CENTRO UNIVERSITÁRIO – CATÓLICA DE SANTA CATARINA

ANDRÉ FURQUIM

IMPLEMENTANDO A METODOLOGIA LEAN NO DESENVOLVIMENTO DE SOFTWARE

JOINVILLE

2015

ANDRÉ FURQUIM

IMPLEMENTANDO A METODOLOGIA LEAN NO DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada ao Curso de Pós-Graduação em Engenharia de Software do Centro Universitário – Católica de Santa Catarina como requisito parcial para obtenção do certificado do curso.
Orientador: Maurício Henning

JOINVILLE

2015

Ficha catalográfica elaborada pela Biblioteca Central do Centro Universitário – Católica de Santa Catarina

Furquim, André.
Implementando a metodologia lean no desenvolvimento de software / André Furquim. – 2015.
51 f. : il.

Orientador: Maurício Henning
Monografia – Centro Universitário – Católica de Santa Catarina, Instituto de Ciências Exatas. Curso de Pós-Graduação em Engenharia de Software, 2015.

1. Lean. 2. Metodologia de Desenvolvimento. 3. Engenharia de Sofware. I. Henning, Maurício. Msc.

ANDRÉ FURQUIM

IMPLEMENTANDO A METODOLOGIA LEAN NO DESENVOLVIMENTO DE SOFTWARE

Monografia apresentada ao Curso de Pós-Graduação em Engenharia de Software do Centro Universitário – Católica de Santa Catarina como requisito parcial para obtenção do certificado do curso.

COMISSÃO EXAMINADORA

Prof. Msc. Maurício Henning - Orientador
Centro Universitário – Católica de Santa Catarina

Prof.

Prof.

AGRADECIMENTOS

Este trabalho é decicado à minha família, meus amigos, meu orientador e a Católica de Santa Catarina.

RESUMO

O software como produto foi ganhando cada vez mais importância com o passar do tempo, assim foi necessário investir em modelos que pudessem melhorar seu desenvolvimento para que fosse possível garantir melhor qualidade, custo e prazo no projeto de software. Um modelo que vem ganhando mais notoriedade atualmente é o modelo de desenvolvimento lean, o qual tem seu foco em criar valor para o cliente através da eliminação de desperdícios, otimização do fluxo de valor, capacitação das pessoas e melhora contínua. Esse modelo, criado pela Toyota, embora aplicado em muitas empresas, ainda possui desafios no que tange a sua utilização no mercado de software. Esse trabalho tem como objetivo esclarecer o leitor sobre essa metodologia (além de outras utilizadas no mercado), fazer um comparativo dessas metodologias com o lean, mostrar seus desafios no mercado de software e apresentar maneiras de como proceder para aplicação da metodologia lean de desenvolvimento em uma empresa de software.

Palavras-chave: Lean, Metodologia de Desenvolvimento de Software, Engenharia de Software.

ABSTRACT

The software as a product has been growing in terms of importance over the time in people's life, thus it was necessary to invest in models in order to improve software development in the sense of guaranteeing a better quality, cost and time in a software project. A model that has been growing in importance over the software community is lean, which focuses on creating customer value through waste elimination, optimization of value streams, people empowerment and continuous improvement. Although this model, created by Toyota, is widely applied in several companies, it's still a challenge regarding to software market. This work aims to explain the reader about this methodology (and other ones used in software market today), compare lean to other software methodologies, show its challenges in the current software market and present ways to implement lean in a software company.

Keywords: Lean, Software Development Methodology, Software Engineering.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AWS	*Amazon Web Services*
BDD	*Bevavior Driven Development*
DSDM	*Dynamic Systems Development Methodology*
FDD	*Feature Driven Development*
GQM+	*Goal Question Metric*
IHC	Interface Humano Computador
JAD	*Joint Application Design*
JIT	*Just in Time*
LSD	*Lean Software Development*
PSP	*Personal Software Process*
SAAS	*Software as a Service*
RUP	*Rational Unified Process*
TDD	*Test Driven Development*
TSP	*Team Software Process*
UAP	*Unified Agile Process*
UML	*Unified Modeling Language*
XP	*Extreme Programming*

SUMÁRIO

1 INTRODUÇÃO

A engenharia de *software* surgiu como um esforço de aplicar processos que pudessem dar maior assertividade nos projetos de *software*. Através desses esforços surgiram vários modelos que contribuíram para melhorar o cenário de desenvolvimento. Pode-se dizer que esses esforços foram também impulsionados através do mercado, o qual se torna cada vez mais competitivo e exige que os projetos de *software* tenham que ser desenvolvidos com extrema qualidade e de forma rápida, repetida e confiável. Dentre as várias metodologias que surgiram, uma metodologia está ganhando bastante destaque no mercado atual: o *lean*. Poppendieck (2007) aponta que iniciativas *lean* (ou enxuta)[1] tem sido utilizadas com sucesso em vários setores como manufatura, logística, serviços e no desenvolvimento de produtos, a fim de garantir uma melhor entrega no que tange ao custo, qualidade e prazo. Uma pergunta que surge naturalmente é: "É possível aplicar os mesmos conceitos já aplicados em outros setores para o *software* ?". Esse trabalho propõe-se a responder essa e outras perguntas sobre aplicação do *lean* no contexto de *software*. Na seção seguinte, são mostrados alguns trabalhos na linha de pesquisa do *lean* e que contribuíram para o trabalho.

1.1 TRABALHOS RELACIONADOS

No trabalho de Poppendieck (2007) é mostrado um breve tutorial para aplicação de *lean* no desenvolvimento de *software*. O *lean* é focado em sete princípios:

- Eliminar desperdício;
- Incorporar qualidade;
- Criar conhecimento;
- Adiar compromisso;
- Entregar rápido;
- Respeitar pessoas e
- Otimizar o todo.

Todo esse grupo de princípios, segundo esse trabalho, fornece uma orientação de como entregar um *software* de forma rápida, melhor e com um custo menor. Neste trabalho é afirmado que o desenvolvimento de *software lean* fornece a teoria por trás das práticas utilizadas pelo desenvolvimento ágil. Além disso, o *lean* fornece para as empresas uma

[1] Neste trabalho é empregado a palavra em inglês lean.

série de princípios para melhorar o processo de engenharia de software com a finalidade de trazer melhora no contexto do cliente, domínio do negócio, capacidade de desenvolvimento e em uma eventual situação única que a empresa enfrente.

Neste trabalho, um tutorial foi feito com uma turma e a aplicabilidade do *lean* no desenvolvimento de *software* é feita através dos seus princípios fundamentais. No princípio do desperdício, além de sua explicação no contexto de *lean*, foi mostrado como enxergar o desperdício no contexto do desenvolvimento e explanado os sete desperdícios no desenvolvimento de *software*. Nesta etapa, a classe foi divida em grupos de até sete pessoas e foi feito um mapeamento de fluxo de valor da experiência de alguém no grupo. Após a elaboração, os grupos apresentaram seu mapa e receberam críticas dos outros participantes.

Para o próximo conceito do *lean*, qualidade, foram apresentados conceitos como: TDD (*test-driven development*), teste unitário automatizado e teste de aceitação. Essa etapa não focou muito em como fazer, mas o porquê da importância dessas práticas para o *lean*.

Uma análise racional, no que se refere ao conhecimento, foi feita sobre a questão de postergar os compromissos (princípio do pensamento *lean*). Também foram apresentados métodos que visam preservar o conhecimento de forma que os times não tenham que reaprender o que já foi aprendido. Além disso, foi feita uma discussão dos benefícios da utilização de uma abordagem que envolve explorar múltiplas soluções em vez de focar em apenas uma.

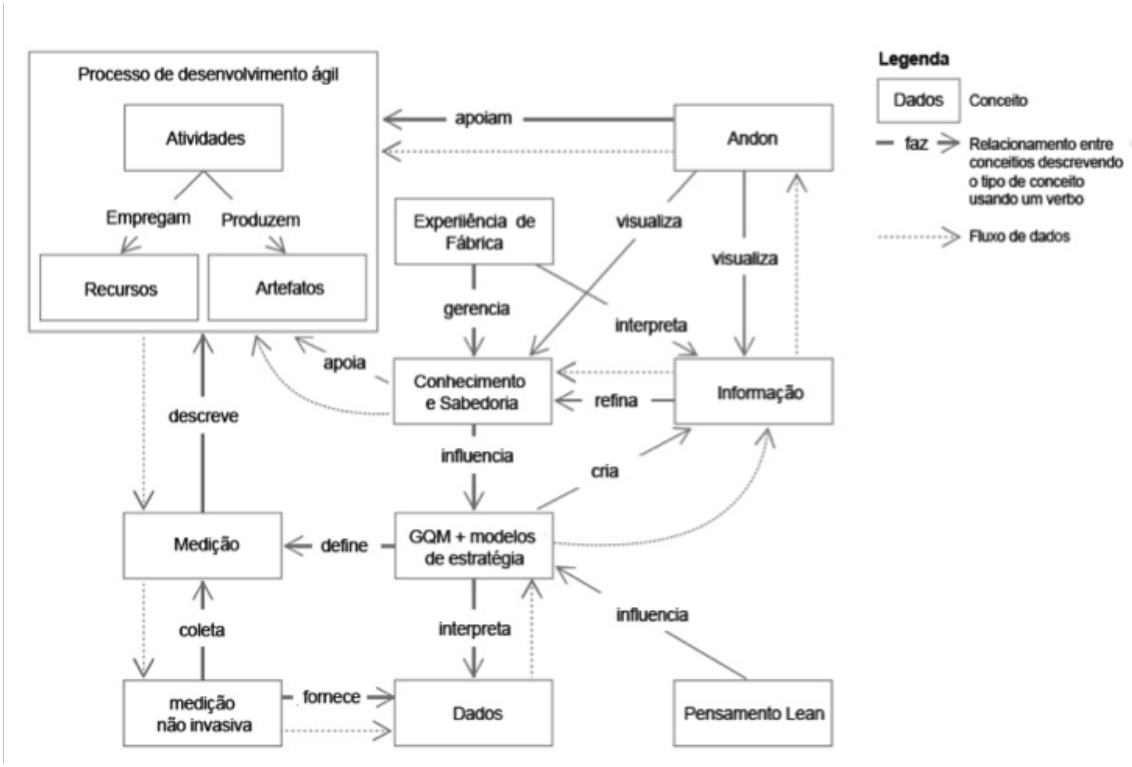
Para questão de velocidade é discutido a aplicação da teoria das filas no desenvolvimento de *software*. Nela foi mostrado o porquê de longas filas serem prejudiciais ao desenvolvimento e mostrado como evita-las. Além disso, é mostrado que ciclos de vida rápidos tendem a dar uma qualidade maior ao projeto de *software* e reduzir seu custo.

Por fim, foi discutido a importância de enxergar um sistema como um todo. Através dessa discussão foi explanado que não é suficiente focar apenas no desenvolvimento em si, assim foi mostrado como mudar o foco do desenvolvimento para os esforços dos objetivos em geral, ou seja, o produto ou o processo apoiado pelo *software*.

A abordagem de migração para o *lean* de uma empresa de *software*, no trabalho de Janes (2015), também foi feita a partir da medição do fluxo de valor da organização com posterior utilização desses dados para migração. Essa equipe já utilizava a metodologia ágil de desenvolvimento. Para medir o fluxo de valor, são medidos aspectos de interesse das atividades, recursos utilizados e artefatos produzidos. É importante notar que, como assinala Janes (2015), o que é de interesse do ponto de vista para os objetivos de uma empresa não necessariamente é para outra. Por isso é utilizado a estratégia GQM+ (*Goal Question Metric Plus*). A estratégia GQM+ é uma nova abordagem de medida baseada

no GQM (*Goal Question Metric*). A diferença segundo Basili *et al.* (2014) é que o GQM+ alinha as medidas do *software* com os objetivos da empresa. Essa abordagem foi necessária a fim de ajudar na correspondência das estratégias com a necessidade de informação e medidas. Os dados obtidos são usados para entender melhor o processo de desenvolvimento de *software*. Uma técnica chamada Andon é então utilizada no sentido de fornecer um mecanismo para visualização dos dados por qualquer pessoa com o menor esforço possível. Através desse mecanismo de visualização, um feedback para o time é feito no sentido de guia-los nos seus objetivos. Na figura 1 é mostrado um mapa que ilustra a dinâmica do *lean* em uma empresa que emprega o processo de desenvolvimento ágil.

Figura 1 – Mapas de conceito descrevendo os blocos de construção do desenvolvimento de software lean



Fonte: Adaptado de Janes, 2015.

Na Figura 1 obverva-se que o processo de desenvolvimento ágil possui atividades, as quais utilizam-se de recursos e produzem artefatos. É possível perceber também que o processo do *lean* não está associado unicamente ao processo de desenvilmento da organização. Por essa razão a abordagem GQM+ foi um diferencial. Através de um sistema de retroalimentação, tem-se um *feedback* que contribui também para melhora contínua do sistema como um todo.

O trabalho de Wang (2011) foca na combinação do *lean* com metodologias ágeis, como o trabalho de Janes (2015). Wang (2011) afirma que existem linhas de pesquisa que consideram que ágil e o *lean* são dois nomes para mesma coisa, enquanto outras linhas

consideram que não. Os autores que consideram que *lean* e ágil não são a mesma coisa possuem basicamente duas visões: a de que *lean* e ágil estão em níveis diferentes dentro da organização e outra que afirma que *lean* e ágil possuem diferentes focos e escopos. (WANG, 2011)

Para visão de que *lean* e ágil estão em diferentes níveis dentro da mesma organização, é utilizado o pensamento *lean* como um guia de princípios para desenvolver e aplicar práticas ágeis. Assim, são propostos sete princípios que são traduzidos para práticas ágeis no contexto do desenvolvimento de um *software*. Para outra visão, que considera que as metodologias possuem focos e escopos diferentes, métodos ágeis estão apenas preocupados com a prática de desenvolvimento e com o gerenciamento do projeto, deixando de lado o contexto global ou de negócio. Nesse sentido, o *lean* se diferenciaria na questão de sua aplicação do escopo, que pode ser desde uma prática de desenvolvimento até uma empresa inteira, onde o desenvolvimento é apenas uma parte do sistema. (WANG, 2011)

Considerando que *lean* e ágil são diferentes, pode-se ter várias combinações possíveis de *lean* em uma empresa. Essas combinações serão vistas mais adiante no trabalho e escolhidas para o contexto do estudo de caso.

Na seção seguinte são abordados os objetivos geral e específicos deste trabalho.

1.2 OBJETIVOS

O objetivo deste trabalho é propor uma metodologia de aplicação do *lean* em uma empresa de *software* que já utiliza-se de algumas práticas ágeis. Assim, é necessário identificar pontos falhos no modelo de desenvolimento da empresa para melhorar os projetos.

Para elaborar esse modelo, algumas perguntas precisam ser antes respondidas como:

- Como aplicar *lean* no ambiente de desenvolvimento ?
- Existe uma formula que possa ser aplicada para qualquer empresa ?
- O quanto as práticas atuais da empresa estão perto ou longe do melhor cenário possível para o *lean* ?
- Quais ferramentas de *software* podem auxiliar o processo ?

Ao longo desse trabalho, essas perguntas são respondidas.

1.2.1 Objetivos Específicos

Afim de melhorar o ciclo de desenvolvimento, é necessário eliminar os desperdícios (um dos princípios do *lean*). Assim é necessário:

- representar o modelo atual de desenvolvimento de uma maneira que fique bem claro para as pessoas envolvida, tanto a nível de negócio como operacional, os problemas existente;
- Propor um novo modelo, que elimine esses desperdícios e
- Sugerir ferramentas que possam melhorar o processo de desenvolvimento.

1.3 CONCLUSÃO DO CAPÍTULO

Esse capítulo teve como objetivo abordar o *lean* de forma superficial. Foi visto que há autores que consideram que *lean* e ágil são dois nomes para mesma coisa e autores que consideram que não. Os autores que consideram ágil e *lean* como coisas distintas possuem basicamente duas visões: que eles focam em nível diferentes dentro da empresa e outra que eles possuem foco e escopos diferentes. Além disso, foi proposto como trabalho uma maneira de melhorar um modelo de desenvolvimento de *software* em uma empresa. Mais detalhes são fornecidos no Capítulo 4. No Capítulo 2 é feita uma breve revisão dos modelos de desenvolvimento, alguns ainda utilizados e outros não pelas empresas a fim de contextualizar a evolução da contrução de *software*.

2 METODOLOGIAS DE DESENVOLVIMENTO

A medida que o *software* começou a ganhar mais importância na vida das pessoas, as empresas tiveram que investir em modelos que melhoram sua capacidade de produção em um mercado que se torna cada vez mais competitivo, como relata Poppendieck (2007). Não é incomum projetos complexos de *software* precisarem ser executados durante um período de tempo para terem sucesso e até fazerem sentido. Um sistema de análise de candidatos a uma eleição, por exemplo, não teria sentido se ficasse pronto depois que a eleição tivesse acabada. Os modelos de desenvolvimento auxiliaram essas empresas a entenderem melhor a dinâmica de desenvolvimento e ajudaram a desenvolver projetos no melhor custo, prazo e qualidade possível. Nesse capítulo são vistos alguns modelos de desenvolvimento, alguns ainda utilizados, e outros que estão em desuso, mas que representaram grande importância histórica para evolução do que há disponível hoje. Esse capítulo tem como objetivo abordar alguns modelos de processo tradicionais: como o cascata, incremental e RUP. Logo depois, são vistos algumas metodologias ágeis que são utilizadas no mercado como SCRUM, XP, LSD etc.

