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.
25 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

Professor Dr. ??
Universidade ???

Professor Dr. ??
Universidade ??

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

FDD	Feature Driven Development
GQM+	Goal Question Metric
IHC	Interface Humano Computador
PSP	Personal Software Process
RUP	Rational Unified Process
TSP	Team Software 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.

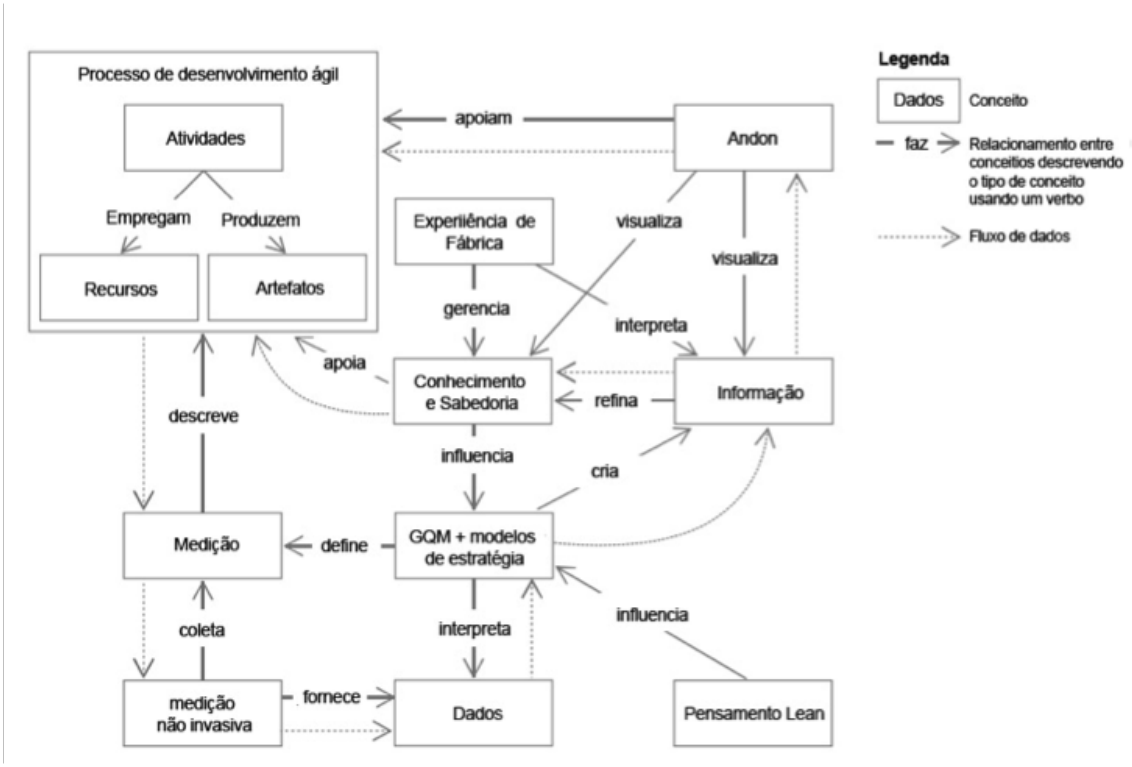
Para questão de velocidade é discutido a aplicação da teoria das filas no desenvolvimento de *software*. Nela foi mostrado o porquê de longas filas serem prejudiciais ao desenvolvimento e mostrado como evita-las. Além disso, é mostrado que ciclos de vida rápidos tendem a dar uma qualidade maior ao projeto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

2.1 PROCESSO DE SOFTWARE

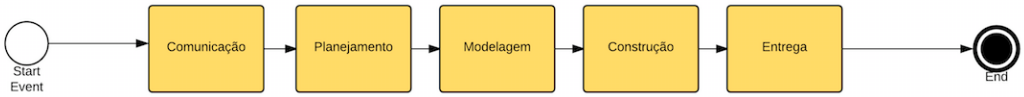
Quando um projeto de *software* precisa ser executado, os engenheiros e seus gerentes utilizam um processo de *software*. Um processo de *sofware*, como define Pressman (2011), é um conjunto de atividades de trabalho, ações e tarefas realizadas a fim de produzir algum artefato de *software*. Cabe ressaltatar 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 metodologias 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

