeis atribuem pouco valor aos esforços de design e documentação, sob a prerrogativa de que essas atividades não estão diretamente relacionadas à codificação e consequentemente, a concepção do produto. Há um claro negligenciamento nesse sentido. Para X-PRO, esforços de design e documentação são muito mais do que artefatos formais ou registros de decisões técnicas. Para X-PRO, os esforços de design e documentação são ativos de conhecimento, ou seja, a fonte ideal para que o time possa buscar informações sobre como funciona o software, sua estrutura, construção e funcionalidades. Além de viabilizar decisões técnicas mais embasadas, os esforços de design e documentação tendem a reduzir o tempo e a curva de aprendizado de novos membros do time, ou mesmo de membros já experientes mas que por ventura precisem analisar e discutir decisões técnicas estruturais do software.

- **Testes são parte do desenvolvimento**

  Alguns processos de software dividem os esforços de desenvolvimento e testes em atividades separadas. Para X-PRO, os testes são parte do desenvolvimento e o desenvolvimento é parte dos testes. Testar não pode ser considerado uma atividade secundária ou mesmo um esforço de qualidade, ou seja, algo que é realizado com propósito de buscar evitar erros. Os testes precisam ser encarados como um atributo do desenvolvimento e que sejam parte efetiva da codificação do software. Para X-PRO, um erro resume-se ao não atendimento de um requisito. Por exemplo: um requisito define que o sistema deve deduzir 10% de todos os clientes com mais 1 ano de contrato, porém, existem casos onde alguns clientes com mais de um ano de contrato, que não efetuaram pagamentos no último mês, tiveram sua cobrança gerada sem o desconto. Habitualmente esse cenário é compreendido como um um erro da aplicação. Para X-PRO, não. O cenário descrito acima é visto simplesmente como não atendimento ao requisito. Se o desenvolvimento é conduzido sob uma perspectiva orientada a testes, o requisito seria implementado em conjunto com cenários de testes que gradativamente iriam exaurir as possibilidades de não atendimento ao requisito – testes falhos – até que a implementação do requisito considerasse todos os cenários em que esse não fosse atingido – teste aprovados.

- **Solução sem satisfação não tem valor**

  O desenvolvimento de software por essência busca construir um produto de software que possa prover diferenciais competitivos, aprimorar e racionalizar processos, controlar operações, ou seja, entregar valor aos usuários. Entretanto, ainda que os clientes expressem seus desejos, os quais são compreendidos como requisitos e posteriormente transcritos em software, é possível que ao final o cliente perceba que o produto não satisfará por completo suas necessidades. Deve-se perceber que a satisfação nesse caso não está relacionada ao time, ou mesmo ao processo de desenvolvimento. O conceito de satisfação apresentado está relacionado a o software atender às expectativas do cliente, mesmo aquelas que ele só terá ciência após receber o produto. Nesse sentido, o time precisa garantir que há satisfação no cliente quanto ao software entregue, sob pena de se conceber uma solução que não será utilizada. Uma insatisfação do cliente deverá acionar o gatilho do princípio de que as mudanças são parte do negócio, e por isso, precisam ser abraçadas.

- **Entregas constantes, valor abundante**

  X-PRO prevê que o cliente precisa perceber constantemente a evolução do software. Melhor do que entregar relatórios ou gráficos de evolução, X-PRO determina que seja entregue software em funcionamento. Essa é a melhor forma do cliente perceber a evolução do seu produto. Nesse sentido, deve-se considerar no planejamento do desenvolvimento que sempre haja entregas ao cliente, por menores que sejam.

- **Comunicação impessoal é mera formalização**

  Para X-PRO só existe uma forma de se comunicar: pessoalmente. Todas as demais formas são consideradas formalização, as quais se associam ao valor de Respaldo, um dos valores do X-PRO. Ainda que seja considerada a utilidade e legitimidade de outros métodos de comunicação – e-mail, fax, carta, entre outros – esses sempre,

devem ser acompanhados da comunicação pessoal. Métodos diferentes do trato pessoal podem levar à incompreensão, dificuldade de se perceber o tom das palavras, mal entendidos, enfim, uma série de intempéries que poderiam ser contornados e rapidamente resolvidos se tratados pessoalmente. Consideram-se métodos de comunicação pessoal: presencial, videoconferência e telefone. Outros métodos podem ser adotados, desde que em conjunto com a comunicação pessoal.

Percebe-se que os valores e princípios do X-PRO guiam e conduzem não só aspectos subjetivos e comportamentais dos seus praticantes, mas principalmente, definem como os resultados previstos para a aplicação do framework do X-PRO podem ser atingidos. Há igualmente o alinhamento dos valores e princípios do X-PRO com os princípios do Manifesto Ágil, que consideram: Indivíduos e interações mais que processos e ferramentas; oftware em funcionamento mais que documentação abrangente; Colaboração com o cliente mais que negociação de contratos; Responder a mudanças mais que seguir um plano. Reforça-se que os valores e princípios do X-PRO são os únicos componentes imutáveis do framework, pois como processo ágil, o X-PRO é adaptável, porém sem a aderência aos seus valores e princípios, os resultados previstos para o processo não poderiam ser atingidos.

6.4. Arquitetura do X-PRO

A arquitetura do X-PRO é a parte do framework diretamente relacionada ao fluxo do processo e a sua execução propriamente dita – ciclo de vida e atividades. Por definição, o X-PRO se enquadra nas características do modelo Iterativo Incremental, porém com a particularidade de dividir sua iteração em duas etapas. Enquanto o ciclo mais exterior está relacionado a aspectos de especificação do projeto e do sistema, o ciclo interno está mais relacionado ao design, desenvolvimento e codificação.

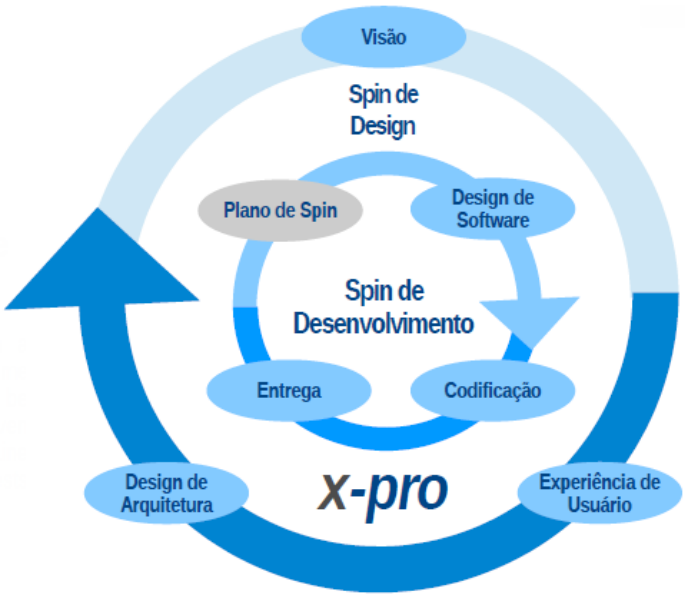


Figura 27: Arquitetura do X-PRO (Extreme Software Process).

Essa característica de ciclo também é encontrada no modelo de ciclo de vida espiral, o qual sugere "*evitar mudanças com tolerância a mudanças*" (SOMMERVILLE, 2009). É conhecimento comum que há um consequente retrabalho e eventuais desperdícios de esforços de desenvolvimento quando o cenário de projeto se dá com mudanças constantes (BEGEL &

NAGAPPAN, 2007). Nesse sentido, para X-PRO dividir as iterações em dois ciclos evita que haja esforços efetivos de codificação antes de um mínimo estabelecimento do que deseja o cliente quanto ao software, ainda assim, permitindo espaço para que as mudanças sejam propostas constantemente.

Ciclo de Vida do X-PRO (Spins de Design / Spins de Desenvolvimento)

O X-PRO é baseado no modelo Iterativo Incremental, de forma que todas as suas características estão contidas no ciclo de vida do X-PRO: as iterações possuem tempo de execução pré-estabelecidos (*timeboxed*); para cada iteração, todas as atividades previstas no processo são executadas; ao final de uma iteração é fundamental que haja entregas ao cliente; e o produto vai evoluindo ao longo de seu ciclo de desenvolvimento. Porém, o X-PRO possui a particularidade de dividir suas iterações em duas etapas, as quais, porém, são executadas sempre em conjunto. Para X-PRO, uma iteração é denominada de **Spin**.

O Spin é um ciclo de execução de projeto, com tempo pré-estabelecido, objetivos específicos e onde todas as atividades do processo são executadas com o propósito de se entregar partes do software em funcionamento para o cliente. Pode-se afirmar que através do conceito de Spin e da execução da atividade de *Visão* – a ser conceituada a seguir – é que são definidas as características de capacidade do X-PRO de ser um framework de processo de software que também cobre os esforços de gestão do projeto de software. Isso porque o conceito do Spin está relacionado a como o X-PRO viabiliza que o projeto de desenvolvimento de software seja gerenciado. Para que isso seja possível, o Spin tem estabelecida uma estrutura semelhante ao conceito de *Sprint*, herdado da metodologia ágil Scrum (SOMMERVILLE, 2009).

- **Objetivos do Spin:** Os Objetivos do Spin descrevem quais são as metas que se deseja atingir com a execução do respectivo Spin. Os objetivos precisam estabelecer conquistas claras e definidas;

- **Duração do Spin:** A Duração do Spin, descreve o período de início e fim de um Spin. Em X-PRO, os Spins devem possuir o prazo máximo de 3 (três) semanas, sendo 1 (uma) semana o prazo máximo de execução de um Spin de Design e 2 (duas) semanas o prazo máximo de um Spin de Desenvolvimento. Ainda que estabeleça prazos máximos, o X-PRO não prescreve que se o time estiver executando atividades do Spin de Desenvolvimento, as atividades do Spin de Design não possam ser executadas, pelo contrário. O que X-PRO define que durante o Spin de Design, o foco principal devem ser as atividades de Design, ou seja, nelas há uma maior concentração de esforço. Essa correlação se assemelha com a dinâmica de execução de um processo de software baseado no Processo Unificado, tal como o RUP (Rational Unified Process). Conforme visto no Capítulo 2, o RUP divide seu processo em fases e prevê um conjunto de atividades, onde nas fases iniciais, as atividades de modelagem e design possuem um esforço maior. Essa característica é igualmente estabelecida em X-PRO;

- **Horas de Trabalho do Spin por Semana:** Uma das características do X-PRO é que ele é um framework que atende tanto esforços de novos produtos como de manutenção de software já produzidos. Nesse sentido, assume-se que um time de desenvolvimento não possui o padrão de 8 (oito) horas diárias para dedicar-se ao desenvolvimento de um novo produto, como pode também precisar de tempo para manter um produto existente. Nesse sentido, X-PRO permite que seja definido

quantas horas de trabalho por semana poderão ser dedicadas ao spin. Para exemplificar, caso um time defina que um spin terá 20 horas semanais por semana – uma média de 4 horas por dia – assume-se que as demais horas do dia serão dedicadas a trabalhar na manutenção de outro produto. Ressalta-se, porém, que as horas de trabalho do spin por semana não precisam ser sequenciais, ou seja, o time pode variar a alocação das horas durante uma semana de spin da forma que achar que deve, reafirmando nesse caso o valor de autonomia;

- **Avisos do Spin:** Os Avisos do Spin descrevem as observações que devem ser revisitadas pelo time durante a execução do Spin. Por exemplo: no Spin anterior identificou-se uma falha técnica na arquitetura e essa precisa ser observada pelo time. Os Spin Warnings podem ser utilizados igualmente para catalogar e informar sobre *bugs* que precisem ser corrigidos;

- **A Fazer:** As demandas descritas como A Fazer, descrevem as estórias de usuário e comportamentos do sistema que deverão ser considerados para execução do Spin;

- **Executando:** As demandas descritas como Fazendo, descrevem as estórias de usuário e comportamentos que estão sendo executadas naquele momento no Spin;

- **Feito:** As demandas descritas como Feito, descrevem as estórias de usuário e comportamentos que já foram executados. Para X-PRO, a Definição de Feito (*Definition of Done*) – conceito herdado do Scrum (SOMMERVILLE, 2009) – se dá quando um comportamento – menor parte de uma estória de usuário – foi plenamente implementado e o Teste de Unidade daquele comportamento está sendo executado com sucesso. O conceito de Comportamento e Teste de Unidade do X-PRO é herdado do método ágil *Behavior-Driven Development* (NORTH, 2006).

Conforme apresentado anteriormente, o X-PRO divide suas iterações – *spins* – em duas partes: **Spin de Design** e o **Spin de Desenvolvimento**. Como foi previamente explanado, a divisão do Spin em duas partes não prevê que esse deverá ser executado um primeiro que o outro, ou que um Spin de Desenvolvimento dependa de um Spin de Design. Essa divisão se dá apenas no sentido de que nos spins iniciais de um projeto de software, é natural que os esforços de design sejam maiores do que os de desenvolvimento.

- **Spin de Design**

  O Spin de Design consiste em uma parte de uma iteração, cujos objetivos estão relacionados a aspectos de design, arquitetura e requisitos do software. Os Spins de Design são mais significativos nos primeiros spins de um projeto de software, haja vista que nas primeiras iterações é natural que não haja uma visão plena do que é o produto e quais os seus requisitos (SHUJA & KREBS, 2008; SOMMERVILLE, 2009). Para X-PRO, um Spin de Design não deve durar mais que uma semana, seja de trabalho ininterrupto, seja pela soma de dias dedicados a atividades de um Spin de Design. As atividades executadas durante um Spin de Design consistem em: Visão, Mockup de Interface e Design de Arquitetura. As atividades serão conceituadas posteriormente nesse capítulo.

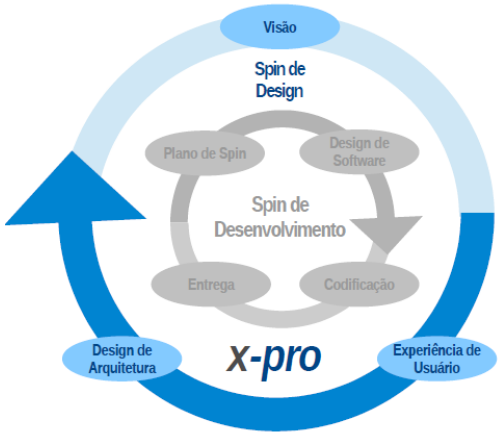


Figura 28: Ciclo de um Spin de Design no X-PRO.

- **Spin de Desenvolvimento**

  O Spin de Desenvolvimento consiste em parte de uma iteração, cujos objetivos estão relacionados a aspectos de codificação e estrutura do software. Sua atividade inicial está relacionada ao planejamento do Spin, ou seja, análise das Estórias de Usuários e especificações realizadas em um Spin de Design anterior. Essa característica é justificada pelo fato de que como o Spin de Design está relacionado a aspectos de design, arquitetura e requisitos do software, não seria possível haver um planejamento do Spin sem saber previamente quais são os requisitos do usuário. A única restrição que se dá ao Spin de Design ante ao Spin de Desenvolvimento, conforme já apresentado, é que o primeiro deve possuir o prazo máximo de 1 (uma) semana, enquanto o Spin de Desenvolvimento, tem seu prazo máximo de execução é de 2 (duas) semanas. Essa característica estabelece que o X-PRO precisa quebrar os requisitos em partes menores – estórias de usuário e comportamentos de sistema – de forma a permitir que pequenos pedaços do software sejam entregues ao final de um spin. As atividades previstas no Spin de Desenvolvimento são: Plano de Spin, Design de Software, Codificação e Entrega, as quais serão conceituadas mais adiante nesse capítulo.

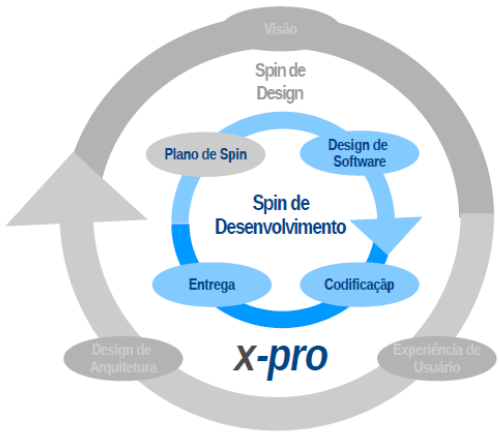


Figura 29: Ciclo de um Spin de Desenvolvimento no X-PRO.

Atividades

As atividades do X-PRO consistem no agrupamento de ações com objetivos específicos. Para X-PRO, as atividades não são descritas com uma abordagem de como devem ser executadas,

mas sim, do que devem prever. Com o propósito de maximizar os resultados oriundos das atividades do X-PRO, estas possuem artefatos relacionados, cujo objetivo não é meramente formalizar e documentar suas execuções, mas sim, atender ao princípio do X-PRO de que documentação são ativos de conhecimento.

- **Visão**

  A atividade de Visão é executada no Spin de Design e tem o propósito de estabelecer os requisitos tanto do projeto como do produto de software resultante dele. O macro objetivo da atividade é compreender aspectos centrais como: objetivos do projeto, principais clientes, áreas de negócio envolvidas com o projeto, propósitos de valor do projeto, cenário do processo, sugestão de solução, impactos da solução proposta, eliciação de requisitos e controle de mudanças do projeto. Os pontos analisados nessa atividade estão relacionadas a aspectos em nível de projeto, que por definição é um "*esforço temporário para se produzir um produto ou serviço*[20]". Entretanto, na atividade de Visão também estão presentes os esforços dedicados à compreensão do que é o produto e quais são seus requisitos. Nesse sentido, os requisitos do cliente são elicitados como Estórias de Usuários, as quais descrevem funcionalidades desejadas pelo usuário com base em uma estrutura descritiva padrão que apresenta o papel, a funcionalidade desejada e o benefício a partir dela, por exemplo: Como Analista Financeiro, desejo os pagamentos de contas, de forma a garantir a segurança das operações financeiras (LEFFINGWELL, 2011; NORTH, 2007). Os artefatos produzidos na atividade de Visão são: Canvas de Visão, cujo objetivo é apresentar os aspectos gerais do projeto; e a Matriz de Estórias de Usuário, que consiste no resumo do escopo de requisitos, estórias de usuário e comportamentos do sistema.

- **Experiência do Usuário**

  A atividade de Experiência do Usuário, está relacionada a compreender a expectativa do usuário – cliente – quanto à apresentação e interface do software. Pode-se dizer que essa é a primeira atividade do ciclo de vida do X-PRO que efetivamente entrega algum produto ao cliente – a interface da aplicação. O macro objetivo dessa atividade é compreender aspectos como: usabilidade, atributos e tipos de dados e dinâmica de interação entre o usuário e o software. Um dos seus principais propósitos é que através do contato do cliente com o que seria a aplicação, ele pode efetivamente identificar se os requisitos, estórias e comportamentos elicitados efetivamente refletem o que ele espera do produto. Quanto ao benefício dessa atividade, está o fato de que possíveis mudanças nos requisitos podem ser identificadas antecipadamente, sem incorrer na necessidade de esforços de codificação, conforme apresentado anteriormente no descritivo da arquitetura do X-PRO. O artefato produzido na atividade de Experiência do Usuário é o Mockup de Interface, que consiste no layout e representação gráfica da interface do software.