2.1 PROCESSO DE SOFTWARE

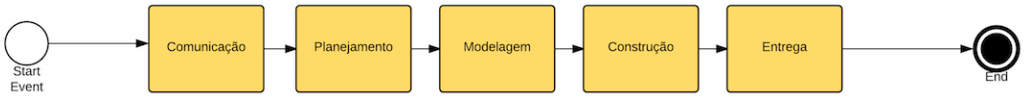
Quando um projeto de *software* precisa ser executado, os engenheiros e seus gerentes utilizam um processo de *software*. Um processo de *sofware*, como define Pressman (2011), é um conjunto de atividades de trabalho, ações e tarefas realizadas a fim de produzir algum artefato de *software*. Cabe ressaltatar que a palavra artefato foi empregada de forma a dar enfase de que não apenas o *software* em si é produzido, mas a documentação etc. Um processo, que muitas vezes é adaptado a algum tipo de projeto, possui uma metodologia de apoio, a qual possui atividades, as quais por sua vez possuem ações e tarefas atreladas.

Geralmente no que tange as atividades do projeto, o engenheiro e os gerentes possuem uma flexibilidade para utilizar as atividades que acharem mais coerente, sendo as principais comunicação, planejamento, modelagem, construção e entrega. Atividades de apoio, e.g. acompanhamento e controle do projeto, administração de risco e garantia de qualidade, gerenciamento de configuração, revisões técnicas etc., são também empregadas durante o processo. (PRESSMAN, 2011)

As atividades de uma metodologia necessitam de um fluxo de processo. O fluxo de processo descreve a sequência das atividades e seus relacionamentos entre si. A Figura 2 mostra um exemplo de um fluxo de processo linear, ou seja, cada atividade é executada em sequência, uma após a outra. Assim, o software só ficaria pronto depois de passar pelas 4 etapas iniciais. Esse é o fluxo de processo utilizado pelo modelo em cascata por exemplo, o qual será visto mais adiante.

Outro fluxo de processo existente é o evolucionário, utilizado bastante em metodo-

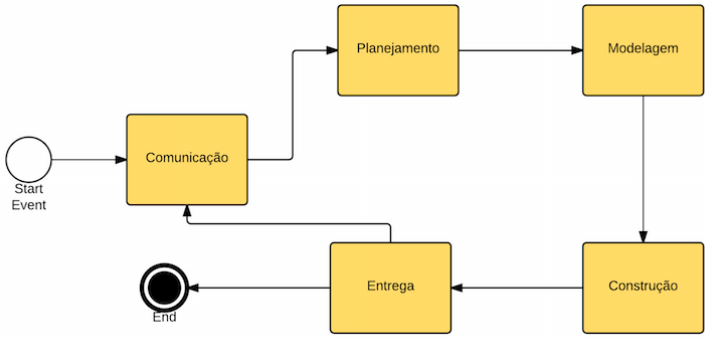
Figura 2 – Fluxo de processo linear



Fonte: Adaptado de Pressman, 2011.

logias ágeis. Nele, um *software* é feito através de várias entregas pontuais. Cada volta no ciclo produz uma versão do *software* mais completa. É ideal para projetos com requisitos não muito bem definidos e complexos. Sua vantagem é que o cliente pode participar e dar *feedback* constantemente e ver como o produto está ficando. A Figura 3 ilustra esse processo.

Figura 3 – Fluxo de processo evolucionário



Fonte: Adaptado de Pressman, 2011.

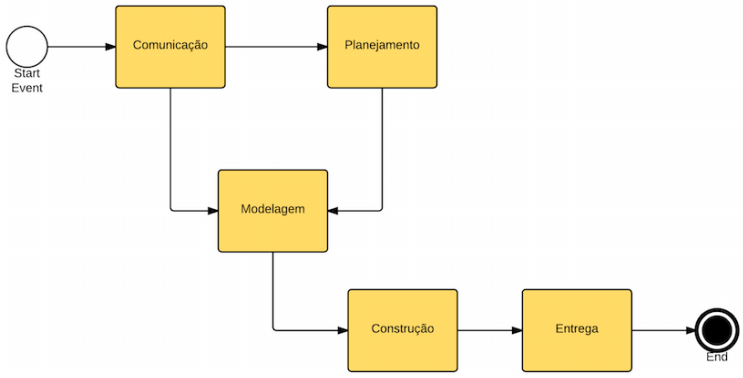
Finalmente há o fluxo de processo paralelo, o qual se diferencia dos outros por possibilitar que uma ou mais tarefas possam ser executadas em paralelo. Na Figura 4 é mostrado um exemplo de fluxo paralelo. Neste ciclo, poderia-se estar na modelagem e na construção ao mesmo tempo, por exemplo.

Na próxima seção são vistos alguns modelos de desenvolvimento, que como visto, empregam algum fluxo de processo.

2.2 MODELOS DE DESENVOLVIMENTOS TRADICIONAIS

Como visto na seção anterior, os modelos de desenvolvimento estão atralado a um ou mais fluxos de processo. Nesta seção são vistos alguns modelos de desenvolvimento que foram utilizados pela indústria de software e outros que ainda são amplamente utilizados. Como há muitos modelos, são descritos os principais segundo Pressman (2011).

Figura 4 – Fluxo de processo paralelo



Fonte: Adaptado de Pressman, 2011.

2.2.1 Cascata

O modelo de desenvolvimento cascata é o ciclo de desenvolvimento mais simples possível. Ele começa basicamente com o levantamento de requisitos por parte do cliente e depois com as etapas de planejamento, modelagem e construção de forma sequencial. Não é possível ir de uma etapa para outra ou pular alguma etapa no processo. Seu fluxo de processo segue o mesmo princípio da Figura 2 e por isso não é mostrado. Dentro da etapa de comunicação há atividades como início do projeto e levantamento de requisitos. Na etapa de planejamento, há atividades como estimativas de cronograma e acompanhamento. Na modelagem há a análise do projeto. A construção possui as atividades de codificação e testes e finalmente no emprego atividades como entrega, suporte e *feedback*.

Pressman (2011) descreve alguns pontos negativos da utilização desse processo como a natureza incerta de projetos de *software* que fazem com que as necessidades sejam difíceis de serem traduzidas de uma vez. Além disso, como mencionado anteriormente o cliente precisa esperar pacientemente até possuir uma versão funcional: que as vezes pode estar bem longe do que ele pediu. Se os requisitos são muito bem definidos e o trabalho deve ser realizado até o final de forma linear, talvez esse seja um processo ideal.

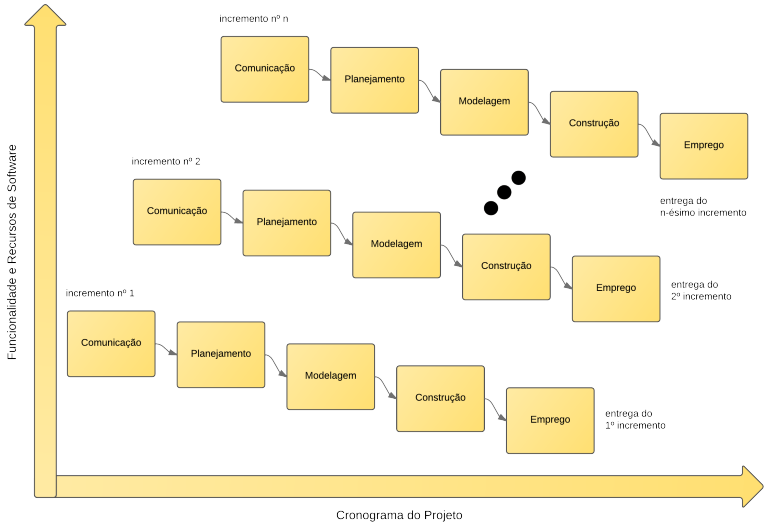
2.2.2 Modelo de Processo Incremental

Ao contrário do modelo em cascata, que geralmente é utilizado em projetos cujos requisitos estão bem definidos, o modelo incremental fica no meio termo. Ele é geralmente empregado em projetos cujos requisitos estão razoavelmente definidos. O modelo combina elementos do fluxo de processo linear e paralelo. Em cada incremento é disponibilizada uma versão aprovada e utilizável do *software* para o usuário, de forma que os futuros incremento serão responsáveis por implementar novas funcionalidades ou expanções das fucionalidades existentes. Como explana Pressman (2011), os primeiros incrementos são

versões funcionais, mas não completas do *software*, assim esse modelo é muito útil quando não há diposponível mão de obra suficiente para entrega de um produto no prazo ou para projetos com baixo orçamento, pois se o sistema for bem aceito, novos incrementos podem ser feitos e mais pessoas podem ser adicionadas ao projeto.

A Figura 5 ilustra o modelo do processo incremental. Pode-se observar que conforme o tempo passa, linha horizontal, novos incrementos são feitos e consequentemente novas funcionalidades e recursos são liberados.

Figura 5 – Fluxo de processo incremental



Fonte: Adaptado de Pressman, 2011.

2.2.3 Modelo de Processo Evolucionário

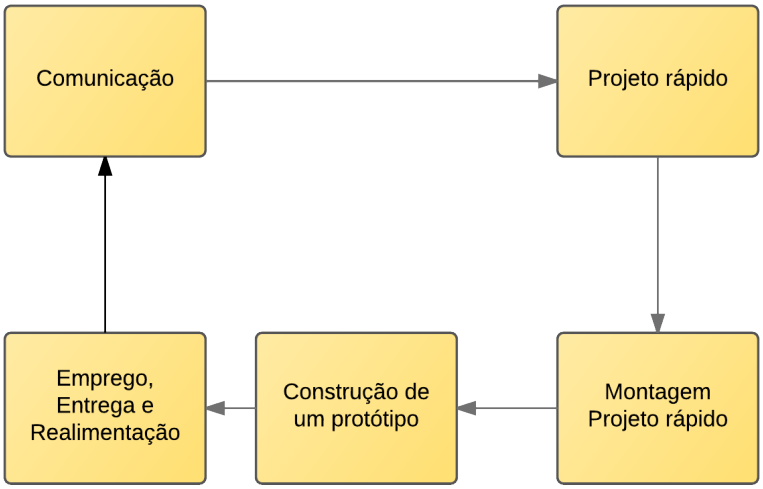
A medida que o mercado demanda que produtos complexos tenham que ser desenvolvidos em um curto prazo, uma versão limitada pode ser desenvolvida para atender as necessidades de negócio. Há projetos cujos detalhes do sistema não estão muito definidos a longo prazo, devido a complexidade, e assim faz-se necessário o uso de um modelo evolucionário. Os modelos evolucionários são iterativos, ou seja, permitem que versões cada vez mais completas do sistema sejam entregues. Dois modelos de processos comuns são o modelo de prototipação e evolucionário.

O modelo de prototipação pode ser usado como um processo dentro de qualquer outros dos modelos citados anteriormente (cascata, incremental etc.) ou sozinho. Sua vantagem é que devido a incerteza de um projeto complexo, um protótipo pode ser construído a fim de validar a aplicação que será construida e até mesmo levantar requisitos através do *feedback* dos *stakeholders*[1]. Um dos problemas com essa abordagem, segundo Pressman (2011), é que os *stakeholders* podem enxergar o sistema como algo operacional, e

[1] O palavra inglesa *stakeholders* denota qualquer pessoa interessada no projeto.

assim, algo que foi feito rapidamente apenas como protótipo (sem foco em qualidade) pode ser utilizado, tornando o código sem qualidade e não performático. A Figura 6 mostra o paradigma da prototipação.

Figura 6 – Paradigma da prototipação



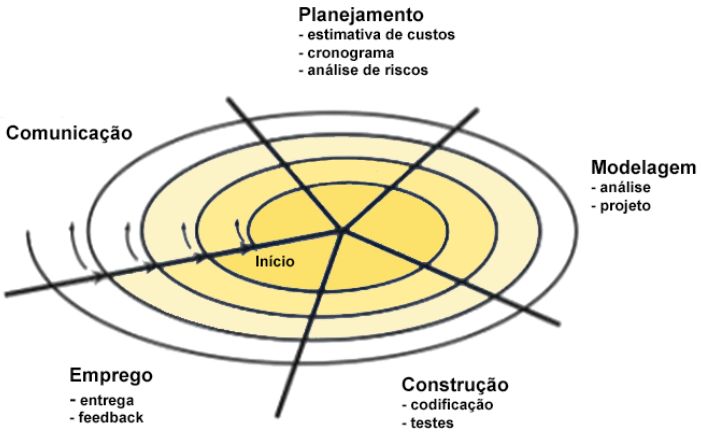
Fonte: Adaptado de Pressman, 2011.

Outro modelo de processo evolucionário é o espiral. Proposto por Barry Boehm, ele agrega a natureza iterativa da prototipação com o modelo em cascata. Esse modelo fornece uma maneira para construção rápida de várias versões do sistema. Nas primeiras versões, um modelo ou um protótipo é contruido. A partir das futuras iterações, esse protótipo torna-se um sistema cada vez mais completo. A Figura 7 mostra um espiral com 4 atividades, as quais podem ser modificadas pela equipe. O ciclo de desenvolvimento começa no centro e a equipe de desenvolvimento define as etapas (eg. comunicação, planejamento, modelagem, construção etc.). Como dito anteriormente, cada volta no ciclo define uma iteração.

2.2.4 PSP e TSP

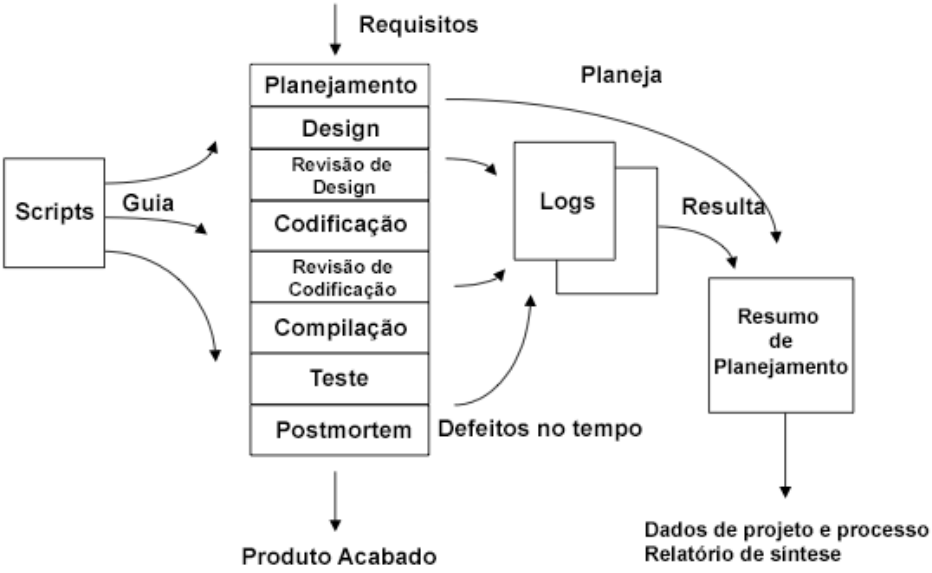
O processo de *software* pessoal, ou *personal software project*, é uma metodologia de desenvolvimento que encoraja o desenvolvedor a conhecer melhor sobre si mesmo para poder estimar e desenvolver *softwares* com mais qualidade e acertividade no que tange as estimativas. A Figura 8 ilustra a dinâmica do modelo. Nela observa-se que o PSP é composto de *scripts*, ou guias, que dizem as ações que o desenvolvedor deve executar para cada etapa. A medida que o desenvolvedor evolui nas etapas, esses guias podem solicitar que informações sejam adicionadas nos logs. Na fase de codificação, por exemplo, cada erro que você encontra no código e seu tempo de correção é anotado. Ao final do projeto, além de ter o sistema, o desenvolvedor tem um relatório que irá ajudá-lo nas estimativas para futuras funcionalidades desse e de outros projetos. (HUMPHREY, 2000)

Figura 7 – Modelo espiral



Fonte: Adaptado de Pressman, 2011.

Figura 8 – Modelo do PSP

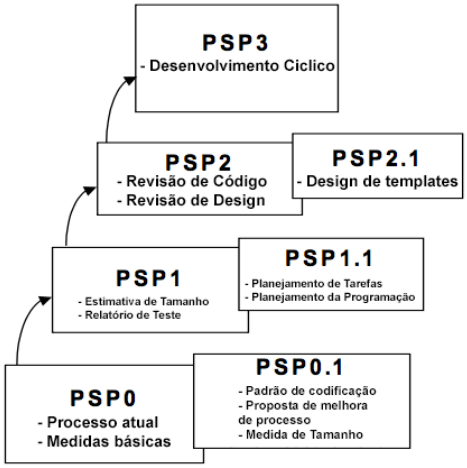


Fonte: Adaptado de Humphrey, 2000.