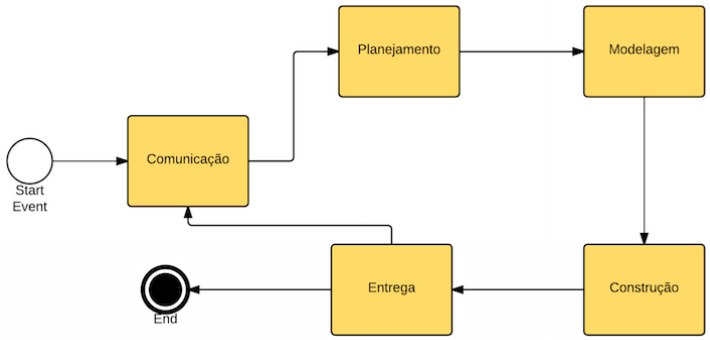
Figura 2 – Fluxo de processo linear



Fonte: Adaptado de Pressman, 2011.

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.

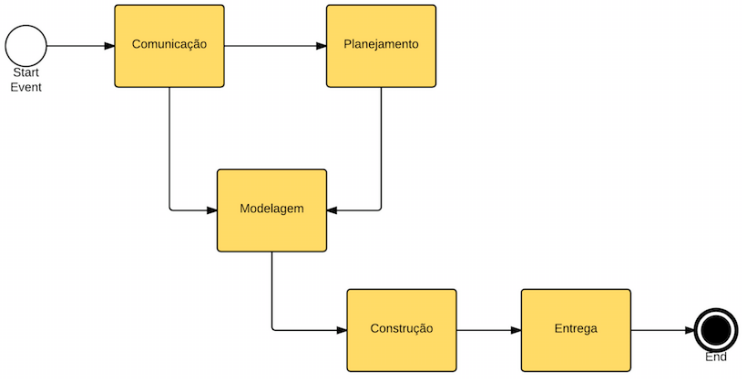
Figura 3 – Fluxo de processo evolucionário



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.

Figura 4 – Fluxo de processo paralelo



Fonte: Adaptado de Pressman, 2011.

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

2.2 MODELOS DE DESENVOLVIMENTO

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).

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*.

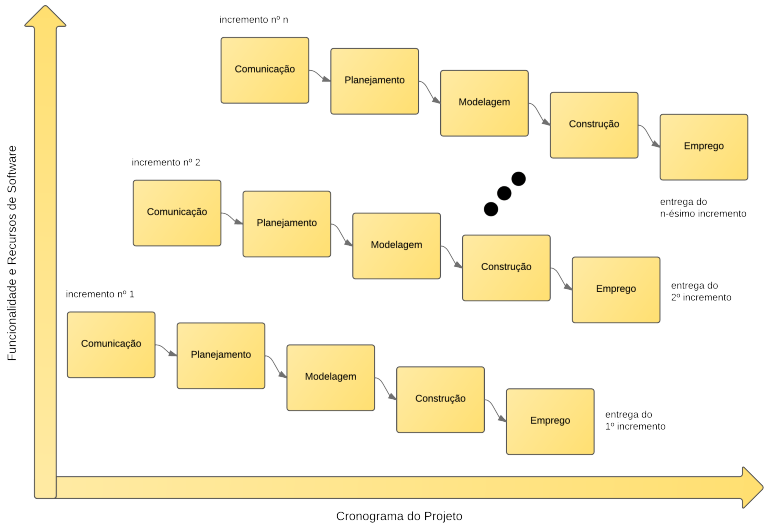
Pressman (2011) descreve alguns pontos negativos da utilização desse processo como a natureza incerta de projetos de *software* que fazem com que as necessidades sejam difíceis de serem traduzidas de uma vez. Além disso, como mencionado anteriormente o cliente precisa esperar pacientemente até possuir 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.

