MAPEAMENTO E APLICAÇÃO DA PRODUÇÃO ENXUTA PARA O PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

FERNANDO LUIZ DE CARVALHO E SILVA

UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE - UENF
CAMPOS DOS GOYTACAZES - RJ
Março - 2011

MAPEAMENTO E APLICAÇÃO DA PRODUÇÃO ENXUTA PARA O PROCESSO DE DESENVOLVIMENTO DE Software

FERNANDO LUIZ DE CARVALHO E SILVA

Dissertação apresentada ao Centro de Ciências e Tecnologia da Universidade Estadual do Norte Fluminense, como parte das exigências para obtenção do titulo de Mestre em Engenharia de Produção .

Orientador: Prof Rogerio Atem de Carvalho, D. Sc.

Campos dos Goytacazes - RJ
MARÇO - 2011

MAPEAMENTO E APLICAÇÃO DA PRODUÇÃO ENXUTA PARA O PROCESSO DE DESENVOLVIMENTO DE *Software*

FERNANDO LUIZ DE CARVALHO E SILVA

Dissertação apresentada ao Centro de Ciências e Tecnologia da Universidade Estadual do Norte Fluminense, como parte das exigências para obtenção do titulo de Mestre em Engenharia de Produção.

Aprovada em 17 de março de 2011

Comissão Examinadora:

Prof.: Helder Gomes Costa, D. Sc. – UFF

Prof.: Romeu Silva e Neto, D. Sc. – IFF

Prof.: José Ramón Arica Chávez, D. Sc. – UENF

Prof: Rogerio Atém de Carvalho, D. Sc. – IFF
Orientador

Campos dos Goytacazes - RJ
MARÇO - 2011

AGRADECIMENTOS

A meu falecido pai, pelos elevados exemplos de moral e ética, paciência e companheirismo.

A minha mãe pelo carinho, dedicação e afeto.

A minha esposa pelo amor, amizade, companheirismo, ensinamentos, colaboração em todos os momentos.

A minhas filhas, a quem dedico todo esforço que faço, como meu pai, visando deixar como exemplo de superação, liberdade e amplas possibilidades.

Ao meu orientador, professor Rogerio Atem, por acreditar no meu trabalho, na contribuição que foi gerada, e pelo apoio constante durante toda a pesquisa.

A todos os amigos que me deram amizade, incentivo, apoio.

Ao amigos e colegas com quem estudei, sem os quais esta caminhada teria sido muito mais pesada e difícil.

Aos amigos e colegas de trabalho do IFF, que me apoiam, incentivam e possibilitaram terminar este trabalho.

Aos professores do LEPROD, por seus valiosos ensinamentos ao longo de meu aprendizado.

Aos funcionários do LEPROD que me ajudaram sempre que precisei.

Resumo

Neste trabalho são mapeados os princípios do pensamento enxuto para o processo de desenvolvimento de *Software*. Portanto, são revistos os conceitos sobre sistemas produtivos, as características do Sistema Toyota de Produção e suas ferramentas. Em um segundo plano são analisados os processos de produção de *Software*, levantando os problemas encontrados. O mapeamento é feito com base no referencial teórico e é discutido o impacto da utilização do pensamento enxuto na melhoria do processo de desenvolvimento de *Software*. Para esta avaliação foi desenvolvida uma pesquisa-ação que monitorou as transformações em um projeto de desenvolvimento de *Software* no Núcleo de Sistemas de Informação do Instituto Federal Fluminense. Durante o experimento foram documentadas as mudanças e monitoradas as vantagens, desvantagens e o desempenho da equipe de desenvolvimento. Este trabalho portanto contribui apresentando soluções para a melhoria do processo de desenvolvimento de *Software* através da utilização do Pensamento Enxuto.

Abstract

This work maps the principles of Lean Thinking to Software Development process. Therefore, concepts of production systems, the characteristics of the Toyota Production System and its tools are reviewed. In background, the software production process and its known problems are analyzed. This mapping is done based on theoretical references, which drive a discussion on the impact of using Lean Thinking to improve the Software Development process. For this evaluation, a action research that tracked changes in a software development project was developed, which took place in the NSI (Information Systems Research Group) of the IFF (Federal Fluminense Institute). During the experiment the changes related to advantages and disadvantages and the development team performance were documented and monitored. This study therefore aims to contribute presenting solutions to improve the process of Software Development through the use of Lean Thinking.

Sumário

Lista de Siglas

ABES - Associação Brasileira de Engenharia de *Software*

BDD - *Behavior-Driven Development*

BDUF - *Big Design Up Front*

CMMI - *Capability Maturity Model Integration*

DbC - *Design by Contract*

DoD - *Department of Defense*

IMVP - *International Motor Vehicle Program*

JIT - *Just-in-Time*

LD - *Lean Software Development*

MPS.Br - Modelo de Processo de maturidade

NVAT - *Non-Value Added*

PE - Pensamento Enxuto

PO - *Product Owner*

QFD - *Quality Function Deployment*

Scrumban - Contração entre *Scrum* e *Kanban*

SI - Sistemas de Informação

STP - Sistema Toyota de Produção

TDD - *Test-Driven Development*

TI - Tecnologia de Informação

TPC - Tambor - Pulmão - Corda

UAT - *Unit Acceptance Test*

VAT - *Value Added Time*

VSM - *Value Stream Mapping*

WIP - *Work in Process*

XP - *eXtreme Programming*

Índice de Figuras

Índice de Tabelas

1. Introdução

Dentre os principais serviços da atualidade encontram-se o desenvolvimento de *Software*, e neste contexto, segundo a Associação Brasileira de Engenharia de *Software* (ABES, 2010), o Brasil encontra-se na 12ª colocação no mercado mundial de serviços de *Software*. Atualmente conta-se com 8.500 empresas, das quais 76% trabalham com desenvolvimento de *Software*.

Michael E. Porter, considerado um dos grandes autores sobre estratégia competitiva, define em seu trabalho junto com Millar (1995) que a Tecnologia de Informação (TI) e Sistemas de Informação (SI) fontes de vantagens competitivas para as organizações.

Entretanto, apesar do senso comum de que exista uma relação direta entre investimentos realizados em TI e vantagens competitivas obtidas, Stethen Roach (1988) abriu a discussão que comprovou a inexistência de uma relação direta entre o investimento entre TI e retorno financeiro proporcional. A obtenção das vantagens está na verdade associada à aderência dos recursos de TI às características do negócio da empresa e de sua demanda. Dado que as empresas estão em constante mudança em adequação às demandas, tais recursos devem ser constantemente atualizados para manutenção de seu valor.

Chou (2005) avalia que dentre os investimentos das organizações em TI, em média 36% são utilizados para suporte e manutenção de *Software* e 21% são utilizados para desenvolvimento de novas funcionalidades, totalizando 57% do total . Destacada a importância da produção de *Software* no contexto econômico, cabe investigar o porque dos problemas (CHARETTE, 2005) encontrados neste processo produtivo.

Diversos autores destacam que os processos tradicionais para o desenvolvimento e manutenção de *Software*, preconizados pela Engenharia de *Software*, tentam melhorar a qualidade do processo e dos produtos de *Software*, no entanto exigem um custo elevado em treinamento, capacitação, e certificação (HIGHSMITH, 2002; BECK, 1999; FOWLER, 2005). Entretanto, segundo SEMEGHINI (2001), 87% das empresas de desenvolvimento de *Software* são de pequeno e médio porte, o que torna difícil a utilização destes conceitos, que acabam sendo utilizados somente por empresas de grande porte.

Este paradoxo é semelhante ao enfrentado pela empresa Toyota Automotors, no início de sua implantação, quando o processo produtivo vigente era de grande escala de produção, padronizada pelo sucesso da Ford Auto Motors. Segundo Ohno (1997), a impossibilidade de seguir os padrões industriais vigentes foi um dos fatores que direcionaram o desenvolvimento do Sistema Toyota de Produção (STP).

A partir das mudanças operadas pelos engenheiros de produção da Toyota, percebeu-se que grande parcela das operações e do tempo consumido na Ford eram na verdade desnecessárias, diante da remodelagem do processo. Enquanto na Ford se produziam grandes lotes de trabalho, criando inevitavelmente estoques de trabalho inacabado em processo, na Toyota se trabalhou em lotes unitários, eliminando o estoque de trabalho em processo e tornando o *Lead Time* muito menor.

No desenvolvimento de *Software* pode-se perceber que o processamento de todas as funcionalidades em cada etapa do desenvolvimento torna imprescindível a utilização de estoques de conhecimentos inacabados em processo. A mudança de paradigma para o trabalho em lotes unitários propicia a eliminação destes estoques de conhecimento inacabado dentro do processo, criando um fluxo contínuo de trabalho e tornando desnecessários os estoques para a transferência de conhecimento.