Humphrey (2000) mostra em seu trabalho que a introdução ao PSP/TSP em universidades segue o roteiro da Figura 9. O curso geralmente possui duração de um semestre e foca na construção de aproximadamente 10 programas. Inicialmente o aluno inicia no que é chamado de PSP 0 (PSP nível 0), o qual ele se utiliza de suas práticas atuais de desenvolvimento e apenas registra o tempo que gastar em cada fase do ciclo de desenvolvimento. Esse processo é melhorado através dos outros níveis até que ele atinja um nível superior e consequentemente aprenda a maioria dos métodos do PSP. No caso de sua aplicação para indústria, há um curso com cerca de 10 programas que levam em torno

de 120 a 150 horas divididos em 14 dias.

Figura 9 – Níveis do PSP



Fonte: Adaptado de Humphrey, 2000.

Além do PSP existe o TSP (*team software process*) que é a aplicação do PSP para criar uma equipe de projetos autodirigida. Pressman (2011) explica que os membros de uma equipe TSP estabelecem objetivos para o projeto, adaptam o processo para atender as expectativas do projeto e trabalham continuamente para aperfeiçoar o precesso de engenharia.

2.2.5 Processo Unificado da Rational (RUP)

O processo unificado é oriundo da discussão de Ivar Jacobson, Grady Booch e James Rumbaugh sobre a necessidade de um processo de software digirido a casos de uso, centrado na arquitetua, iterativo e iterativo incremental. Assim, o processo unificado utiliza-se de recursos e características de modelos tradicionais de processo de software e alguns dos princípios de metodologias ágeis como comunicação com o cliente. Esse processo é proprietário e foi criado inicialmente pela empresa *Rational Software Development*, qual foi adquirida pela IBM. (PRESSMAN, 2011) (MACHADO, 2013)

O processo unificado é composto segundo Machado (2013) por:

- **Papéis**: define quem é responsável por uma tarefa;
- **Artefatos**: o que será gerado durante cada tarefa. Pode ser um documento, um modelo do sistema etc;
- **Atividades**: o que cada responsável executa para atingir os objetivos do projeto e
- **Fluxo de atividades**: orienta a execução das atividades.

Além dos papéis, o RUP possui práticas que visam resolver problemas que ameaçam projetos como: alta complexidade, falta de definição formal de um processo de gerenciamento de mudanças, inconsistências não detectadas em requisitos (design e implementação), testes insuficientes, controle subjetivo, falha ao atacar riscos do projeto, comunicação ambigua e imprecisa e automação insuficiente. A seguir são vistas as práticas de acordo com Machado (2013).

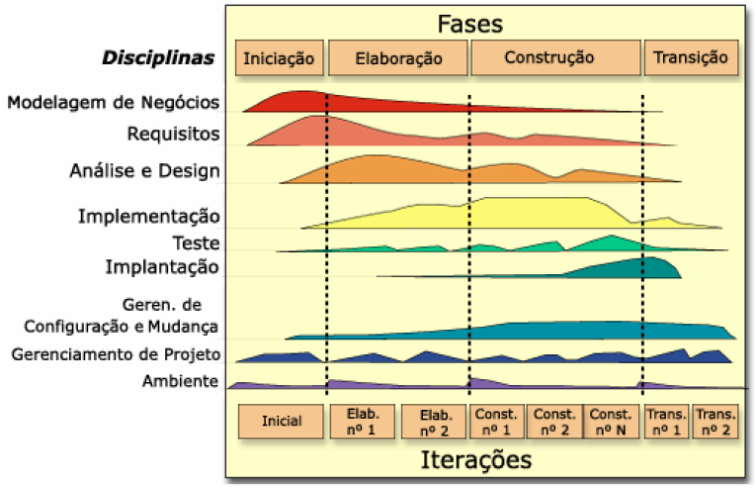
1. **Desenvolvimento iterativo**: Utilização do modelo em espiral e a construção do *software* se em iterações. Para mais informações sobre o modelo, consulte a seção 2.2.3.
2. **Gerenciar os requisitos**: Consiste em elicitar, organizar e documentar a funcionalidade e restrições do *software*.
3. **Usar arquitetuas baseadas em componentes**: Sistema é organizado em módulos coesos e fracamente acoplados. Um modelo pode ser desenvolvido por um terceiro, aproveitado de outro projeto e desenvolvido desde o início.
4. **Modelar software visualmente**: Utilização da modelos (e.g UML) para descrever o sistema de forma não ambigua.
5. **Verificar qualidade do software continuamente**: Testes em cenários de utilização a cada incremento.
6. **Controlar mudanças**: Cada mudança do *software* necessita de um requisito de mudança que descreve criteriosamente as mesmas.

Além disso, o RUP é um processo orientado a casos de uso e pode ser adaptado (com restrições) pela organização. A Figura 10 ilustra a estrutura do RUP. O projeto é dividido nas fases de iniciação, elaboração, construção e transição. Cada é composta por disciplinas que dão diretrizes de como proceder. Cada fase pode ter uma ou mais iterações. Observa-se na Figura 10 que inicialmente os maiores esforços estão na modelagem, requisitos e análise. A fase de iniciação é fundamental para mitigar os riscos. A elaboração é responsável pelo planejamento das atividades e recursos necessários para o projeto, especificação detalhada dos requisitos e projeto de arquitetura. A fase de construção em como objetivo evoluir o sistema que está sendo proposto. A transição marca a entrega para os usuários de um produto quase pronto com apenas questões de empacotamento, entrega, treinamento, suporte e manutenção.

2.3 DESENVOLVIMENTO ÁGIL

O desenvolvimento ágil surgiu através da assinatura do “Manifesto para o Desenvolvimento Ágil de Software” por dezesseis desenvolvedores, autores e consultores da área

Figura 10 – Estrutua do RIP



Fonte: Adaptado de Machado, 2013.

de *software* e Kent Beck. Esse manifesto tinha como objetivo quebrar o velho conceito que havia em relação ao desenvolvimento dos projetos de *software*, os quais possuiam muitos formalismos, documentação excessiva etc. Pode-se dizer que o movimento ágil é uma resposta ao mercado que se tornou, com o passar do tempo, mais competitivo e exigiu que projetos fossem executados rapidamente e ao mesmo tempo com capacidade de se adaptar as mudanças de forma rápida e menos custosa. (PRESSMAN, 2011)

O manifesto ágil possui os seguintes valores segundo Beck e Beedle (2015):

- Indivíduos e iterações sobre processos e ferramentas;
- Software operacional acima de documentação completa;
- Colaboração dos clientes acima de negociação contratual e
- Respostas a mudanças acima de seguir um plano.

Além dos valores pregados pelo manifesta, ele possui os seguintes princípios segundo Beck e Beedle (2015):

- Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado;
- Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente;
- Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo;
- Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto;

- Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho;
- O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face;
- Software funcionando é a medida primária de progresso;
- Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente;
- Contínua atenção à excelência técnica e bom design aumenta a agilidade;
- Simplicidade – a arte de maximizar a quantidade de trabalho não realizado – é essencial;
- As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis e
- Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Nas próximas seções são vistos alguns processos ágeis utilizados no mercado.

2.3.1 Extreme Programming – XP (Programação Extrema)

Extreme Programming é uma metodologia ágil de desenvolvimento de projetos de *software*, criada por Kent Beck em 1996, ideal para pequenas e médias equipes de desenvolvimento. Como a maioria das metodologias ágeis, é uma metodologia com foco em projetos cujos requisitos não são bem definidos no início do projeto e cujos requisitos tendem a mudar constantemente. As principais características da XP segundo Sbrocco e Macedo (2012) são:

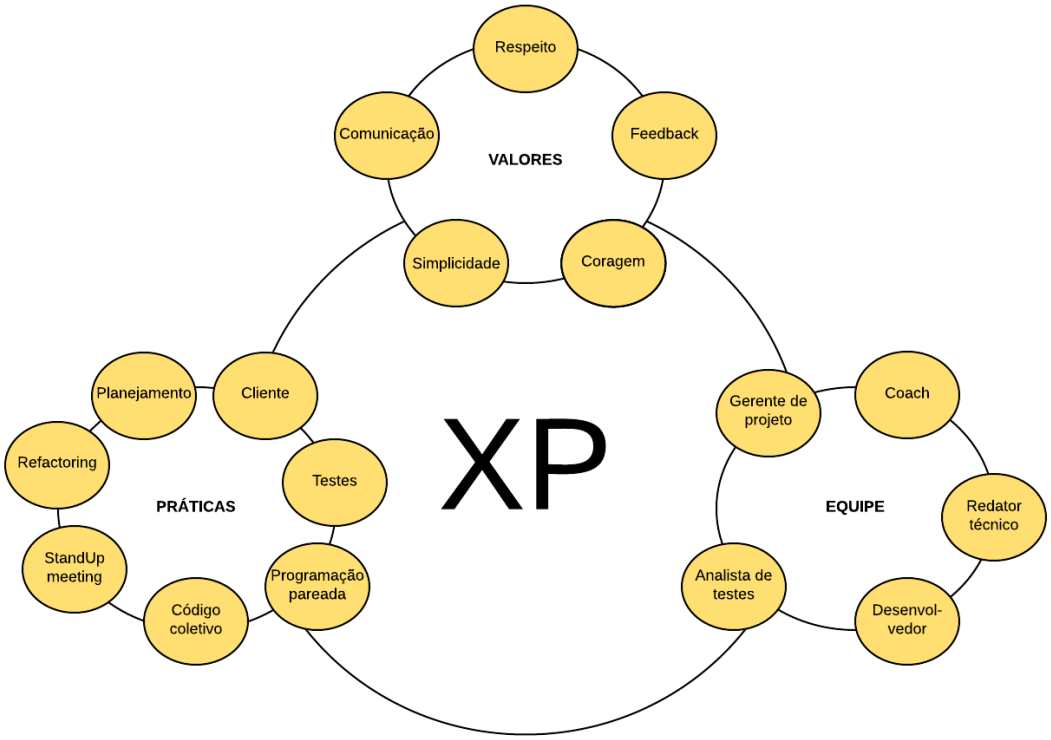
- *Feedback* constante;
- Abordagem incremental e
- Encojajamento da comunicação entre as pessoas envolvidas.

Por abordagem incremental entende-se o modelo discutido anteriormente nesse capítulo, ou seja, a evolução do sistema se dá através de escolhas de funcionalidades principais do sistema, as quais são cada vez melhoradas e extendidas conforme o projeto evolui. Em cada final de incremento, a equipe tem um sistema "utilizável", o qual os *stakeholders* podem validar e dar opiniões: o que facilita sua mudança em relação aos requisitos e encora a comunicação como identificado por Sbrocco e Macedo (2012).

A XP é uma metodologia de projetos de *software* que dá preferência a utilização do paradigma de orientação a objetos e pequenas equipes de até 12 desenvolvedores. Além disso, a XP é uma metodologia flexível que pode ser adaptada a um projeto conforme a

necessidade. A metodologia possui divisões, as quais possuem características atreladas. As divisões são: valores, equipe e prática, as quais são detalhadas mais adiante segundo Sbrocco e Macedo (2012). A estrutura da metodologia é observada na Figura 11.

Figura 11 – Estrutura da XP



Fonte: Adaptado de Sbrocco e Macedo, 2012.

Para que uma equipe de desenvolvimento diga que construiu um *software* utilizando XP, é necessário que a equipe siga os seus valores sob a forma de diretrizes, mesmo que ela adapte a metodologia com sua necessidade.

Os valores da XP, como dito anteriormente e mostrado na Figura 11, são comunicação, *feedback*, simplicidade e coragem. Na questão de comunicacão, é importante que haja a comunicação entre os desenvolvedores e o cliente de forma direta (até mesmo face a face). O cliente deve tornar-se e sentir-se peça fundamental no processo de desenvolvimento.

Outro valor importante da XP é o *feedback*. Por utilizar uma abordagem incremental, é fundamental que a comunicação flua por parte dos desenvolvedores com o cliente, assim como do cliente para os desenvolvedores. A cada liberação do incremento de *software*, o cliente pode testar o que foi produzido para validar e ajudar os desenvolvedores no rumo que o sistema deve tomar.

A simplicidade é outro valor importante da XP. Deve-se implementar uma funcionalidade da forma mais simples possível para funcionar. O XP, como uma metodologia ágil, parte do princípio de que os requisitos são incertos e estão sempre mudando, assim

deve-se sempre pensar no agora e deixar funcionalidades que somente serão utilizadas no futuro para depois. O foco está somente em funcionalidades que agregam valor ao cliente no momento.

Outro valor importante da XP é a coragem. Toda a equipe (desenvolvedores e equipe de projeto) devem tomar atitudes complexas e de alto risco. Alguns problemas que surgem devem ser pensados durante ou antes do desenvolvimento. Todo projeto deve ser discutido e analisado ao receber a proposta do cliente. A coragem é um valor que deve ser tomado em qualquer alteração ou sugestão de alteração no escopo do produto, processo e também nas pessoas. O desenvolvimento deve seguir um ritmo sustentável e todos os artefatos (e.g. documetação, fontes etc.) devem estar disponíveis para os membros da equipe. Outro fator importante é a refatoração, ou seja, os desenvolvedores e projetistas devem tomar um tempo para melhorar o código ou arquitetura do sistema. O contato dos desenvolvedores com o cliente é fundamental nessa metodologia para gerar *feedback*. Testes automatizados e programação em pares (com um codificando e outro revisando) são encorajados também. O respeito é outro valor fundamental que faz com que haja respeito entre o cliente e a equipe de desenvolvimento para estimular a crítica e sua aceitação de ambas as partes.

Além dos valores, há boas práticas que os seguidores da XP devem seguir durante o projeto. O padrão de desenvolvimento, depois que a equipe é formada, é imporante para que os desenvolvedores consigam entender o códigos um dos outros. Além disso, a equipe deve escolher a melhor linguagem de programação e as ferramentas que se adequam melhor ao projeto que vai ser construido. A programação deve seguir um *design* mais simples possível, pois os requisitos sempre mudam em projetos incertos e seu custo é alto. Quanto mais simples um código é, mais fácil sua manutenção.

O cliente é um participante ativo na construção do projeto. O incremento de *software* começa com a história de usuário (feita juntamente com o cliente), a qual será quebrada em atividades. Todas as histórias são priorizadas pelo cliente, de modo que as funcionalidades mais importantes para o sistema sejam feitas em primeiro lugar. Todos os dias é realizada uma reunião chamada de *stand up meeting*, ou reunião em pé, para discutir o que foi feito ontem, os problemas e as histórias que serão desenvolvidas por cada um. As práticas de *pair-programming* (programação em pares), refatoração de código, testes (unitário ou aceitação), metáfora (palavras que facilitam comunicação com o cliente), ritmo sustentável (carga horária de no máximo 40 horas semanais), integração contínua (toda equipe sabe o que foi desenvolvido e o impacto da funcionalidade com o sistema atual é sempre testado) e *releases* curtos são outras práticas utilizadas pela metodologia.

A equipe de um projeto XP é composta pelo gerente de projeto, o qual é responsável por ser o maior elo entre a equipe de desenvolvimento e o cliente. Prazos, custos, cobranças da equipe e ritmo de trabalho são responsábilidades desse papél. Outro papél importante

na XP é o *Coach*, ou técnico, que é um profissional com grande experiência e conhecimento da metodologia. Qualquer dúvida deve ser sanada por esse profissional. O desenvolvedor é o profissional responsável por colocar a ideia em prática. Não há diferença entre um desenvolvedor, analista, projetista. O analista de teste (reponsável por validar o produto) e redator técnico (responsável pela documentacão) completam a equipe.

Além do XP, existe também o IXP(*Industrial Extreme Programming* ou programação extrema industrial. Pode-se dizer que é o XP incorporado ao DNA de uma empresa (IXP, 2015). A IXP difere da XP original por contar com uma maior inclusão de gerenciamento, papel expandido para clientes e técnicas atualizadas. A IXP adiciona as seguintes novas práticas no desenvolvimento conforme Pressman (2011):

- **Avaliação imediata**: Feita antes do projeto para verificar se o o ambiente de desenvolvimento é apropriado para IXP, se a equipe é composta por interessados, se existe um programa de qualidade diferenciado que apoia melhora contínua, se existe uma cultura de apoio aos novos valores de uma equipe ágil etc.
- **Comunidade de projeto**: Equivale ao papél de equipe na XP, mas pode ser composta, por exemplo, por um tecnólogo, clientes fundamentais e outros envolvidos que desempenham um papel importante no projeto.
- **Mapeamento de projeto**: Avaliação do projeto pela equipe IXP para verificação da justificativa conforme os objetivos da organização. A verificação da consequência em relação a sistemas ou processos existentes também é feita.
- **Gerenciamento orientado a testes**: Criticos avaliam o estado do projeto frequentemente para obter o progresso. É importante para determinar se os objetivos foram atingidos ou não.
- **Retrospectivas**: Tem como objetivo revisar itens, eventos e lições aprendidas a cada iteração de *software* ou do desenvolvimento da versão completa. Tem como objetivo melhorar o processo de aplicação da IXP.
- **Aprendizagem contínua**: Incentiva as pessoas envolvidas a aprendere novos métodos e técnicas para melhorar qualidade do proceso aplicado.