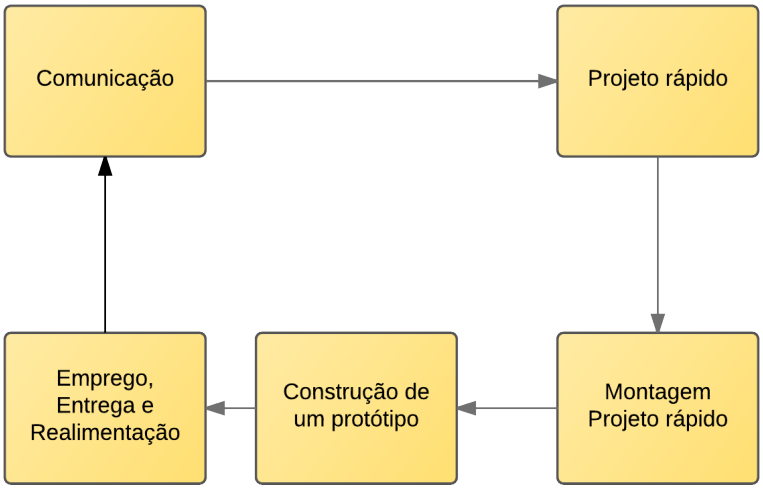
2.2.3 Modelo de Processo Evolucionário

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 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.

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

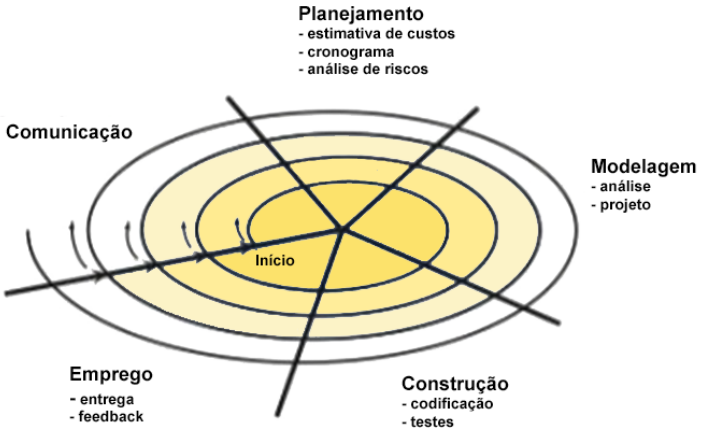
Figura 6 – Paradigma da prototipação



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.

Figura 7 – Modelo espiral

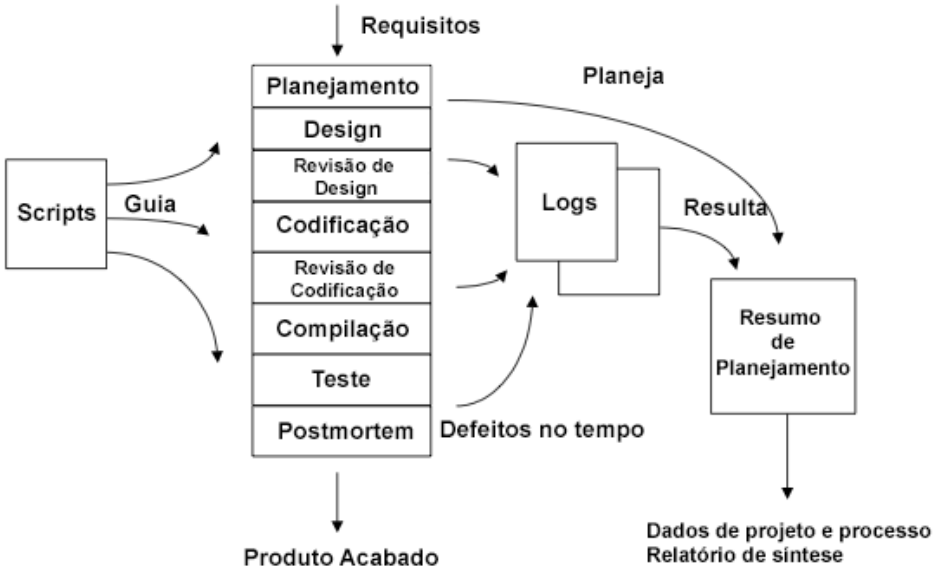


Fonte: Adaptado de Pressman, 2011.