Outro aspecto importante a ser discutido é que os projetos de desenvolvimento de *Software* vem historicamente apresentando baixa satisfação do usuário, atrasos e aumento de custos. Charette (2005) discute o risco de projetos de desenvolvimento de *Software*, apresentando projetos que falharam utilizando os métodos tradicionais, e destaca que a própria natureza do processo é propícia para falhas.

O mercado atual é altamente dinâmico, tornando a sobrevivência das empresas uma questão de constante atualização. Em função desta constante busca pela atualização, os requisitos de *Software* não podem ser estáticos e imóveis, sendo portanto necessário um processo de desenvolvimento que possibilite a mudança dinâmica destes requisitos. Em consequência, os processo tradicionais de desenvolvimento normalmente não aceitam mudanças e demoram para entregar resultados. Os objetivos da produção são afetados diretamente, fazendo com que gerentes de projeto e desenvolvedores reclamem das constantes mudanças solicitadas pelos patrocinadores do desenvolvimento.

Poppendieck e Poppendieck (2003) discutem a natureza dos processo de desenvolvimento tradicionais que refletem características da produção em massa, e com sobrecarga dos desenvolvedores. De fato, Humphrey (2005), discute que faltam mais camadas de poderes gerenciais, e mais cobrança e controle com seriedade das atividades executadas pelos desenvolvedores. Este autor explica que os projetos de *Software* falham porque o planejamento feito pelas camadas gerenciais precisa ser aceito pelos desenvolvedores como um desafio à sua produtividade.

Neste trabalho se discute o contexto do desenvolvimento de *Software* como um serviço que precisa de um processo para ser administrado corretamente. Entretanto, este processo visa especificação, teste, aceite de diversas partes, gerando um produto (o *Software*) como consequência, corroborando com a expectativa de que se possa tratá-lo como um processo de produção. Ao mesmo tempo, entende-se que, contrariamente à visão corrente um processo estruturado verticalmente, entende-se que este processo possa ser administrado em fluxo contínuo de lotes unitários, conforme a visão de gestão por processos.

1.1. Problema de Pesquisa

Portanto, esta pesquisa inicia-se entendendo que os processos de desenvolvimento de software tradicionalmente aplicados não apresentam estabilidade que garantam atingir aos objetivos de forma previsível. A solução adotada é a mesma da produção em massa, optando pelo trabalho em grandes lotes e usando estoques para contornar as ineficiências e demoras do seu processo produtivo.

Buscando enfrentar e resolver tais problemas entendeu-se que o pensamento enxuto seria uma das soluções possíveis para mudar a arquitetura deste processo, e portanto fez-se a seguinte pergunta:

É possível obter melhorias no processo de desenvolvimento de *Software* pela aplicação dos princípios descritos no Pensamento Enxuto (PE)?

A partir desta pergunta iniciou-se uma pesquisa sobre o pensamento enxuto, o processo de desenvolvimento de *Software* e os problemas advindos deste processo.

Entretanto, a aplicação do pensamento enxuto não retorna imediatamente um processo novo e correto. O pensamento enxuto fornece princípios, práticas e

técnicas para que o processo melhore continuamente em busca da satisfação do cliente e da redução de custos para a organização.

Portanto, foi necessário recortar a área de estudo visando à aplicação específica de técnicas que possibilitassem uma primeira simplificação e otimização do processo, sem agregar complexidade e burocracia.

Foram feitos os seguintes recortes:

- Sobre o pensamento enxuto foram levantados os autores que descreveram inicialmente o tema e as ferramentas utilizadas por contribuir com processo de desenvolvimento e de produção, principalmente para serviços;
- Para o desenvolvimento de *Software* foram descritos alguns processos tradicionais e foi abordado, em linhas gerais, o padrão ISO/IEC 12207;
- Para a engenharia do produto foram avaliadas técnicas ágeis, principalmente as descritas na Programação eXtrema (XP);

1.2. Objetivos

1.2.1. Objetivo principal

Verificar através de um mapeamento e de um estudo de casos se os conceitos e ferramentas do pensamento enxuto podem contribuir para melhorar o processo de desenvolvimento de *Software*.

1.2.2. Objetivos secundários

Avaliar a filosofia do pensamento enxuto para desenvolvimento de *Software.*

Levantar a literatura científica existente sobre o tema.

Relacionar as ferramentas e técnicas do Sistema Toyota de Produção para o processo de desenvolvimento de *Software*.

Observar em campo a aplicação destes conceitos, analisar e discutir estes resultados.

Apresentar em um processo os aspectos do pensamento enxuto e seu impacto na melhoria do processo.

Verificar a adequação do uso da ferramenta *Kanban*.

Verificar se usando os princípios do pensamento enxuto os desenvolvedores participam da melhoria do processo.

Verificar se o processamento em lotes pequenos e unitários propiciam um controle melhor do trabalho.

Verificar a adequação do direcionamento à satisfação do cliente e aos valores por ele estabelecidos, como um fator de melhoria do processo.

Verificar se, conforme preconizado pelo pensamento enxuto, a colaboração em pares e planejamento em equipe, promove um aprendizado coletivo, melhoria nas estimativas e com isso maior estabilidade do processo.

Verificar se o jidoka pode ser aplicado para o desenvolvimento de *software* e suas consequências.

Que impactos no tempo de produção e na satisfação do cliente podem ser obtidos pela aplicação do pensamento enxuto ao processo de desenvolvimento de *software*.

1.3. Justificativa

Ocorre que o *Software* assume papel estratégico na sociedade e na sobrevivência das empresas. Ao mesmo tempo a engenharia de *Software* não possui ainda a capacidade de promover o desenvolvimento de forma eficiente, repetitiva e rápida dos valores esperados pelo cliente no *Software.* Conforme discutido neste introdução o processo de desenvolvimento de *Software* tem atingido resultados negativos e inesperados em diversos casos.

O Pensamento Enxuto aponta como uma ferramenta para otimizar processos e é até o presente momento pouco explorado para apoiar e otimizar o processo de desenvolvimento de *Software*.

Diversos autores na área de engenharia de *Software* já apontam para a utilização do pensamento enxuto para o processo de desenvolvimento de *Software*, entretanto o conhecimento necessário para avaliar processos produtivos, como PE, *Six Sigma*, Teoria das Restrições, não são conceitos estudados pela Engenharia de *Software*.

Portanto, ao responder a pergunta desta pesquisa, busca-se trazer luz a utilização de processo de produção da indústria de *Software*, na qual os processo

são definidos empiricamente, sem a bagagem teórica para definição, otimização e gerenciamento de processos produtivos.

1.4. Metodologia

Segundo Thiollent (2004) uma metodologia é um nível abstrato que agrupa, descreve e discute métodos que podem ser aplicados para resolver algum problema.

Westbrook (1995) afirma que as contribuições para problemas unicamente acadêmicos bem estruturados são normalmente simplificações pouco úteis diante de problemas desestruturados encontrados em campo, na vida real.

A estratégia de pesquisa-ação tem como vantagem a participação do pesquisador não só como observador, mas como agente de um processo monitorado, buscando observar e documentar os resultados obtidos.

Assim, neste subitem será descrita e conceituada a estratégia de Pesquisa-Ação (PA), adotada para este trabalho, visando acompanhar o processo e interferir quando necessário para avaliar a aplicação da teoria do pensamento enxuto na prática de um processo vivo de desenvolvimento de *Software* em um estudo de casos.

1.4.1. Descrição

Inicialmente foi feita uma revisão bibliográfica, buscando evidências sobre os seguintes assuntos:

1. O Sistema Toyota de Produção, com vistas a entender sua origem e abrangência;
2. O Processo de desenvolvimento de *Software* segundo a Engenharia de *Software*, para compreender o processo de desenvolvimento de *Software* e seu estado atual ;
3. A utilização de métodos ágeis que atualmente utilizam de conceitos do pensamento enxuto para produzir *Software* ;
4. Trabalhos que associem o pensamento enxuto para orientar e otimizar o processo de desenvolvimento de *Software*.

Um estudo de caso único foi realizado com vista a obtenção de dados qualitativos sobre a utilização destes conceitos mapeados do pensamento enxuto, durante a aplicação destes conceitos em um caso real, descrito no último capítulo.

Segundo Yin (2003), o estudo de caso único é valido em uma investigação científica de um fenômeno contemporâneo, sendo realizado durante o período do projeto, tentando utilizar métodos e práticas emergentes e desconhecidas para os próprios elementos do projeto.