Não obstante a essas características, o IXP modifica a XP no uso da prática de SDD (*story-driven development*). Na SDD, histórias para testes de aceitação são realizadas antes da prática de desenvolvimento. Além disso, é encorajado o uso do DDD (*domain-driven design*), ou seja a construção de um modelo de domínio que representa o pensamento de especialistas sobre um assunto da disciplina. A Figura 12 mostra os valores (em vermelho) e práticas da IXP.

Figura 12 – Valores e práticas da IXP

Fonte: Adaptado de IXP, 2015.

2.3.2 Feature Driven Development (FDD)

A metodologia FDD (*feature driven development*)[2] é uma metodologia criada nos anos noventa por Jeff De Luca e Peter Coad para implementação de um sistema bancário internacional que não poderia ser implementado no prazo. Essa metodologia proporciona aos envolvidos no projeto formas de interação e controle fáceis que se traduzem em regras de fácil entendimento e resultados rápidos. Como outras metodologias ágeis, o desenvolvimento é focado em entregas que agregam valor ao cliente através da disponibilização de códigos bem desenvolvidos e testados. Sua utilização se dá tanto em equipes pequenas quanto grandes. Uma característica importante da metodologia é seu foco na qualidade, a qual é enfatizada desde o começo do projeto. Cada nova funcionalidade feita pela equipe deve ser testada e disponibilizada para o usuário. Além disso, a metodologia permite o acompanhamento do progresso do desenvolvimento através de gráficos por parte da equipe e do cliente. (SBROCCO; MACEDO, 2012)

Como outras metodologias ágeis, a FDD é centrada em práticas. No FDD são cinco:

1. Modelagem de objetos de domínio;
2. Desenvolvimento por funcionalidade;
3. Entregas regulares (builds);
4. Formação da equipe de projeto e

[2] O projeto FDD possui um site on o leitor pode obter mais informações *http://bit.ly/1g7hlE1*.

5. Posse individual do código (classes/*features*).

A modelagem de objetos tem como objetivo estudar, analisar e modelar o sistema que vai ser desenvolvido. Essa atividade convida os responsáveis pelo projeto a realizar tarefas como entrevista, diagramas UML e outros métodos que ajudem os desenvolvedores na implementação. (SBROCCO; MACEDO, 2012)

O desenvolvimento por funcionalidade tem como objetivo construir uma lista de funcionalidades a partir dos requisitos. Esses requisitos são classificados de acordo com a área de negócio que pode ser:

- Áreas de Negócio (*Business Areas, Major Feature Sets*);
- Atividades de Negócio (*Business Activities, Features Sets*) e
- Passos da Atividade de Negócio (*Activity Steps, Features*).

A entrega regular é caracterizada por manter sempre a entregas para o cliente e trabalhar sempre com o sistema mais recente. Um sistema de versionamento como o GIT ou SVN são extremamente necessários para que isso ocorra.

A metodologia FDD possui cinco papéis: gerente de projeto, arquiteto-chefe/ especialista do negócio, equipe de modelagem/planejamento, programador-chefe e equipe de funcionalidades "*features*". O gerente de projetos é o responsável por alocar as pessoas de acordo com suas competências para o projeto e atividades do projeto. Ele é reposponsável por entrar em contato direto com o cliente e extrair os requisitos e acomplanhar o processo de desenvolvimento para que esteja de acordo com o FDD. Um documento contendo as regras de negócio do projeto é feito por ele e os especialistas também para auxiliar os desenvolvedores. O arquiteto-chefe é uma pessoa que deve ser consultada para dúvidas em relação a arquitetura do sistema ou de regra de negócios. Pode não ser necessário, caso o sistema seja muito simples. A equipe de modelagem é responsável por produzir diagramas ou outros artefatos necessários para tornar a implementacão mais fácil. Esse profissional também é responsável por dividiras *features* para a equipe de funcionalide, que são os desenvolvedores propriamente ditos.

A última prática do FDD é a posse individual do código (classe/features). Uma lista de funcinalidades e seus reponsáveis por ela é elaborada. Inspeções regulares através de validação do que foi produzido, gerenciamento de configuração e mudança (através de testes e versionamento) e um relatório/visibilidade de resultados, através de gráficos que indicam o progresso do projeto e suas *features* também são utilizados.

2.3.3 Crystal

A Crystal, criada por Alistair Cockburn em 1998, é uma família de tecnologias criada com a necessidade de suprimir o mercado de trabalho, o qual tornava-se na época cada vez mais complexo devido a abrangência da utilização de tecnologias e automatização de processos não apenas por grandes empresas, mas por empresas de pequeno e médio porte. A dificuldade de utilização de metodologias tradicionais fez com que uma metodologia que priorizasse a adaptabilidade surgisse. (PRESSMAN, 2011) (SBROCCO; MACEDO, 2012)

A fim de atingir o objetivo da adaptabilidade, foram criadas várias metodologias com papéis, padrões de processos, produtos de trabalhos e práticas diferentes que pudessem ser escolhidos de acordo com o projeto. (PRESSMAN, 2011)

Sbrocco e Macedo (2012) mostra que a metodologia possui um código genérico com o objetivo de atender vários tipos de projetos, e que a metodologia foi dividida em cores: de acordo com o quão crítico o sistema é.

Tabela 1 – Divisão da família Crystal

Cores	Nº Desenv.	Em caso de falha
Clear	1–6	Perdem dinheiro, mas recuperam facilmente
Yellow	7–20	Pedem dinheiro discretamente
Orange	21–40	Perdem dinheiro substancialmete
Red	41–100	Perda de dinheiro e até vidas humanas

Fonte: Sbrocco e Macedo, 2012.

A metodologia Crystal possui, como as outras metodologias, alguns princípios segundo Sbrocco e Macedo (2012):

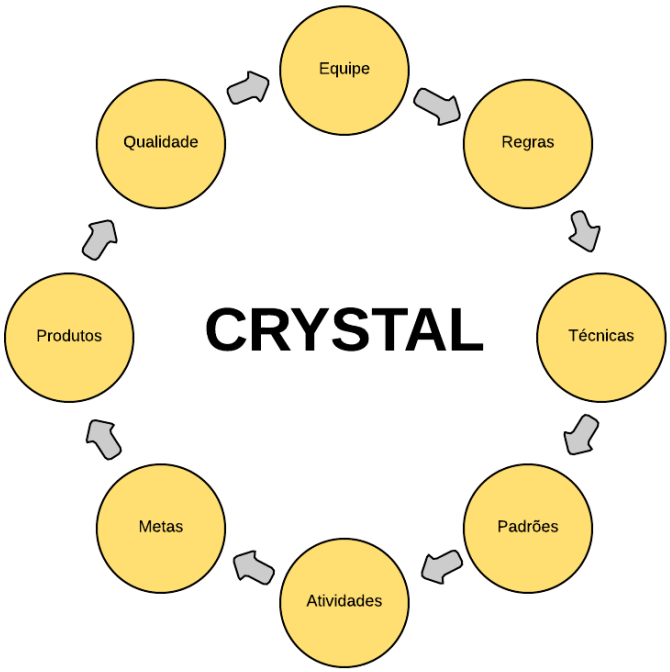
1. Trabalho face a face com o cliente: considera que envolver o cliente nas iterações e nas decisões é muito mais produtivo;
2. Peso significa custo: quanto maior a complexidade, maior o custo;
3. Usar metodologias diferenciadas para equipes maiores;
4. Mais cerimônias maior criticidade: quanto mais diálogos com os envolvidos melhor;
5. Comunicação eficiente (*feedback*) é melhor que entregas que não funcionam;
6. Habitabilidade: tolerância em lidar com seres humanos e
7. Eficiência no desenvolvimento.

A metodologia Crystal possui, segundo Sbrocco e Macedo (2012), além de seus princípios alguns pensamentos ou filosofias por trás. Num projeto de *software* é importante considerar que cada pessoa trabalha de maneira diferente da outra, assim as limitações de cada um devem ser consideradas em uma empresa com muitos projetos. Outro pensamento é que mesmo que um projeto de *software* seja da mesma área ou tipo, eles podem diferir

em necessidades. Além disso, a metodologia, como muitas metodologias ágeis, consideram o desenvolvimento um projeto um processo social que envolve intensa comunicação dos envolvidos, assim é sempre importante também ouvir ideias de usuários (mesmo que inequívoca e inviável no momento). Foco na qualidade que agrega valor ao negócio, consideração de novas técnicas e tecnologias são também importantes em uma área que está em constante evolução e mudança. Por fim, a melhoria do processo vem, em suma, através da experiência da metodologia nos projetos. Na Figura 13 é mostrado o ciclo de vida dessa família de metodologia que é baseado em integrações. Todo o processo deve funcinar de forma cíclica com um time de qualidade que siga as filosofias e princípios.

A equipe da metodologia crystal vai depender complexidade (como visto na Tabela 1). Caso a equipe seja menor, alguns membros podem assumir mais de uma função. O patrocinador é reponsável pelo investimento financeiro do projeto. O coordenador do projeto possui o papel de definir as entregas e entrar em contato com o patrocinador. Analistas de negócios, usuário *stakeholder* (um usuário do sistema pelo menos), *designer/projetista* (responsável pela arquitetura e elementos de IHC), programadores, testadores (para teste de regressão) e redatores (responsáveis por documentar atas, tabular questinários e entrevistas) são outros participantes desse processo. (SBROCCO; MACEDO, 2012)

Figura 13 – Ciclo de vida da família de metodologia Crystal



Fonte: Adaptado de Sbrocco e Macedo, 2012.

2.3.4 Processo Unificado Ágil (UAP)

O processo unificado ágil (ou em inglês *Agile Unified Process*) é uma abordagem híbrida criada por Scott Ambler através da combinação do RUP (visto na seção 2.2.5) com métodos ágeis. Através da combinação de metodologias, criou-se um *framework* capaz de ser aplicado em pequenos e grandes projetos de *software*. Assim, todos os princípios do movimento ágil (visto na seção 2.3) foram agregados ao AUP. Quando criado por Ambler, o UAP foi centrado nos seguintes princípios segundo Edeki (2013):

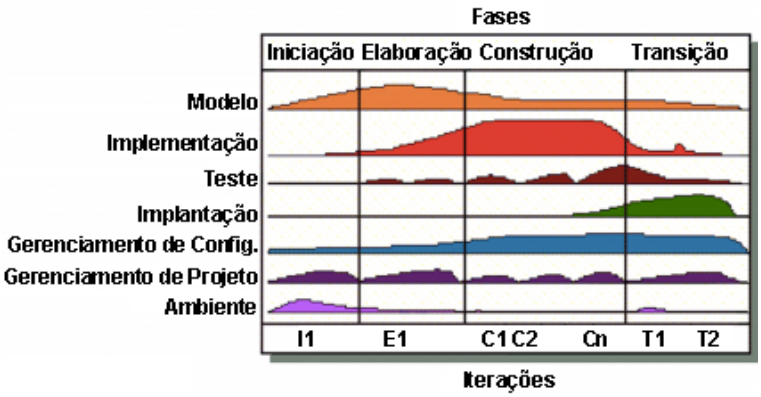
- A maioria das pessoas não lêem documentações detalhadas, no entanto elas precisam de direção e treinamento;
- O projeto deve ser descrito de forma simples em algumas páginas;
- O AUP deve estar em conforme com o manifesto ágil;
- O projeto deve entregar coisa que agregam valor ao negócio ao invés de funcionalidades desnecessárias;
- Desenvolvedores devem utilizar as melhores ferramentas disponíveis para tarefa que possuem e
- UAP é adaptado facimente através de ferramentas de edição de HTML.

Assim como o RUP, o UAP possui fases e disciplinas. Uma diferença do UAP com o RUP é que o UAP combina as disciplinas de modelagem, requisitos e análise e *design* em apenas uma chamada de Modelo. O fluxo de trabalho segue através de iterações de duas semanas, o qual segue o processo de acordo com as fases. A Figura 14 mostra o ciclo de desenvolvimento do UAP com suas fases e disciplinas. Na fase de iniciação, começa-se com a disciplina de modelo, a qual é responsável pelo delineamento do escopo de projeto, mitigação de riscos, custo cronograma e viabilidade. A fase de construção é a fase mais longa e é responsável por produzir o código executável. A disciplina de teste ratifica que o que foi produzido está nos conformes com o padrão de qualidade. A disciplina de implantação ratifica que o sistema desenvolvido foi implantado, enquanto a configuração ratifica que os artefatos produzidos estão estão mapeados e versinados. O gerenciamento de projeto direciona as atividades que ocorrem durante o ciclo de desenvolvimento do *software*. Por fim, a disciplina de ambiente é reposável por garantir que o time de projeto possui tudo que ele precisa para o projeto ter êxito. (EDEKI, 2013)

2.3.5 Método de Desenvolvimento de Sistemas Dinâmicos (DSDM)

O DSDM (*Dynamic Systems Development Method*) ou Método de Desenvolvimento de Sistemas Dinâmicos é uma adordagem de desenvolvimento ágil utilizada para desenvolver

Figura 14 – Ciclo do UAP



Fonte: Adaptado de Edeki, 2013.

e manter projetos de *software* cujo prazo é curto. Essa metodologia surgiu em 1994 como uma demanda de líderes de empresas que estavam insatisfeitos com métodos de desenvolvimento que eram caros, rígidos e não confiáveis. Essa metodologia baseia-se na versão modificada do princípio de Pareto. Em outras palavras, a metodologia propõe que 80% de uma aplicação pode ser entregue em 20% do tempo que levaria para que a aplicação completa fosse feita. A metodologia utiliza o modelo iterativo de desenvolvimento (pequenas entregas) e, além disso, somente o suficiente é feito. Detalhes e requisitos obscuros são deixadas para próxima iteração. Existe um grupo chamado DSDM Consortium formado por algumas empresas que utilizam-se dessa metodologia e que "mantêm" o *framework*. Mais informações podem ser encontradas em *http://bit.ly/1JWOGIU*. (PRESSMAN, 2011) (DSDM, 2015)

O DSDM basea-se em nove princípios conforme Sbrocco e Macedo (2012):

- Participação ativa dos usuários e *stakeholders*;
- Abordagem cooperativa e compartilhada;
- Equipes com poder de decisão;
- Entregas contínuas que fazem diferença;
- Desenvolvimento iterativo e incremental;
- *Feedback*;
- Todas as possíveis alterações durante desenvolvimento podem ser reversíveis;
- Fixar os requisitos essenciais e
- Teste em todo ciclo de vida.

Em relação aos testes, como o prazo dos projetos que utilizam-se de DSDM são rígidos, testes são feitos desde o começo (testes de regressão são bastante utilizados). (SBROCCO; MACEDO, 2012)

O ciclo de desenvolvimento segundo Pressman (2011) é composto por três ciclos iterativos que são precedidos por duas atividades de ciclos adicionais. São elas:

- **Estudo de viabilidade**: Tem como objetivo estabelecer os requisitos básicos de negócio, restrições e a viabilidade do uso do *framework* DSDM.
- **Estudo de negócio**: Tem como objetivo estabelecer requisitos funcionais e de informação. A arquitetura do sistema é definida também nessa etapa.
- **Iteração de modelos funcionais**: Tem como objetivo produzir um conjunto de protótipos funcionais que serão melhorados através do *feedback* dos usuários.
- **Iteração de projeto e desenvolvimento**: Pode ocorrer ao mesmo tempo que a iteração de modelo funcional. Tem como objetivo revisar o que foi feito na iteração anterior e certificar-se da existência de um processo de engenharia na produção de algum valor ao negócio do cliente.
- **Implementação**: É responsável por alocar o último incremento do *software* no ambiente operacional. Caso o *software* não esteja funcionando conforme o esperado ou venha alguma solicitação de mudança, o processo volta para a etapa de iteração de modelos funcionais.

2.3.6 Desenvolvimento de Software Adaptativo (ASD)

O desenvolvimento adaptativo de *software* ou (*adaptative software development*) é uma métodologia proposta por Sam Bayer e James Highsmith em 1997 para criação de projetos de *software* de maneira mais rápida. Essa maneira mais rápida de desenvolvimento torna-se fundamental em um ambiente que exige a construção de programas complexos e em um prazo curto onde os programadores não tem muito poder de decisão. (PRESSMAN, 2011) (SBROCCO; MACEDO, 2012)

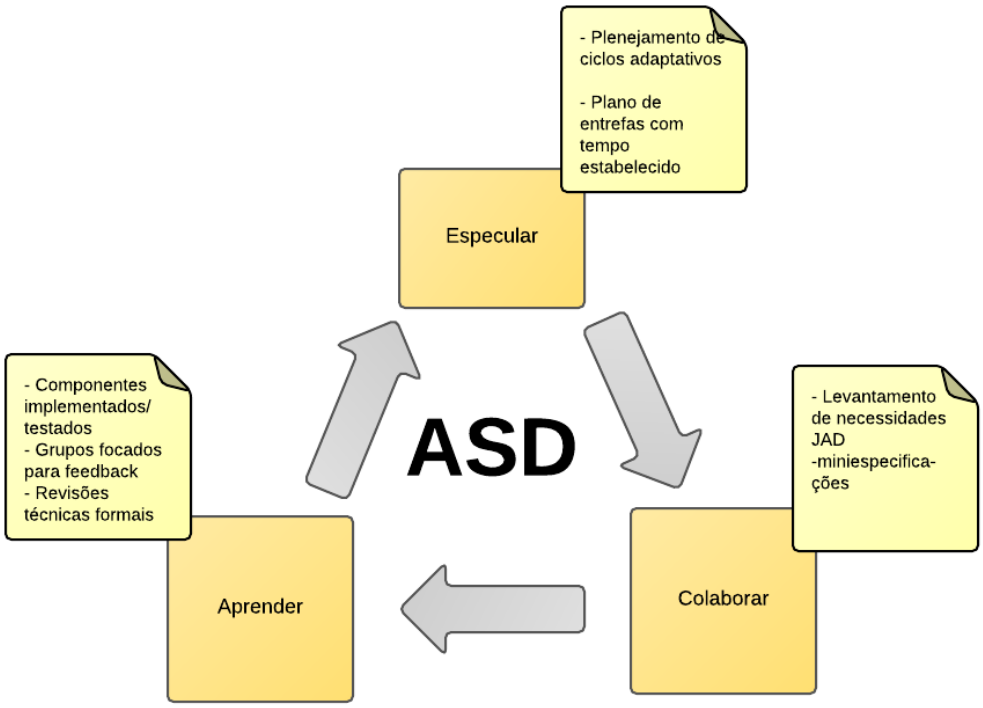
O ciclo da metodologia ASD possui três fases: especulação, colaboração e aprendizagem. A seguir são vistos cada um deles segundo (PRESSMAN, 2011).