- **Design de Arquitetura**

  A atividade de Design de Arquietura objetiva modelar e definir a macro arquitetura do software, contemplando aspectos relacionados a um modelo generalista que considera variáveis de hardware, software, comunicação e demais componentes que possam influenciar na concepção estrutural do software. Por se tratar de uma atividade

[20] PMI. A Guide to the Project Management Body of Knowledge, 5th Edition, 2012.

executada no Spin de Design, compreende-se que essa atividade tende a ser executada nos spins iniciais do projeto, ainda que possa ser continuamente realizada, caso haja mudanças na arquitetura da aplicação ao longo do desenvolvimento. Como o X-PRO é indicado tanto para esforços de concepção de um novo software como para manter produtos existentes, essa atividade tende a perder valor ou ter uma execução limitada em manutenção de sistemas, haja vista que a arquitetura do produto já está consolidada. O artefato produzido na atividade de Design de Arquitetura é o Diagrama de Implantação, previsto na biblioteca de UML (Unified Modeling Languagem) (AMBLER, 2004).

- **Plano de Spin**

  A atividade de Plano de Spin tem como propósito definir os objetivos e estratégias para a execução de um Spin. Como o X-PRO tem como escopo atender todo o ciclo de vida de um projeto de software, a atividade de Plano de Spin está relacionada com esforços de gestão de projetos, estabelecendo aspectos como: objetivos do spin, duração do spin, horas de trabalho por semana no spin, avisos do spin, atividades a realizar, em realização e realizadas. As atividades a serem realizadas no Spin são extraídas da Matriz de Estórias de Usuário, a qual funciona como um backlog do produto de software em desenvolvimento. Essa atividade possui uma correlação com a atividade de *Sprint Planning* do SCRUM (SOMMERVILLE, 2009), onde um período de tempo é estabelecido (*timeboxing*) para a execução de uma iteração, baseada na visão de atividades dispostas de forma objetiva – a fazer, fazendo e feito. O artefato produzido na atividade de Plano de Spin é o Canvas de Tarefas.

- **Design de Software**

  A atividade de Design de Software tem como propósito definir aspectos técnicos e estruturais do software. Trata-se de um detalhamento da atividade de Design de Arquitetura, com o objetivo claro de estabelecer questões mais objetivas e técnicas do software, principalmente relacionadas à estrutura de dados e representação do projeto de análise da aplicação. O X-PRO estabelece nessa atividade duas convenções padrão para o framework: o modelo relacional[21] é o padrão para definição da estrutura de dados e o paradigma de orientação a objetos é o padrão de representação de projeto de análise e arquitetura. A atividade de Design de Software prevê esforços inerentes ao desenvolvimento de qualquer produto de software: modelar a estrutura de dados e o projeto de análise. É notório conforme identificado no Capítulo 5, que esses dois esforços possuem valor inquestionável em projetos de software, mas que porém, não são executados de forma estruturada, principalmente no que diz respeito a representar essas definições em modelos e diagramas. Nesse sentido, X-PRO remete ao princípio de que design e documentação das regras de negócio são ativos de conhecimento. Os artefatos produzidos na atividade de Design de Software são os Modelo de Softwares, os quais tratam-se do Diagrama de Classes e o Diagrama de Modelo de Dados (ER, Entidade Relacionamento), os quais assim como o Diagrama de Implantação previsto na atividade de Design de Arquitetura, são previstos na biblioteca de UML (Unified Modeling Languagem) (AMBLER, 2004).

[21] CODD, E.F. Relational Model of Data for Large Shared Data Banks, Communications of the ACM, Vol. 13, No. 6, pp. 377-387, June, 1970.

- **Codificação**

  A atividade de Codificação está relacionada aos esforços de implementação. Nessa atividade o X-PRO determina outra convenção: o desenvolvimento deve ser orientado a testes e a comportamentos. A importância de seguir essa convenção é tão relevante para os praticantes do X-PRO que o framework propositalmente tem no título da atividade de desenvolvimento a orientação de como esse deve ocorrer – orientado a testes e a comportamentos. Essa característica do processo de codificação do X-PRO se dá pelos notórios benefícios da produtividade e redução de falhas viabilizados pela adoção de práticas de TDD (Test-Driven Development) e BDD (Behavior-Driven Development). Não há nenhum artefato previsto diretamente relacionado à atividade de Codificação.

- **Entrega**

  A atividade de Entrega está relacionada às entregas do projeto. Nessa atividade o propósito é simples: entregar valor para o cliente. O valor, nesse caso, trata-se de software em execução, exceto no caso de entregas ocorridas entre Spins de Design, as quais estão relacionadas aos mockups de interfaces para validação do cliente. O artefato produzido na atividade de Entrega é o Notas de Entrega, que consiste em pequenas notas quanto ao que foi implementado no pacote que está sendo liberado.

Linha do Tempo

Como foi apresentado, o X-PRO prevê em sua arquitetura um ciclo de vida de execução de atividades. Esse ciclo de vida é baseado em esforços com duração pré-estabelecida (*timeboxed*), intitulado de Spin, o qual é dividido em Spin de Design e Spin de Desenvolvimento. Essa divisão objetiva atribuir o volume de esforços específicos necessários às respectivas atividades de cada spin.

Adicionalmente, o framework do X-PRO propõe uma Linha do Tempo, que consiste em uma sugestão de como deve ser executado um ciclo de vida de projeto no X-PRO. Inicialmente a proposta da Linha do Tempo tem fins didáticos e não é obrigatória, ou seja, objetiva ilustrar como evolui um projeto ao longo do tempo.

A Linha do Tempo pode ser adotada como referência de execução do processo, haja vista que no caso de novos adeptos, pode haver alguma dificuldade para visualizar como o processo é aplicado. Cada projeto pode – e deve – adequar e estabelecer a sua Linha do Tempo de acordo com a necessidade deste, de forma a fornecer uma visão geral das entregas do projeto.

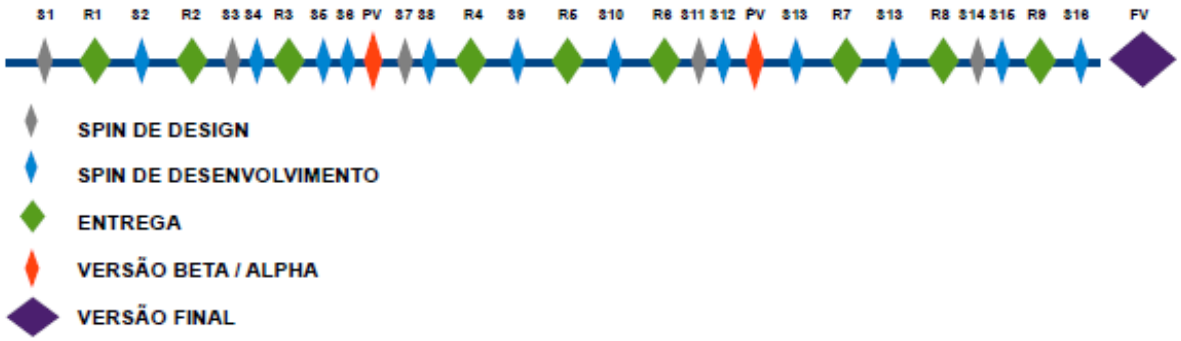


Figura 30: Linha do Tempo dos ciclos de Spin do X-PRO.

Na representação gráfica apresentada da Linha do Tempo do X-PRO, é possível identificar alguns pontos relevantes para a compreensão do processo. Percebe-se que o primeiro spin é um Spin de Design, ou seja, está se compreendendo as nuances e escopo do que será o projeto e consequentemente, o produto e seus requisitos. Em seguida, há uma primeira entrega – entrega – a qual conforme descrito nas atividades, está relacionada à Experiência do Usuário – experiência do usuário, cujo propósito é compreender por meio de uma representação gráfica de interface, se efetivamente o que foi elicitado atende às expectativas do cliente.

Ocorre, então, o primeiro Spin de Desenvolvimento, o qual entregará a segunda entrega, nesse caso, já se tratando de alguma parte do software rodando. Adiante, a Linha do Tempo apresenta dois spins, um de Design e outro de Desenvolvimento, ou seja, compreende-se que está havendo um amadurecimento das ideias e desejos do cliente à medida em que ele passa a receber entregas do projeto e que isso dá margem para solicitações de mudança, as quais requerem esforços de Design. A Linha do Tempo ilustrada na Figura 30 apresenta dois spins de Desenvolvimento para então haver o primeiro pacote de versão do produto, ou uma Versão de Protótipo, que é representada como uma versão Beta. Em X-PRO, as Versões de Protótipo objetivam estabelecer um marco no projeto, onde, por exemplo, poderia ser representado por uma entrega de um módulo do software. Assim, mais do que entregas pequenas ao longo do projeto, para X-PRO é importante haver marcos de entregas mais substanciais, mesmo que essas consistam no agrupamento de outras entregas menores em um pacote único, que para o framework é chamado de Versão de Protótipo, como explanado anteriormente.

O conceito de Linha do Tempo pode ser utilizado igualmente para apoiar a adoção do X-PRO em projetos de larga escala. Nesses cenários de projeto, sugere-se a decomposição do projeto em partes menores, as quais seriam planejadas individualmente como projetos X-PRO menores. Nesse caso o uso do recurso de Linha do Tempo auxilia o entendimento macro do projeto, contemplando todas as diferentes frentes de decomposição do projeto.

6.5. Papeis e Responsabilidades no X-PRO

No X-PRO, toda a estrutura do framework é adaptável e pode ser customizada conforme a necessidade do projeto, desde que esteja alinhada, porém, aos valores e princípios do X-PRO. Igualmente no caso dos papéis e responsabilidades em um projeto X-PRO, esses são passíveis de adaptação e definição conforme necessidade do projeto. Entretanto, o framework propõe alguns papéis e responsabilidades fundamentais, os quais invariavelmente são encontrados em qualquer projeto de software. No entanto, uma pessoa pode absorver mais de um dos papéis propostos no framework do X-PRO, haja vista que esses definem responsabilidades e não cargos dentro da estrutura do projeto.

- **Patrocinador:** O Patrocinador é a pessoa de referência na concepção do produto. Este papel tem como objetivo ser a principal fonte de requisitos do software, bem como a pessoa que possui maior respaldo para decidir sobre pontos contraditórios do projeto ou do próprio software. O Patrocinador é encarregado de definir uma equipe de Clientes Principais, os quais serão envolvidos quanto a elicitação dos requisitos, estórias de usuários e comportamentos do software. Ainda que haja um time de Clientes Principais, o Patrocinador é sempre o principal decisor quanto ao escopo de um requisito do software. Pode-se fazer uma correlação desse papel com o *Product Owner* da metodologia ágil Scrum.

- **Clientes Principais:** Os Clientes Principais consistem em um conjunto de pessoas definidas pelo Patrocinador que serão responsáveis por serem fontes de informação de processo e requisitos do software. Esse papel geralmente é atribuído a pessoas que posteriormente serão usuárias da aplicação, de forma que sua participação é de fundamental importância. Todavia, dado o fato de que trata-se de um grupo, é possível que hajam conflitos sobre como é a operação de um processo de negócio ou mesmo como deve ser um requisito do software. Nesse caso, deve ser convocada a participação do Patrocinador para que esse defina qual a medida deve ser tomada e a decisão final a cerca do impasse criado. É indicado que o papel de Clientes Principais seja atribuído a pessoas que conheçam bem o processo de negócio a ser suportado pelo software em desenvolvimento.

- **Mestre X:** O Mestre X é um papel de liderança dentro de um processo X-PRO. O objetivo desse papel é tanto gerir o projeto de software como principalmente, garantir que a estrutura e o processo do X-PRO seja respeitado e seguido conforme estabelecido. Pode-se fazer uma correlação do papelo de Mestre X com o *Scrum Master* da metodologia ágil do Scrum, porém, que não se restringe a garantir apenas o processo, mas igualmente, a gestão e o resultado do projeto. É encarregado pela atividade de Plano de Spin, liderando a definição das variáveis consideradas no planejamento de um Spin, é também uma importante responsabilidade do Mestre X garantir o bom desempenho do time X-PRO, por meio de ações que motivem e forneçam o constante apoio para que o time possa alcançar os objetivos estabelecidos. O Mestre X deve tratar conflitos internos, remover empecilhos para que o time possa desenvolver seu trabalho, bem como ser o intermediário entre o time, os Clientes Principais e o Patrocinador do projeto.

- **Analista de Negócios:** O Analista de Negócios é responsável por realizar as atividades de Visão, Experiência do Usuário e Design de Arquitetura. É fundamental que esse papel possua boas habilidades quanto ao processo de negócio a ser suportado pelo software, servindo como uma referência do processo do cliente para o time X-PRO. Além de habilidades de negócio, o Analista de Negócios deve possuir habilidades de elicitação de requisitos, modelagem de negócios, design de interface e UML, haja vista que esse papel será responsável por produzir artefatos que posteriormente serão utilizados como referência documental pelo time de desenvolvimento do projeto.

- **Desenvolvedor:** O Desenvolvedor é o papel encarregado pelas atividades de Design de Software, Codificação e Entrega. É o responsável pela efetiva construção e codificação do software. É importante que o desenvolvedor possua igualmente habilidades de análise e projetos de software, além de conhecimentos de UML, haja vista que o desenvolvedor será encarregado de produzir os artefatos previstos na atividade de Design de Software.

6.6. Artefatos do X-PRO

Uma parte relevante do framework do X-PRO diz respeito aos artefatos propostos para a execução do processo em um projeto de software. Conforme observado por Akif e Majeed (2012), na busca por uma maior agilidade nos processos e consequentemente nos projetos de software, as metodologias ágeis tendem a negligenciar aspectos relevantes em Engenharia de Software, tais como o Design e Documentação. Nesse sentido, uma das propostas do X-PRO

é prever um processo ágil e eficiente, mas que igualmente considere a relevância das atividades de design e documentação.

Como apresentado na Seção 6.4, a qual expõe os aspectos gerais quanto à arquitetura do X-PRO, os esforços de design são parte significativa do processo. A outra lacuna das metodologias ágeis que X-PRO busca tratar é fornecer um conjunto de artefatos em seu framework, os quais sejam significativos do ponto de vista de prover conhecimento sobre o software, mas que ao mesmo tempo não comprometa as características de agilidade e eficiência buscada pelo framework. Assim, X-PRO prevê um conjunto de artefatos mínimos e fundamentais para a devida compreensão e documentação de todo o seu ciclo de vida. Os artefatos serão apresentados por ordem de concepção nas atividades previstas no framework do X-PRO.

- **Canvas de Visão**

  Habitualmente os modelos e frameworks de processos de software propõem uma série de documentos com propósito de apresentar as variáveis do projeto, bem como características preliminares do que é o cenário a ser coberto pelo software (SOMMERVILLE, 2009). Modelos baseados no Processo Unificado, tais como o RUP (Rational Unified Process), preveem diversos artefatos distintos, tais como Documento de Visão, Documento de Requisitos, Documento de Planejamento de Projeto (SHUJA & KREBS, 2008; KRUTCHEN, 1999). Já em se tratando de metodologias ágeis, principalmente aquelas que tratam os esforços relacionados a gerência de projetos, tais como Scrum (JANOFF & RISING, 2010) ou mesmo Extreme Programming (BECK, 2000), foca-se mais especificamente no processo de execução do projeto do que em uma visão ampla de planejamento do mesmo.

  Essa dificuldade em possibilitar uma visão ampla do contexto do projeto sem necessariamente requerer a produção de um volume extenso de documentação é igualmente encontrada em outras áreas de conhecimento, como a Administração. Para concepção de um novo negócio, por exemplo, sugere-se que seja produzido um documento de Plano de Negócios, o qual especifique todas variáveis inerentes à concepção e realização de um novo negócio, dentre elas: objetivos, público alvo, produtos e serviços propostos, análise de riscos, análise financeira, entre outras variáveis[22]. Considerando esse cenário de complexidade, Osterwalder e Pigneur (2010) propuseram o Business Model Canvas, o qual consiste em um mapa visual pré-formatado contendo blocos descritivos de distintas variáveis de negócios. Aplicando esse modelo, uma organização pode facilmente descrever seu modelo de negócios. Os benefícios da sua aplicação são os mais diversos tais como: simplificação da descrição de um negócio ou proposta de negócio; exposição visual do que é o negócio para todo um grupo de trabalho; bem como a possibilidade de expor o modelo em uma superfície para que um grupo de pessoas discuta os elementos do modelo de negócios e vá adicionando pontos que devem existir no novo negócio, entre outros.

  Com base no modelo proposto por Osterwalder e Pigneur (2010), o framework do X-PRO prevê o artefato Canvas de Visão, o qual objetiva compilar todas as principais variáveis do projeto de software, separando-as por blocos lógicos, centralizadas em um único documento. Assim como no modelo de Osterwalder e Pigneur (2010), a

[22] Pinson, L. Anatomy of a Business Plan: A Step-by-Step Guide to Building a Business and Securing Your Company's Future. 6th Edition, Dearborn Trade: Chicago, USA, 2004.

elaboração do Canvas de Visão se dá de uma forma conjunta entre um grupo de pessoas – no caso do X-PRO, do time de projeto – onde por meio de sessões de trabalho, o time juntamente com o cliente possa elicitar e documentar todas as variáveis de contexto do projeto. A produção do Canvas de Visão é o primeiro esforço que deve existir em um projeto X-PRO, sendo parte fundamental da atividade de Visão. Não há prescrição quanto a quantas sessões devem ser realizadas para a concepção de um Canvas de Visão, haja vista que trata-se de um documento que constantemente deve ser incrementado e analisado por parte do time do projeto. Entretanto, assim como descreve o X-PRO, naturalmente há um volume maior de esforços na sua produção nos primeiros spins do projeto, sempre sendo esse tratado no spin de Design. A estrutura do Canvas de Visão, conforme ilustrado na Figura 31, prevê as seguintes variáveis:

- **Objetivos do Projeto:** Objetivos estabelecidos para o projeto;
- **Clientes Principais:** Quem são os principais clientes do projeto, assim como quais serão suas interações com o software;
- **Áreas de Negócio:** Quais são as áreas que estão sendo afetadas pelo projeto e quais são os principais aspectos que afetarão as áreas de negócio;
- **Proposições de Valor:** Qual valor está sendo entregue aos clientes, quais dos problemas do cliente estão sendo ajudados a serem solucionados e quais necessidades dos clientes estãos endo satisfeitas;
- **Cenário de Processo:** Descrição geral do cenário de processo a ser coberto pelo software;
- **Sugestão de Soluções:** Descrição geral da sugestão de soluão proposta para o projeto;
- **Impactos da Solução:** Descrição geral de quais impactos serão causados pela solução – sejam eles positivos ou negativos;
- **Requisitos MoSCoW:** Descrição de quais requisitos gerais precisam (M - Must), deveriam (S – Should), poderiam (C - Could) e haveriam (W – Would) de ter o software;
- **Mudanças:** Descrição de mudanças em aspectos do projeto ao longo de sua execução;
- **Informações:** Informações gerais relativas ao projeto.