O ambiente no qual tais práticas foram aplicadas buscava a renovação das antigas práticas que eram consideradas obsoletas, sendo portanto um caminho natural e real de busca, sofrendo entretanto influência da pesquisa.

Neste caso os limites do fenômeno observado não estavam claramente definidos, pois ainda não existe uma consolidação sobre a utilização do PE para o processo de desenvolvimento de *Software*, faltando metodologias para sua aplicação, e existindo mesmo poucos fenômenos como este observáveis e acessíveis para a pesquisa.

Neste estudo percebe-se também a quantidade de pontos de interesse de observação dos diversos conceitos envolvidos desde filosofia, fatores humanos, variáveis de controle de processo, desempenho, qualidade e adequação, compondo diversas fontes de evidência neste mesmo caso.

Foram portanto observados os vários pressupostos teóricos, observados na revisão bibliográfica sobre o pensamento enxuto e o STP, para a observação e documentação do estudo de casos.

Com base no trabalho de Yin (2003), avaliou-se o estudo de caso único como um estudo qualitativo, buscando responder a pergunta do trabalho, se o pensamento enxuto pode orientar e otimizar o processo de desenvolvimento de *Software*, tornando-o mais gerenciável, aumentando a velocidade, a qualidade e diminuindo desperdícios comumente observados.

1.5. Organização do trabalho

Esta dissertação está organizada em 8 capítulos. Neste primeiro capítulo, introdutório foi apresentado o contexto no qual se pesquisa, a pergunta que motivou o trabalho e os objetivos que se busca alcançar para responder a pergunta.

No segundo capítulo são apresentadas as etapas da pesquisa, as técnicas utilizadas e o relacionamento entre as técnicas e os objetivos de pesquisa.

No terceiro capítulo são descritos os conhecimentos abordados em tópicos, visando introduzir o leitor a complexidade dos assuntos abordados.

Durante os capítulos dois e três é feita uma revisão bibliográfica do estado da arte do pensamento enxuto aplicado a serviços e ao desenvolvimento de *Software*, com o objetivo de apresentar ao leitor o que a academia já tem pesquisado sobre o assunto e dando destaque aos experimentos práticos realizados.

O capítulo quatro o autor apresenta sua contribuição, ressaltando o mapeamento do pensamento enxuto e das ferramentas da produção enxuta para o processo de desenvolvimento de *Software*.

No capítulo cinco é apresentado um estudo de caso realizado no NSI, onde está sendo desenvolvido, entre outros projetos, o projeto da Biblioteca Digital.

No último capítulo, são discutidos os resultados do estudo de casos apresentado.

2. Referencial Teórico – Sistemas de Produção

Com o objetivo de tornar claros os conceitos sobre o Pensamento Enxuto e sobre o processo de desenvolvimento de *Software*, serão revistos neste capítulo conceitos preliminares da Engenharia de Produção e Engenharia de *Software* embasadores desta discussão.

Durante esta revisão serão também mesclados o entendimento deste autor e observações colhidas em outros trabalhos encontrados.

2.1. Ferramentas do STP

Durante a evolução do STP foram desenvolvidos vários conceitos e ferramentas visando a melhoria da qualidade e da eficiência dos processos. Algumas destas ferramentas e conceitos serão explicados a seguir.

2.1.1. *Kanban*

O *Kanban* é considerado por Ohno (1997) como o responsável pelo comando do STP, como uma ferramenta de gerenciamento visual, de rápida assimilação, que dá transparência ao fluxo de execução do processo.

Ainda segundo Ohno (op. cit.), a utilização do cartão *Kanban* implementa o *Just-in-Time*, flexibilizando a produção, tornando dispensáveis os documentos utilizados anteriormente para controlar o processo produtivo, e tornando dispensável o comando dos gerentes.

O papel de gerente, criado originalmente por Taylor, é considerado na Toyota como treinador, orientador e desafiador dos trabalhadores. Os gerentes devem encontrar oportunidades e oferecer desafios técnicos para os trabalhadores que se auto organizam para encontrar as soluções de menor esforço e melhor desempenho.

Neste ponto pode-se observar que os gerentes deixaram de executar a função controle, e parte das outras funções gerenciais como planejamento da produção também foram assimiladas pelos operadores.

Ohno (1997) destaca ainda que o uso dos quadros *Kanban* faz com que todos vejam e entendam melhor o processo produtivo, e com isso, os operadores passam a entender melhor sobre o processo e contribuir, tomando decisões próprias,

aumentando sua participação, gerando maior desempenho e tornando-se mais satisfeitos com seu trabalho. Após o início do uso dos cartões *Kanban* se estabelece uma relação de dependência com os trabalhadores que tornam-se essenciais ao processo.

A otimização do processo produtivo pela identificação de gargalos, demoras, esperas, fica mais fácil a partir do monitoramento do uso dos quadros *Kanban*, pois estes dão visibilidade ao processo como um todo.

As soluções de otimização passam a ser vistas por todos, o que gera maior colaboração dos operadores, que sentem-se instigados à colaborar com a melhoria do processo e eliminação de desperdícios.

As funções do *Kanban* (OHNO, 1997) são:

1. Fornecer informações sobre quanto produzir ou transportar ;
2. Fornecer informação sobre a produção;
3. Impedir a superprodução e o transporte excessivo;
4. Servir como uma ordem de fabricação afixada às mercadorias;
5. Impedir produtos defeituosos pela identificação no processo;
6. Revelar problemas existentes e manter o controle de estoques.

Cabe observar que cada operação deve implementar a técnica de *Heijunka* visando atualizar o ritmo de produção para conseguir sempre abastecer seus clientes internos.

Ao receber uma solicitação de produção através de um cartão *Kanban*, a operação fornece imediatamente o produto devolvendo junto o cartão *Kanban* da solicitação. Após o fornecimento, este supermercado estará desabastecido, podendo ser realizada uma nova produção para reabastecê-lo.

Para realizar a produção de reabastecimento são utilizados os cartões *Kanban* locais da operação, referentes aos insumos necessários à produção, que são usados para solicitar estes insumos às operações fornecedoras responsáveis. Cada cartão é levado à um fornecedor interno, que deverá devolvê-lo com o insumo solicitado.

Após receber os insumos necessários, devidamente acompanhados dos cartões *Kanban* locais, a produção é iniciada e não deve ser interrompida até o seu término. Ao terminar, o produto pode ser usado para repor o supermercado ou ser entregue para alguma operação solicitante.

A quantidade de cartões *Kanban* é mantida sempre a mesma para evitar superprodução, não sendo possível criar novos cartões ou perder cartões.

As regras para o uso do *Kanban* (SHINGO, 1996) são:

1. O processo cliente solicita somente o número de itens indicado no cartão *Kanban* de movimentação do processo fornecedor ;
2. O processo fornecedor produz e fornece itens somente na quantidade de produção e sequência indicadas pelo cartão *Kanban* ;
3. Nenhum item é produzido ou transportado sem um cartão *Kanban* ;
4. O *Kanban* sempre acompanha os próprios produtos ;
5. Produtos defeituosos nunca podem ser enviados para o processo seguinte. O resultado é mercadorias 100% livres de defeitos ; e
6. Reduzir o número de *Kanban* aumenta sua sensibilidade aos problemas.

2.1.2. Autonomação (*Jidoka*)

A autonomação, ou *Jidoka*, é considerada como um dos pilares do STP e de sua filosofia (OHNO, 1997). A palavra *Jidoka* (do Japonês) significa a autonomia da máquina, e visa a separação do trabalho do homem e da máquina. Dentre as soluções desenvolvidas no STP, a autonomação talvez seja a solução mais antiga, pois este conceito foi assimilado por Ohno a partir de seu trabalho na tecelagem Toyoda, tendo sido desenvolvido originalmente por Sakishi Toyoda.

A autonomação representa a autonomia das máquinas, monitoramento e prevenção de anormalidades na produção. O conceito de autonomia está presente em vários aspectos do STP, principalmente na relação entre funcionários e gerência. Conforme já discutido, o STP se desenvolveu buscando soluções para problemas reais, na impossibilidade da aderência ao paradigma da administrativa de Ford/Taylor.

Este conceito é considerado por Shingo (1996) como o início da revolução de melhoria dos processos (quinta revolução), onde o objetivo principal deixa de ser o mero funcionamento e a produção a qualquer custo.

Segundo Shingo e Ohno, prevenir defeitos é mais importante do que encontrar erros, e por isso é necessária a parada da produção sempre que uma anormalidade for detectada, visando a investigação, a análise da causa raiz e a

eliminação das condições que possibilitem a anormalidade dentro do processo produtivo.