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 8 – Modelo do PSP

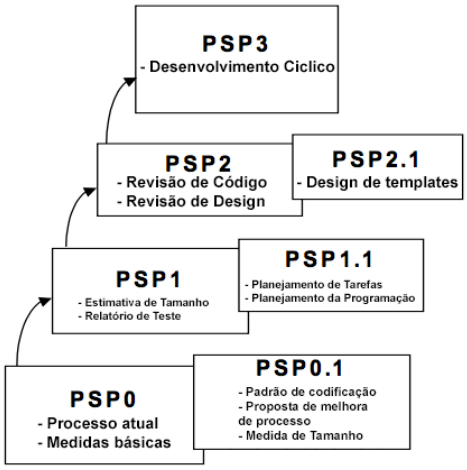


Fonte: Adaptado de Humphrey, 2000.

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.

Além do PSP existe o TSP (*team software process*) que é a aplicação do PSP para

Figura 9 – Níveis do PSP



Fonte: Adaptado de Humphrey, 2000.

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 sessão 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.

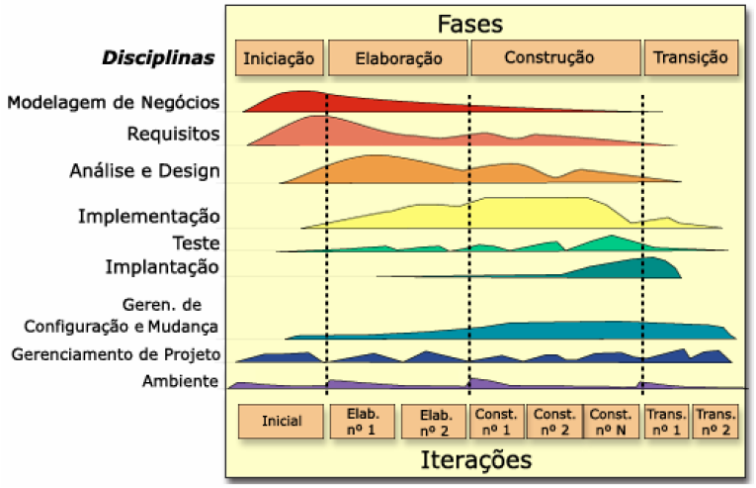
6. **Controlar mudanças**: Cada mudança do *software* necessita de um requisito de mudança que descreve criteriosamente as mesmas.

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 de *software* e Kent Beck. Esse manifesto tinha como objetivo quebrar o velho conceito

Figura 10 – Estrutua do RIP



Fonte: Adaptado de Machado, 2013.

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)

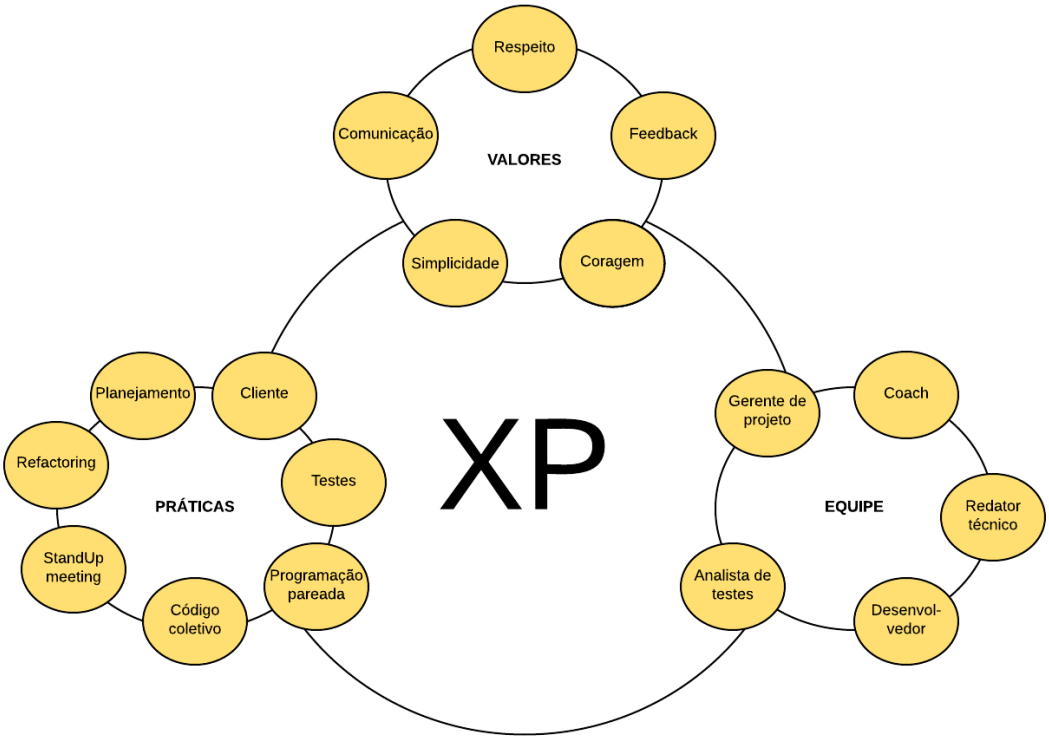
Extreme Programming é uma metodologia ágil de desenvolvimento de projetos de *software*, criada por Kent Beck em 1996, ideal para pequenas e médias equipes de desenvolvimento. Como a maioria das metodologias ágeis, é uma metodologia com foco em projetos cujos requisitos não são bem definidos no início 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.