Na fase de especulação há o início do projeto com o planejamento de ciclos adaptáveis. Esse planejamento envolve o levantamento de informações como: missão do cliente, restrições do projeto (e.g. data de entrega ou descrições de usuário) e requisitos básicos. Tudo isso será levantado para definição dos ciclos (incrementos) que farão parte do projeto de *software*. Na etapa de colaboração é feito o levantamento de necessidades, JAD e miniespecificações. A colaboração, como parte de uma metodologia ágil, é importante

e nessa metodologia não é diferente, assim gerentes de projeto e *stakeholders* atuam na priorização das funcionalidades do sistema para possibilitar a organização do projeto. As funcionalidades que são mais utilizadas e importantes vão ter uma prioridade maior. O aprendizado é uma fase importante pois possibilita que todo artefato produzido seja apresentado para equipe. Revisões junto com clientes e testes são versão beta são feitas nessa etapa. Nessa fase o desenvolvedor certamente aprende cada vez mais sobre o cliente, seus desejos e os próximos passos que o projeto deve passar. Os ciclos de revisão e teste são curtos e servem apenas para aprender com erros pequenos. (SBROCCO; MACEDO, 2012) (PRESSMAN, 2011)

A Figura 15 mostra o ciclo de desenvolvimento da metodologia. Depois da fase de aprendizado, há uma liberação de incremento de *software* que é passível que sofrer ajustes para ciclos subsequentes.

Figura 15 – Ciclo do ASD



Fonte: Adaptado de Pressman, 2011.

2.3.7 Scrum

A metodologia SCRUM[3], além de outras metodologias ágeis, foi fortemente influenciada por práticas do *lean* (ou manufatura enxuta). Essa metodologia foi proposta por Jeff Sutherland e posteriormente expandida por Schwaber e Beeddle. Como uma metodologia ágil, o SCRUM é ideal para projetos cujos requisitos são incertos – tendo

[3] O nome SCRUM é oriundo de uma atividade que acontece em uma partida de rugby.

assim uma grande capacidade de adaptar-se a mudanças de requisitos. Além disso, é ideal para projetos com prazo apertado. Sua principal função é orientar o desenvolvimento de *software* dentro de uma processo que possui as seguintes atividades estruturais segundo Pressman (2011):

1. Requisitos;
2. Análise;
3. Projeto;
4. Evolução e;
5. Entrega.

Em cada atividade metodológica, ocorrem tarefas que são realizadas dentro de um padrão de processo chamado de *sprint*[4] que são iterações de trabalho com duração de duas até quatro semanas. (PRESSMAN, 2011)

O SCRUM é baseado segundo Sbrocco e Macedo (2012) em seis características:

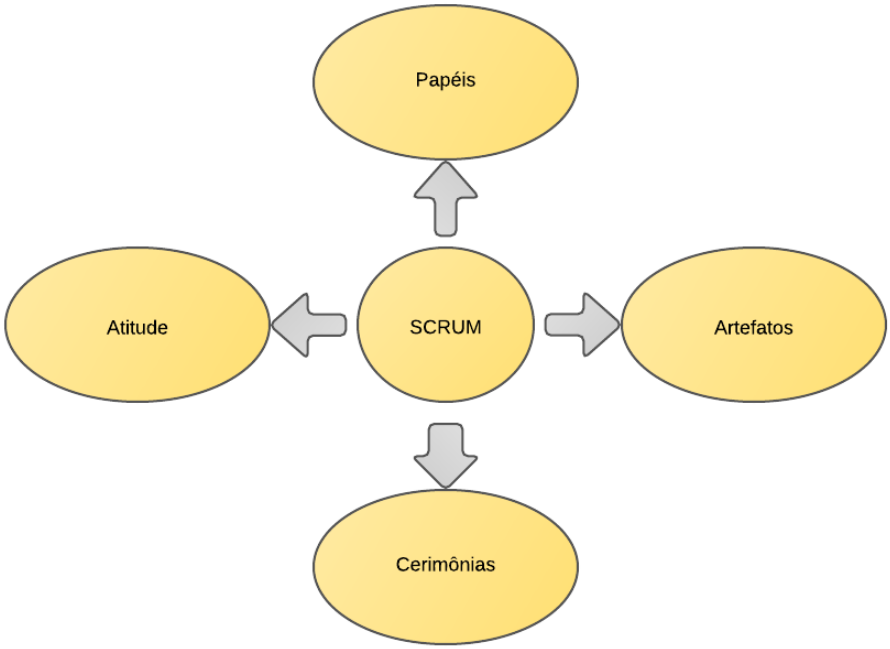
1. Flexibilidade de prazos;
2. Times pequenos;
3. Revisões frequentes;
4. Colaboração e
5. Orientação a objetos;

A Figure 16 mostra os fundamentos do SCRUM. Os papéis se dividem em três principais: *Product Owner*, SCRUM *Master* e *Team* ou equipe. Além desses três papéis há a possibilidade do cliente ser incluso, pois ele exerce uma participação importante na construção do sistema. As cerimômias são reuniões que acontecem durante o ciclo de desenvolvimento do projeto e são: *Daily Meeting*(ou *Daily Scrum*), *Sprint Review*, *Sprint Planning Meeting* e *Sprint Retrospective*. Os artefatos são documentos gerados através dessas reuniões ou cerimônias e são: *Product Backlog*, *Sprint Backlog* e *Burndown Chart*. (SBROCCO; MACEDO, 2012)

O ciclo de desenvolvimento do SCRUM é mostrado na Figura 17. Tudo começa com uma reunião entre os clientes e desenvolvedores para definição do *backlog* do produto. Esse backlog de produto nada mais é do que a lista de funcionalidades que o sistema deve ter.

[4] *Sprint* são corridas de velocidade em inglês.

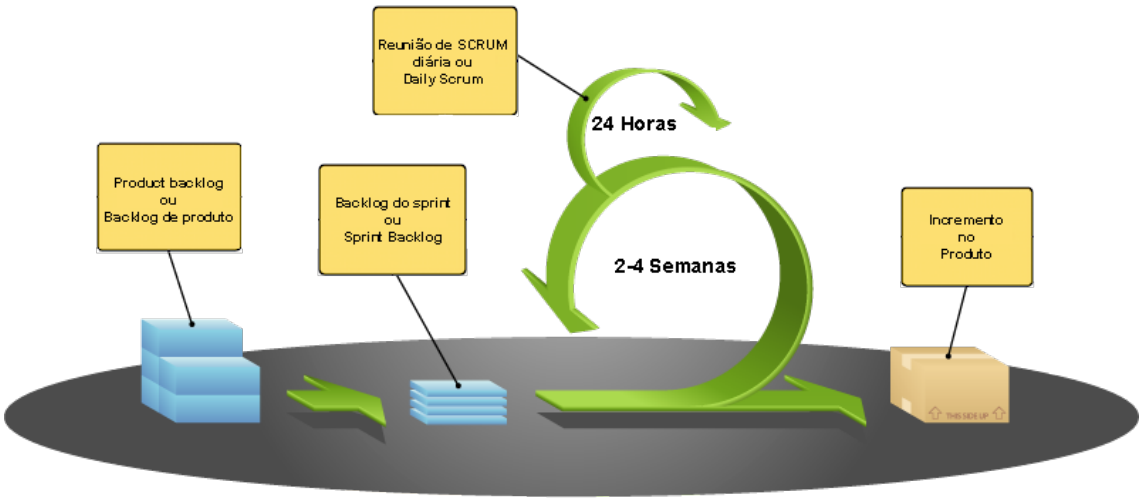
Figura 16 – Fundamentos do SCRUM



Fonte: Adaptado de Sbrocco e Macedo, 2012.

Após acordado com o cliente sobre as entregas, riscos e prazos, é eleito o SCRUM Master entre a equipe alocada para o projeto. Com o *backlog* de produto feito, é definido o que será feito na *sprint*: assim é feito o *backlog* de *sprint* através da cerimônia de *sprint planning* em que todos participam e tem duração de no máximo oito horas. Nela, o *product owner* irá selecionar items do *backlog* de produto e priorizar de acordo com a necessidade. Após isso, os desenvolvedores irão então quebrar as funcionalidades em atividades. Técnicas como *planning poker* são utilizadas para definir o tamanho de uma atividade e medir seu esforço. O *planning poker* é basicamente um jogo de cartas que os desenvolvedores utilizam para dar "nota" de complexidade para uma funcionalidade. Enquanto as notas das cartas divergirem entre eles, uma nova rodada é feita. A numeração de Fibonacci é geralmente empregada nesse processo. Em cada dia de trabalho é feita uma reunião (como mostra a Figura 17) chamada de *daily scrum* entre o SCRUM Master e o Time de no máximo 15 minutos cujo objetivo principal é saber o que cada um está fazendo, o que será feito hoje, se há algum impedimento e quais. Ao final de 2 a 4 semanas tem-se um incremento de *software*. Ao final da *sprint* é realizada a cerimônia de *sprint review* cujo objetivo é mostrar os resultados da *sprint* para o *Product Owner* e outras pessoas convidadas. A cerimônia de retrospectiva de *sprint* ou *retrospective sprint* é também realizada com a participação do time e SCRUM master. O *product owner* pode estar presente, embora sua participação não seja obrigatória. Essa cerimônia tem como objetivo ver o que deu certo e o que deu errado para melhorar as próximas *sprints*. (SBROCCO; MACEDO, 2012) (COHN, 2011)

Figura 17 – Dinâmica do SCRUM



Fonte: Adaptado de Sbrocco e Macedo, 2012.

Alguns artefatos produzidos, como mencionado anteriormente, com o SCRUM são *backlog* de produto, *backlog* de *sprint*, *task board* (utilizado mapear o estado das atividades na sprint) e o gráfico Burndown que é feito pelo SCRUM Master para verificar o esforço de cada interação e fazer o acompanhamente de forma gráfica. (SBROCCO; MACEDO, 2012)

2.3.8 Desenvolvimento de Software Enxuto (LSD)

O desenvolvimento de *software* enxuto ou LSD (*lean software development*) é uma metodologia de desenvolvimento que visa aplicar os princípios da fabricação enxuta para a engenharia de *software*. Os princípios que norteiam o LSD e que vieram do pensamento *lean* são:

- Eliminiar desperdício;
- Incorporar qualidade;
- Criar conhecimento;
- Adiar compromisso;
- Engregar rápido;
- Respeitar as pessoas e
- Otimizar o todo.

O LSD adapta cada princípio do pensamento *lean* ao processo de *software*. Para eliminar desperdício pode-ser executar as seguintes acões segundo Pressman (2011):

1. Não adicionar recursos ou funções que possivelmente não serão percebidas ou utilizadas pelos usuários;
2. Avaliar o impacto do custo e do cronograma de qualquer requisito solicitado;
3. Eliminar qualquer etapa de processo que não seja realmente necessária;
4. Estabelecer mecanismos para elicitação de requisitos;
5. Escrever e executar testes de qualidade;
6. Reduzir tempo para solicitar e receber uma decisão que afete o *software* e
7. Racionalizar o fluxo de informação entre as pessoas envolvidas no processo

No próximo capítulo é abordada essa metodologia com mais detalhes.

2.4 CONCLUSÃO DO CAPÍTULO

As pessoas envolvidas com o desenvolvimento de *software* sempre procuraram melhorar a entrega em termos de prazo, custo e qualidade e assim novos modelos foram sendo criados e adaptados para melhorar o processo de desenvolvimento. Toda essa evolução e criação de novos processos foram importantes para o entendimento da natureza do *software*, que ao contrário das outras engenharias é fruto de um trabalho social que envolve a participação massiva do cliente: principalmente nas metodologias ágeis. Pode-se dizer que não existe uma metologia de processo melhor ou pior e que tudo vai depender do tipo de projeto e complexidade. O modelo RUP, embora completo, não seria ideal para o desenvolvimento de um sistema simples, devido ao seu custo maior de implantação por exemplo. Dependendo da empresa, um ou outro processo se torna mais ideal para uma equipe de desenvolvimento. No caso do SCRUM, por exemplo, é essencial que todas as pessoas da equipe estejam sempre motivadas e sejam ágeis o suficiente para se adaptar a mudanças no projeto e isso não se adquire de uma hora para outra. Além disso, a comunicação entre todas as pessoas é essencial (principalemte com o cliente). O próximo capítulo aborda melhor o desenvolvimento enxuto de *software* ou LSD (*lean software development*).

3 LEAN SOFTWARE DEVELOPMENT

Não é possível falar do desenvolvimento de *software* enxuto (ou *lean software development*) sem antes falar do sistema de produção criado pela Toyota e que deu origem ao *lean*. Esse capítulo tem como objetivo aprofundar o leitor sobre a metodologia *lean* de desenvolvimento de *software* e dar uma introdução de seu surgimento na manufatura até sua utilização no desenvolvimento de *software* conforme Poppendieck e Poppendieck (2007).

3.1 INTRODUÇÃO

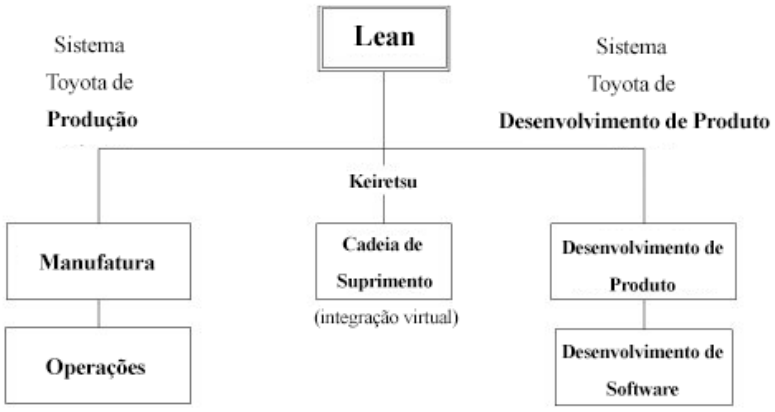
O sistema Toyota de produção, que deu origem ao *lean*, surgiu como uma alternativa ao modelo de produção em massa aplicado por Henry Ford em 1914. Henry Ford conseguiu, através da produção em massa, aumentar o salário de 2,4 dolares por 9 horas de trabalho para 5 dolares por 8 horas de trabalho diário. Tudo isso foi conseguido através da redução de 85% do trabalho sobre a produção do carro e da redução de 12 horas para apenas 90 minutos na sua produção, a qual passava através de uma linha de montagem com trabalho especializado. Nenhum trabalhador precisava saber fazer um carro inteiro, sendo necessário apenas treinar o funcionário para executar um tipo de trabalho na linha de produção. No sistema fordista de produção, o trabalho passou a ser essencialmente especializado e feito por qualquer tipo de pessoa, que poderia ser subtituída facilmente. Esse sistema de produção em massa demonstrou ser viável durante um certo tempo, mas apresentou problemas como a dificuldade de adaptação do modelo em relação a complexidade, que pode ser corroborada pela seguinte frase atribuída a Ford: “O cliente pode ter o carro da cor que quiser, contanto que seja preto”. No modelo de produção em massa, o custo de uma mercadoria é reduzido de 15% até 25% conforme o número de unidades dobra, porém o custo de produção sobe de 20% para 35% cada vez que a variedade dobra.

O sistema de produção da Toyota apresenta vários conceitos e ideias que possibilitaram melhorar o sistema de produção e deixar as empresas mais competitivas como o fluxo JIT (*Just in Time*) e o Jidoka (Autonomia). O fluxo JIT nada mais é do que eliminar o estoque e repensar o processo de produção em pequenos lotes. De modo prático, no sistema fordista tinha-se uma máquina especializada em fazer apenas uma peça do carro em massa, já no sistena da Toyota uma máquina poderia ser adaptada rapidamente para fazer diferentes partes rapidamente. O Jidoka, ou autonomia implantou a ideia de qualidade, fazendo com que qualquer evento anormal no processo de produção fizesse com que a produção fosse interrompida automaticamente, o que eliminava desperdício. O sistema deve ser projetado então para ser a prova de falhas, sem necessidade de um humano testando ou supervisionando para detectar erros, ou seja ele é isento de inspeção.