O desvio momentâneo de um problema pode afastar o engenheiro responsável da causa real do problema, tornando a melhoria do processo produtivo muito mais difícil. Segundo Shingo, as melhores oportunidades para se detectar a causa raiz dos problemas são as primeiras aparições do problema. Quanto mais se contorna o problema principal pelo ataque aos efeitos causados, mais se distancia da causa raiz, tornando a identificação da causa cada vez mais obscura.

Esta mudança de ponto de vista, de melhoria da operação para a busca de um fluxo produtivo contínuo e estável, associado a busca de defeitos e a separação do trabalho do homem e da máquina é a filosofia do *Jidoka*.

Como consequência o trabalho do homem torna-se mais eficaz no sentido de agregar mais valor, pois atividades repetitivas e de inspeção praticadas anteriormente foram automatizadas e passaram a ser realizadas pela máquina.

Além da mudança na forma de pensar, entre melhorias operacionais ou locais para melhorias no processo ou globais, a relação entre o operador das máquinas com a empresa também estava sendo alterada.

As técnicas de *Poke-Yoke*, complementares do conceito de *Jidoka*, são discutidas no tópico a seguir.

2.1.3. *Baka-Yoke* ou *Poke-Yoke*

A partir do conceito de *Jidoka*, visto no tópico anterior, que Ohno aprendeu gerenciando a tecelagem, Ohno cria os instrumentos *Baka-Yoke* (anti-idiota), posteriormente chamados *Poke-Yoke* (anti-erro), que impedem que insumos sejam introduzidos de forma incorreta nas máquinas e monitoram o resultado do processamento.

Os dispositivos *Poke-Yoke* possibilitam a autoinspeção reforçada, que atua sobre 100% da produção através de controles físicos e mecânicos (SHINGO, 1996).

Estes dispositivos fazem parte de um objetivo maior, a meta de Defeito Zero. O conceito de *Jidoka* implica na autonomia não só de máquinas mas também de pessoas. As máquinas e as pessoas devem ser autônomas para encontrar anormalidades, e parar a produção assim que as encontrarem.

Existe três tipos de *Poke-Yoke*: o de contato, o de quantidade fixa e os de sequência fixa.

Poke-Yoke de contato são os mais comuns, pois verificam através de contato se as condições de encaixe de uma entrada ou as condições finais de uma saída estão de acordo com especificações.

Os *Poke-Yoke* de quantidade asseguram que uma quantidade nem maior nem menor de um dado produto estejam presentes antes ou depois do processamento. Também pode-se assegurar que um número de movimentos ou atividades sejam feitas.

O *Poke-Yoke* de sequência verifica e alerta sobre uma sequência específica de atividades ou passos que devem ser conduzidos, para assegurar que algo seja feito da forma correta.

Dispositivos *Poke-Yoke* devem apresentar os seguintes atributos:

- Inexpressividade (não atrapalhar a produção; quase não sendo notados);
- Simplicidade de implementação (serem facilmente implementados);
- Serem específicos para a sua necessidade (dedicados para uma dada situação);
- Desenvolvidos pelos operadores (o operador deve ser especialista no que faz) .

Os *Poke-Yoke* também podem ser específicos para uma das funções:

- Não aceitar entrada de defeitos ;
- Detectar e alertar sobre a geração de defeitos ;
- Não deixar que um defeito passe a diante .

Cabe esclarecer a diferença entre erros e defeitos. Defeito é um produto que não atende a especificação. Erros são considerados desvios dos objetivos de um processo, e por isso geram defeitos.

Para tanto, a implementação do método *Poke-Yoke* segue as seguintes fases:

- Detecção: busca identificar o erro antes que este se torne um defeito.
- Minimização: busca minimizar o efeito do erro.

- Facilitação: busca adoção de técnicas que facilitem a execução das tarefas nos processos de manufatura ou no fornecimento de serviços.
- Prevenção: busca ações para impedir que o erro ocorra.
- Substituição: busca substituir processos ou sistemas por outros mais consistentes.
- Eliminação: busca a eliminação da possibilidade de ocorrência de erros pelo redesenho do produto, do processo de obtenção ou da prestação de serviços.

As técnicas de *Poke-Yoke* complementam podem ainda usar os quadros e sinais *Andon* para sinalizar anormalidades ou paradas na produção.

2.1.4. *Andon*

A palavra *Andon* em Japonês significa Luz. Este recurso representa mais uma contribuição dos operadores para a melhoria do processo como um todo. Sua atuação reforça a eficiência do gerenciamento visual. Os dispositivos *Andon* são quadros luminosos utilizados para sinalizar problemas na linha de produção que revelam o estado de máquinas e do processo.

Conforme o tamanho da planta industrial cresce a complexidade para os operários e engenheiros de processo, e pode tornar-se difícil receber o *feedback* das máquinas sobre anomalias encontradas. Para organizar a comunicação entre as máquinas e os operadores e/ou engenheiros de processo, a sinalização das máquinas e dos operadores foi organizada em painéis de fácil visualização, normalmente colocados em lugares altos, para que todos possam visualizá-los. Estes quadros são normalmente encontrados junto com sirenes para chamar atenção.

Segundo Shingo, nem todo o problema precisa parar a produção. Alguns problemas podem ser configurados para avisar ao operador mas continuar o processamento. Alguns problemas podem também ser acionados manualmente pelo operador através de interruptores ou cordas que acionem o alerta de problemas.

Ao se detectar um problema mais grave, o *Andon* indica a máquina para que os engenheiros de processo investiguem o caso. Caso não existam problemas ele

sinalizará o estado atual do desempenho da planta industrial e as suspeitas de mal funcionamento que tiver conhecimento.

Cabe ressaltar o caráter visual e simples da técnica, e que o principal objetivo é a melhoria do processo, não só de problemas locais.

Estes quadros também servem para levar informações aos trabalhadores, como o ritmo atual de produção, se a linha estiver com problemas ou precisar ser parada o *Andon* anuncia a parada.

2.1.5. Padronização de Operações

No STP, conforme descrito no tópico busca pela perfeição (2.5.4 - 5°passo), tem-se a perfeita noção de que a perfeição não existe. Entretanto este é o real motivo da filosofia da busca pela perfeição. Como a perfeição não existe este é um alvo móvel, sendo sempre atualizado pelas novas sugestões sobre como se pode avançar a partir da qualidade atual.

Desta forma, a cada patamar de qualidade atingido, representa o estado do processo reflete a melhor forma de se proceder conhecida e especificada até então. Esta padronização deve ser do conhecimento de todos para atingir três objetivos básicos:

. Possibilitar o surgimento de ideias para melhoria destes mesmos processos ;

. Garantir que todos executem suas operações da melhor forma conhecida ;

. Homogeneizar o conhecimento entre os que já tem muita experiência e os que tem menos experiência.

A padronização de operações foi uma das características que ajudou a trazer o desempenho do STP para os níveis atuais. Sem a padronização os diversos esforços como a Troca Rápida de Ferramentas, (SHINGO, 1996), se perderiam nas práticas conhecidas por alguns e desconhecidas por outros.

A padronização de operações se situa no contexto de ensino e aprendizado dentro da organização, e demonstra o valor dos trabalhadores na estrutura.

2.1.6. *Kaizen*

Kaizen significa melhoria incremental, ou evolução. As mudanças *Kaizen* são normalmente mudanças pequenas e locais, sugeridas pelos técnicos, que devem ser motivados a encontrar melhorias para o processo. Tais melhorias devem visar a

eliminação ou minimização de custos e desperdícios e aumento da satisfação do cliente.

Uma mudança *Kaizen* precisa ser discutida com a equipe, pois os trabalhadores da equipe são os maiores especialistas sobre o processo.

Quando um trabalhador tem uma ideia de como eliminar um desperdício, aumentar o fluxo do processo, melhorar a adequação da satisfação para o cliente, prevenir alguma falhar, monitorar anormalidades ou automatizar uma operação, ele deve expressar sua ideia em uma proposta de *Kaizen*. Deve haver um consenso entre os trabalhadores sobre os prós e contras. Os engenheiros de produção são questionados sobre riscos e a validade da mudança.

O processo é medido antes da melhoria e os resultados são comparados após a implantação provisória da melhoria. Caso a melhoria seja comprovada ela torna-se permanente e é documentada como um novo processo padronizado.

O comportamento para as melhorias deve ser :

. Observar uma oportunidade de melhoria ;

. Medir os benefícios e desperdícios atuais ;

. Estimar os resultados da melhoria (prós e contras);

. Planejar a implantação da melhoria e medidas defensivas ;

. Obter o consenso de todos ;

. Implementar a melhoria ;

. Medir os resultados ;

. Comparar com a situação anterior ;

. Obter consenso sobre os resultados ;

. Prevenir recorrências ;

. Padronizar a operação ;

. Aplicar nas operações similares ;

. Buscar novas oportunidades de melhoria.