Figura 11 – Estrutura da XP



Fonte: Adaptado de Sbrocco e Macedo, 2012.

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.

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 (INDUSTRIALXP, 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 IndustrialXP, 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):

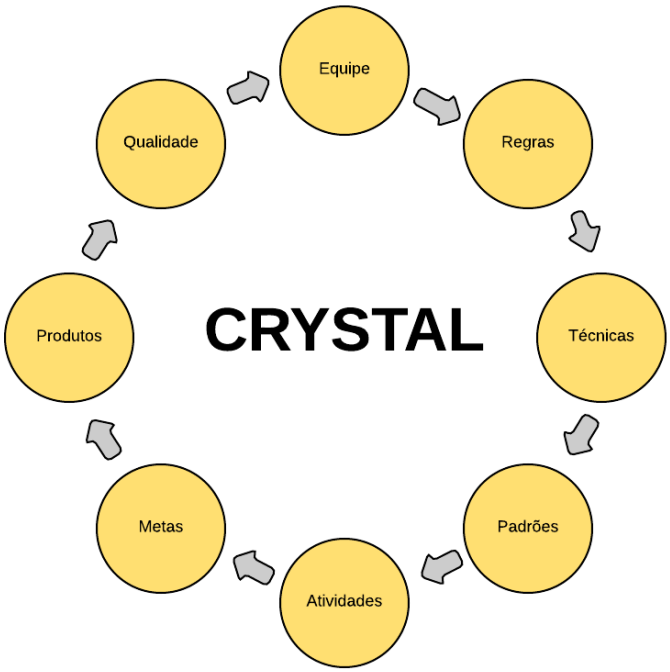
1. Trabalho face a face com o cliente: considera que envolver o cliente nas iterações e nas decisões é muito mais produtivo;
2. Peso significa custo: quanto maior a complexidade, maior o custo;
3. Usar metodologias diferenciadas para equipes maiores;
4. Mais cerimônias maior criticidade: quanto mais 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.

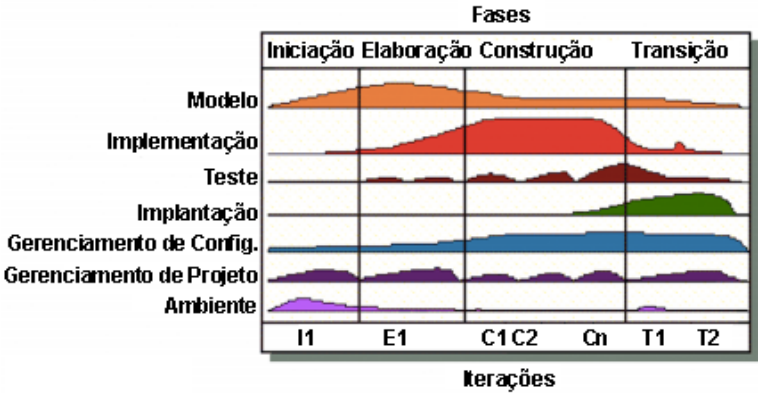
2.3.4 Processo Unificado Ágil (UAP)

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 sessão 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 sessão 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)

Figura 14 – Ciclo do UAP



Fonte: Adaptado de Edeki, 2013.

REFERÊNCIAS

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.

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>.

INDUSTRIALXP. 2015. Disponível em: <http://industrialxp.org/>. Acesso em: 31 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.

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.

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.