3.2 LEAN

A palavra *lean*, empregada hoje, foi resultado da nova atribuição dada pelo livro lançado em 1990 e entitulado *The Machine That Changed The World* para o que era chamado de JIT (*Just-in-Time*) ou sistema Toyota de produção. Depois de seu lançamento, o que antes era chamado de sistema de produção Toyota passou a ser chamar produção enxuta ou *Lean Production*. No início houve muita aversão ao emprego desse novo modelo de produção, mas conforme o tempo passou o pensamento *lean* atingiu sucesso e foi empregado da manufatura para outras áreas como: cadeia de suprimentos, desenvolvimento de produtos e no desenvolvimento de *software*. A Figura 18 mostra a extensão do modelo *lean* para outras áreas através da árvore da família *lean*.

Figura 18 – Árvore da família *lean*



Fonte: Poppendieck e Poppendieck, 2007.

Na área de manufatura, por exemplo, há o exemplo da companhia Dell de computadores que consegue entregar computadores customizáveis dentro de poucos dias. No que tange a cadeia de suprimentos, não adianta uma empresa ser *lean* se os fornecedores da qual ela depende não forem. No exemplo da Dell, as empresas envolvidas na cadeia de produção do computador aprendem como trabalhar além da fronteira da empresa para alinhar seus interesses com a cadeira inteira de suprimentos. A manufatura enxuta e desenvolvimento enxuto compartilham certas características, o que torna fácil seu emprego no desenvolvimento de produto como redução no tempo de manufatura e redução no tempo de desenvolvimento. O *software* nada mais é do que o desenvolvimento de um produto, sendo o desenvolvimento apenas um subconjunto de todo o processo. Nesse sentido, não há como melhorar a capacidade de desenvolvimento sem melhorar e entender o que constitui o desenvolvimento eficaz de um produto. Por isso, os princípios que regem o sistema Toyota de produção e o sistema Toyota de desenvolvimento são os mesmos. Os princípios do *lean* para o desenvolvimento de *software* são vistos a seguir.

3.3 OS PRINCÍPIOS DO *LEAN* NO DESENVOLVIMENTO

A seguir são vistos os princípios do *lean* no contexto do desenvolvimento de *software* que são:

- Eliminação de desperdício;
- Incorporar qualidade;
- Criar conhecimento;
- Adiar compromisso
- Entregar rápido
- Respeitar as pessoas e
- Otimizar o todo.

3.3.1 Eliminação de Desperdício

O princípio da eliminação de desperdício é relacionado a tudo que não agrega valor para o cliente. No desenvolvimento de *software* é importante entender o que significa esse valor, o qual pode mudar pelo fato dos clientes nem sempre saberem o que querem. Após entender esse desperdício, é preciso saber enxergá-lo dentro da organização. No desenvolvimento tudo que interfere no sentido de impedir que os usuários recebam esse valor ou qualquer atraso para que isso ocorra é considerado desperdício. Um exemplo comum de desperdício no desenvolvimento são funcionalidades extras. Segundo Poppendieck e Poppendieck (2007) apenas 20% das funcionalidades extras são usadas regularmente. Essas funcionalidades são basicamente coisas que não foram planejadas no início do desenvolvimento, mas que fazem com que a complexidade e manutenção do código se tornem custosa (além de testes desnecessários). Numa situação como essa, a empresa pode estar potencialmente gastando bastante com suporte e manutenção de funcionalidades que não agregam valor, ao invés de investir em coisas realmente úteis. Um mito que existe no desenvolvimento é que quanto mais cedo a especificação ocorrer, menor será o desperdício. Isso não é verdade, especialmente porque o cliente quase sempre não sabe o que precisa. Um escopo de funcionalidades que o sistema potencialmente pode ter pode ficar extenso e sofrer bastante modificações.

A real motivação por trás da emilinação de desperdício é descobrir e eliminar desperdícios que podem reduzir custos e tornar os produtos mais efetivos. A Tabela 2 mostra a equivalência dos desperdícios na manufatura e no desenvolvimento de *software*. Mais abaixo, é descrito um pouco sobre cada um desses desperdícios.

Tabela 2 – Os Sete Desperdícios

Manufatura	**Desenvolvimento de Software**
Estoques no processo	Trabalho parcialmente realizado
Superprodução	Funcionalidades extras
Excesso de Processamento	Reaprendizagem
Transporte	Transferência de controle
Movimentação	Troca de Tarefas
Esperas	Atrasos
Defeitos	Defeitos

Fonte: Poppendieck e Poppendieck, 2007.

Alguns exemplos de trabalhos parcialmente realizados podem ser: códigos não testados, códigos não documentados, códigos não mandados para ambiente de deploy (possivelmente por apresentarem conflitos na hora de mergear com outra *branch*) etc.

O pior desperdício de todos é o de funcionalidades extras, o qual remete a funcionalidades que não são necessárias para entrega do trabalho ao cliente. Se não houver uma razão clara ou necessidade economica, essa funcionalidade não deve ser desenvolvida.

A questão de reaprendizagem se refere a toda vez seja descoberto algum conhecimento que já foi aplicado anteriormente, mas que foi reaprendido. Uma alternativa para esse problema é aplicar técnicas de gestão de conhecimento na organização e no desenvolvimento.

A transferência de controle tange na dificuldade de comunicação de conhecimento tácito, ou seja todo aquele que é adquirido pela experiência e que é difícil de se adquirir através de documentação.

A troca de tarefas é um desperdício que acontece quando os desenvolvedores precisar trocar de uma tarefa para outra. Pode acontecer também de acontecer quando se tenta fazer mais de uma tarefa ao mesmo tempo. O planejamento das tarefas por um gestor de projetos pode eliminar esse problema com a aplicação de métodos como caminho crítico etc.

O atraso e defeitos são os outros problemas comuns de desperdício e que estão relacionados com o mal uso dos recursos humanos ou planejamento errado e falta de investimento em testes de prevenção.

3.3.2 Incorporar Qualidade

Esse princípio do *lean* no desenvolvimento de *software* está relacionado com a incorporação de qualidade no código desde o início. Assim, o teste não deve ser feito apenas no final do processo de desenvolvimento. A inspeção no processo de qualidade

deve-se dar principalmente antes que o defeito ocorra (controlando todas as condições de ambiente necessárias). No desenvolvimento de *software* existem técnicas como o TDD (*Test Driven Development*) para resolver esse problema de qualidade. O TDD tem seu início com a escrita do SUnit, uma biblioteca de testes, em Smalltalk por Kent Beck nos anos 90. Seu uso era encorajado para facilitar a execução de testes de *software* automatizados, que eram feitos de forma manual até então. Posteriormente houve a criação do SUnit para Java (chamado de JUnit) juntamente com Erich Gamma. Mais tarde houve a portabilidade para outras linguagens como Ruby, Python, C++, Perl e Php.[1]. Com o amadurecimento das bibliotecas SUnit, essa ferramenta deixou de servir apenas como uma forma de automatização de testes e passou a ser utilizada principalmente para atividades de *design* do código. O TDD é uma prática de desenvolvimento que envolve criar testes antes de escrever o código que será testado. Nela, o desenvolvedor escreve um pequeno trecho de teste para um código que ainda não existe. O desenvolvedor depois roda o teste, e naturalmente ele falha. Depois disso, o desenvolvedor apenas escreve código suficiente para que o teste passe. (BARAÚNA, 2013)

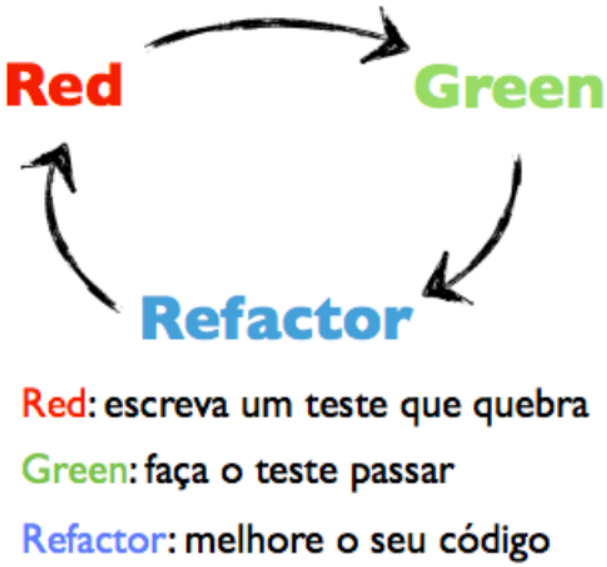
A vantagem de escrever os testes antes do código de produção é que ao final do desenvolvimento tem-se um código testado e que serve como uma documentação viva do sistema. Toda vez que seja preciso fazer uma modificação no sistema ou adição de novas funcionalidades, os testes são executados e, caso essa nova funcionalidade afete o comportamente do que já foi testado, o teste falha, o que ajuda na liberação de funcinalidades que possuem menos probabilidade de dar problema em produção. A Figura 19 mostra o ciclo *red – green – refactor* utilizado no TDD. Primeiramente o desenvolvedor escreve um código que falhe. Depois disso, ele faz o teste passar. Quando o teste passar, ele melhora o código para torná-lo mais legível. A ideia de fazer o código falhar é entender e projetar o código aos poucos. Quando se procede dessa maneira, pode-se descobrir todos os casos possíveis de um problema e assim, os códigos ficam melhores e menos propensos a erros.

Outra abordagem para incorporar qualidade é o BDD (*Behavior Driven Development*) que é uma evolução do TDD. O BDD começou como uma tentativa de melhor entender e explicar o processo do TDD. O problema era basicamente a palavra "Teste" no TDD que fazia com que os desenvolvedores não utilizassem o TDD como uma ferramente de *design* ou projeto para o *software*. Com o passar do tempo, o BDD evoluiu no sentido de testar o comportamento de um objeto ao invés de sua estrutura interna (como é o foco do TDD). (CHELIMSKY, 2012)

A Figura 20 ilustra o processo do BDD. Primeiramente o desenvolvedor descreve um cenário de teste. Após a escrita desse cenário, é escrita uma *step-definition*, que

[1] O padrão da família formada pela portabilidade dessa biblioteca de teste para todas as linguagens se chama SUnit.

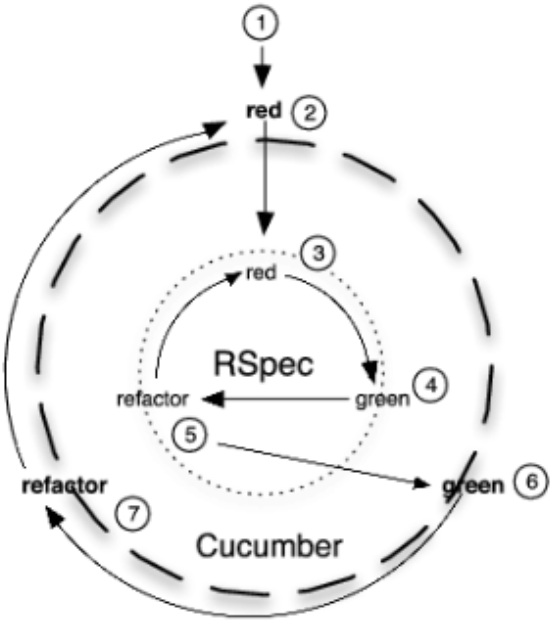
Figura 19 – Ciclo red – green – refactor



Fonte: Baraúna, 2013.

descreve um comportamento de uma funcionalidade do sistema. Nessa *step-definition*, o desenvolvedor escreve somente o um trecho de código necessário para o teste falhar. Após esse processo, entra-se num ciclo do BDD que foi mostrado anteriormente. Na linguagem ruby pode-se utilizar o Cucumber e o Rspec como *frameworks* para BDD e TDD respectivamente.

Figura 20 – Ciclo do BDD



Fonte: Chelimsky, 2012.

O trabalho do teste não pode-se limitar apenas a encontrar defeitos de produção no sistema e sim fazer que os erros e problemas posssam ser previnidos e detectados para que não entrem em produção.

3.3.3 Criar Conhecimento

O conhecimento do desenvolvimento de *software* não é criado em apenas uma etapa. Um dos problemas com o modelo de desenvolvimento em cascata (visto na sessão 2.2.1) é que ele assume que todo conhecimento é adquirido na etapa de análise, porém a construção de um *software* é um processo de criação de conhecimento. Mesmo que todos os detalhes sejam vistos na etapa de análise, os detalhes de *design* sempre ocorrem na etapa de contrução (quando o *software* está sendo codificado). Um processo de desenvolvimento que seja focado na criação de conhecimento deve esperar que o *design* evolua durante a codificação, principalmente porque o cliente quase sempre não sabe o que quer, assim o *feedbacks* dos *stakeholders* são importantes nesse processo e dão direcionamento para o que está sendo elaborado.

As empresas que possuem excelência no desenvolvimento de *software* aprendem no processo de construção do *software* e utilizam o conhecimento aprendido para futuros projetos. O processo de desenvolvimento precisa encorajar o aprendizado através do ciclo de desenvolvimento.

3.3.4 Adiar Compromisso

Um sistema de *software* é passivel de sofrer mudanças: todos os requisitos e riscos dificilmente vão ser mitigados em uma primeira etapa de análise. Porém, toda e qualquer decisão que torne o processo de mudança mais difícil ou irreverssível deve ser postergada até o último momento. É melhor deixar uma decisão importante para o último momento do que tomar uma decisão prematura que não leve a lugar algum. Isso não significa que todas as decisões devem ser postergadas, o time ou o líder de tecnologia deve comprometer-se a fazer com que a maioria das decisões sejam reversíveis com um sistema e arquitetura que sejam o mais flexível possível. As vezes a pressa pela eliminação de riscos a fim de diminuir a incerteza pode levar a decisões erradas que são irreversíveis. O melhor a se fazer é experimentar várias soluções, deixando as opções em aberto para serem tomadas o mais tarde possível com os testes e avaliações necessários.

3.3.5 Entregar rápido

Todo desenvolvimento de *software* deve ser feito da forma mais rápida possível para que o cliente não tenha tempo de mudar de ideia. Num mundo rápido e cada vez mais competitivo, a rapidez de implementação de uma ideia pode ser fundamental

para seu sucesso. Isso pode ser conseguido através da eliminação do desperdício e gastos desnecessários, diminuição das taxas de erros (através de inspeções no código) etc.

A teoria das filas é uma teoria que pode ajudar nesse sentido. Muitas vezes no desenvolvimento há novas demandas de cliente ou defeitos para serem corrigidos e organizar essas é de certa maneira desafiador. A teoria das filas ajuda no gerenciamento dessa lista de "requisitos".

3.3.6 Respeitar pessoas

Três dos quatro pilares do sistema de desenvolvimento de produtos da Toyota dizem respeito as pessoas envolvidas no processo de desenvolvimento de um produto. Assim, esse é um princípio muito importante que norteia o desenvolvimento *lean*.

As pessaos são motivadas a trabalharem em produtos de sucesso e para conseguir-se produtos de sucesso é necessário bons líderes. Empresas que respeitam seus empregados desenvolvem grandes líderes que por sua vez são capazes de conduzir grandes projetos. Além disso, é necessário assegurar-se que o conhecimento técnico necessário esteja presente na pessoa que designa a função. Distrubuir a pessoa de acordo com seu perfil e capacidade para um projeto é fundamental para o sucesso do projeto. O respeito as pessoas está relacionado com a capacidade do time em se organizar em relação aos planos e objetivos da empresa. Todas as pessoas precisam fazer parte do processo: isso envolve desenvolvedores, *stakeholders* etc. O processo de construção de *software* também é um processo de certa forma "social".

3.3.7 Otimizar o todo

Uma empresa que utiliza-se do *lean* otimiza a cadaia de valor desde o princípio até seu final. Se algum processo dentro dessa cadeia não é otimizado, o processo como um todo é afetado. Aqui é feita uma análise do processo de desenvolvimento *lean*, mas não adianta o processo de desenvolvimento ser otimizado, se outros processos que também fazem parte do processo de negócio apresentarem deficiência. Uma das ferramentas presentes e que foi mostrado no Capítulo 1 é a utilização de mapas de cadeia de valor. Através da análise da cadeia de valor é possível enxergar desperdícios com uma visão holistica de todo o processo da empresa.

3.4 DISCIPLINA, QUALIDADE e 5S

No desenvolvimento de *software lean* não é possível acelerar o desenvolvimento sem incorporar qualidade no produto (como foi visto nos princípios do *lean* acima). Para tornar-se *lean* é indispensável que haja disciplina no que diz respeito a maneira como o *software* é desenvolvido. Ao entrar em ambiente de desenvolvimento é possível perceber o

nível de disciplina que uma equipe possui. Se a sala de desenvolvimento está bagunçada, possívelmente a equipe é desleixada, o que pode fazer com que o código também possua essa qualidade negativa. O Cinco S é uma ferramenta clássica do *lean* que ajuda a organizar o espaço de trabalho de uma equipe garantindo que tudo que seja necessário esteja em mãoes no momento que a equipe precise. Os Cinco S são referências as palavras japonesas *seiri, seiton, seiso, seiketsu e shitsuke*. Essas palavras foram traduzidas para a língua inglesa e são: *sort* (organização), *sistematize* (classificação), *shine* (limpeza), *standardize* (padronização) e *sustain* (autodisciplina). No desenvolvimento de *software*, o 5S não se aplica apenas ao ambiente físico, mas também ao ambiente lógico por trás da tela. Para organização poderia ser, por exemplo, remover e fazer *backup* de códigos e arquivos que não são mais utilizados, para classificação envolve organizar as coisas nos servidores de forma que o ambiente possa ser utilizado por qualquer pessoa da empresa. Limpeza pode ser manter o ambiente de desenvolvimento limpo (sem copos, marcas de dedo na tela etc.). Padronização envolve, por exemplo, colocar automatização para garantir que cada estação de trabalho tenha a versão mais recente ou a que foi acordada para o desenvolvimento do produto, *backup regular* etc. A autodisciplina só precisa ser então mantida no ambiente.