As melhorias radicais do processo são consideradas operações *Kaikaku*, vistas a seguir.

2.1.7. *Kaikaku*

Kaikaku significa melhoria radical, ou revolução. A técnica de *Kaikaku* visa planejar, implantar, avaliar uma mudança radical no processo. Diferentemente do *Kaizen* onde as mudanças são pequenas e pontuais, e pode-se estimar e medir localmente o impacto das mudanças, nas operações de *Kaikaku*, normalmente as mudanças são sentidas no processo como um todo, sendo necessário um maior planejamento. Portanto, tais operações são menos frequentes e devem ser conduzidas com o apoio de especialistas e a participação da equipe.

Operações *Kaikaku* normalmente envolvem um treinamento para a equipe de trabalho, visando seu preparo para uma nova realidade.

Uma mudança de orientação do processo, por exemplo de vertical (funcional) para horizontal em fluxo, seria uma mudança *Kaikaku*.

Estas mudanças, ao contrário das *Kaizen*, apresentam resultados imediatos

Aplicar JIT à um processo é considerado um *Kaikaku*, pois produzir para estoques requer toda uma infraestrutura custosa em termos de espaço, trabalhadores, tempo para estocagem. Portanto, para criar uma estrutura JIT, apesar de baratear e tornar o processo muito mais rápido, trará à tona diversos problemas que devem ser antecipados, e requer uma reestruturação radical da empresa.

Criar uma nova cadeia de valor agregado pode implicar em uma mudança radical, pois pode afetar várias operações, eliminando operações, otimizando e reconfigurando-as, na busca pela eliminação de desperdícios e agregação do valor desejado pelo cliente.

2.1.8. Efeito dos Lotes de Trabalho

Quando um trabalho é realizado em lotes, só se tem a oportunidade de identificar as falhas depois do termino do lote. Na prática, após a produção os lotes são armazenados para futura recuperação. Ao recuperar um produto para uso, pode-se ter a surpresa de identificar um erro não detectado. Neste caso, provavelmente, outros produtos do mesmo lote deverão apresentar o mesmo problema. Entretanto a falha que originou o erro já pode ter se modificado, tornando-se mais difícil de ser identificada.

Diz-se portanto que lotes grandes escondem falhas, como o nível do mar esconde as rochas.

Os lotes grandes em ambientes de serviço, ainda são piores, pois implicam em chaveamento de contexto mental. Vários trabalhadores administrativos em vários departamentos, recebem várias listagens contendo vários objetos de trabalho a serem processados.

Cada trabalhador faz somente a sua parte do trabalho para cada item do relatório. Entretanto nenhum deles tem a noção completa de um objeto de trabalho. Nenhum deles compreende o problema completamente, a ponto de sugerir melhorias para o processo. Eles somente podem sugerir melhorias locais. Entretanto melhorias locais podem ir de encontro as necessidades do processo, fazendo com que o desempenho global caia.

O pior efeito é sobre o desempenho do sistema. Este sistema só emite output, a sua primeira saída demora o tempo relativo a soma do processamento de todos os registros em todos os departamentos anteriores mais o tempo do primeiro processamento no último departamento.

Nos trabalhos em lotes não se consegue enxergar os desperdícios pois eles estão ocultos no resultado do trabalho em lotes.

Outra desvantagem do trabalho em grandes lotes é dificuldade de reação a demanda. Na verdade, empresas tornam-se reféns da prática de trabalhar em lotes, gerando estoques e não tendo como reagir a falta ou excesso de demanda.

2.2. Pensamento Enxuto

2.2.1. Eliminação de desperdícios (Muda)

Desde o início da produção de automóveis na empresa Toyota os recursos financeiros, as máquinas e a própria demanda dos consumidores configuravam um quadro bem diferente da realidade da empresa de Ford. Enquanto empresas ao redor do mundo seguiam o exemplo da Ford, o Japão estava saindo da segunda guerra mundial, na qual o país sofreu o ataque que dizimou duas de suas maiores cidades, desestruturando sua sociedade e economia.

Este quadro de falta de recursos levou a empresa Toyota a considerar hábitos e operações diversos dos padrões praticados no ocidente. Para resolver as dificuldades então encontradas, os responsáveis pela produção, bem como os próprios operadores, buscaram formas de eliminar os desperdícios, encontrando através de experimentos com tentativas e erros, padrões operacionais mais eficientes.

Vários autores citam que a base do STP é na verdade a busca pela eliminação de desperdícios, visando com isso diminuir o custo da empresa. Esta atitude, que foi desenvolvida pelo bom senso dos japoneses, atualmente parece simples, porém contrastou com o costume e o pensamento científico da época, segundo o qual a geração de lucros somente seria eficiente quanto atingisse escalas maiores de produção. As equações que definem os dois pensamentos podem ser vistas na Figuras 1 e 2.

$$\begin{gathered} \textit{Fórmula da FORD} \\ PV = CP + L \\ \textit{Preço de Venda} = \textit{Custo de Produção} + \textit{Lucro} \\ \textit{Lógica da Contabilidade de Custos} \\ (\textit{utilizada até } 1973) \end{gathered}$$

Figura 1: Fórmula de Ford

$$\begin{gathered} \textit{Fórmula da Toyota} \\ L = PV - CP \\ \textit{Lucro} = \textit{Preço de Venda} - \textit{Custo de Produção} \\ \textit{Lógica da Contabilidade Enxuta} \\ (\textit{utilizada após } 1973) \end{gathered}$$

Figura 2: Fórmula da Toyota

Segundo a fórmula da produção em massa, o lucro deve ser obtido pela diminuição do custo de compra ao comprar grandes volumes junto aos fornecedores. Neste caso o preço de venda leva em consideração o Lucro desejado. Na Toyota, compra-se somente sob demanda, entretanto diminui-se custo pela

eliminação de desperdícios durante a produção, gerando custos de produção cada vez menores e mantendo-se o preço levemente abaixo do mercado.

Ohno (1997) considerou a existência de três formas de trabalho: trabalho líquido, trabalho que não adiciona valor porém que suporta o trabalho efetivo e perdas. Portanto segundo Ohno o trabalho que adiciona valor sob o ponto de vista do cliente deve ser otimizado, as operações que não adicionam valor mas que ainda não podem ser eliminadas devem ser alvo de melhorias e automatização, e os desperdícios serem eliminados.

Segundo Shigeo Shingo (1996), foram documentados sete tipos de desperdícios, que foram portanto incorporados a filosofia da empresa e se buscou desde então sua erradicação. Uma das primeiras definições de Shingo é que todas as operações que não agregam valor ao cliente sejam desperdícios.

As perdas considerados desperdícios e algumas soluções são descritos por Shingo são (Tabela 1):

Tabela 1: Desperdícios segundo Shigueo Shingo

Desperdícios	Soluções	Definições	Motivadores
1.Superprodução	Sincronizar processos Diminuição de estoques	Produção além do demandado pelo cliente	Diminuir o esforço e o tempo de produção.
2.Espera	Lote unitário	Tempo no qual uma operação aguarda o resultado de outra.	Eliminar o tempo improdutivo para atender ao cliente no menor tempo possível.
3.Transporte	*Layout* da planta Transportes racionais	Movimento de produtos ou insumos pela planta de produção.	Eliminar movimento para otimizar esforço e o tempo de produção.
4.Processamento	Engenharia de Valor Análise de Valor	Fazer exatamente e somente oque o cliente deseja.	Não fazer nem mais nem menos do que o cliente deseja.
5.Estoque	Caçar variabilidade Sincronizarprocessos Pequenos Lotes	Eliminar estoques de insumos, produtos semi-acabados ou acabados.	Diminuir o investimento, aumentar a qualidade do produto, foco no produto e melhoria da qualidade do produto, aumentar a velocidade do processamento, diminuir o transporte,
6.Movimentação	Linearização do processo	Diminuir a movimentação dos trabalhadores	Aumenta o desempenho do processo, diminui o tempo de produção.
7.Defeitos	Inspeção, eliminar defeitos, não descobri-los *Poke-Yoke*	Detectar defeitos automaticamente e evitar sua repetição.	Aumento sensível da qualidade e diminuição de erros.
8. Desperdício de criatividade	Autonomia técnica, auto-gerenciamento	Valorização do conhecimento técnico e do conhecimento acumulado pelos trabalhadores para fins técnicos. Diminuição ou eliminação da cadeia de comando e controle.	Possibilita soluções técnicas rápidas, dinâmicas e de qualidade. Aumento do foco na qualidade. Aumento da sinergia na equipe.