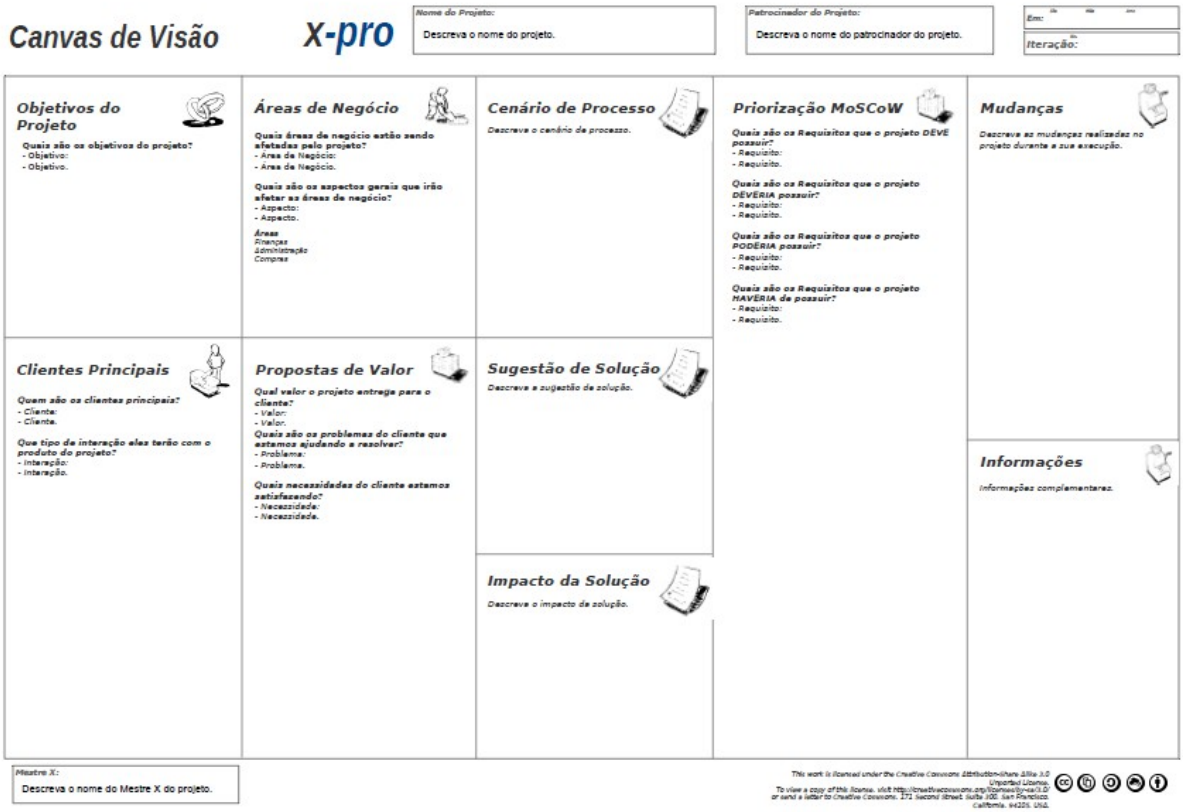


Figura 31: Canvas de Visão.

- **Matriz de Estórias de Usuário**

  A Matriz de Estórias de Usuárioé o segundo artefato previsto na atividade de Visão e tem como objetivo compilar todos os Requisitos, Estórias de Usuário e seus comportamentos. A proposta de utilização de uma matriz tem como objetivo tornar ainda mais ágil o processo de elicitação e documentação das funcionalidades a serem previstas pelo software, haja vista que todos os recursos que o software deve possuir são compilados em um único artefato. Além da facilidade de compilar em um único documento todos os requisitos e estórias do usuário, a adoção da Matriz de Estórias de Usuário contempla informações quanto à priorização das estórias e documentação dos testes de aceitação, práticas essas a serem conceituadas na Seção 6.7. A estrutura da Matriz de Estórias de Usuário prevê:

  - **Requisito:** Descrição do requisito geral solicitado pelo cliente;
  - **Estória de Usuário – Como Papel:** Descrição da parte da Estória do Usuário relativa ao papel executor do processo;
  - **Estória de Usuário – Eu Desejo, Recurso:** Descrição da Estória do Usuário relativa ao recurso desejado pelo cliente;
  - **Estória de Usuário – De forma que, Benefício:** Descrição da Estória do Usuário relativa ao benefício proporcionado pelo recurso a ser desenvolvido;
  - **MoSCoW:** Descrição de qual a classificação de prioridade da Estória de Usuário (M – Must, S – Should, C – Could, W – Would);
  - **Teste de Aceitação:** Descrição do teste de aceitação a ser realizado posteriormente a produção do recurso. O Teste de Aceitação consiste na validação a ser realizada pelo cliente para certificar que o recurso

desenvolvido atende aos requisitos inicialmente elicitados. O teste de aceitação é descrito em nível de Estória de Usuário;

- **Comportamento – Dado que:** Descrição da condição em que o comportamento deve ser executado, como parte da prática de Behavior-Driven Development (NORTH, 2006);
- **Comportamento – Quando:** Descrição da condição de quando o comportamento deve ser executado, como parte da prática de Behavior-Driven Development (NORTH, 2006);
- **Comportamento – E:** Descrição da condição de complemento a quando o comportamento deve ser executado, como parte da prática de Behavior-Driven Development (NORTH, 2006);
- **Comportamento – Então:** Descrição da ação executada mediante gatilho do comportamento, como parte da prática de Behavior-Driven Development (NORTH, 2006);
- **Feito:** Informação da situação atual de produção de uma Estória de Usuário. Os status possíveis são: Previsto (Previsto para o spin atual); Fazendo, Feito; Backlog (A ser realizado em um próximo spin).

- **Mockup de Inteface**

O Mockup de Interface é um artefato previsto na atividade de Experiência de Usuário, executada durante os spins de Design. O objetivo do Mockup de Interface é ilustrar o projeto de interface desejado pelo usuário para o sistema. Assim como proposto por Behavior-Driven Development (NORTH, 2006), o projeto de interface nas primeiras iterações do projeto é de fundamental importância para amadurecimento dos requisitos e estórias por parte dos clientes, haja vista que diante de uma visão do que possivelmente será o software, o cliente pode identificar nuances de processo, condições e comportamentos complementares para a aplicação. É igualmente uma ferramenta para descrever as variáveis de dados e tipos, informações necessárias para execução da atividade de Design de Software, onde é prevista a concepção do Diagrama de Classes e Diagrama de Dados (ER).

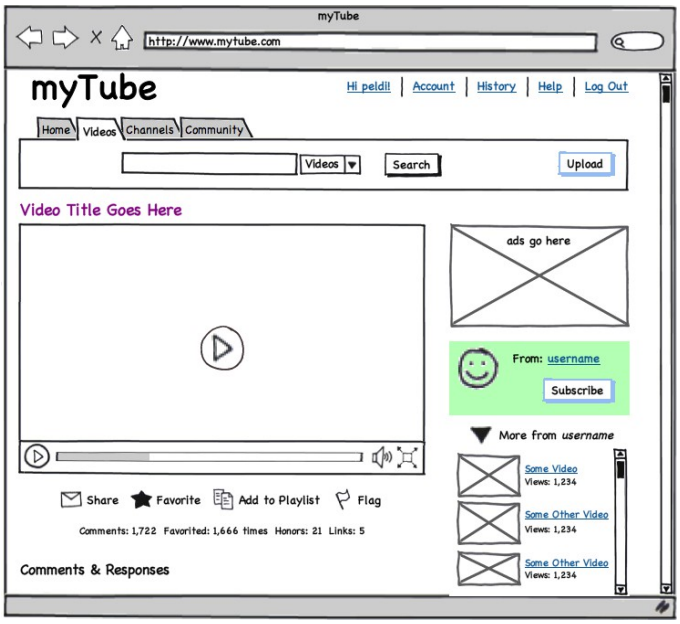


Figura 32: Exemplo de Mockup de Interface.

- **Canvas de Tarefas**

  O Canvas de Tarefas é um artefato produzido durante a atividade de Plano de Spin e descreve os aspectos gerais previstos para o próximo spin do projeto. Baseado no modelo de Osterwalder e Pigneur (2010), o Canvas de Tarefas é um artefato que é dinâmico e atualizado constantemente ao longo da execução do spin, haja vista se tratar de uma ferramenta para gestão tanto por parte dos desenvolvedores como dos gestores do projeto. Para X-PRO, o Canvas de Tarefas deve ser exposto visualmente para que todo o time possa visualizá-lo. Para isso, sugere-se a aplicação de *post-it*, pequenos pedaços de papel coláveis que sejam manipulados entre os estados de desenvolvimento previstos em um spin (A Fazer, Fazendo e Feito). O Canvas de Tarefas possui a seguinte estrutura:

  - **Objetivos do Spin:** Descreve quais são os objetivos – mensuráveis – para o spin;
  - **Duração do Spin:** Estabelece a duração do spin e quantas horas produtivas por semana o time de projeto terá;
  - **Avisos do Spin:** Descreve avisos que o time de projeto deve estar atento durante a execução do spin;
  - **A Fazer:** Estórias de usuário e comportamentos previstos para serem executados no spin;
  - **Fazendo:** Estórias de usuário e comportamentos que estão sendo executados no spin;
  - **Fazendo:** Estórias de usuário e comportamentos que já foram executados no spin.

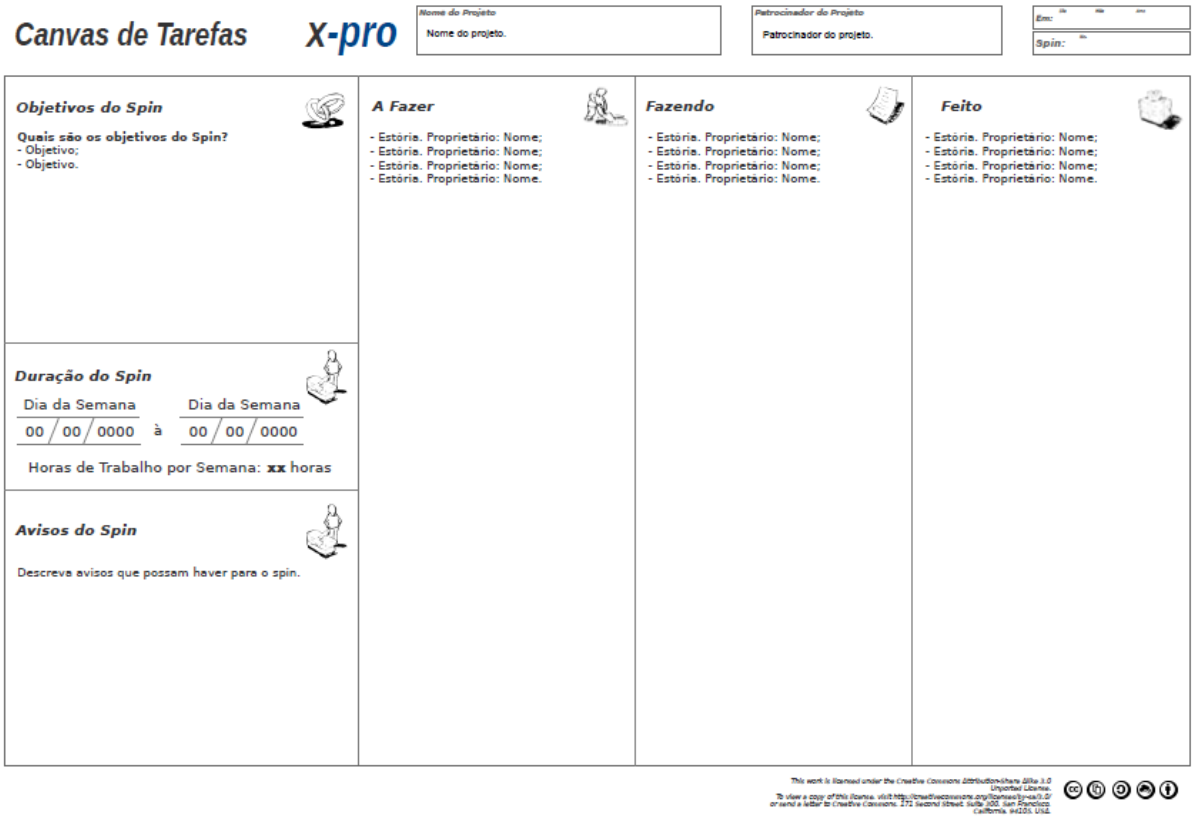


Figura 33: Canvas de Tarefas.

- **Modelo Arquitetural (Diagrama de Implatação)**

  O Modelo Arquitetural trata-se de um artefato produzido na atividade de Design de Arquitetura, cujo objetivo é apresentar uma visão geral da arquitetura do software. Por ser um artefato concebido em spins de design, há uma consequente característica de que através desse artefato, é possível conhecer detalhes da estrutura geral da aplicação, informação que é importante tanto no início do projeto, o que possibilita compreender a dimensão deste, como para manutenção posterior do produto concebido. O Modelo Arquitetural trata-se do Diagrama de Implantação previsto na biblioteca de UML (AMBLER, 2004). Apesar de em outros modelos de processo o Diagrama de Implantação ser produzido nas iterações finais do projeto, para X-PRO a visualização antecipada de como seria a estrutura de implantação – a qual considera diversos aspectos que não só o software – auxiliam no melhor entendimento do que será necessário o software considerar em termos do ambiente a ser implantado – software, hardware, comunicação, integração, entre outras variáveis. Em X-PRO, o Modelo Arquitetural (Diagrama de Implantação) deve ser produzido utilizando as práticas de Agile Modeling (AMBLER, 2002), a qual faz parte do conjunto de práticas previstas por X-PRO, que serão apresentadas na Seção 6.7.

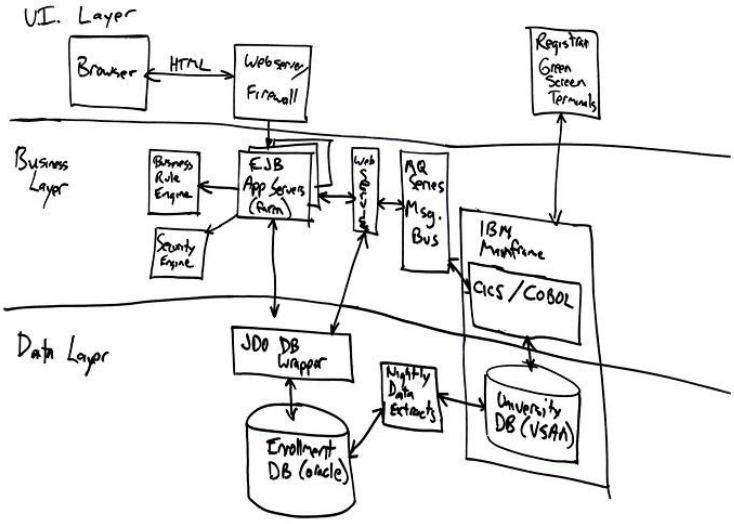


Figura 34: Exemplo Diagrama de Implantação modelado com base nos princípios de Agile Modeling.

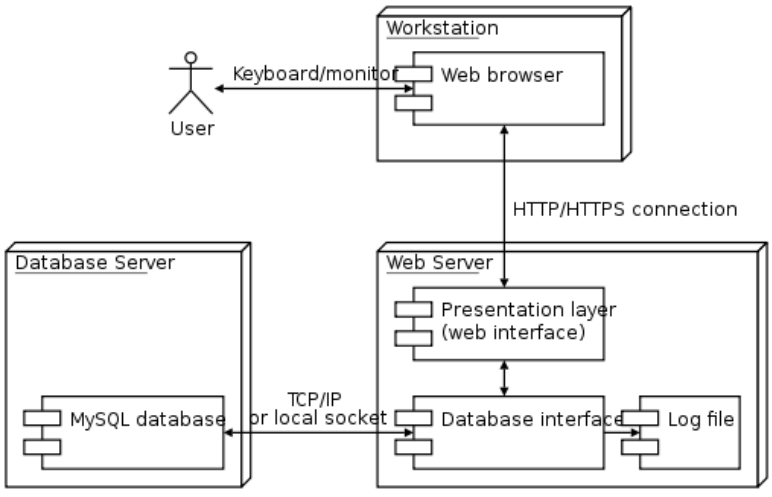


Figura 35: Exemplo Diagrama de Implantação modelado em UML em ferramenta CASE.

- **Modelo de Software (Diagrama de Classe / ER)**

  O Modelo de Software (Diagrama de Classe / ER) é um artefato produzido na atividade de Design de Software, cujo objetivo é modelar o Diagrama de Classes e o Diagrama de Dados (ER, Entidade Relacionamento) do software. Essa atividade é inerente a qualquer projeto e processo de software, cujo propósito é definir a estrutura lógica da aplicação, por isso tem parte relevante na estrutura do framework do X-PRO. Assim como o Modelo Arquitetural (Diagrama de Implantação), o Modelo de Software prevê diagramas parte da biblioteca do UML (AMBLER, 2004), porém concebidos através da aplicação da prática Agile Modeling (AMBLER, 2002).

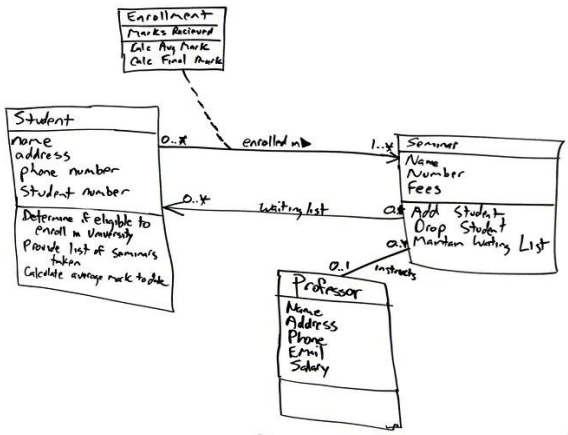


Figura 36: Exemplo Diagrama de Classes modelado com base nos princípios de Agile Modeling.

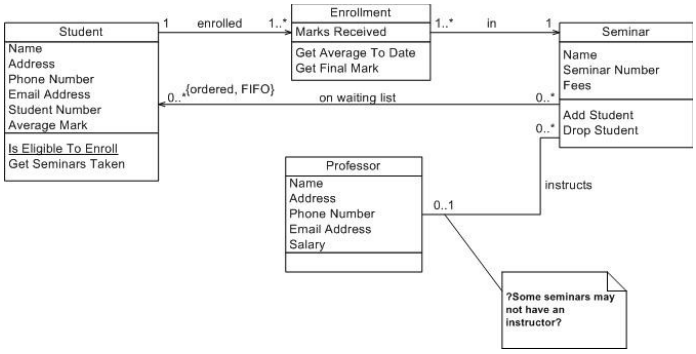


Figura 37: Exemplo Diagrama de Classes modelado em UML em ferramenta CASE.

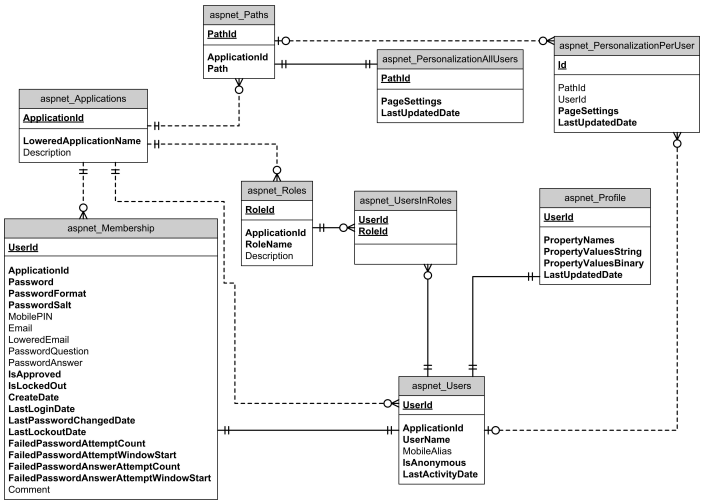


Figura 38: Exemplo Diagrama de Dados (ER) modelado em UML em ferramenta CASE.

- **Notas de Entrega**

  O artefato Notas de Entrega (Notas de Lançamento) refere-se a um documento elaborado na atividade de Entrega, executada em spins de Development, cujo objetivo é descrever e documentar quais Estórias de Usuário foram implementadas no último spin e como essa implementação ocorreu. Trata-se de um importante documento para relatar ao cliente quanto ao que foi entregue, bem como formalizar as entregas. A produção do artefato Notas de Entrega apoia a aplicação dos valores de Respaldo (no sentido de prover um mecanismo que documenta as ações executadas pelo time) e Antecipação (no sentido de fornecer feedback ao cliente quanto aos esforços promovidos pelo time).