Em Poppendieck e Poppendieck (2007) é mostrado também os 5S em relação a linguagem Java, mas que serve para a programação em geral. Abaixo é mostrado cada S com sua respectiva ação para melhoria.

3.4.1 Organização

A organização na linguagem Java envolve reduzir o tamanho do código do repositório. Para isso, é necessário eliminar tudo que for:

- Códito morto (que não faz nada);
- Imports que não são utilizados;
- Variáveis que não são utilizadas;
- Métodos que não são utilizados;
- Classes que não são utilizadas e
- Refatorar códigos redundantes.

3.4.2 Classificação

A classificação ou sistematização envolve organizar o projeto e pacotes. Tudo precisa ter uma local e tudo precisa estar no seu devido lugar. Para isso é necessário:

- Resolver ciclos de dependência de pacotes e

- Minimizar depenpências.

3.4.3 Limpeza

Os problemas são mais visíveis quando o código está limpo e claro. Para resolver problemas de limpeza pode-se executar as seguintes ações:

- Resolver testes que estão falhando e erros;
- Melhorar cobertura de código;
- Melhorar performance de teste e
- Resolver os "*warnings*" que aparecem no teste ou nos *logs*.

3.4.4 Padronizar

Uma vez que o ambiente encontra-se limpo, deve-se mantê-lo asssim. Reduzir a complexidade é importante para melhorar o processo de manutenção. Uma técnica bastante utilizada para resolver questões desse tipo é médir o *software*. Primeiramente é necessário saber o que medir e para quê. Esses dados são importantes e podem dar uma direção para equipe no que tange a qualidade do código. Uma técnica de medição, chamada de complexidade ciclomática pode ajudar segundo Laird e Brennan (2006). Quanto maior a complexidade do código, mais difícil é mantê-lo.

3.4.5 Autodisciplina

Para autodisciplina é necessário seguir e usar procedimentos.

3.5 CONCLUSÃO DO CAPÍTULO

Nesse capítulo foi visto sobre a metodologia *lean* de desenvolvimento. O *lean* foi uma metodologia de desenvolvimento que nasceu na manufatura e que hoje está cada vez mais sendo aplicada no *software*. Foram visto os princípios do *lean* como eliminação de desperdícios, incorporar qualidade, criar conhecimento, adiar compreomisso, entregar rápido, repeitar pessoas e otimizar o todo e como eles são aplicados no desenvolvimento. A importância de testes e detecção de falhas também foi explicada. Além disso, foi discutida a importância do mapeamento do fluxo de valor para uma empresa de forma a entender e eliminar os despedícios. Utilizar uma prática do *lean* não necessáriamente vai tornar uma empresa *lean* se os desperdícios não forem eliminados e o processo não for melhorado. Sem fazer esse mapeamento do fluxo de valor, não há como melhorar o processo. No próximo capítulo são mostradas algumas melhorias que foram identificas e até implementadas no que diz respeito aos princípios do *lean*.

4 IMPLEMENTANDO LEAN EM UMA EMPRESA DE SOFTWARE

Esse trabalho, como identificado na sessão 1.2, tem como objetivo propor uma melhoria no modelo de desenvolvimento de uma empresa de *software* aplicando conceitos do *lean*. Assim, esse capítulo aborda o processo de desenvolvimento atual e tenta melhora-lo com alguns dos princípios do desenvolvimento *lean* de *software* vistos no Capítulo 3.

4.1 A EMPRESA

Nesse capítulo é descrito o processo de desenvolvimento de uma empresa de *software* do ramo de auto publicação de livros. O nome da empresa não é mencionado, mas isso não contribui de forma negativa para o trabalho proposto. A empresa possui atualmente três sistemas, os quais foram desenvolvidos utilizando-se a linguagem de programação ruby. Além da linguagem ruby, é utilizado o *framework* Rails, já que os produtos são plataformas SAAS (*Software as a Service*). Outras tecnologias utilizadas são: AWS (*Amazon Web Services*), para armazenamento de arquivos e imagens na núvem, Postgres SQL (para banco de dados) e outras tecnologias.

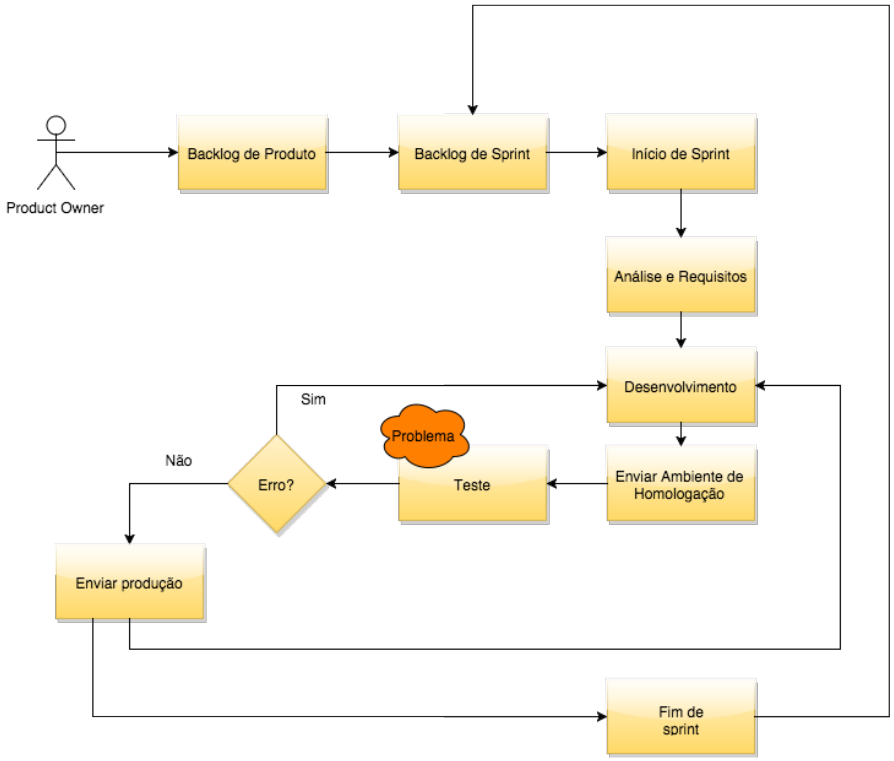
No que tange a equipe de desenvolvimento, a equipe foi composta até pouco tempo por três programadores, e hoje conta apenas com um engenhairo de *software* que é responsável por atender solicitações do dono do negócio (ou *Product Owner* no Scrum). Esse profissional é responsável pela parte de *backend* (lógica de negócios), *frontend* e suporte técnico (quando necessário). A análise técnica e estimativa também é feita por esse profissional. Um time de atendimento ao usuário (duas pessoas) também existe, o qual passa as demandas de suporte técnico (quando existem) via Zendesk (ferramenta de suporte). Alguns pontos de melhorias anteriores a esse trabalho contribuíram para a implantação do *lean*, então essas melhorias também são mencionadas. A sessão seguinte mostra o processo utilizado atualmente.

4.2 O PROCESSO DE SOFTWARE ATUAL

Para implementar o desenvolvimento *lean* em uma empresa, como visto no capítulo anterior, é fundamental identificar e conhecer o processo de *software* e identificar os desperdícios como visto na sessão 3.3.1. Na Figura 21 é mostrado o fluxo básico da empresa atualmente. Tudo começa com o *backlog* de produto. O *backlog* de produto foi um componente tirado do Scrum (visto na sessão 2.3.7). Atualmente utiliza-se um modelo de Scrum, mas sem algumas cerimônias como *daily meeting*, já que todo o processo é feito hoje apenas por uma pessoa. O *product owner* é responsável por colocar os itens e priorizar o *backlog*. Através de uma reunião, os items são colocados em um *backlog* de *sprint*, que vão compor uma *sprint* que dura de 2 até 3 semanas. Quando iniciada a *sprint*, o desenvolvimento de uma funcionalidade vai seguir um modelo incremental:

não necessáriamente em um ciclo a funcionalidade toda vai acabar. Cada vez que uma funcionalidade estiver pronta, pode-se enviar para o ambiente de homologação para os *stakeholders* validarem e darem um *feedback*. Testes efetuados por terceiros são feitos em projetos maiores (embora não estejam representados no processo). Caso a funcionalidade seja validada, ela pode ser enviada para produção. Quando todas as funcionalidades estiverem prontas, a *sprint* termina.

Figura 21 – Processo de desenvolvimento atual com desperdício identificado



Fonte: Autoria própria

4.3 IDENTIFICANDO DESPERDÍCIOS

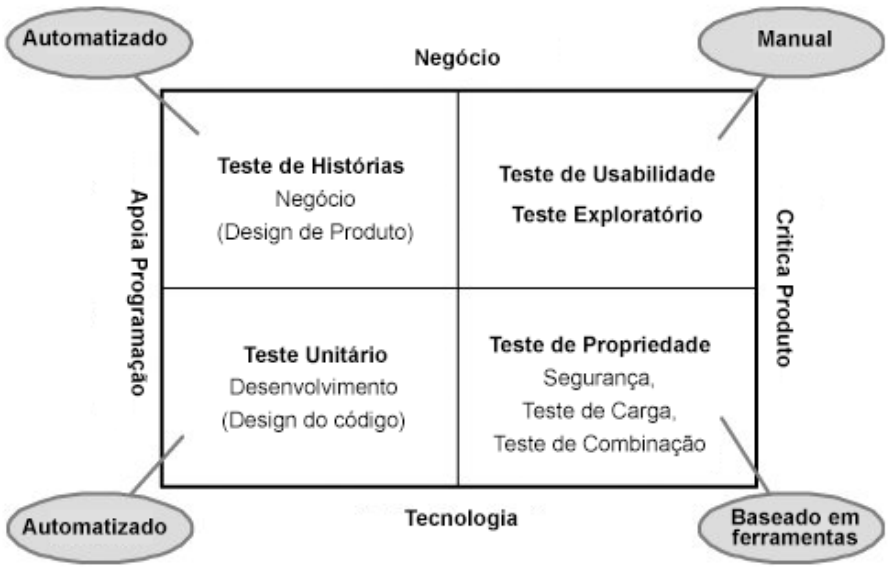
Um dos problemas apontados na Figura 21 da sessão 4.2 é apontado na nuvem identificada com a palavra "problema". O sistema principal da empresa, que está em foco nesse trabalho e representa mais de 80% da renda da empresa, foi construido sem as técnicas apresentadas na sessão 3.3.2 como TDD e BDD. Toda vez que uma alteração no sistema precisa ser feita, não há como prever se alguma funcionalidade existente pode ser afetada. Se o sistema fosse construído com o uso de TDD e BDD, os testes seriam rodadas em cada *deploy*, e caso algum comportamento do sistema falhasse, o código teria que ser reescreito, o que evitaria de enviar novos problemas para produção. Esse tipo de problema fere claramente o princípio da eliminação de desperdício e da incorporação de

qualidade. No *lean*, como visto no Capítulo 3, é preciso não apenas garantir o teste depois da codificação, mas trabalhar na questão de detecção do erro. Em uma analogia com o sistema Toyota de manufatura, toda vez que houver algum problema na linha de produção, esse problema deve ser detectado e a linha de produção deve parar para que o problema ou defeito seja corrigido. Uma das maneiras de se identificar esses desperdícios é montando o fluxo da empresa em um gráfico que pode ser um diagrama de fluxo de valor como mostra a Figura 1 no Capítulo 1 ou de forma mais simples como identificado na Figura 21.

4.4 RESOLVENDO DESPERDÍCIO

Como mencionado na sessão 4.3, existe um problema que faz com que novos suportes e *tickets* no Zendesk sejam frequentes cada vez que uma ou mais novas funcionalidades entrem no ar. Devido a complexidade do sistema, é praticamente impossível e até mesmo inviável (devido a presença de apenas uma pessoa na equipe) que se teste o sistema inteiro sempre que uma nova funcionalidade seja enviada para o ambiente de produção. Mas qual a melhor alternativa para resolver problemas de teste em sistemas legados com nenhuma cobertura de testes ?

Figura 22 – Tipos de testes



Fonte: Poppendieck e Poppendieck, 2007.

A Figura 22 mostra os tipos de testes existentes e suas finalidades. Como pode-se observar, os testes podem estar mais relacionados com questões de negócio ou tecnológicas. Ao mesmo tempo ele pode apoiar mais a programação em si ou fazer críticas ao *software*. Um teste de usabilidade (relacionado ao segundo quadrante superior), por exemplo, poderia

encontrar problemas relacionados a dificuldade de se encontrar alguma coisa na tela, ou a realizar alguma ação etc. Esse tipo de testes geralmente é executado por alguma pessoa, e já é realizado na empresa. Os testes de negócio e unitários foram descritos na sessão 3.3.2. Os testes de propriedade podem testar questões relacionadas a segurança do sistema, carga e combinação. Frequentemente pode-se utilizar ferramentas prontas para, por exemplo, achar vulnerabilidades conhecidas em códigos, pastas com permissões indevidas etc. Um exemplo desse tipo de ferramente de teste de propriedade é o WPSCan[1], que procura vulnerabilidades em um sistema feito com Wordpress.

No caso do sistema da empresa em questão, é importante garantir que as novas funcionalidades sejam feitas utilizando-se do princípio do TDD e BDD, pois ele garante que o desenvolvimento seja maduro através do processo de *design* do código e da refatoração, que se encaixa no *seiri* (organização) no 5S do *lean*. Em casos como esse, em que há pouca cobertura de teste, é preciso saber o que testar. Nesse caso, é indispensável que funcionalidades essenciais do sistema estejam coberta por testes.

Como mencionado no início do capítulo, o sistema da empresa foi implementado com o *framework* Rails. Houve uma demanda do suporte de mudança no cadastro de usuários: que é uma parte crítica do sistema. Basicamente a ficha cadastral foi combinada com outra ficha de cadastro relacionada aos autores (que antes era separada). Uma das maneiras de se testar que tudo sempre vai funcionar como esperado é implementar teste de *views*. Utilizando o *Selenium Web Browser* com uma ferramenta chamada Capybara[2], que torna a escrita de testes que utilizam-se do Selenium mais fácil, foi possível escrever testes que são codificados uma vez e que vão executar automaticamente de forma lógica.

A Figura 23 mostra um teste de *view* para o cadastro. Basicamente ele testa a criação de um usuário normal sem erro algum e a criação de um usuário do tipo pessoa jurídica sem erro algum. Como o sistema trata da criação de autores também, que podem ser pessoa física ou jurídica, os testes para esses casos também devem ser implementados. Ao final deve haver os seguintes testes: pessoa física não autora, pessoa física autora, pessoa jurídica não autora e pessoa jurídica autora.

Ao rodar os testes, o *browser* abrirá e os dois casos são testados. Caso o comportamente no bloco expect não aconteça, o teste falhará.

Esse tipo de abordagem e teste demonstra ser interessante em sistemas legados e com poucos teste, pois futuramente não precisa se gastar tempo testando manualmente se uma dada funcionalidade do sistema está funcionando ou não. Esse código ainda pode ser melhorado para eliminar duplicidade (conforme o *seiso* ou limpeza do 5S). Muitos campos estão sendo testados com o mesmo nome, logo eles poderiam estar numa função que é chamada antes de entrar em cada bloco it.

[1] O WPScan pode ser encontrado e baixado no endereço http://bit.ly/1MPrh2w.
[2] O projeto Capybara pode ser encontrado no endereço http://bit.ly/1ht90TA.