(Adaptado de SHINGO, 1996)

2.2.2. Eliminação de Sobrecarga (*Muri*)

Segundo Imai (1997) muri representa a uma condição extenuante de trabalho, tanto para operários, máquinas e para o processo. Esta sobrecarga deve ser medida pela real capacidade de cada operário ou máquina, visando que este possa cumprir suas metas com qualidade. Qualquer sinal de esforço demasiado deve ser observado como uma sobrecarga. Os operadores devem ter tempo para se organizarem e pensarem sobre o que fazem e as máquinas devem ter tempo para manutenção preventiva, e não devem ser usadas senão diante de demandas reais.

Morgan e Liker (2008) compreendem muri como empurrar processos, operadores e máquinas além dos seus limites naturais. Pessoas sobrecarregadas produzem trabalhos imprecisos, imperfeitos, com problemas de qualidade e riscos de segurança. Equipamentos sobrecarregados causam apagões e defeitos. E processos sobrecarregados geram filas de trabalho, aumentam o tempo de ciclo, geram erros e não respondem de forma consistente, não sendo portanto previsíveis.

2.2.3. Diminuição de Irregularidade (*Mura*)

Masaaki Imai (1997) entende que existam duas formas de mura. Uma é a interrupção do fluxo de trabalho, do trabalho de um operador ou de uma máquina. A outra é a inconstância de ritmo
como a variação no tempo para realizar uma tarefa, causando demoras ou sobrecargas de trabalho.

Morgan e Liker (2008) consideram que sempre ocorrem variações e que estas devem ser estudadas visando sua compreensão e o dimensionamento de recursos. Os recursos necessários deverão estar disponíveis visando não baixar a qualidade dos trabalhos.

2.2.4. Cinco princípios básicos

O Pensamento Enxuto (PE) pode ser explicado como a identificação do valor real, o alinhamento da melhor sequência de operações que criam o valor, a realização desta cadeia de operações sem interrupção ou demoras, somente diante da solicitação do cliente, buscando a melhoria contínua do processo através da eliminação de desperdícios e a redução dos custos de produção.

Os passos do pensamento enxuto oferecem suporte para a melhoria de um processo estabelecido. Entretanto três fatores críticos de sucesso precisam ser atendidos: transparência no processo, fortalecimento da equipe e suporte da administração.

O pensamento enxuto deve ser conhecido por todos, seus objetivos e as ações de melhoria devem ter a maior transparência possível apresentando *feedback* imediato visível por todos, e os trabalhadores devem participar ativamente do processo.

A equipe de trabalhadores que executa as operações do processo deve ser valorizada e respeitada como a autoridade técnica sobre como o trabalho deve ser feito. Os trabalhadores devem buscar aprender e se desenvolver para serem especialistas no trabalho que realizam.

O gerente e a administração devem dar suporte para os trabalhadores removendo os impedimentos apontados por eles e oferecendo desafios para que eles resolvam tecnicamente, como por exemplo a eliminação de desperdícios, melhorias no fluxo e melhorias na adequação do valor desejado pelo cliente.

Desta forma Womack e Jones (1998) identificaram os cinco princípios abaixo:

1°- Identificação de Valor

Segundo Womack e Jones (1998), o valor é o produto ou serviço que o cliente deseja ou precisa segundo seu ponto de vista, na quantidade, hora, local em que precisa e à um custo aceitável. Percebe-se nesta definição de valor que são considerados o escopo do desejo, o momento, o local, o custo e a quantidade.

O local como componente do valor se divide em duas acepções descritas no pensamento enxuto. Porém clientes precisam ser caracterizados, para fins desta discussão como clientes internos e externos (SHINGO, 1996).

Segundo Shingeo Shingo (1996), clientes e fornecedores externos são pessoas físicas ou organizações fora da empresa produtora. Clientes e fornecedores internos são identificados a cada operação ou processo dentro de qualquer empresa. Um fornecedor interno é um funcionário ou departamento que fornece informações ou insumos para a execução de uma dada operação. Clientes internos são funcionários ou departamentos que dependem que informações ou insumos sejam terminados e entregues. Neste contexto o valor é considerado como

informações ou insumos, e também precisa ser definido sob o ponto de vista do cliente.

O custo de um bem ou serviço é a preocupação de produção mais comum, e foco principal no desenvolvimento no planejamento da produção. De fato, o custo pode inviabilizar a demanda, excluindo uma organização do mercado, fazendo com que ela arque com custos de manutenção de estoques inviáveis ou recursos humanos inativos.

O planejamento de fluxos de trabalho internos nas organizações deve levar em consideração o custo para o atendimento de demandas internas, o que caracteriza a busca por processos produtivos mais rápidos e econômicos.

Entretanto, a partir da definição de valor do Pensamento Enxuto, pode-se observar que um valor definido com precisão pode apresentar-se como indesejado se não for avaliado sob a ótica do cliente. Por exemplo um programa de controle de estoques pode ser ótimo para um cliente e inútil para outro. É portanto necessária a investigação junto ao cliente sobre o que é especificamente o valor por ele desejado.

O momento no qual o cliente deseja um bem ou serviço é também essencial, pois depois de um período de desenvolvimento e produção do bem ou serviço, aquela necessidade pode ficar desatualizada ou até mesmo extinguir sua necessidade. Na indústria de *Software* projetos de desenvolvimento de médio porte podem levar mais de um ano para serem terminados. Tais projetos dificilmente estarão aderentes às necessidades originais das organizações para as quais tenham sido idealizados.

Para clientes internos, a espera por fornecedores internos que trabalhem em lotes pode atrasar o fluxo de trabalho, fazendo com que os clientes fiquem parados ou assumam outras atividades paralelas, aumentando o risco de erros e a diminuição da qualidade em seu trabalho.

O local onde o valor é entregue, para clientes externos, é importante pois não se deseja pagar por transportes longos ou mesmo esperar muito tempo para receber o que se deseja.

Para clientes internos, o local onde os fornecedores entregam o valor é importante para a organização pois poderá definir eficiência, velocidade e desperdício dentro da organização. A entrega de insumos em locais distantes de onde serão utilizados é considerado um desperdício por Ohno (1997). Fornecedores

internos que trabalham gerando lotes intermediários normalmente demoram para terminar seus lotes, gerando normalmente estoques intermediários. Desta forma ocorrem duas movimentações que não seriam necessárias no caso de lotes unitários: da produção para o estoque e do estoque para o cliente. Neste caso, a solução em lotes unitários é a não formação de estoques e a aproximação de fornecedor e cliente, fazendo com que o produção unitária flua imediatamente para o cliente.

No processamento de informações estes fatos também ocorrem, como pode ser observado em relatórios que estocam problemas a serem tratados. O processamento de uma dada informação fica estocado em relatórios, sem que aja foco e percepção específica de sua realidade individual. Com isso as informações são tratadas com menor qualidade e velocidade. Os clientes de uma dada informação são forçados a esperar os resultados de todo o lote.

Womack e Jones (1998) citam 3 tipos de distorção comumente encontradas na administração de processos produtivos:

- A Mentalidade Financeira predomina sobre a Realidade Cotidiana
- A Realidade Técnica predomina sobre o Desejo do Consumidor
- Onde o valor é criado

A primeira distorção, Financeira sobre Cotidiana, diz respeito a empresários que estão distantes da produção e da realidade do cliente. Desta forma concentram sua habilidade e suas decisões em questões financeiras da organização. Como efeito geral, o valor gerado para o cliente neste tipo de organização encontra-se desfocado do que o cliente realmente deseja.

A segunda distorção, onde a Realidade Técnica predomina sobre o Desejo do Consumidor, representa a supervalorização da técnica. Este problema pode ser observado em processos intelectuais e técnicos, onde administradores não tenham como acompanhar pelo fator técnico obscuro. Desta forma a comunidade técnica afasta-se de objetivos mais sensatos e perde-se na técnica pelo prazer técnico.

A terceira distorção citada por Womack e Jones é sobre o local onde o valor é criado. Os autores argumentam que a geração do valor pode ser melhor aproveitada quando gerado próximo ao seu mercado consumidor, o que pode ser também fora do Japão. A produção precisam ser pensada sob a ótica do cliente, pois baixar os preços em função de baixa demanda é desperdício, e sinal de planejamento

ineficiente. No caso da Toyota, os vários gerentes japoneses, treinados dentro da Toyota, preferiam sempre desenvolver suas plantas industriais dentro do Japão, ao invés de pensarem em outros países, sendo esta um distorção do conceito de valor.