NOTAS DE ENTREGA NOTAS DE ENTREGA DE ESTÓRIAS DE USUÁRIO			x-pro
PROJETO:	Descrição.	**PATROCINADOR:**	Descrição.
DATA:	Descrição.	**PERÍODO DO SPIN:**	___/___/_____ à ___/___/_____
MESTRE X:	Descrição.	**CLIENTES PRINCIPAIS:**	Descrição.

ESTÓRIAS DE USUÁRIO	IMPLEMENTAÇÃO	DESENVOLVEDOR
Descrição.	Descrição.	Descrição.
Descrição.	Descrição.	Descrição.
Descrição.	Descrição.	Descrição.
Descrição.	Descrição.	Descrição.
Descrição.	Descrição.	Descrição.
Descrição.	Descrição.	Descrição.

Figura 39: Notas de Entrega de Estórias de Usuários que constam na entrega resultado do Spin.

6.7. Práticas do X-PRO

Uma das premissas do escopo proposto pelo framework do X-PRO é contemplar as melhores práticas previstas nas metodologias ágeis de desenvolvimento de software. O conceito do que é uma prática para o X-PRO consiste em um método para execução de uma atividade. As práticas adotadas por X-PRO foram selecionadas para compor o seu escopo de execução, ou seja, constituem parte fundamental do framework. São práticas previstas no X-PRO:

- **Estórias de Usuário**
  Prática aplicada na atividade de Visão

  Os requisitos do cliente são detalhados em forma de Estórias de Usuário, que consistem em uma ou mais sentenças para descrever em linguagem de negócio o que o usuário deseja ou precisa para o sistema, como parte das suas funções de trabalho (NORTH, 2007). As sentenças são descritas seguindo uma estrutura padrão: Eu como [Função], desejo [descrição da necessidade], de forma a [descrição dos ganhos obtidos pela implementação dessa estória].” Em X-PRO, a prática de Estórias de Usuário é adotada na execução da atividade de Visão, especificamente na produção do artefato de Matriz de Estórias de Usuário (Matriz de Estórias de Usuário).

- **Agile Modeling**
  Prática aplicada nas atividades de Design de Arquitetura e Design de Software

  Conforme apresentado na Seção 3.12.1, a ideia central de Agile Modeling é encorajar a produção de modelos suficientes para suportar a compreensão de problemas de design e propósitos de documentação, porém, mantendo a quantidade de modelos produzidos a mais baixa possível (ABRAHAMSSON ET AL., 2002). Esse conceito está totalmente alinhado com a proposta do X-PRO para os esforços relacionados a design e documentação: ainda que sejam fundamentais e necessários, modelos de análise e design precisam ser objetivos e essenciais. A pesquisa descrita no Capítulo 5 deste trabalho direciona quais os modelos o framework do X-PRO considera como necessários de serem produzidos nas atividades de Design de Arquitetura (Diagrama de Implantação) e Design de Software (Diagrama de Classes e Modelo de Dados – ER).

  Ainda que não tenha como propósito principal ser apenas um conjunto de práticas e princípios, o foco de Agile Modeling é garantir que as atividades de Análise e Design de um projeto de software sejam garantidas – haja vista que algumas metodologias ágeis não preveem essas atividades em seus processos (ABRAHAMSSON ET AL., 2002) – porém, sejam executadas de uma forma alinhada aos princípios e valores do desenvolvimento ágil de software. Nesse sentido, sugerem-se detalhamentos para a prática de Agile Modeling na execução de um projeto baseado em X-PRO:

  - As atividades de modelagem executadas nas atividades de Design de Arquitetura e Design de Software devem ser realizadas em conjunto por pelo menos dois membros do time de desenvolvimento, de forma a possibilitar a troca de percepções e entendimentos a cerca da análise e design do software;
  - Deve-se priorizar o uso de design à mão livre, sem o uso de ferramentas de software para produção dos diagramas. Só após um entendimento conjunto entre os participantes das sessões de análise e design, o modelo então desenhado à mão deve ser diagramado em uma ferramenta;
  - Não se deve buscar um modelo ideal de análise e design do software. Como parte de um processo iterativo e incremental, a prática de modelagem deve ser interpretada como evolutiva, de forma que pode representar uma ilustração mais simplória nos primeiros spins e detalhada à medida em que os spins são executados.

- **MoSCoW Prioritization**
  Prática aplicada na atividade de Visão e Plano de Spin
  A prática de MoSCoW[23] Prioritization (do português, Priorização MoSCoW) descreve um método para priorização de requisitos. O nome desse método é composto pelas iniciais de cada tipo de priorização, em inglês: Must (Precisa), Should (Deve), Could (Pode) e Would (Poderia) (ou Won't como citam muitos autores). Este método possibilita maior clareza quanto a o que é essencial para o projeto e é executado durante a atividade de Visão, sendo considerado tanto no artefato de Canvas de Visão, o qual estabelece os aspectos gerais do projeto; como também durante a elicitação dos requisitos e estórias de usuário para produção do artefato de Matriz de Estórias de Usuário. Ainda que não seja executada diretamente na atividade de Plano de Spin, a

[23] IIBA, International Institute of Business Analysis. A Guide to the Business Analysis Body of Knowledge, 2009.

priorização MoSCoW é referenciada nessa atividade, haja vista que há a definição de quais estórias irão compor um spin, podendo sua classificação MoSCoW influenciar no caso de ser necessário definir qual estória seria priorizada na situação de um problema que inviabilizasse o atendimento pleno as metas de um spin.

- **Behavior-Driven Development**
  Prática aplicada na atividade de Codificação

  A prática de Behavior-Driven Development, apresentada na Seção 3.10 deste trabalho, consiste em uma metodologia ágil de desenvolvimento de software, cuja fundamentação está baseada em Test-Driven Development, a qual prevê que o desenvolvimento da aplicação seja orientado pela escrita de testes de unidade (HELLENSOY & WYNNE, 2012). Essa metodologia considera que os esforços de desenvolvimento e testes não podem ser vistos de forma separada. Em BDD, resolve-se o problema fundamental de TDD, onde considera-se diversos ambientes de projeto, é possível haver a confusão quanto ao que testar, o que não testar, quando testar e como compreender que um teste efetivamente falhou (NORTH, 2006). Nesse sentido, BDD utiliza o conceito de Comportamento (Behavior), o qual nada mais é do que a descrição textual de um Teste de Unidade.

  A adoção de BDD em X-PRO como prática é justificada pois o framework tem como premissa ser o mais eficiente possível. Nesse sentido, em vez de se prever uma atividade de Desenvolvimento e outra para Testes, pensando na eficiência, X-PRO adota BDD como prática para os esforços de codificação. Considera-se que com o uso de BDD, não se fará necessária a execução de uma atividade específica para testes, exceto os Testes de Aceitação (Acceptance Test), a serem descritos a seguir. Conceitualmente, um teste é realizado para verificar o não atendimento a um requisito, o que por consequência geraria um erro – *bug* – na aplicação. Com a adoção de BDD, ao escrever um teste de unidade para um comportamento específico, e fazendo este teste passar – por meio da implementação do seu código – se satisfaz o requisito inicial. Outro ponto que exclui a necessidade de uma disciplina formal de testes é que com BDD, cada parte do requisito – comportamento – possui ao menos um teste de unidade implementado, o que é logicamente mais prático de se pensar, em detrimento de uma abordagem tradicional de testes, onde seria implementado um teste de unidade que pode validar múltiplos requisitos.

- **Integração Contínua**
  Prática aplicada na atividade de Codificação

  A prática de Integração Contínua, herdada de Extreme Programming, consiste em "*uma prática de desenvolvimento de software onde os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia*[24]". Em X-PRO, a prática de integração contínua é realizada durante a atividade de Codificação, porém não é prescrita formalmente, cabendo aos desenvolvedores acordarem como melhor realizarem suas integrações. X-PRO sugere, porém, que sejam realizadas ao menos duas vezes por semana sessões de integração entre os trabalhos de desenvolvedores, devendo obrigatoriamente até o final do spin, todo o trabalho previsto para ele estar

[24] FOWLER, 2006, Tradução do Autor, disponível em: http://martinfowler.com/articles/continuousIntegration.html.

devidamente integrado. Nesse sentido, sempre deve haver na atividade de Plano de Spin, a previsão de execução de sessões de integração contínua, de forma a garantir a execução dessa prática.

- **Service-Oriented Architecture**
  Prática aplicada nas atividades de Design de Arquitetura, Design de Software e Codificação

  O conceito de Service-Oriented Architeture (do português Arquitetura Orientada a Serviços) consiste em um estilo de arquitetura de software cujo princípio roga que as funcionalidades sejam implementadas na forma de serviços[25]. As implemenações desses serviços – componentes do software – são disponibilizadas por meio de interfaces (fronteiras de comunicação entre duas entidades) e são acessíveis por meio de *web services*[26]. A arquitetura SOA baseia-se nos princípios de computação distribuída e utiliza um paradigma de *request/reply* (requisição/resposta) para estabelecer comunicação entre sistemas clientes e os sistemas que implementam os serviços[27]. A proposta de adoção de SOA como referência arquitetural para projetos baseados em X-PRO está relacionada ao fato de que o processo propõe uma visão onde o software é por natureza enxergado de forma modular – requisitos, estórias e comportamentos. Nesse sentido, entende-se que a adoção de uma arquitetura baseada em serviços favorecerá o desenvolvimento modular da aplicação, além de reforçar a premissa fundamental do framework que é a eficiência, uma vez que serviços cujas implementações forem finalizadas podem ser disponibilizados aos clientes ou mesmo acessados por outras aplicações. Essa prática igualmente possui uma relação complementar com as práticas de Integração Contínua e RTC (Relevance Tracking Component), igualmente previstas no framework do X-PRO.

- **User Acceptance Test**
  Prática aplicada na atividade de Entrega

  A prática de User Acceptance Test (do português Testes de Aceitação de Usuário) objetiva verificar se os requisitos compreendidos pelos analistas realmente atendem aos requisitos do cliente. Trata-se de uma rodada de testes realizada pelo próprio usuário durante a atividade de Entrega e é realizada em nível de Estória de Usuário – ou seja, cada estória será individualmente testada pelo usuário. Assim como no caso da prática de Integração Contínua, X-PRO propõe que na atividade de Plano de Spin, seja prevista uma sessão de User Acceptance Test, para que a implementação da referida estória possa ser efetivamente dada como concluída.

---

25 SOA Working Group of The Open Group. Disponível em: http://opengroup.org/projects/soa/doc.tpl?gdid=10632, acessado em 9 de Fevereiro de 2014.

26 Web service é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis. Os Web services são componentes que permitem às aplicações enviar e receber dados em formato XML. Cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, o formato XML. Fonte: Verbete "Web Service" na Wikipedia, disponível em: http://pt.wikipedia.org/wiki/Web_service, acessado em 9 de Janeiro de 2014.

27 Raghu R. Kodali. What is service-oriented architecture? Disponível em: http://www.javaworld.com/javaworld/jw-06-2005/jw-0613-soa.html, acessado em 9 de Janeiro de 2014.

Outras Práticas propostas por X-PRO

Além da herança de práticas previstas em outras metodologias ágeis de desenvolvimento de software, o framework do X-PRO propõem a adoção de práticas até então não previstas em outras metodologias ágeis.

- **Imersão de Negócio**
  Prática aplicada em qualquer atividade do X-PRO

  Times de desenvolvimento invariavelmente recebem novos membros, que eventualmente não conhecem das regras de negócio nem tampouco os processos que são suportados pelo software que este novo membro passará a manter ou desenvolver. Como X-PRO não prevê uma documentação detalhada dos processos de negócio, sugere-se a prática de Imersão de Negócio. Essa prática consiste em sessões lideradas por membros mais experientes do time, os quais se utilizam dos artefatos produzidos no X-PRO – Canvas de Visão, Matriz de Estórias de Usuário, Modelo Arquitetural e Modelo de Software e Mockup de Interface – para instruir e capacitar os novos membros sobre as variáveis de contexto que compõem o universo em que o software está inserido. Essa prática reforça o valor de Unidade do time, onde já a partir da entrada do novo membro no time, cria-se uma atmosfera que favoreça a interação entre os membros e principalmente, torne a experiência de conhecer o software mais atrativa para o novo membro do grupo.

- **Reporte Diário**
  Prática aplicada nas atividades do Spin de Desenvolvimento

  Em Scrum, o acompanhamento contínuo das atividades do time de projeto é realizado através das *Daily Scrum Meetings*, as quais tratam-se de reuniões realizadas diariamente pelo time, onde cada membro reporta o que fez, o que pretende fazer naquele dia e se há algum impedimento para realizar a tarefa (SOMMERVILLE 2009). Entretanto, conforme observado por Akif e Majeed (2012), uma das críticas às práticas do Scrum é a realização excessiva de cerimônias e reuniões, o que eventualmente tira o foco e concentração do time, além de reduzir o tempo útil de desenvolvimento, haja vista que o tempo da iteração é consumido por reuniões.

  Objetivando tratar esse aspecto do excesso de formalidades e reuniões para acompanhamento do time, X-PRO propõe o *Reporte Diário*. Trata-se de um feedback diário fornecido por cada membro do time, reportando o que fez no dia anterior, o que pretende fazer naquele dia e se há algum empecilho. O reporte pode ser feito por meio de qualquer recurso de comunicação que formalize o feedback, seja e-mail ou mesmo uso de alguma ferramenta de apoio. O objetivo do Reporte Diário é prover um mecanismo em que o time aplique o valor de Antecipação proposto por X-PRO, ou seja, proativamente forneça feedback aos agentes envolvidos no projeto.

- **RTC (*Relevance Tracking Component*)**
  Prática aplicada na atividade de Codificação

  Existem duas variáveis de extrema importância em qualquer processo de software, as quais dizem respeito a como uma mudança repercute em outros pontos da aplicação, e consequentemente, quais as práticas adotadas para realizar uma análise prévia de

impacto (IEEE, 2004). Esse contexto é um dos principais motivadores para a concepção da disciplina de Gestão de Mudanças como parte da prática padrão de Engenharia de Software (SOMMERVILE, 2009; IEEE, 2004). É notório, contudo, ressaltar que o processo de gestão de mudanças requer o estabelecimento de diversos aspectos de controle, de forma a garantir que esse seja realizado de forma eficaz, tais como: registro e gestão das requisições de mudança, análise dos requisitos para verificação da sua interação com outros requisitos do sistema, entre outras atividades (SOMMERVILE, 2009; IEEE, 2004). Todavia, por serem essas práticas tradicionais mais formais e demandantes de tarefas que eventualmente conflitam com os valores, princípios e práticas do X-PRO, compreende-se a necessidade de um mecanismo mais alinhado com a premissa de eficiência do X-PRO.

Nesse sentido, X-PRO sugere uma prática intitulada Relevance Tracking Component (RTC). O RTC consiste na proposta de implementação de um componente, o qual deve ser parte de qualquer software produzido com X-PRO. O objetivo desse componente é registrar e catalogar o consumo dos principais – ou mesmo todos, se assim desejar o time – serviços (*services*, herdado do conceito de SOA) do software. Em administração de sistemas, é comum o surgimento de demandas que requeiram a mudança de algum componente do sistema (SOMMERVILE, 2009). Pela adoção de Behavior-Driven Development como abordagem de implementação, é fato que qualquer alteração no sistema que mudasse algum comportamento do software acarretaria na falha de testes de unidade implementados para validar esses comportamentos (HELLESOY & WYNNE, 2012). Entretanto, exceto no caso de erros de compilação, o software poderia ser colocado em produção mesmo contendo erros em testes de unidade. Como então garantir que uma mudança não repercutirá negativamente em outras partes do sistema? A abordagem tradicional em outros processos de software sugere o uso de Rastreabilidade de Requisitos, onde por meio de uma ferramenta CASE (*Computer-Aided Software Engineering*), seria verificado o impacto de alterações entre requisitos (SOMMERVILE, 2009).

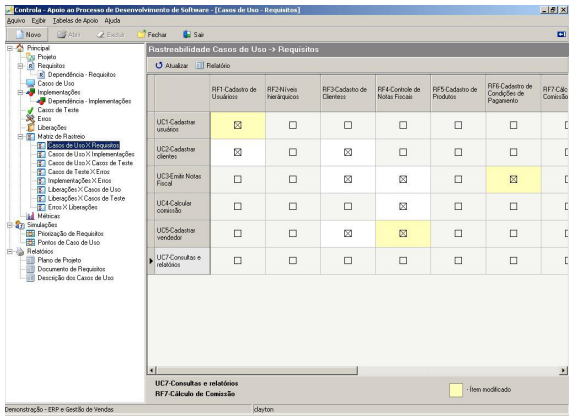


Figura 40: Ferramenta CASE intitulada “Controla”, com exemplo de matriz de rastreabilidade de requisitos.

Compreende-se, todavia, que a adoção de uma prática como Rastreabilidade de Requisitos para avaliação de impacto de mudanças tende a não ser algo ágil, haja vista a natural necessidade de revisão do requisito, acesso à ferramenta, avaliação da integração entre os requisitos, entre outros fatores. E ainda assim, mesmo compreendendo que um determinado requisito é integrado com diversos outros, não necessariamente esses requisitos são de uso frequente por parte dos usuários do software. Ou seja, ainda que relevantes para o software, esses requisitos podem ter

uma utilização reduzida pelos clientes da aplicação. Para X-PRO, a eficiência precisa ser mantida como premissa fundamental, nesse sentido, o real valor a ser questionado seria: "*Considerando que uma alteração em um dado serviço poderia afetar um outro, qual a frequência de uso desse outro serviço? Ele é efetivamente relevante para o cliente?*"

A proposta da implementação do RTC é responder essa pergunta. O conceito de RTC é herdado das práticas de *billing* (do português "bilhetagem", mas que conceitualmente quer dizer processo de cobrança) em Computação em Nuvem. Conforme apresentado por Petterson (2011), o processo de cobrança por consumo de uma aplicação ofertada em um ambiente de Computação em Nuvem como SaaS (*Software as a Service*, do português Software como Serviço), pode ser realizado baseado na medição de transações realizada em determinados componentes do software. Petterson (2011) exemplifica o caso do desenvolvimento de uma aplicação orientada a serviços, onde essa precisa ser efetiva tanto no que diz respeito a compreender qual a demanda por cada serviço – podendo assim definir estratégia de escalabilidade – bem como cobrar pelo consumo de cada serviço específico. A sugestão de Petterson é monitorar cada requisição ao serviço, gerando por consequência o volume de requisições feitas a cada serviço.

> "Como os recursos - sejam eles de hardware do servidor, uma solicitação de banco de dados, um pedido de fila de mensagens, ou serviços de monitoramento - são cobrados com base no uso real, você deve incluir um ID para a transação consumidora em cada invocação de recursos. Por exemplo, se você chamar um serviço para obter dados do banco de dados, a solicitação HTTP associada deve incluir o ID da transação, bem como a identificação do consumidor para posterior correlação dessas métricas. Claro, você deve ter um segmento adicional no aplicativo para capturar dados de correlação de transação de modo que nem o desempenho da transação núcleo nem o tempo de resposta seja afetado." (PETTERSON, 2011, pág. 2, Tradução do Autor)

A Figura 43 mostra um exemplo de uma transação, que inclui os diferentes serviços SOA e utiliza um ID de transação. Agentes são implantados em todos os nós para capturar dados da transação para cada transação. Neste caso, o código "t1234" é o ID que identifica a transação, onde cada serviço amarra o tempo decorrido da CPU para o ID de transação para posterior medição.