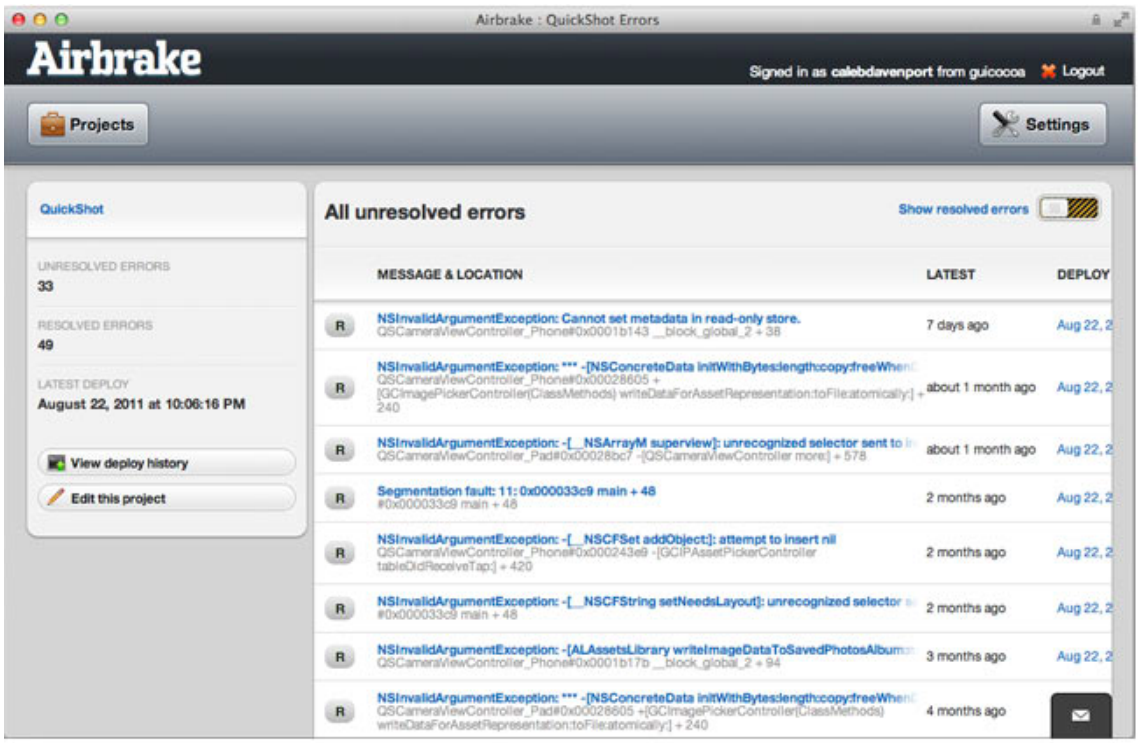
Figura 23 – Teste de *View*

```
require 'spec_helper'

describe 'sign up' do

  it 'creates a company user (with cpf) without any error', :js => true  do
    visit '/users/sign_up'
    fill_in('user_name', :with => 'Usuario    ')
    fill_in('user_email', :with => 'usuario@           .comm')
    fill_in('user_password', :with => 'cda1234')
    fill_in('user_password_confirmation', :with => '       ')
    fill_in('user_cpf', :with => '21241843422')
    select("BRASIL", :from => 'select_country')
    select('Santa Catarina', :from => 'select_state')
    select('Joinville', :from => 'select_city')
    fill_in('user_address_zipcode', :with => '89201580')
    fill_in('user_address_street', :with => 'rua do    ')
    fill_in('user_address_number', :with => '90')
    fill_in('user_address_district', :with => 'Nome do bairro')

    click_button('Cadastrar-se')

    expect(page).to have_content('Bem vindo, Usuario CDA!')

  end

  it 'creates a company user (with cnpj)', :js => true do
    visit '/users/sign_up'
    fill_in('user_name', :with => 'Usuario    ')
    fill_in('user_email', :with => 'usuario@           .comm')
    fill_in('user_password', :with => 'cda1234')
    fill_in('user_password_confirmation', :with => '       ')
    click_link('show_pj')
    fill_in('user_cnpj', :with => '42524498000131')
    select("BRASIL", :from => 'select_country')
    select('Santa Catarina', :from => 'select_state')
    select('Joinville', :from => 'select_city')
    fill_in('user_address_zipcode', :with => '89201580')
    fill_in('user_address_street', :with => 'rua do cda')
    fill_in('user_address_number', :with => '90')
    fill_in('user_address_district', :with => 'Nome do bairro')

    click_button('Cadastrar-se')

    expect(page).to have_content('Bem vindo, Usuario CDA!')
  end
end
```

Fonte: Autoria própria.

Outra forma de incorporar qualidade no serviço é utiliza-se de ferramentas de métricas que avaliam a complexidade do código e ferramentas que analisam os erros que ocorrem em ambiente de produção. Uma ferramenta desse tipo pode ser o Airbrake, que é mostrada na Figura 24. Caso aconteça algum erro em algum ambiente, a empresa é notificada e pode tomar alguma ação antes de receber uma reclamação ou que alguma coisa mais séria aconteça. Programas que avaliam métrica do sistema como complexidade cicliomática, podem indicar locais do código com alta complexidade, e que possivelmente são mais difíceis de serem mantidos ou que possuem algum erro.

Figura 24 – Ferramenta de detecção de bugs



Fonte: Autoria própria.

4.5 ORGANIZANDO O TRABALHO

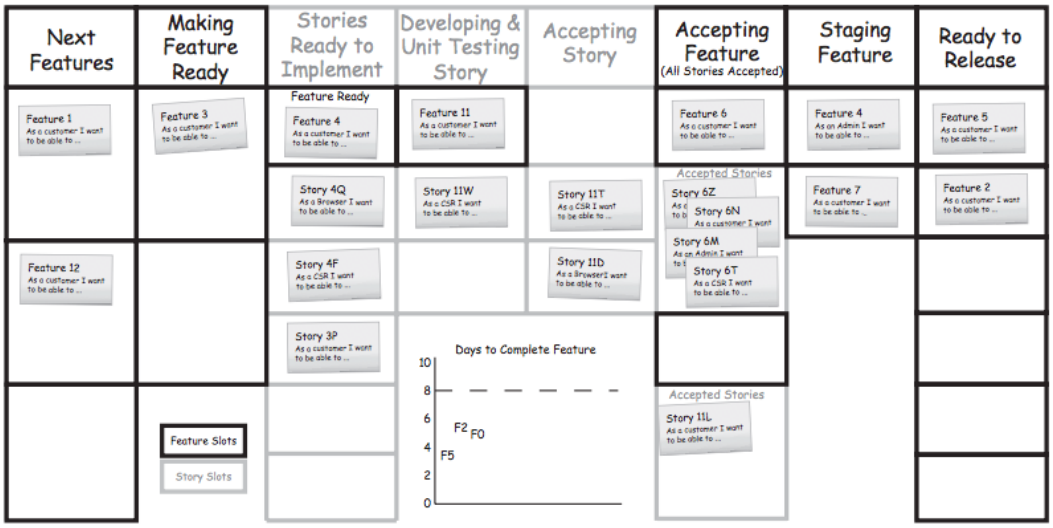
Um componente muito utilizado pelas empresas, e que foi implantado de modo eletrônico, vindo do *lean* foi o Kanban. Segundo Poppendieck e Poppendieck (2010), a palavra kanban significa cartão e é utilizada na manufatura para indicar o trabalho que precisa ser feito. O kanban pode ser utilizado no desenvolvimento de *software* para indicar o andamento de uma funcionalidade no fluxo de produção. Para o *kanban*, há uma coluna para cada passo dentro do fluxo de trabalho da empresa. Tudo isso pode ser feito em nível mais abrangente, como as histórias de usuário, como mais técnicos como as implementações necessárias para uma funcionalidade.

Os sistemas de *kanban* são projetados para limitar o trabalho que está sendo executado, pois quanto mais processos estão sendo executados, mais lento é o fluxo de trabalho. Essa é a principal razão da invenção do *kanban*, segundo Poppendieck e Poppendieck (2010), ou seja limitar o trabalho que está sendo executado e melhorar o fluxo no processo. A Figura 25 mostra um exemplo de um quadro. Cada coluna representa um passo, ou etapa no fluxo de trabalho de uma empresa. No exemplo da figura cada funcionalidade chega ao quadro pelo lado esquerdo na coluna "*Next Features*". O número máximo de funcionalidades que devem ficar nessa coluna são três. Quando há espaço suficiente, a funcionalidade muda de estado. No próxima etapa, a funcionalidade é

decomposta em histórias e depois são postas na próxima coluna. O fluxo segue através do desenvolvimento e testes unitários e depois para o teste de aceitação. Uma vez que uma funcionalidade passe no teste de aceitação, elas são transformadas em funcionalidades novamente e passam pelo teste de aceitação de funcionalidade e *staging*. Finalmente, a funcionalidade é colocado na coluna de pronto (*Ready to Release*).

O membro do time pode trabalhar em qualquer etapa que tiver habilidade. Quando uma funcionalidade termina, o tempo para que ela se movesse através do quadro de *kanban* é anotado. O dia em que a funcionalidade foi posta em "*Next Features*" é subtraído do dia em que ela entrou em "*Ready to Release*". O resultado é então colocado no gráfico "*Days to Complete Feature*" (dias para completar a funcionalidade).

Figura 25 – Exemplo de um kabban



Fonte: Poppendieck e Poppendieck, 2010.

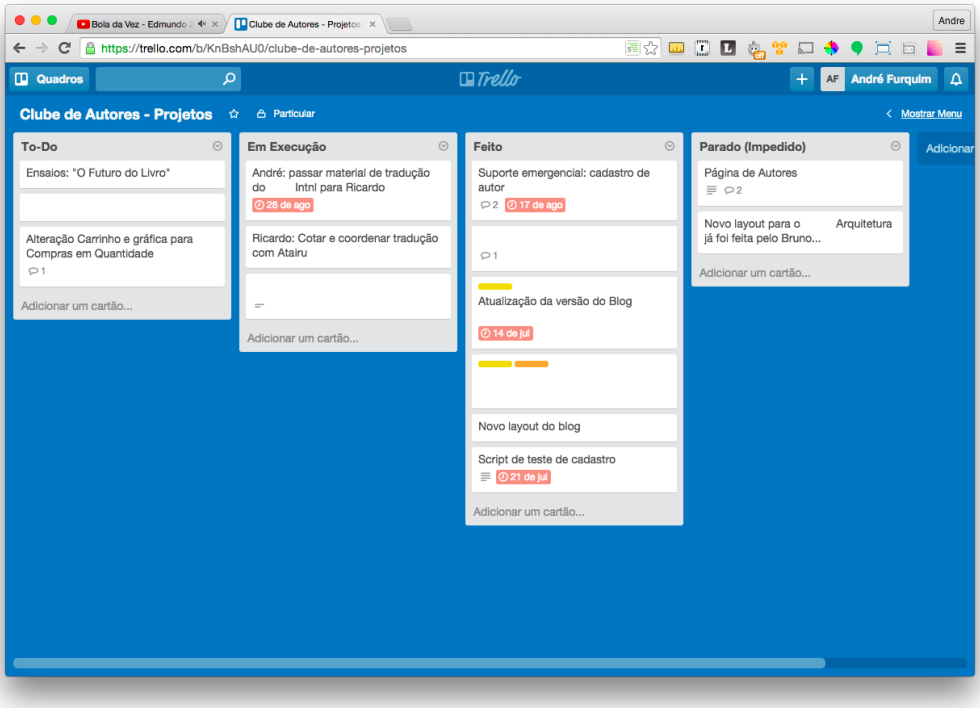
Existem outras variações do *kanban* e na empresa foi implementada uma versão eletrônica utilizando-se da ferramenta Trello[3]. Na empresa, foi utilizado os seguintes passos (para uma funcionalidade), To–Do, Em Execução, Feito e parado. Algumas informações forem omitidas na imagem para preservar a empresa. O exemplo do *kanban* pode ser visto na Figura 26.

4.6 CONCLUSÃO DO CAPÍTULO

Nesse capítulo foram vistos alguns problemas da empresa e suas soluções aplicando os princípios *lean* de desenvolvimento de *software*. Foi visto que algumas demandas de suporte foram diminuidas através da utilização de teste não apenas depois da implementação

[3] O trello é uma ferramenta livre e pode ser utilizado através do endereço http://bit.ly/1eyOBPE

Figura 26 – Exemplo de kabban na empresa



Fonte: Autoria própria.

de uma funcionalidade, para ver se tudo funciona conforme o desenvolvimento, mas de maneira preventiva ou trabalhando-se na sua detecção. Para melhorar o processo da empresa é fundamental que seja entendido o seu fluxo e que seus desperdícios sejam eliminados. Existe várias outras técnicas, em nível de programação, que também pode auxiliar no processo e que acabaram sendo incorporadas e que foram identificadas na sessão 3.4.1. No próximo capítulo é feita a conclusão sobre o trabalho realizado.

5 CONCLUSÃO

Esse trabalho teve como objetivo aplicar métodos de desenvolvimento *lean* em uma empresa de *software*. Através da aplicação de alguns desses princípios, foi notado uma melhora significativa no que tange a manutenção do sistema assim como nos *tickets* recebidos de problemas de suporte relacionado a funcionalidade do teste em questão.

Para entender e melhorar o processo de desenvolvimento de uma empresa foi importante estudar as diferentes metodologias e suas práticas, como por exemplo o TDD no *Extreme Programming*, vistos no Capítulo 2. Para se aplicar o *lean* em uma empresa de desenvolvimento não existe uma fórmula correta, pois tudo vai depender da empresa e do negócio. Utilizar-se dos princípios do *lean*, como *kanban* por exemplo, não necessariamente vai ajudar a empresa a eliminar desperdício ou melhorar sua organização. Além de tudo, o desenvolvimento é feito por pessoas e investir na equipe e entender as necessidades da pessoa é fundamental para qualquer projeto.

Para se aplicar *lean* em uma empresa de *software* é necessário entender o desenvolvimento, a equipe e todas as pessoas envolvidas e contrastar com os seus princípios e perguntar-se "Qual é a melhor maneira de eliminar esse desperdício?". Práticas como TDD, BDD, *clean code* etc., sem dúvida melhoram o desenvolvimento e qualidade do *software* e vem se tornando um padrão na indústria de desenvolvimento para incorporar qualidade no sistema.

5.1 Trabalhos Fufutos

Os seguinte trabalhos futuros podem ser implementados para a conformidade com o desenvolvimento *lean* e até na área do *lean*:

- Implementar integração contínua nos produtos da empresa;
- Desenvolver um manual de práticas de codificação com práticas de desenvolvimento para uma empresa;
- Aprimorar ainda mais as técnicas de desenvolvimento do *lean* com o desenvolvimento da empresa e
- Implementar um modelo para gerenciamento do conhecimento na empresa conforme o princípio do *lean*.

REFERÊNCIAS

BARAÚNA, H. **Cucumber e RSpec**: Construa aplicações ruby on rails com testes e especificações. [S.l.]: Casa do Código, 2013. ISBN 978-85-66250-34-3.

BASILI, V. R.; HEIDRICH, J.; LINDVALL, M.; MÜNCH, J.; REGARDIE, M.; ROMBACH, H. D.; SEAMAN, C. B.; TRENDOWICZ, A. Gqm+strategies: A comprehensive methodology for aligning business strategies with software measurement. **CoRR**, abs/1402.0292, 2014. Disponível em: <http://arxiv.org/abs/1402.0292>.

BECK, K.; BEEDLE, M. **Manifesto for Agile Software Development**. 2015. Disponível em: <http://agilemanifesto.org/>. Acesso em: 16 jul. 2015.

CHELIMSKY, D. **The RSpec Book**. 1. ed. [S.l.]: Pragmatic Bookshelf, 2012. ISBN 978-1934356371.

COHN, M. **Desenvolvimento de software com Scrum**: aplicando métodos ágeis com sucesso. 1. ed. Porto Alegre: Bookman, 2011. ISBN 978-85-7780-807-6.

DSDM. **DSDM CONSORTiUM**. 2015. Disponível em: <http://www.dsdm.org/>. Acesso em: 01 ago. 2015.

EDEKI, C. Agile unified process. In: **International Journal of Computer Science and Mobile Applications – UCSMA**. [S.l.: s.n.], 2013. v. 1, p. 13–17.

HUMPHREY, W. S. **The Personal Software Project**. Pittsburgh, PA 15213-3890, 2000. Disponível em: <http://www.sei.cmu.edu/reports/00tr022.pdf>.

IXP. **What is Industrial XP?** 2015. Disponível em: <http://industrialxp.org/>. Acesso em: 27 jul. 2015.

JANES, A. A guide to lean software development in action. In: **Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on**. [S.l.: s.n.], 2015. p. 1–2.

LAIRD, L.; BRENNAN, M. C. **Software Measurement and Estimation**. Hoboken, New Jersey: John Wiley and Sons, Inc, 2006. ISBN 0-471-67622-5.

MACHADO, F. N. R. **Análise e gestão de requisitos de software**: onde nascem os sistemas. 1. ed. São Paulo: Érica, 2013. ISBN 987-85-365-0362-2.

POPPENDIECK, M. Lean software development. In: **Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on**. [S.l.: s.n.], 2007. p. 165–166.

POPPENDIECK, M.; POPPENDIECK, T. **Implementing lean software development**: from concept to cash. 1. ed. Boston, MA 02116: Pearson Education, Inc., 2007. ISBN 0-321-43738-1.

POPPENDIECK, M.; POPPENDIECK, T. **Leading lean software development**: results are not the point. 1. ed. Boston, MA 02116: Pearson Education, Inc., 2010. ISBN 0-321-62070-4.

PRESSMAN, R. S. **Engenharia de Software**: Uma abordagem profissional. 7. ed. Porto Alegre: McGraw-Hill, 2011.

SBROCCO, J.; MACEDO, P. **Metodologias Ágeis**: Engenharia de software sob medida. 1. ed. São Paulo: Editora Érica, 2012. ISBN 978-85-365-0398-1.

WANG, X. The combination of agile and lean in software development: An experience report analysis. In: **Agile Conference (AGILE), 2011**. [S.l.: s.n.], 2011. p. 1–9.