Womack e Jones (1998) discutem que a cadeia de valor deve ser entendida em seus fornecedores, em um regime de parceria que busca a eliminação do desperdício específico para o negócio. Alguns exemplos são fornecidos demonstrando que a investigação mais aprofundada na cadeia de agregação de valor dos fornecedores apresenta valores desnecessários para a empresa Toyota enquanto cliente.

2º- Mapeamento da Cadeia de Valor Agregado (Value Stream Mapping)

A cadeia de valor representa o conjunto das operações do processo produtivo sendo analisado. Cabe esclarecer que cada processo produtivo pode ter mais de um processo, responsáveis por exemplo por linhas de produto ou linhas de serviços.

O objetivo da identificação inicialmente é conhecer as operações, mapeando:

. As informações necessárias de entrada e saída ;

. Os materiais necessários e produzidos ;

. O relacionamento entre as operações ;

. O tempo de espera até o início da execução da operação ;

. O tempo médio de execução de cada operação ;

. Quanto cada operação agrega de valor ao produto sob o ponto de vista do cliente.

Cada operação deve ser analisada visando sua classificação quanto aos três tipos básicos de operações propostos por Ohno (1997):

1) Operações que agregam valor ;
2) Operações que não agregam valor, mas são temporariamente necessárias ;
3) Operações que não agregam valor e podem ser eliminadas.

Após identificadas as operações as demoras e os tempos de execução, pode-se imaginar um novo modelo que contemple uma versão melhorada da cadeia de

valor, onde se possa encontrar formas de eliminar ou otimizar as operações que não agregam valor, tipo 2 e 3.

As operações dos tipos 2 e 3 devem ser eliminadas, ou automatizadas, ou também padronizadas visando que seu desempenho mínimo seja fixado, estabilizando o processo.

As sugestões dos trabalhadores serão essenciais para a melhoria e eliminação destas atividades, refazendo os procedimentos padronizados.

Um novo mapa com as atividades propostas deve ser iniciado, visando a melhoria do processo em uma melhoria *Kaikaku* (tópico 2.3.9).

No próximo passo, fluxo, serão otimizadas as operações internamente.

3º- Fluxo contínuo de valor

A definição de fluxo de valor, segundo Womack e Jones (1998) é fazer com que as atividades que agregam valor sejam executadas em um fluxo constante desde o início até o fim, com a entrega ao cliente. Ohno (1997) diz que tudo o que o STP faz visa diminuir o tempo entre a solicitação do cliente e a entrega do valor esperado.

O passo de definição de fluxo tem como objetivo portanto eliminar as demoras e esperas entre operações, tornando o fluxo “contínuo” no sentido de ininterrupto, e buscando eliminar o Mura do processo (ver tópico 2.5.3). O fluxo do processo deve ser portanto contínuo e estável, visando sua previsibilidade.

O maior problema citado por Womack e Jones neste passo é a conscientização de que lotes grandes não geram eficiência, mas sim desperdício. O paradigma de grandes lotes foi introduzido no inconsciente coletivo por Ford, com o sucesso da produção em massa. A Toyota provou que esta não é a forma mais eficiente de produção.

O processo de produção em grandes lotes gera estoques intermediários e demora mais para a disponibilidade do lote. O lote sendo gerado somente torna-se disponível após o tempo necessário para sua conclusão. Se a produção de um lote de 50 peças demora 50 minutos e uma peça demora 1 minuto para ser feita, na produção em lote haverá o desperdício de 49 minutos para a utilização de uma peça.

Este mesmo problema é abordado por Goldratt (1998) ao discutir a multitarefa nociva, no contexto da corrente crítica, ao discutir filas (tópico 2.6.3).

Percebe-se uma ligação íntima entre o lote unitário do Pensamento Enxuto e a Corrente Critica, no sentido da demora para obtenção de resultados e na baixa qualidade do que é produzido, pela falta de foco, impedindo o que Liker (2005) chama de linearização do processo.

Segundo Liker (2005), um processo dividido em várias operações que trabalham em lotes, deverá consumir muito mais tempo do que o necessário para a produção em lote unitário. De fato a produção destas mesmas operações em lote unitário elimina os estoques intermediários, conforme discutido no tópico sobre desperdícios de estoque (2.5.1.5).

Os operadores devem ser incentivados a padronizar e melhorar as operações documentadas, visando a melhoria contínua no sentido de atingir aos valores desejados pelo cliente, gerando menos esforço humano, maior velocidade e qualidade.

Womack e Jones (1998) consideram que o maior problema para a implantação do fluxo são os agrupamentos funcionais como departamentos. Segundo os autores a visão de departamentos cria um desempenho subótimo para os processos.

De fato a estrutura departamental é deficiente pela falta de visão do todo, incentivando lotes de trabalho, gerando paradas no fluxo em suas fronteiras, entre outros problemas.

Outro problema segundo os autores é o uso de processos desconexos e agregados, que segundo os autores não enxergam ou estimulam o fluxo da agregação de valor como um todo, frequentemente parando o fluxo nas fronteiras dos processos, dos departamentos e das empresas.

Portanto, um processo produtivo idealmente deve ser coeso sob a ótica de geração de valor para o cliente e prover fluxo dentro de seus limites.

Womack e Jones citam 3 etapas para criar fluxo:

1) identificar o início e o fim da cadeia de agregação de valor ;
2) ignorar fronteiras de departamentos, organizações, profissões e funções, e eliminar todos os empecílhos e esperas de fluxo contínuo ; e

3) repensar as ferramentas, técnicas e decisões para eliminar retro fluxo, falhas, sucata, paralisações, de forma que não se ande para traz.

Entretanto fluxo não é o suficiente, a seguir será explicado como seguir a demanda.

4º- Estabelecimento de Demanda (Just-in-Time)

A ideia do funcionamento *Just-in-Time* (JIT), segundo Ohno (1997) nasce de Kiichiro Toyoda, que em conversa pessoal com Ohno diz que em um processo produtivo inteligente as peças e ferramentas deveriam chegar as mãos dos operários no momento de sua aplicação. Esta ideia evolui com a visualização da solução nos supermercados americanos, onde os produtos eram repostos somente depois que os níveis das prateleiras baixavam.

A ideia do JIT, foi testada durante algum tempo na oficina de Taiichi Ohno (1950-1955), com tentativas e erros, até que todos os problemas foram equalizados, quando em 1955 o sistema provou sua eficiência na oficina gerenciada por Ohno e foi então implantada na fábrica na qual trabalhava.

Cabe ressaltar a diferença estrutural que causa a implantação desta técnica. Implantar produção somente a partir da demanda implica em não produzir enquanto não houver demanda. Somente a demanda deve comandar o processo produtivo anterior, com a ordem de produção. Isto é válido para a demanda entre clientes e fornecedores internos e externos. A produção de algum bem ou serviço somente poderá começar a partir da demanda pelo cliente. A produção de insumos somente pode começar pela necessidade real de operações ou processos a jusante. Portanto esta é a mudança que diferencia o STP da Ford.

Desde o ano de 1908, a empresa Ford já havia descoberto o conceito de fluxo, amplamente utilizado na produção em massa. Os ideais de Henry Ford foram aprendidos pelos engenheiros de produção da Toyota, Taiichi Ohno, Kiichiro e Eiji Toyoda, e Shigueo Shingo. Entretanto os engenheiros da Ford não estudaram estes mesmos ideais do fundador da Ford.

Segundo Ohno (1997), o JIT é um sistema de produção puxada, onde a produção é iniciada para atender à um cliente, seja este externo ou interno. A ideia de clientes internos nasce da necessidade de fornecimento interna de componentes

necessários a montagem de produtos a jusante no processo, com a dependência de componentes fabricados a montante. Os processos dependentes são clientes dos processos fornecedores.

As demandas são atendidas pelos supermercados de peças que são posteriormente repostos por fluxos de fornecimento ou produção desacoplados.

5º- Busca pela perfeição

A perfeição é um alvo móvel, nunca atingido, porém eternamente perseguido. Os passos anteriores não precisam ser atingidos de uma só vez. Ao contrário, pode-se crescer na medida do entendimento da filosofia e do aprendizado sobre o próprio processo. O pensamento enxuto da suporte ao bom-senso, que deve ser estimulado a quebrar as barreiras do senso coletivo e das verdades inabaláveis.

A busca pela perfeição implica no gerenciamento de infinitas etapas de aproximação do perfeito através de melhorias incrementais e radicais. A otimização se dá par e passo com o aprendizado dos trabalhadores e da administração sobre o processo. Quanto mais o cliente puxa valor da organização mais revela os impedimentos para o fluxo, e quanto maior a velocidade do fluxo mais desperdícios se tornam aparentes na cadeia de valor, e quanto mais se executa a cadeia de valor mais *feedback* se obtém sobre a satisfação do cliente.