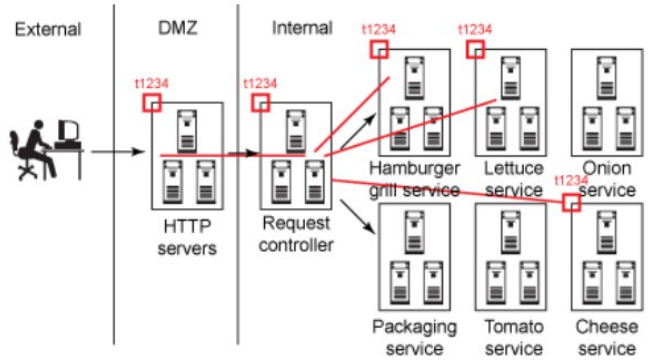


Figura 41: Exemplo de transação utilizando serviços e transações SOA (PETTERSON, 2011).

Para X-PRO, a implementação sugerida por Petterson (2011) é a proposta do Relevance Tracking Component para apoiar o processo decisório quanto ao impacto de uma mudança não do ponto de vista da aplicação, mas do ponto de vista do cliente. Considerando um cenário exemplo, onde um software com um RTC implementado apresentou um problema ou precisa de uma alteração de urgência. Com os métodos tradicionais – como rastreabilidade de requisitos – o desenvolvedor verificaria que há uma série de chamadas aquele serviço, o levaria a ter receio quanto a alguma mudança. Com o uso do RTC, bastaria ao desenvolvedor verificar o histórico – ranking – de consumo de serviços da aplicação por parte dos usuários. Se a mudança que está sendo realizada trata de um serviço com alto volume de consumo, consequentemente é compreendido que a mudança pode acarretar em um problema para a operação do software. Entretanto, caso o serviço tenha um consumo baixo, o desenvolvedor pode optar por realizar a mudança, mesmo que essa acarrete no comprometimento de outro serviço, mas que pela baixa utilização, não seria prioritário do ponto de vista do cliente. Em suma, para X-PRO, a avaliação de impacto de mudanças pode ser feita de duas maneiras: a primeira com a verificação de quais testes de unidade deixam de passar após a implementação da mudança, e a segunda por meio da consulta ao ranking de consumo de serviços da aplicação através dos dados coletados pelo RTC.

6.8. Considerações Finais

A proposta do modelo X-PRO objetiva a concepção de uma nova abordagem de processo de software, o qual estende os conceitos e práticas das metodologias ágeis de desenvolvimento de software para uma abordagem de processo eficiente. O conceito de eficiência proposto pelo X-PRO está diretamente relacionado ao que é efetivamente fundamental existir – e que efetivamente existe, na prática, conforme avaliação de estudos empíricos – em qualquer projeto de software. A concepção do framework do X-PRO se deu através da revisão da literatura a cerca das metodologias ágeis de software, no sentido de identificar suas características, escopo e gaps; e através da aplicação de pesquisa com um grupo de desenvolvedores, conforme apresentado no Capítulo 5.

7. APLICAÇÃO DO X-PRO

Neste capítulo é apresentada a aplicação do framework proposto por X-PRO, o qual foi adotado na prática em um projeto de desenvolvimento de software de uma fábrica de software brasileira com sede em Recife/PE. O objetivo do estudo foi avaliar a execução do modelo, bem como as assertivas estruturais propostas neste trabalho.

7.1. Introdução

Com objetivo de avaliar a consistência, aplicabilidade e principalmente a viabilidade do modelo proposto por X-PRO, o framework proposto foi adotado em um cenário de projeto de desenvolvimento de uma fábrica de software brasileira com sede em Recife/PE. O contexto de avaliação consiste no instanciamento do X-PRO em um projeto de médio porte, o qual teve o propósito de avaliar as assertivas estruturais do modelo proposto por este trabalho. Será descrita a abordagem utilizada para execução do cenário proposto, a qual é apresentada em maiores detalhes, assim como o processo de execução do modelo e os resultados encontrados.

Abordagem Utilizada na Aplicação do X-PRO

O cenário considerado para avaliação do modelo se deu através da aplicação do método de Estudo de Caso, com a adoção do framework proposto por X-PRO como processo de software em um projeto de desenvolvimento de uma fábrica de software brasileira. A construção desse estudo foi composta pelas seguintes etapas:

- **1ª Etapa – Seleção da Organização para Aplicação do X-PRO:** Inicialmente foi estabelecida a necessidade de um contexto organizacional onde o X-PRO pudesse ser demandado em toda a sua totalidade, haja vista a necessidade de validar todos os aspectos relacionados à sua estrutura. Nesse sentido, a disciplina de Engenharia de Software deveria fazer parte de forma intensa da organização escolhida como estudo de caso de aplicação do X-PRO. Nesse sentido, foi identificada uma fábrica de software brasileira de múltiplos produtos, com sede em Recife/PE, a qual foi convidada para o estudo dada a sua característica operacional poder validar todas as variáveis contidas no modelo proposto nesse trabalho;

- **2ª Etapa – Definição do Projeto para Adoção do X-PRO:** Após a definição da organização, essa deveria indicar um projeto de médio a grande porte no qual o X-PRO pudesse ser adotado como processo de software do início ao fim do projeto. Nesse sentido, a organização indicou um projeto de desenvolvimento de uma plataforma (software) de integração entre múltiplos sistemas de informação de um dos seus clientes;

- **3ª Etapa – Capacitação do Time de Projeto no Escopo do X-PRO:** Definido o projeto a adotar o X-PRO como processo, foi realizada a capacitação do time quanto aos conceitos, estrutura, características e demais variáveis relacionadas ao modelo proposto neste trabalho;

- **4ª Etapa – Execução do Projeto com a aplicação do X-PRO:** Foram realizados os testes, acompanhamentos e monitoramento da aplicação prática do X-PRO durante o período de realização do projeto, o qual teve uma duração de 3 (três) meses;

- **5ª Etapa – Avaliação dos Resultados:** Com os dados extraídos e coletados da execução do projeto com a aplicação do X-PRO, foram analisados os resultados encontrados com o uso do cenário exemplo a partir do processo proposto por este trabalho. Essa análise constitue em determinar se a aplicação do X-PRO e assertivas estruturais propostas realmente garantiram a resolução dos problemas de pesquisa levantados por este trabalho.

7.2. Cenário de Execução

Para a realização do estudo aplicado do X-PRO, foi realizada uma análise dos perfis de organizações que poderiam contribuir no sentido de explorar toda a estrutura e variáveis de contexto possíveis para validar o modelo proposto por este trabalho. Baseado nessa premissa, considerou-se que o cenário ideal para execução plena do X-PRO seria uma organização diretamente envolvida com desenvolvimento de software, ainda que isso não configure nenhuma exigência quanto a limitações do X-PRO para outros contextos, como áreas de Sistemas de empresas que não são de TI, sendo, porém, o objetivo dessa premissa apenas relacionado à possibilidade de exploração plena do modelo proposto por este trabalho.

Elaborou-se uma apresentação da proposta do estudo, bem como uma apresentação geral do X-PRO e esse material foi encaminhado a três organizações diferentes, cuja principal atuação é como fábrica de software. Uma das organizações era de pequeno porte, com um quadro funcional de 20 funcionários e desenvolvimento de 2 produtos de software próprios em seu portfólio; outra era de médio porte, com um quadro funcional de 120 funcionários e desenvolvimento como fábrica de software terceirizada de 8 produtos de software de diferentes clientes; e por fim uma organização de grande porte, com um quadro funcional de mais de 400 funcionários e desenvolvimento de um ERP (Enterprise Resource Planning, em português conhecido como Sistema de Gestão Integrada) com uma carteira de aproximadamente 100 clientes.

Ainda que as organizações convidadas a participar do estudo atuem com desenvolvimento de software, identificou-se que a organização de médio porte era a mais indicada, haja vista que além dessa possuir um cenário de diferentes contextos de produtos de software – dada a sua atuação com terceirização – percebeu-se que essa dinamicidade do seu ambiente operacional poderia contribuir com o propósito do estudo de validar e explorar plenamente a estrutura e variáveis de contexto do X-PRO. Por solicitação da organização participante do estudo, sua identidade foi mantida em sigilo, de forma que este trabalho faz referência a seu nome como Software Company. Ainda que seu nome seja fictício pelo motivo explanado acima, todos dados, informações, características e demais variáveis mencionadas a seguir são verídicas e de fato refletem a realidade da organização participante no estudo.

Dados da Organização Participante

- **Nome:** Software Company;
- **Atuação:** Terceirização de Fábrica de Software, Fábrica de Testes, Educação e Capacitação;
- **Empresa Pública ou Privada?** Privada;

- **Sede:** Recife/PE;
- **Funcionários:** 120 pessoas;
- **Funcionários diretamente envolvidos com a Fábrica de Software:** 98;
- **Número de Clientes:** 7;
- **Número de Produtos:** 8;
- **Se possui certificação no Processo de Desenvolvimento:** Não;
- **Se possui processo de software definido:** Sim, sendo baseado em Rational Unified Process (RUP), porém com a prática de gestão de projeto baseada no Scrum;
- **Se adota alguma prática ágil de desenvolvimento de software:** Sim, em novos projetos passou a adotar as práticas de Test-Driven Development e Programação em Pares, sendo a segunda utilizada para capacitação de novos desenvolvedores em frameworks internos.

Dados do Projeto Selecionado

Definida a organização participante do estudo, passou-se a analisar o projeto no qual o X-PRO seria aplicado como processo de software. Por sugestão da organização participante em conjunto com o autor deste trabalho, foi escolhido um projeto de desenvolvimento de uma plataforma (software) de integração entre os múltiplos sistemas de informação de um dos clientes da fábrica de software. A seguir são listados os dados deste projeto:

- **Objetivo do Projeto:** Desenvolvimento de uma plataforma (software) de integração entre os múltiplos sistemas de informação de um cliente da fábrica de software;
- **Quantidade de Funcionários da Fábrica de Software Envolvidos:** 12;
- **Quantidade de Funcionários do Cliente Envolvidos:** 4;
- **Tempo Estimado do Projeto:** 3 meses;
- **Início do Projeto:** Novembro de 2013;
- **Término do Projeto:** Janeiro de 2014;
- **Tecnologia Adotada:** Java.

7.3. Execução do Projeto com a Aplicação do X-PRO

Conforme apresentado na seção anterior, a escolha do projeto foi realizada em comum acordo entre as lideranças da empresa Software Company e o autor deste trabalho, no sentido da possibilidade de exploração do X-PRO no contexto do projeto selecionado.

Na Software Company, o início técnico de um projeto é realizado com a definição de um Gerente de Projeto, o qual assumiria o papel de Mestre X, que em X-PRO é responsável por gerir tanto o projeto como garantir que as definições do processo são seguidas conforme estabelecido. Nesse sentido, fez-se necessário realizar uma capacitação individual para o referido papel, de forma que esse pudesse dominar o framework sem a necessidade de intervenção direta do autor do trabalho, garantindo assim o máximo de naturalidade à execução do projeto.

Após a atribuição do Mestre X, foi definida a escolha do time, a qual foi realizada sem nenhuma influência direta pela iniciativa de execução do X-PRO, tendo essa seleção dos participantes sido feita conforme padrão da própria organização, a qual considera: requisitos técnicos dos membros do time; mescla entre profissionais mais e menos experientes; conhecimento ou experiência prévia sobre o objeto o projeto; disponibilidade de agenda. Em se tratando da qualificação dos participantes da pesquisa, 85% possuem nível superior em

cursos relacionados a Ciência da Computação e 15% estão com graduação superior em andamento – igualmente em áreas relacionadas a Computação.

Capacitação do Time de Projeto no X-PRO

Com o estabelecimento do time do projeto, o Mestre X realizou uma reunião de abertura do projeto, trâmite costumeiro no processo da Software Company. Conforme havia sido acertado previamente com o autor deste trabalho, a equipe do projeto não deveria ser comunicada quanto ao fato de que o X-PRO tratava-se de um modelo em estudo, mas sim, de um processo de software novo que a organização estava implantando. O propósito dessa definição objetivou não induzir a equipe do projeto a ter no processo o principal foco, mas sim, seguir sua dinâmica costumeira que objetiva a entrega do produto do projeto.

O Mestre X apresentou ao time que o X-PRO tratava-se de um novo processo de desenvolvimento de software que a Software Company passaria a adotar, o qual seguia a tendência das iniciativas mais recentes da organização no sentido de ampliar o uso de práticas ágeis, objetivando aprimorar seu processo de desenvolvimento. O autor deste trabalho esteve presente na reunião de abertura do projeto, tendo sido apresentado como um consultor e mentor na prática do processo que a Software Company passaria a adotar, bem como ficaria responsável por ministrar o treinamento e capacitação do time. Foi comunicado ao time que ao longo do projeto seriam realizadas algumas avaliações no sentido de identificar os resultados alcançados a partir da adoção do X-PRO, bem como seria realizada uma avaliação ao final do projeto para identificar a percepção de todos os membros do time.

A capacitação do time ocorreu durante o período de uma semana, sendo realizada de segunda a sexta-feira no período da manhã, contando com a participação de todos os membros do time de projeto – no caso, os colaboradores da Software Company, sendo 12 no total. Ao final do treinamento foi realizada uma avaliação de 10 (dez) questões, onde cada questão valia 1 (um) ponto e tinha média 7 (sete) para que o colaborador fosse considerado apto a trabalhar sob a prática do modelo proposto por X-PRO. A média geral da turma foi de 9,125, sendo a menor nota 8,0 e a maior 10,0, onde 41,67% da turma teve nota máxima (10,0).

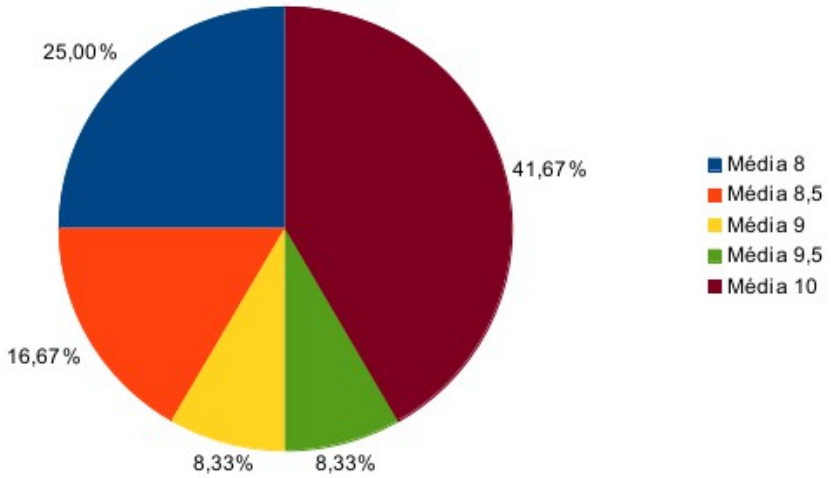


Figura 42: Gráfico das Notas da Avaliação da Capacitação no framework do X-PRO para o time executor do projeto prático de aplicação do modelo.

Execução do Projeto

Concluída a fase de capacitação do time envolvido com o projeto, o projeto foi realizado conforme planejado. Estabeleceu-se que ao final de cada spin, seria realizada uma avaliação com o time do projeto, onde esses deveriam avaliar por meio de atribuição de notas, quais foram suas percepções a cerca de toda a estrutura prevista no framework do X-PRO – valores, princípios, ciclo de vida, atividades, papéis, artefatos e práticas – porém considerando apenas a experiência contida naquele spin – haja vista que uma observação realizada em uma spin não necessariamente seria válida para outro.

Dado o objetivo da avaliação estar relacionado a execução do processo do X-PRO, essa foi realizada apenas com os membros internos do time de projeto, excluindo-se os membros do cliente – Patrocinador e Clientes Principais. Para realização da avaliação, foi considerada uma escala que seguiu a seguinte ordem de atribuição: 0 – Totalmente Insatisfeito; 1 – Insatisfeito; 2 – Pouco Satisfeito; 3 – Satisfeito; 4 – Bastante Satisfeito; 5 – Totalmente Satisfeito. Para os casos em que as notas atribuídas pelos avaliadores fosse entre os dois primeiros e o último nível (0, 1 e 5), obrigatoriamente o participante deveria relatar com um comentário o porque da sua avaliação ter sido aquela. O objetivo foi compreender onde o modelo não atingia seus objetivos e onde que ele superava as expectativas, o que possibilitou validar a proposta de resolução do problema de pesquisa abordado por este trabalho. Após a entrega e finalização do projeto, foi aplicado o mesmo questionário, porém considerando todo o projeto, de forma a identificar dos participantes as suas percepções quanto a aspectos negativos e positivos do modelo. O projeto teve sua Linha do Tempo estabelecida em 4 spins, 3 entregas, 1 versão beta e 1 versão final. Conforme planejamento, o spin foi realizado ao longo de 3 meses.

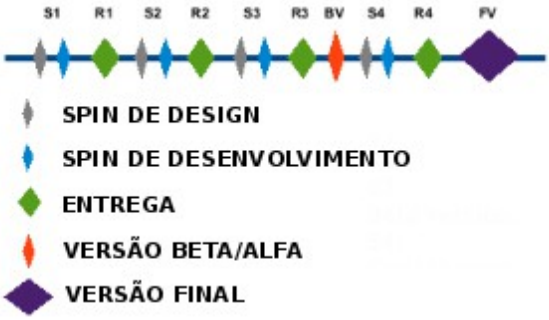


Figura 43: Linha do Tempo do Projeto cujo X-PRO foi aplicado.

Quanto à estrutura de papéis e responsabilidades do projeto, essa foi estabelecida na reunião de abertura, conforme prática já realizada pela Software Company. O time do projeto foi estabelecido conforme a seguir:

- **Mestre X**. Quantidade: 1 (Membro da Software Company);
- **Patrocinador**. Quantidade: 1 (Membro do Cliente);
- **Clientes Principais**. Quantidade: 3 (Membro do Cliente);
- **Analista de Negócios**. Quantidade: 3 (Membro do Cliente);
- **Desenvolvedores**. Quantidade: 8 (Membro do Cliente).

Estabeleceu-se que toda a estrutura do X-PRO deveria ser adotada e executada naquele projeto, haja vista que havia o propósito de identificar se o processo proposto por X-PRO estava aderente as características da Software Company. Entretanto, foi salientado aos

participantes que essa determinação se resumia àquele primeiro projeto, haja vista que o X-PRO não é prescritivo quanto à estrutura do framework, exceto quanto à aplicação dos seus valores e princípios, de forma que outros projetos poderiam adotar plena ou parcialmente o modelo.

7.4. Análise dos Resultados obtidos na Avaliação do X-PRO

A avaliação realizada para identificar a aplicação prática do X-PRO se deu através do acompanhamento junto à equipe do projeto, tendo seus dados de execução sido coletados ao longo de todo o ciclo de desenvolvimento do projeto – que conforme apresentado na seção anterior, ocorreu em 4 spins. Ao final de cada spin, o questionário foi aplicado a fim de identificar a percepção dos participantes quanto a cada um dos componentes do framework do X-PRO. O primeiro resultado verificado é que não foram identificadas variações da percepção dos envolvidos ao longo dos spins, o que leva a crer que a capacitação quanto à estrutura do framework realizada antes do início do projeto foi suficiente para formar a opinião dos participantes a cerca da estrutura do X-PRO.

Conforme estabelecido e explanado na Seção 7.3, apenas as notas classificadas em 0, 1 e 5 (Totalmente Insatisfeito; Insatisfeito; Totalmente Satisfeito, respectivamente) deveriam contemplar comentários por parte dos participantes, haja vista que estariam posicionadas entre os dois extremos do contexto de avaliação. Esses comentários e considerações, porém, serão apresentados na Seção 7.5 deste trabalho.

No que diz respeito à percepção dos participantes a cerca dos Valores e Princípios previstos no X-PRO, conforme apresentado na tabela 16, os Valores do X-PRO obtiveram uma média geral de 3,93 (entre Satisfeito e Bastante Satisfeito), sendo a menor média 3,66 para o valor de Antecipação e a maior para o valor de Autonomia, com média de 4,25. Dado que a maior parte dos participantes da pesquisa eram desenvolvedores, o posicionamento dos valores de Autonomia e Unidade como destaques pode levar a conclusão de que esses são percebidos como fundamentais principalmente para os diretamente envolvidos com esforços de desenvolvimento.

Quanto aos Princípios, o destaque da avaliação ficou com o princípio de "*Ágil, mas com ordem*", o que confirma o entendimento de equipes de projetos de desenvolvimento que o fato de se adotar uma metodologia ágil não está relacionada à ausência completa de alguma estrutura lógica no ciclo de vida do processo de desenvolvimento de software, dada a característica de adaptabilidade dessas metodologias; e o princípio "*Design e documentação das regras de negócio são ativos de conhecimento*", o que confirma a percepção de importância de design e documentação, este que é um dos problemas de pesquisa cujo X-PRO objetivou tratar: a ausência ou subestimação de esforços de design e documentação em metodologias ágeis de desenvolvimento de software.

		Média Geral
Valores	Autonomia	4,25
	Unidade	4,00
	Foco	3,83
	Respaldo	3,91
	Antecipação	3,66
	Média Geral dos Valores	***3,93***
Princípios	Ágil, mas com ordem	4,5
	As mudanças são parte do negócio	3,83
	Design e documentação das regras de negócio são ativos do conhecimento	4,66
	Testes são parte do desenvolvimento	3,66
	Solução sem satisfação não tem valor	3,83
	Entregas constantes, valor abundante	3,91
	Comunicação impessoal é mera formalização	3,66
	Média Geral dos Valores	***4,00***

Tabela 16: Resultados coletados na pesquisa de avaliação do modelo X-PRO quanto aos elementos de Valores e Princípios do framework.

Na avaliação relacionada ao ciclo de vida do X-PRO, conforme apresentado na tabela 17, o destaque positivo ficou por parte da atividade de Visão, a qual foi avaliada na média geral com a nota de 4,5 (entre Bastante Satisfeito e Totalmente Satisfeito).

		Média Geral
Ciclo de Vida - Spins	Spin de Design	4,00
	Spin de Desenvolvimento	4,00
	Objetivos do Spin	3,58
	Duração do Spin	3,75
	Horas de Trabalho do Spin por Semana	3,91
	Média Geral do Ciclo de Vida - Spins	***3,84***
Ciclo de Vida - Atividades	Visão	4,5
	Experiência do Usuário	4,25
	Design de Arquitetura	4,08
	Plano de Spin	4,00
	Design de Software	4,00
	Codificação	3,83
	Entrega	3,58
	Média Geral do Ciclo de Vida - Atividades	***4,03***

Tabela 17: Resultados coletados na pesquisa de avaliação do modelo X-PRO quanto aos elementos de Ciclo de Vida (Spins e Atividades) do framework.

Os Papéis e Artefatos previstos no framework do X-PRO foram avaliados com médias geral de 3,83 (entre Satisfeito e Bastante Satisfeito) e 4,08 (entre Bastante Satisfeito e Extremamente Satisfeito), respectivamente. A maior avaliação nesses quesitos ficou por conta do artefato de Canvas de Visão, o qual foi destacado como extremamente positivo para os participantes do projeto no que diz respeito à sua simplicidade, riqueza de informações e por não se tratar de um documento amplo e extenso para compreensão do que é o projeto.

		Média Geral
Papéis	Mestre X	4,00
	Patrocinador	3,83
	Clientes Principais	3,83
	Analista de Negócios	3,75
	Desenvolvedor	3,75
	Média Geral dos Papéis	***3,83***
Artefatos	Canvas de Visão	4,5
	Matriz de Estórias de Usuário	4,25
	Mockup de Interface	4,25
	Canvas de Tarefas	4,00
	Modelo de Arquitetura	4,00
	Modelo de Software	4,00
	Notas de Entrega	3,58
	Média Geral dos Artefatos	***4,08***

Tabela 18: Resultados coletados na pesquisa de avaliação do modelo X-PRO quanto aos elementos de Papéis e Artefatos do framework.

A avaliação das práticas do X-PRO obtive a classificação média geral de 3,96 (Satisfeito e Bastante Satisfeito), tendo a prática RTC (Relevance Tracking Componente, proposta neste trabalho, obtido a melhor classificação entre as práticas previstas no framework. Avaliações positivas nas práticas de Arquitetura Orientada a Serviços (SOA – Service-Oriented Architecture), Agile Modeling, MoSCoW Prioritization (Priorização MoSCoW) e Integração Contínua, referenciam a representatividade de práticas já consolidadas em outras metodologias ágeis de desenvolvimento de software.

Outra prática bem classificada na avaliação foi a prática de Reporte Diário, a qual faz referência a prática de Daily Spin Meeting do Scrum (SOMMERVILLE, 2009), porém, substituindo a reunião diária por um relatório simples e objetivo encaminhado pelos desenvolvedores ao Mestre X do projeto, de forma que esse possa dispor de um mecanismo de acompanhamento da evolução das tarefas sendo desenvolvidas no spin atual.

		Média Geral
Práticas	Estórias de Usuário	3,66
	Agile Modeling	3,91
	MoSCoW Prioritization	3,91
	Behavior-Driven Development	3,83
	Integração Contínua	3,83
	Service-Oriented Architecture	4,08
	User Acceptance Test	3,91
	Imersão de Negócio	3,91
	Reporte Diário	4,0
	Relevance Tracking Component	4,50
	Média Geral das Práticas	3,96

Tabela 19: Resultados coletados na pesquisa de avaliação do modelo X-PRO quanto aos elementos de Práticas do framework.

Considerando que a estrutura do framework do X-PRO prevê 7 diferentes atributos – Valores, Princípios, Ciclo de Vida – Spins, Ciclo de Vida – Atividades, Papéis, Artefatos e Práticas – a avaliação geral do modelo ficou classificada com a nota de 3,95, o que posiciona os participantes do projeto entre Satisfeitos para Bastante Satisfeitos. Percebeu-se que essa satisfação está associada à proposta híbrida do X-PRO em compilar as principais características positivas de outras metodologias ágeis de desenvolvimento de software, porém, incorporando características complementares que objetivam resolver os principais problemas destas, conforme apresentado como problema de pesquisa deste trabalho.

7.5. Considerações e Comentários dos Participantes da Avaliação

Definiu-se que para a execução da avaliação da aplicação prática do X-PRO, qualquer atributo que fosse avaliado com as escalas de 0, 1 e 5 (Totalmente Insatisfeito; Insatisfeito; Totalmente Satisfeito, respectivamente) deveria contemplar comentários por parte dos participantes, haja vista que estaria posicionado entre os dois extremos do contexto de avaliação. Os comentários e considerações dos foram compilados a seguir, sendo apresentada igualmente a escala atribuída para o referido comentário[28]. Ressalta-se que a avaliação não identificou nenhum comentário ou consideração especificamente sobre os papéis do X-PRO.

- **Valores**
    - O valor de Respaldo é uma importante referência para equipe de projeto, dado que, conforme avaliado pelos participantes, equipes técnicas não costumam ter o hábito de documentar ações intermediárias dentro do projeto, tais como: reuniões, contatos, deliberações verbais, mudanças de prioridade ao longo do tempo, entre outros. Nesse sentido, foi relatada a importância de que todas as ações da equipe precisam estar respaldadas em algum registro formal.

[28]Os comentários dos participantes foram padronizadas com a mesma estrutura textual, haja vista representarem as mesmas visões, ainda que apresentadas de formas distintas.

- O valor de Autonomia foi apontado como fundamental para que o desenvolvedor pudesse conduzir seus trabalhos sem necessitar o tempo todo reportar "como" está fazendo, mas sim, garantindo "o que" está fazendo, no sentido de apresentar ao gestor do projeto que o resultado está sendo alcançado.

- **Princípios**

  - Foi relatado que o fato do X-PRO incorporar características de várias metodologias ágeis, além de propor um framework estruturado porém, não prescritivo, viabilizou um sentimento de que a agilidade fazia parte do processo, mas que havia um processo garantindo a sua execução íntegra.

  - Foi observado que a ressalva do X-PRO quanto às práticas de Design e Documentação foram consideradas importantes, dado que as metodologias ágeis tendem a subestimar esses esforços (AKIF & MAJEED, 2012). Especialmente os esforços de design foram mencionados como um importante ativo de conhecimento para a manutenção do software por parte dos desenvolvedores.

- **Ciclo de Vida – Spins**

  - Foi considerada interessante a abordagem de uma iteração – spin – contemplar esforços iniciais de design e visão de negócio, haja vista que essa característica tende a ser inevitável na execução dos projetos e precisa ser realizada.
  - Foi observada a importância do estabelecimento de horas úteis dentro de um spin, haja vista que para os desenvolvedores que trabalham tanto com esforços de Desenvolvimento como de Manutenção de software, é importante estabelecer períodos em que eles possam se dedicar exclusivamente ao desenvolvimento.

- **Ciclo de Vida – Atividades**

  - A abordagem proposta pela atividade de Visão foi avaliada como positiva, consolidando tanto o entendimento do projeto como os requisitos desejados para o software em uma atividade simples e que produz poucos artefatos.

  - Foi relatada a importância do desenho da interface como algo relevante na especificação, dado que o cliente pode perceber como ficará sua aplicação, fazendo com que ele tenha mais confiança nos requisitos, além de possibilitar a antecipação da identificação de comportamentos necessários para a aplicação.

  - O desenvolvimento orientado a testes / comportamento foi avaliado como positivo no sentido de garantir um produto de maior qualidade, dado a sua característica de considerar a validação dos requisitos durante o desenvolvimento e não como uma atividade à parte. Outro aspecto positivo observado diz respeito ao fato de que o desenvolvimento orientado a testes / comportamento provê um mecanismo mais assertivo de que alterações não comprometam outras partes do sistema, dado que qualquer mudança que reflita no comportamento de algum componente, consequentemente fará com que algum dos seus testes de unidade falhe.

- **Artefatos**

    - A abordagem proposta pelo Canvas de Visão foi avaliada positivamente por vários dos participantes, haja vista este artefato consolidar todas as informações do projeto em um único documento, o qual é apresentado de forma simples e objetiva, facilitando tanto a sua concepção como manutenção.

    - A proposta da Matriz de Estórias de Usuário foi classificada como positiva no sentido de consolidar em uma visão única os requisitos, estórias de usuário e comportamentos do sistema, o que facilita a visualização das funcionalidades gerais do sistema.

- **Práticas**

    - A adoção das práticas de Agile Modeling para os esforços de Design simplificam e agilizam a execução dessas atividades, principalmente pelo fato de ser sugerida a produção de modelos à mão livre, sendo realizada a transferência para uma ferramenta CASE apenas após a estabilização do modelo.

    - O uso de SOA (Service-Oriented Architecture) como prática indicada pelo framework incentiva a adoção desse padrão, o qual tem sido largamente adotado, principalmente pela tendência de oferta de software como serviço (Software as a Service) e Computação em Nuvem (Cloud Computing).

    - A prática de Imersão de Negócio foi considerada uma abordagem interessante para apoiar não só no entendimento do contexto técnico e arquitetural do software, como igualmente para estreitar a relação entre os membros do time.

    - Foi relatada que a adoção do Reporte Diário é mais prática e produtiva do que reuniões diárias, que podem consumir tempo em excesso do time. A proposta de um relatório simples e objetivo encaminhado pelos desenvolvedores ao Mestre X do projeto, foi pontuada como positiva como mecanismo de acompanhamento da evolução das tarefas sendo desenvolvidas no spin atual.

    - A prática de RTC foi largamente classificada como importante e inovadora no sentido de auxiliar a equipe de desenvolvimento em duas frentes distintas: 1) entendimento quanto ao consumo de módulos do software por parte do cliente; 2) possibilitar o entendimento do que é relevante para cada cliente, considerando não apenas a visão arquitetural, técnica e dos requisitos – as quais são generalistas e não tratam a especificidade de uso do software por cada cliente do respectivo.

7.6. Considerações Finais

Os resultados encontrados a partir da execução do X-PRO evidenciaram aspectos da aplicação prática do framework, bem como possibilitaram validar as assertivas estruturais propostas por este trabalho. A opção por não revelar ao time do projeto que a aplicação do X-PRO tratava-se de um estudo de caso possibilitou compreender de forma imparcial se as características e estruturas proposta pelo modelo atendem a uma realidade prática de projetos de desenvolvimento de software.

Observou-se que um fator determinante para que a aplicação e execução do X-PRO ocorresse de forma bem sucedida foi dedicar um esforço inicial em capacitar todo o time de projeto no sentido de haver uma compreensão plena da estrutura do modelo, seus objetivos, propostas, ciclo de vida e demais características. Conforme pode ser observado, aproximadamente metade dos participantes avaliou com nota máxima os conceitos do X-PRO, além de que nenhum deles ficou abaixo da média 7,0 na avaliação dos conhecimentos após a etapa de capacitação.

No que diz respeito à execução, aplicação prática e avaliação do X-PRO, pôde-se observar que os participantes atribuíram a avaliação ao modelo como bastante satisfatória, tendo a avaliação atingido a média geral de 3,95, sendo 0,0 a mínima e 5,0 a máxima. A partir desse estudo, considera-se que o X-PRO é um framework de processo de desenvolvimento de software apto para ser aplicado em outros cenários de projetos.

A partir da perspectiva dos problemas de pesquisa abordados por esse trabalho, foi possível constatar que todos foram plenamente cobertos e atendidos pela proposta do framework. O primeiro problema de pesquisa aborda que cada metodologia ágil disponível na literatura objetiva resolver aspectos específicos de iniciativas de desenvolvimento de software. Exemplos dessa abordagem são Test-Driven Development e XP (Extreme Programming), as quais estão mais focadas aos esforços de codificação do que do projeto de software como um todo (BEGEL & NEGAPPAN, 2007). O objetivo era que o framework X-PRO definisse diretrizes e fosse aplicável tanto para os esforços de gestão de projeto quanto de codificação. Conforme proposto no framework do X-PRO, essa característica foi observada pelos participantes da pesquisa na avaliação de aspectos como os Valores e Princípios, mas principalmente, na estrutura do ciclo de vida do X-PRO, bem como através da atividade de Visão.

O segundo problema de pesquisa aborda que buscando maior agilidade nos projetos de software, as metodologias ágeis tendem a negligenciar aspectos relevantes em Engenharia de Software, tais como o Design e Documentação (AKIF & MAJEED, 2012). O framework X-PRO se propôs a abordar em seu escopo as práticas sugeridas por modelagem ágil (Agile Modeling), bem como considerou em sua estrutura a produção e manutenção de artefatos de documentação de software. Essa característica foi igualmente observada na aplicação prática do X-PRO, onde os participantes relataram como positiva a abordagem do framework no que diz respeito a princípios – como o que afirma que "*Design e documentação das regras de negócio são ativos do conhecimento*" – mas principalmente, quanto às atividades de Design de Arquitetura, Design de Software e prática de Agile Modeling.

Por fim, o terceiro problema de pesquisa abordado considera que as metodologias ágeis habitualmente são compreendidas como aplicáveis a projetos de software de pequena e média escala, sendo inclusive desaconselhadas como referência única de processo em projetos de larga escala, com times acima de 40 pessoas (LINDVALL ET AL., 2002). Ainda que essa assertiva não tenha sido validada na aplicação prática do X-PRO, não há no framework nenhuma restrição no sentido de que esse seja adotado em projetos de larga escala ou para times acima de 40 pessoas. Conclui-se, conforme apresentado, que a proposta do X-PRO como modelo eficiente de desenvolvimento de software se mostrou satisfatória no sentido de atender seus objetivos e premissas, bem como resolver os problemas de pesquisa abordados por esse trabalho.

8. CONCLUSÕES

Neste capítulo são apresentadas as considerações finais deste trabalho, descrevendo suas principais contribuições, limitações encontradas, análise quanto a trabalhos relacionados e outros processos híbridos de desenvolvimento de software, bem como a proposta de trabalhos futuros.

8.1. Principais Contribuições

A proposta de um modelo eficiente de desenvolvimento de software baseado em metodologias ágeis visa contribuir para área de conhecimento da Engenharia de Software, colaborando com o propósito de atender e resolver aspectos críticos habitualmente levantados a cerca das metodologias ágeis de desenvolvimento de software, conforme levantado pelos problemas da pesquisa deste trabalho.

O framework definido neste trabalho permite a seus adeptos estabelecer um processo que possa convergir as características de produtividade das metodologias ágeis, porém, também contemple aspectos tais como design, documentação e um ciclo de vida estruturado. Além disso, a definição de um conjunto de assertivas estruturais para uso do framework proposto permite que este seja adaptado de forma a garantir consistência nos processos resultantes, além de manter o princípio de adaptabilidade das metodologias ágeis de desenvolvimento de software. Partindo do contexto apresentado, este trabalho contribui igualmente no sentido de atender a uma demanda organizacional crescente no que diz respeito a propostas de novos modelos e melhorias nos processos de desenvolvimento de software.

8.2. Limitações Encontradas

Ainda que este trabalho tenha apresentado a resolução dos problemas de pesquisa abordados, tanto através da revisão da bibliografia como por meio da aplicação prática do framework proposto, algumas limitações puderam ser evidenciadas. As limitações são pontuadas a seguir:

- **Revisão Sistemática da Literatura –** Este trabalho restringiu sua revisão de literatura aos autores, trabalhos e fontes de referência bibliográfica a cerca das metodologias ágeis de desenvolvimento de software e temas correlatos. Essa opção se deu em função da amplitude da pesquisa no que diz respeito a restrições de tempo para execução deste trabalho além da amplitude das diferentes frentes realizadas neste trabalho, tais como: revisão da literatura, avaliação comparativa das metodologias ágeis de software, pesquisa com grupo de desenvolvedores, proposta de modelo e estudo de caso prático de aplicação do modelo.
- **Amplitude da aplicação prática do modelo proposto –** Este trabalho restringiu-se ao cenário de uma única organização quando da aplicação prática do modelo proposto, o que por consequência pode levar a resultados específicos ao contexto tratado. Ainda assim, as evidências comparativas às referências bibliográficas abordadas evidenciam que os resultados positivos encontrados no estudo de caso prático realizado neste trabalho tendem a se repetir em outros contextos e organizações, dado que o modelo proposto objetivou definir uma estrutura generalista de processo de desenvolvimento eficiente de software.

- **Múltiplos escopos e tamanhos de projeto -** Não foi possível validar modelo proposto por este trabalho em outros contextos de escopo e tamanho de projeto, o que acarreta na ausência de resultados práticos nesse sentido. Entretanto, conforme abordado no tópico anterior, as evidências embasadas pela literatura referenciada pelo modelo evidenciam a tendência de obtenção de resultados positivos quanto a aplicação do X-PRO em projetos de múltiplos escopos e tamanhos.

- **Prática de Behavior-Driven Development –** O estudo de caso prático de aplicação do modelo proposto por este trabalho possui relatos e considerações dos participantes de que a obrigatoriedade do uso de Behavior-Driven Development limita a adoção do processo, principalmente pelo fato de que fábrica de software brasileiras não costumam adotar essa abordagem. Todavia, não foram apresentadas nem tampouco identificadas pesquisas ou embasamento na literatura para essa assertiva a cerca da baixa adoção de Behavior-Driven Development no Brasil.

- **Prática de Estórias de Usuário –** Outro ponto relatado pelos participantes do estudo de caso prático do modelo proposto por este trabalho refere-se ao fato de que para novos desenvolvedores envolvidos com a manutenção, apenas as Estórias de Usuário não seriam suficientes para entendimento da operação do software. Esses mesmos participantes, porém, observaram que a prática de Imersão de Negócio proposta neste trabalho é uma abordagem interessante para apoiar e preencher eventuais lacunas das Estórias de Usuário, ainda que os participantes tenham reafirmado que o ideal nesse caso seria a utilização Casos de Uso.

- **Aplicação do modelo em projetos de larga escala e times acima de 40 pessoas –** Este trabalho não apresentou evidências em seu estudo de caso prático de que o modelo proposto é aplicável em projetos em larga escala e times acima de 40 pessoas. Ainda assim, a fundamentação teórica deste trabalho apresenta assertivas que direcionam e embasam o entendimento de que o modelo proposto é sim aplicável em projetos de larga escala e times acima de 40 pessoas.

8.3. Trabalhos Relacionados e Outros Processos Híbridos de Desenvolvimento de Software

O modelo proposto por este trabalho objetivou a concepção de um processo eficiente de desenvolvimento de software baseado em metodologias ágeis. Uma das suas principais características é convergir as abordagens, conceitos e práticas positivas das metodologias ágeis descritas na literatura, o que por consequência pode caracterizar o modelo proposto por este trabalho - X - PRO - como híbrido, haja vista sua proposta é compilar características de outros modelos.

A literatura prevê outros modelos que igualmente possuem em suas propostas a abordagem híbrida, com a adoção de princípios e práticas de outras metodologias ágeis de desenvolvimento de software. Duas dessas abordagens possuem um relativo destaque e serão abordadas por este trabalho: AUP (Ágile Unified Processo, do português Processo Unificado Ágil) e o OpenUP. A seguir serão conceituados estes processos e como esses são posicionados ante a proposta deste trabalho.

- **Agile Unified Process**

  O AUP é uma versão simplificada do IBM Rational Unified Process (RUP), modelo proposto e desenvolvido por Scott Ambler (AMBLER, 2006). O AUP descreve uma abordagem simples e fácil de se compreender para desenvolvimento de aplicações de negócio utilizando técnicas ágeis e conceitos relacionados ao RUP. O AUP aplica práticas como Test-Driven Development, Agile Modeling, entre outras práticas ágeis.

  - **Análise comparativa do Agile Unified Process ante ao X-PRO**

    A proposta de Ambler (2006) para o Agile Unified Process, conforme descrito anteriormente, foi de unicamente simplificar a abordagem já proposta pelo IBM Rational Unified Process (RUP), o qual trata-se de um processo prescritivo – orientado à gestão – e que estabelece rotinas, práticas e execuções formais pré-estabelecidas. Ainda que permita adaptação do seu framework, o AUP – e consequentemente o RUP – não é plenamente aderente aos princípios ágeis, tais como: produção de documentação apenas efetivamente relevante para o projeto; processos orientados a viabilizar a autonomia do time; estabelecimento de atividades e ações efetivamente fundamentais para viabilizar a entrega do projeto; entre outros aspectos. O X-PRO, em contrapartida, estabelece em seu framework uma abordagem totalmente aderente aos princípios ágeis, descrevendo-se como um modelo eficiente de desenvolvimento de software justamente por considerar apenas as atividades, práticas, artefatos e conceitos essenciais para a execução de um projeto de software.

    Outro ponto de ressalva quanto ao AUP adotar práticas ágeis, tais como Test-Driven Development e Agile Modeling, está relacionado ao fato de que o modelo não descreve como essas práticas podem efetivamente contribuir com o processo baseado em AUP, o que leva a conclusão de que a proposta de adoção de práticas ágeis neste caso restringe-se a uma sugestão, e não sendo a prática parte do processo. No caso do X-PRO, as práticas ágeis são efetivamente parte do processo, sendo Behavior-Driven Development a prática estabelecida para as atividades de desenvolvimento – codificação – e a prática de Agile Modeling para a execução das atividades de Design de Arquitetura e Design de Software.

- **OpenUP**

  O OpenUP (do português Processo Unificado Aberto), é uma parte do Eclipse Process Framework[29], o qual estabelece um framework de processo *open source* (código aberto) baseado nas melhores práticas de diversos líderes em desenvolvimento de software. O OpenUP preserva características essenciais do IBM Rational Unified Process (RUP) que inclui desenvolvimento iterativo e incremental, casos de usos e cenários na direção do desenvolvimento, gerenciamento de riscos e outras características igualmente encontradas no RUP. O OpenUP é definido como uma alternativa aberta e livre ao RUP – o qual é propriedade da IBM.

  - **Análise comparativa do OpenUP ante ao X-PRO**

[29] Ecplise Process Framework: https://www.eclipse.org/epf/.

Ainda que o OpenUP se defina como um processo aderente aos princípios ágeis, conforme apresentado por Santos (2009), a principal característica do OpenUP que remete aos princípios ágeis é o desenvolvimento iterativo e incremental. Há ainda na sua definição que este se trata de um processo de baixa cerimônia, o que, porém, não é efetivamente encontrado na prática, dada a sua característica de processo baseado no Processo Unificado. Nesse sentido, X-PRO apresenta-se como uma alternativa mais aderente aos princípios ágeis, dada a sua abordagem orientada a eficiência, além de não ser uma proposta baseada no Processo Unificado. As características do OpenUP herdadas do RUP também o define como um processo orientado a gestão, dado que diversas das suas atividades e práticas são direcionadas a produzir documentação e alocar tempo de projeto para permitir acompanhamento da gestão.

Uma abordagem proposta pelo OpenUP, entretanto, se assemelha ao X-PRO no sentido de decompor os requisitos – no caso do X-PRO detalhados como Estórias de Usuário, já no OpenUP como Casos de Uso – em partes menores que viabilizem pequenas unidades de trabalho, o que não só facilita a percepção de evolução do produto de software, como possibilita uma gestão mais granular do projeto. O OpenUP define essas partes como Work Items (do português Itens de Trabalho), que são itens contidos em uma listagem de tudo o que precisa ser realizado no projeto. Cada Work Item pode conter referências a informações relevantes para seguir com o trabalho – como Documento de Casos de Uso, por exemplo. A visão do OpenUP para esses pequenos entregáveis está mais relacionada a atividades gerais de projeto do que funcionalidades de software, efetivamente. No X-PRO, a menor parte de uma funcionalidade é um Comportamento – originado dos conceitos propostos por Behavior-Driven Development – estando este mais efetivamente relacionado ao software do que a um item do projeto – como um artefato ou a execução de alguma atividade específica.

Por fim, assim como encontrado no AUP (Agile Unified Process), o OpenUP sugere a adoção de práticas ágeis, porém não descreve como essas podem efetivamente contribuir com o processo, levando a conclusão de que a proposta de adoção de práticas ágeis neste caso restringe-se a uma sugestão, não sendo efetivamente a prática parte do processo, diferentemente do X-PRO onde práticas ágeis são características fundamentais do seu framework.

8.4. Trabalhos Futuros

A partir da conclusão do presente trabalho, identifica-se potencial para expansão da linha de pesquisa apresentada, focada na temática de processos eficientes de desenvolvimento de software. Como proposta de trabalhos futuros, identificam-se:

- **Estudo comparativo do X-PRO com outras abordagens híbridas de processos de software baseadas em metodologias ágeis –** Sugere-se a execução posterior de um estudo comparativo mais detalhado do X-PRO com outras abordagens híbridas de processo de software baseadas em metodologias ágeis, tais como Agile Unified Process (AUP) e OpenUP. O objetivo é identificar como as características e definições

estabelecidas por X-PRO contribuem ou mesmo são mais efetivas no que diz respeito ao objetivo de estabelecimento de um processo eficiente e orientado à agilidade.

- **Estudo comparativo do X-PRO com metodologias ágeis –** Sugere-se a execução posterior de um estudo comparativo mais detalhado do X-PRO ante a outra metodologia ágil. O objetivo é ratificar que as metodologias ágeis atualmente disponíveis na literatura cobrem aspectos específicos do ciclo de vida de um projeto de software, enquanto que o X-PRO propõe uma abordagem ampla.
- **Estudo de caso prático em projetos de larga escala e times acima de 40 pessoas** – Identifica-se como relevante a execução posterior de um estudo de caso prático de adoção do X-PRO como framework de processo de software em projetos de larga escala e times acima de 40 pessoas. A proposta do X-PRO o descreve como aderente a esse tipo de cenário, ainda que não tenha sido possível validar e certificar essa características no estudo de caso prático realizado nesse trabalho.

- **Estudo analítico da adoção do X-PRO em organizações que possuam certificação em modelos de capacidade de processo de software** – Conforme observado por Mcmahon (2011), existe uma linha de pesquisa ampla e contínua a cerca de como as metodologias ágeis ou processos baseados em práticas ágeis podem ser adotados por organizações que possuam certificação em modelos de capacidade de processos de software, tais como o CMMI (Capability Maturity Model - Integration, do português Modelo de Maturidade e Capacidade – Integração) e o MPS.BR (Melhoria do Processo de Software Brasileiro). Nesse sentido, identifica-se como relevante um trabalho que verifique a adoção do X-PRO em organizações que possuam essas certificações, de forma a identificar como as características de agilidade e adaptação do X-PRO se comportam ante a natural prescrição de processos certificados e abordagem de execução pré-estabelecida.

A continuidade dos estudos das práticas e propostas abordadas neste trabalho indica uma contribuição para a área de Engenharia de Software, no sentido de avançar com pesquisas sobre processos de desenvolvimento de software, considerando principalmente a assertiva de que diversos autores abordam quanto à qualidade do produto de software estar fortemente relacionada com a qualidade do processo utilizado para o seu desenvolvimento (SOMMERVILLE, 2009).

REFERÊNCIAS

ABRAHAMSSON, P., SALO, O., RONKAINEN, J., WARSTA, J. **Agile software development methods: Review and analysis**. VTT Publications, 2002.

AGILE ALLIANCE. **Guide to Agile Practice: Behavior-Driven Development.** Agile Alliance and Institute Agile, 2013. Disponível em: http://guide.agilealliance.org/guide/bdd.html, acessado em 6 de Janeiro de 2014.

AKIF, R., MAJEED, H. **Issues and Challenges in Scrum Implementation** . International Journal of Scientific & Engineering Research, Volume 3, Issue 8, August, 2012 .

AMBLER, S. **Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process**. Wiley Computing Publishing, New York, 2002.

AMBLER, S. **The Object Primer: Agile Model Driven Development with UML 2**. Cambridge University Press, 2004.

AMBLER, S. **Feature-Driven Development (FDD) and Agile Modeling**. Ambysoft, 2005. Disponível em http://www.agilemodeling.com/essays/fdd.htm e acessado em 2 de Janeiro de 2014.

AMBLER, S. **Agile Unified Process, version 1.1**. Ambysoft Inc., 2006.

BECK, K. **Extreme Programming Explained**. Addison-Wesley, 2000.

BECK, K. **Test-Driven Development by Example.** Addison-Wesley, 2003.

BEGEL, A., NAGAPPAN, N. **Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study** . Microsoft Research, September, 2007.

BOEHM, B., TURNER, R., BOOCH, G., COCKBURN, A., PYSTER, A. **Balancing Agility and Discipline: A Guide for the Perplexed.** Addison-Wesley, 1st Edition, 2003.

BUFFARDI, K, EDWARDS, S. H. **Impacts of Teaching Test-Driven Development to Novice Programmers** . International Journal of Information and Computer Science , Volume 1, Issue 6, PP. 135-143 , September, 2012.

CARVALHO, R. A., MANHÃES, R. S., SILVA, F. L. C. **Filling the Gap between Business Process Modeling and Behavior-Driven Development.** CoRR, 2008.

CARVALHO, R. A., MANHÃES, R. S., SILVA, F. L. C. **Mapping Business Process Modeling constructs to Behavior-Driven Development Ubiquitous Language.** CoRR, 2010.

CHRISTENSEN, H. B. **Flexible, Reliable Software: Using Patterns and Agile Development.** Chapman and Hall/CRC, 1st Edition, May 4, 2010.

COCKBURN, A. **Crystal Clear: A Human-Powered Methodology For Small Teams, including The Seven Properties of Effective Software Projects**. Addison-Wesley, 2004.

COLEMAN, G., VERBRUGGEN, R. **A Quality Software Process for Rapid Application Development**. Springer Software Quality Journal, Volume 7, Issue 2, p. 107-122, July, 1998.

DYBA, T., DINGSOYR, T. **Empirical studies of agile software development: A systematic review**. Elsevier, Information and Software Technology, Volume 50, Issues 9-10, Pages 833-859, August, 2008.

FIRDAUS, A., GHANI, I., YASIN, N. I. M. **Developing Secure Websites Using Feature-Driven Development (FDD): A Case Study**. Jornal of Clean Energy Technologies, Vol. 1, No. 4, October, 2013.

GEORGE, B., WILLIAMS, L. **An Initial Investigation of Test-Driven Development in Industry**. Proceedings of the 2003 ACM Symposium on Applied Computing, p. 1135-1139, ACM Digital Library, 2003.

GEORGE, B., WILLIAMS, L. **A structured experiment of test-driven development**. Elsevier Information and Software Technology, Volume 46, Issue 5, Pages 337–342, 15 April, 2004.

GRENYER, P. **Test-Driven Development Doesn't Mean Test First**. Norfolk Tech Journal, Digital Edition, Issue 02, p. 27, November, 2013.

HELLESOY, A., WYNNE, M. **The Cucumber Book: Behavior-Driven Development for Testers and Developers.** The Pragmatic Programmers, 2012.

HIGHSMITH, J. A. **Messy, Exciting and Anxiety-Ridden: Adaptive Software Development.** American Programmer, Volume X, No. 1, January, 1997. Disponível em: http://www.adaptivesd.com/articles/messy.htm, acessado em 1 de Janeiro de 2014.

HIGHSMITH, J. A. **Adaptive Software Development: A Collaborative Approach to Managing Complex Systems**. New York Dorset House, 2000.

HUNT, A., THOMAS, D. **The Pragmatic Programmer: From Journeyman to Master.** Addison-Wesley Professional, 1st Edition, October 30, 1999.

IEEE, Institute of Electrical and Electronics Engineers. **Software Engineering Body of Knowledge**. 3rd Version, 2004.

JANOFF, N. S., RISING, L., **The Scrum Software Development Process for Small Teams**. IEEE Software Publishings, July/August, 2000.

JANZEN, D. S., SAIEDIAN, H. **On the Influence of Test-Driven Development on Design de Software**. 19Th Conference on Software Engineering Education & Technology, IEEE Computer Society, 2006.

KRUTCHEN, Phillippe. **Rational Unified Process: An Introduction**. Addison-Wesley, 1999.

LARMAN, C., KRUCHTEN, P., BITTNER, K. **How to fail with the Rational Unified Process: Seven steps to pain and suffering**. Valtech Technologies and Rational Software, 2001.

LAYMAN, L., WILLIAMS, L., CUNNINGHAM, L. **Extreme Programming in Context: An Industrial Case Study**. Proceedings of the 2004 Agile Development Conference, IEEE Computer Society, Pages 32-41, 2004.

LEFFINGWELL, D. **Agile software requirements: lean requirements practices for teams, programs, and the enterprise**. Pearson Education, 2011.

LEVINE, L. **Reflections on Software Agility and Agile Methods: Challenges, Dilemmas & the Way Ahead** . Carnegie Mellon University, Software Engineering Institute, May 11, 2005.

LINDVALL, M., BASILI, V., BOEHM, B., COSTA, P., DANGLE, K., SHULL, F., TESORIERO, R., WILLIAMS, L., ZELKOWITZ, M. **Empirical Findings in Agile Methods**. Extreme Programming and Agile Methods — XP/Agile Universe, Lecture Notes in Computer Science Volume 2418, pp 197-207, 2002.

MADEYSKI, L., SZALA, L. **The Impact of Test-Driven Development on Software Development Productivity – An Empirical Study.** Lecture Notes in Computer Science, LNCS 4764. Springer, pp.200-211 , 2007.

MADEYSKI, L. **Test-Driven Development: An Empirical Evaluation of Agile Practice.** Springer Publishing, 2010.

MCMAHON, P. E. **Integrating CMMI and agile development: case studies and proven techniques for faster performance improvement**. Addison-Wesley, 2011.

NERUR, S., MAHAPATRA, R., MANGALARAJ, G. **Chalenges of Migrating to Agile Methodologies**. Magazine Communications of the ACM - Adaptive Complex Enterprises, Volume 48, Issue 5, May 2005.

NORTH, D. **Introducing Behavior-Driven Development.** Dan North and Associates, 2006. Disponível em: http://dannorth.net/introducing-bdd/, acessado em 6 de Janeiro de 2014.

NORTH, D. **What's in a story?** Dan North and Associates, 2007. Disponível em: http://dannorth.net/whats-in-a-story/, acessado em 6 de Janeiro de 2014.

OSTERWALDER, A., PIGNEUR, Y. **Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers**. Wiley Publishing, 1st Edition, July 13, 2010.

PALMER, S. R., FELSING, J. M. **A Practical Guide to Feature-Driven Development.** Prentice Hall Publishing, 2002.

PANG, J., BLAIR, L. **Refining Feature-Driven Development – A methodology for early aspects.** Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, p. 86-91, 2004.

PEREIRA, E. B. **Uma Proposta para Adaptação de Processos de Desenvolvimento de Software Baseados no Rational Unified Process**. Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática. Dissertação de Mestrado, 2005.

PETERSSON, J. **Cloud metering and billing: Billing metrics for compute resources in the cloud**. IBM Developer Works Publishing, August, 08, 2011.

POLLICE, G. **Using the Rational Unified Process for small projects: Expanding upon eXtreme Programming**. Rational Software Corporation, 2001.

PRESSMAN, Roger S. **Software Engineering: A Practitioner's Approach 6th edition**. McGraw-Hill Science, 2004.

REIS, C. A. L. **Uma Abordagem Flexível para Execução de Processos de Software Evolutivos**. Universidade Federal do Rio Grande do Sul, Instituto de Informática. Programa de Pós-Graduação em Computação, 2003.

REIS, R. Q., REIS, C. A. L., SCHLEBBE, H., NUNES, D. J. **Towards an aspect-oriented approach to improve the reusability of Software Process Models**. Workshop on early aspects: aspect-oriented requirements engineering and architecture design, 2002.

ROCHA, A. R., MONTONI, M., SANTOS, G., MAFRA, S., FIGUEIREDO, S., BESSA, A., MIAN, P. **Estação TABA: Uma infraestrutura para Implantação do Modelo de Referência para Melhoria de Processo de Software**. Programa de Engenharia de Sistemas e Computação, Universidade Federal do Rio de Janeiro (COPPE/UFRJ), 2005.

SANTOS, S. S. **OpenUP: Um processo ágil.** IBM Development Works Publication, 2009. Disponível em: http://www.ibm.com/developerworks/br/rational/local/open_up/index.html?ca=dat, acessado em 12 de Janeiro de 2014.

SCHWABER, K., SUTHERLAND, J. **The Scrum Guide - The definitive guide to Scrum: The rules of the Game**. Scrum.org, 2013.

SHUJA, A. K., KREBS, J. **IBM Rational Unified Process: Reference and Certification Guide**. IBM Press, 2008.

SOLIS, C., WANG, X. **A study of the Characteristics of Behavior-Driven Development.** 37th Euromicro Conference on Software and Engineering and Advanced Applications (SEAA), p. 383-387, August, 30, 2011.

SOMMERVILLE, Ian. **Software Engineering**. Addison-Wesley, 9th Edition, 2011.

SONG, X., OSTERWEIL, L. J. **Comparing design methodologies through process modeling.** 1st International Conference on Software Process, Los Alamitos, Calif., IEEE CS Press, 1991.

SONG, X., OSTERWEIL, L. J. **Toward objective, systematic design-method comparisons.** IEEE Software 9(3): 43–53, 1992.

SHORE, J., WARDEN, S. **The Art of Agile Development: Test-Driven Development**. O'Reilly Publishing, 2008.

SHRIVASTAVA, S. V., DATE, H. **Distributed Agile Software Development: A Review** . Journal of Computer Science and Engineering, Volume 1, Issue 1, May, 2010 .

STAPLETON, J. **Dynamic systems development method – The method in practice**. Addison-Wesley, 1997.

TOMHAVE, B. **Alphabet Soup: Making Sense of Models, Frameworks and Methodologies.** Secure Consulting, 2005.

TURK, D., FRANCE, R., RUMPE, B. **Limitations of Agile Software Processes.** Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, p. 43-46, 2000.

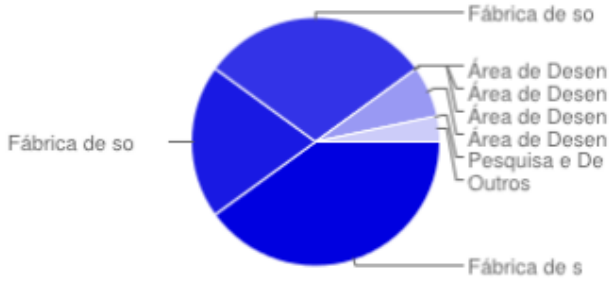
WAKE, W. C. **Extreme Programming Explored**. Addison-Wesley, 2002.

WESTERHEIM, H. HANSSEN, G. K. **The Introduction and Use of a Tailored Unified Process – A Case Study**. 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005.

APÊNDICE A: PESQUISA E AVALIAÇÃO SOBRE ADOÇÃO DE PRÁTICAS E METODOLOGIAS ÁGEIS DE SOFTWARE

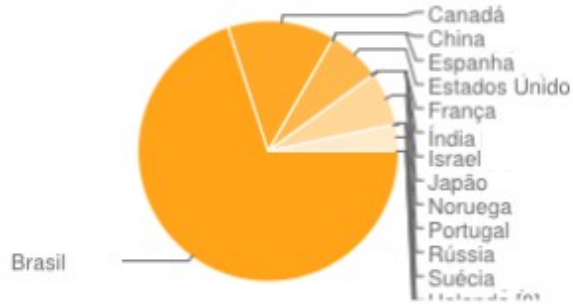
Visão geral sobre sua organização

Em qual segmento da indústria de software sua organização atua?



Fábrica de software de múltiplos produtos	40%
Fábrica de software de produto único	20%
Fábrica de software de terceirização	30%
Área de Desenvolvimento de um departamento de Tecnologia da Informação, Indústria	0%
Área de Desenvolvimento de um departamento de Tecnologia da Informação, Serviços	0%
Área de Desenvolvimento de um departamento de Tecnologia da Informação, Varejo	0%
Área de Desenvolvimento de um departamento de Tecnologia da Informação, Outros	7%
Pesquisa e Desenvolvimento, Academia	0%
Outros	3%

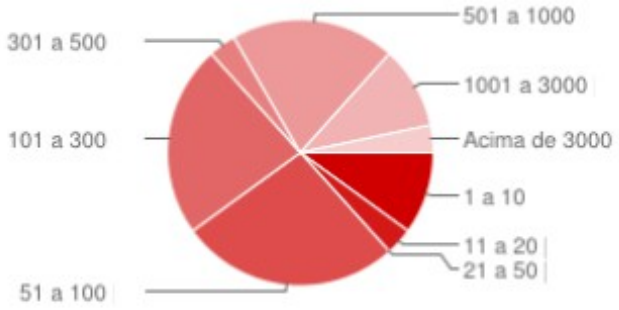
Com base na pesquisa "PwC Global 100 Software Leaders", foram listados os países onde estão sediadas as principais organizações mundiais de software. A sua organização está sediada em qual dessas localidades?



Alemanha	0%
Bélgica	0%
Brasil	70%
Canadá	13%
China	0%
Espanha	0%
Estados Unidos	7%

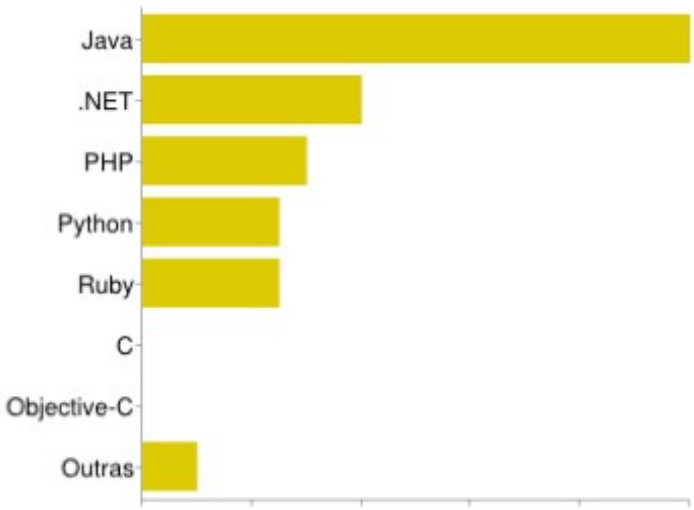
França	0%
Índia	0%
Israel	0%
Japão	0%
Noruega	0%
Portugal	7%
Rússia	0%
Suécia	0%
Holanda	0%
Outros	3%

Quantos funcionários sua organização possui diretamente ligados aos esforços de desenvolvimento de software?



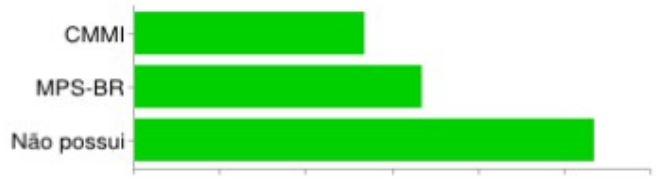
1 a 10	10%
11 a 20	3%
21 a 50	0%
51 a 100	27%
101 a 300	23%
301 a 500	3%
501 a 1000	20%
1001 a 3000	10%
Acima de 3000	3%

Qual a principal linguagem de desenvolvimento adotada nos projetos da sua organização?



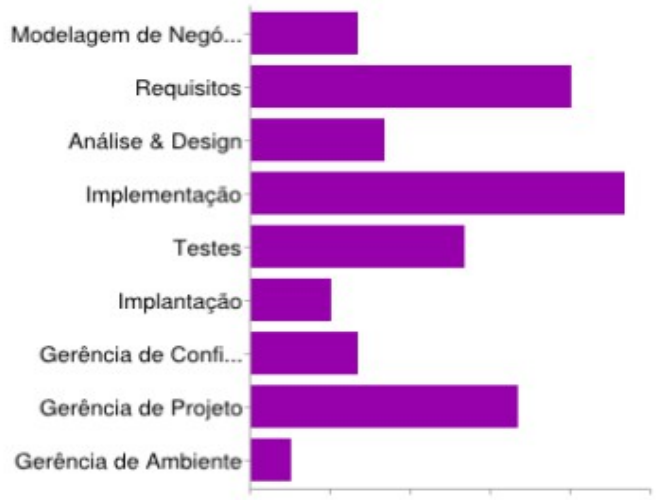
Java	43%
.NET	17%
PHP	13%
Python	11%
Ruby	11%
C	0%
Objective-C	0%
Outras	4%

Sua organização possui alguma certificação de processo de desenvolvimento de software?



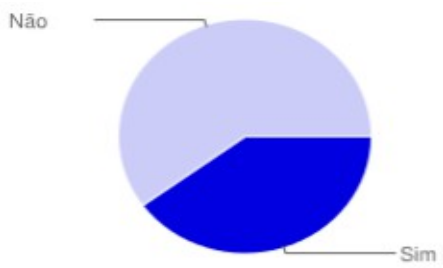
CMMI	24%
MPS-BR	29%
Não possui	47%

No processo de desenvolvimento de software da sua organização, quais das atividades abaixo você pode afirmar ter uma execução padronizada e estabelecida?



Modelagem de Negócios	7%
Requisitos	20%
Análise & Design	8%
Implementação	23%
Testes	13%
Implantação	5%
Gerência de Configuração	7%
Gerência de Projeto	16%
Gerência de Ambiente	2%

Nos projetos de software da sua organização, existe alguma atividade ou esforço dedicado a Análise e Design?



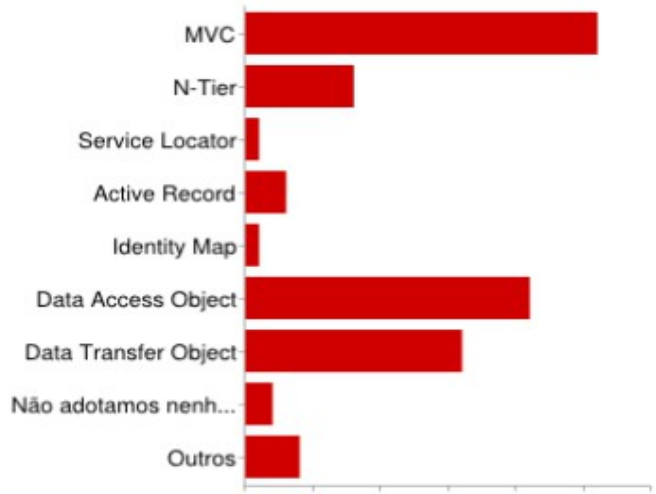
Sim	40%
Não	60%

Caso sua resposta tenha sido sim na pergunta anterior (se sua organização possui atividades ou esforços de Análise e Design), quais são os modelos habitualmente concebidos?

Diagrama de Classe	10%
Diagrama de Componentes	8%
Diagrama de Objetos	3%
Diagrama de Pacotes	3%
Diagrama de Implantação	3%
Diagrama de Estrutura Composta	0%

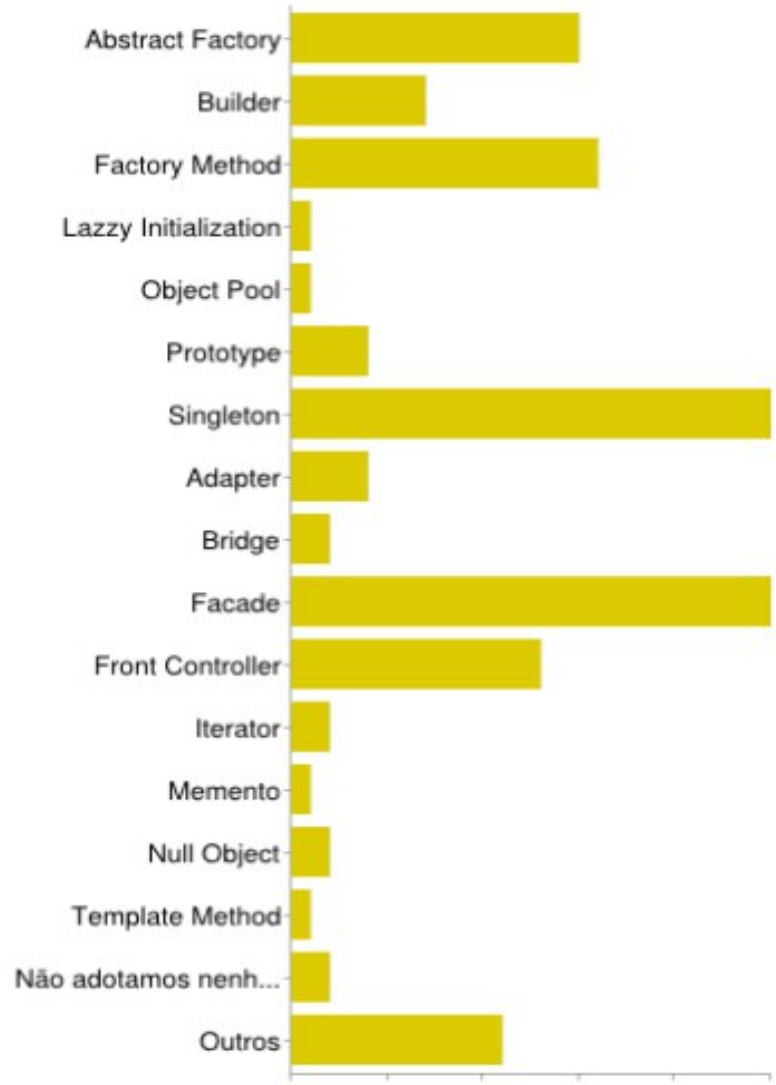
Diagrama de Perfil	3%
Diagrama de Casos de Uso	7%
Diagrama de Estados	1%
Diagrama de Atividades	4%
Diagrama de Sequência	3%
Diagrama de Interatividade	0%
Diagrama de Colaboração	0%
Diagrama de Tempo	1%
Mockup de Interface	4%
Diagrama de Modelo de Dados (ER)	27%
Não produzimos nenhum modelo dos esforços de Análise e Design	23%

Quais são os Padrões Arquiteturais (Architectural Patterns) que sua organização utiliza?



MVC	32%
N-Tier	10%
Service Locator	1%
Active Record	4%
Identity Map	1%
Data Access Object	26%
Data Transfer Object	20%
Não adotamos nenhum Padrão Arquitetural (Architectural Pattern)	2%
Outros	5%

Quais são os Padrões de Projeto (Design Patterns) que sua organização utiliza?



Abstract Factory	11%
Builder	5%
Factory Method	12%
Lazzy Initialization	1%
Object Pool	1%
Prototype	3%
Singleton	19%
Adapter	3%
Bridge	2%
Facade	19%
Front Controller	10%
Iterator	2%
Memento	1%
Null Object	2%
Template Method	1%
Não adotamos nenhum Padrão de Projeto (Design Pattern)	2%

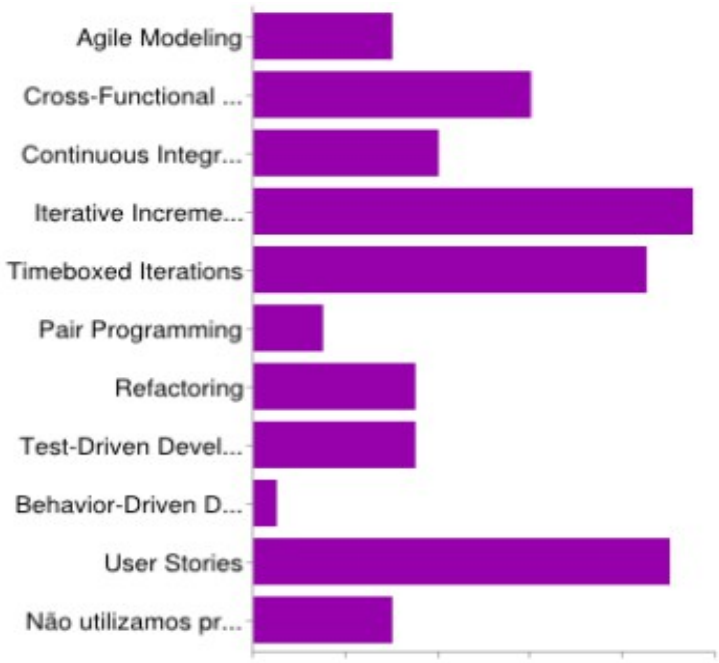
Outros	8%

Adoção de práticas e metodologias ágeis na sua organização

Sua organização adota alguma metodologia ágil de desenvolvimento de software?

Scrum	49%
Extreme Programming	16%
Crystal	0%
Dinamic Systems Development Method	0%
Adaptive Software Development	0%
Feature-Driven Development	0%
Test-Driven Development	16%
Behavior-Driven Development	2%
Não adotamos metodologias ágeis, mas sim um modelo baseado no Processo Unificado (Unified Process)	9%
Não adotamos metodologias ágeis, mas sim um modelo baseado em Ciclo de Vida em Cascata	5%
Não temos processo de software	2%
Outros	0%

Sua organização utiliza alguma prática ágil específica no seu processo de desenvolvimento de software?



Agile Modeling	6%
Cross-Functional Teams	12%
Continuous Integration	8%
Iterative Incremental Development	18%
Timeboxed Iterations	16%
Pair Programming	3%
Refactoring	7%
Test-Driven Development	7%
Behavior-Driven Development	1%
User Stories	17%
Não utilizamos práticas ágeis de desenvolvimento de software	6%

Experiência pessoal

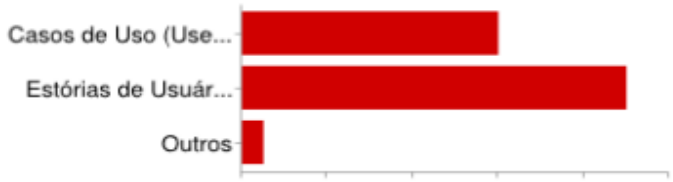
Com base na sua experiência pessoal, quais as atividades fundamentais que devem existir no ciclo de vida de um projeto de software? Você deve selecionar as atividades que na sua percepção não podem deixar de ser executadas em nenhum projeto de software.

Modelagem de Negócios	5%
Requisitos	19%
Análise & Design	18%
Implementação	18%
Testes	13%
Implantação	6%
Gerência de Configuração	6%
Gerência de Projeto	13%
Gerência de Ambiente	2%

Você foi escalado para um novo projeto, cujo objetivo é migrar a tecnologia de desenvolvimento de uma aplicação. Nesse caso, compreende-se que você precisa conhecer como foi concebida e como está disposta a estrutura da aplicação. Quais seriam os modelos que na sua opinião seriam fundamentais para auxiliá-lo no conhecimento da aplicação, por consequência, na sua manutenção?

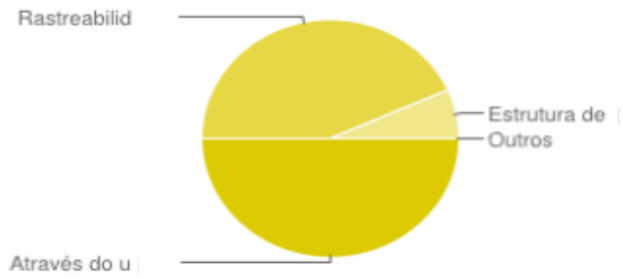
Diagrama de Classe	21%
Diagrama de Componentes	2%
Diagrama de Objetos	5%
Diagrama de Pacotes	2%
Diagrama de Implantação	18%
Diagrama de Estrutura Composta	0%
Diagrama de Perfil	0%
Diagrama de Casos de Uso	6%
Diagrama de Estados	2%
Diagrama de Atividades	6%
Diagrama de Sequência	5%
Diagrama de Interatividade	0%
Diagrama de Colaboração	0%
Diagrama de Tempo	0%
Mockup de Interface	12%
Diagrama de Modelo de Dados (ER)	22%

Se você puder escolher qual dos identificadores de funcionalidade produzir e utilizar como desenvolvedor, qual seria a sua escolha?



Casos de Uso (Use Cases)	39%
Estórias de Usuário (User Stories)	58%
Outros	3%

Você recebeu a atribuição de realizar uma mudança em um componente de uma aplicação. Qual a técnica adotada por você para Análise de Impacto dessa alteração?



Através do uso de um SCM (Software Configuration Management)	50%
Rastreabilidade de Requisitos	43%
Estrutura de Configuração de Itens de Software	7%
Outros	0%

Você acredita que é possível utilizar práticas e metodologias ágeis de desenvolvimento de software para projetos com grandes times (acima de 40 pessoas) e projetos de grande escopo?

Sim	87%
Não	13%