No âmbito organizacional é necessário esclarecer que os fatores críticos de sucesso são a autonomia técnica e o comprometimento da equipe de trabalhadores. Os trabalhadores devem ser respeitados como os especialistas na engenharia das operações, estimulados a serem cada vez melhores e assumirem o papel de melhorar continuamente suas operações e o processo.

Cada trabalhador deve pensar no processo como um todo e sempre conhecer as necessidades dos colegas que trabalham na sequência da cadeia de valor. O objetivo de cada operação deve ser claramente agregar valor para os clientes externo e interno.

A cadeia de valor deve ser constantemente otimizada pelos trabalhadores, verificando oportunidades de otimização e comparando os resultados com os procedimentos padronizados anteriores. Cada novo procedimento padronizado deve ser consenso de todos os trabalhadores. Os desperdícios devem ser estudados e eliminados em discussão por toda a equipe técnica.

Cabe ao gerente apontar os problemas e oferecer desafios, respeitando a soberania da equipe no aspecto técnico. Os desafios oferecidos devem ser estimulantes para adequar a agregação de valor, diminuir desperdícios, aumentar o fluxo, atender melhor aos clientes internos e externos.

Todo este processo precisa entretanto seguir a regra da transparência para todos, conforme comentado no início, e planejar o *feedback* esperado para cada melhoria proposta. As melhorias devem representar de fato um incremento positivo no resultado final do processo, não somente para uma operação.

Outra forma de melhoria é a melhoria radical, aqui chamada de *Kaikaku*, que reflete uma reengenharia no processo vista no tópico 2.3.9. Enquanto a melhoria contínua gera benefícios pequenos constantemente, uma reengenharia deve gerar benefícios maiores em curto prazo.

3. Referencial Teórico - Processos de Software

Neste capítulo será discutido o histórico da Engenharia de *Software*, suas contribuições, os processos de desenvolvimento de *Software* tradicionais, e os processos ágeis, considerados revolucionários.

3.1. Histórico

Tomando especificamente foco para o desenvolvimento e manutenção de *Software*, percebe-se a relação da TI com a Engenharia de *Software* (ES), termo criado pelo comitê de ciências da North Atlantic Treaty Organization (NAUR & RANDALL, 1969).

Os objetivos da engenharia de *Software*, na sua criação (NAUR & RANDALL, 1969) eram resolver os problemas de falhas graves que já ocorriam em *Software* e a falta de um processo que tornasse estável a produção de *Software*. Esta estabilidade visava a previsibilidade e reprodutibilidade essenciais ao estabelecimento de qualquer engenharia.

Estes objetivos foram sendo alterados e atualmente, segundo a IEEE (1990) seus objetivos são :

- A aplicação de uma orientação sistemática, disciplinada, quantificável e reprodutível para o processo de desenvolvimento e manutenção de *Software*;
- O estudo destas orientações.

Apesar destas disposições, os profissionais e acadêmicos desta área vem buscando encontrar métodos em outras áreas de conhecimento que deem suporte para a criação de *Software*, sem entretanto obter muito sucesso pois os métodos utilizados em grande maioria provêm da administração ao invés da engenharia (POLLICE, 2005).

O desenvolvimento de *Software* surge como uma atividade associada ao uso e operação dos computadores e é tratado inicialmente de forma empírica e artesanal. A dedicação dos profissionais de desenvolvimento de *Software* à condução de trabalhos mais corretos, com melhores atributos de qualidade como precisão, manutenibilidade, são perceptíveis nos trabalhos como os relatos de

trabalhos antigos como os desenvolvidos no Departamento de Defesa (DoD) dos EUA, NASA, IBM, e de pesquisadores como Djikstra, por exemplo.

A busca por excelência no desenvolvimento de *Software* direcionou estudos acadêmicos a busca de processos de desenvolvimento com o objetivo de estabelecer características que assegurassem a produção de *Software* com qualidade.

Segundo Larman e Basili (2003), dentre as práticas mais antigas do desenvolvimento de *Software* encontra-se o Desenvolvimento Iterativo e Incremental, sendo possível identificar indícios de sua utilização pelos idos da década de 1960 em diversos trabalhos e relatos.

Ainda segundo Larman e Basili, em Royce (1970), este processo passa a ser considerado pela comunidade de desenvolvimento de *Software* como um processo empírico e ultrapassado, sendo o processo cascata definido por Winston Royce um novo processo, semelhante ao empregado nas engenharias, e portanto esperava-se que com mais chances de gerar qualidade.

A partir do trabalho de Winston Royce, aparece o Ciclo de Vida de Desenvolvimento de *Software*, conforme descrito em Pressman (2006) e as técnicas de Análise Estruturada (Yourdon, 1975; GANE e SARSON, 1979).

De acordo com Larman e Basili (2003), a má interpretação do artigo de Royce (1970) dá origem à um novo paradigma para o desenvolvimento de *Software* que impõe alguns problemas, a saber:

- estabelecido a partir do processo de engenharias estáticas ;
- com características da produção em massa ;
- centrado em documentos ;
- com alto nível de chaveamento de contexto ;
- retorno do investimento somente no final do processo .

3.2. Processo de Desenvolvimento de Software

Este processo tem seu contexto na produção de *Software*, ferramenta necessária a quase totalidade das atividades humanas, seja para lazer, trabalho, indústria de manufatura e serviços.

Segundo Pressman (2006), a engenharia de *Software* prevê métodos que definem como se deve desenvolver *Software*. Estes métodos quando implementados criam um processo de produção.

Dijkstra e colaboradores (1972) conceitua o desenvolvimento de *Software* como a diminuição da distância intelectual entre um problema do mundo real e sua solução computacional.

Royce (1970), preocupado com o gerenciamento de uma equipe de trabalho muito grande, e com dificuldades no gerenciamento de alguns casos dentro desta equipe resolveu o problema aproveitando um modelo de processo, baseado em processos em fases, que foi conhecido como o modelo *Waterfall* ou Cascata (Figura 3).

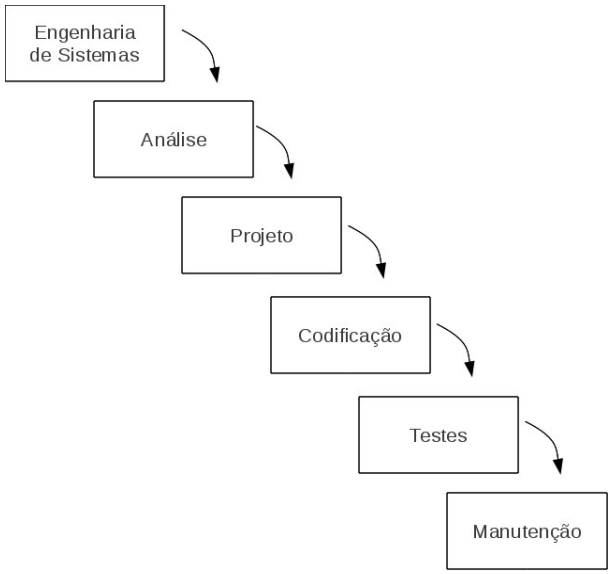


Figura 3: Ciclo de Vida Clássico (PRESSMAN, 2006)

Este processo de desenvolvimento propõe a transformação de requisitos declarados pelos clientes ou patrocinadores em *Software*, utilizando para isso estágios com operações bem definidas, nas quais o desenvolvimento é dividido visando otimizar o trabalho de cada grupo de profissionais, e prevendo antecipar as necessidades de informação das fases posteriores.

Segundo Kruchten (2000), este processo se baseia em processos da engenharia civil, onde os métodos, técnicas, postulados, restrições e riscos são bem comportados e apresentam variação mínima. Entretanto, na engenharia de *Software* o contexto de trabalho é muito dinâmico, pois deve representar a evolução e sobrevivência das empresas no mercado, não podendo portanto ser tratados por processos pouco responsivos à mudanças.

3.2.1. Engenharia do processo

. Engenharia de Sistemas

O engenheiro de sistemas colhe junto aos clientes os requisitos de sistema sem um maior aprofundamento, e planeja uma arquitetura de sistema composta por hardware, *Software* preexistente e *Software* à ser desenvolvido, e seres humanos, visando atender a estes requisitos. A parte de *Software* é conceitual e lógica, prevendo implicações futuras. Nesta etapa são planejadas as funcionalidades do *Software* e a interface entre os diversos elementos *Software*, hardware e seres humanos. Esta etapa tem como saída um documento com as observações do engenheiro de sistemas.

. Análise de Requisitos de *Software*

O engenheiro de *Software*, recebe a documentação de sistema gerada na fase anterior, e colhe novamente os requisitos para o *Software*, desta vez com maior profundidade de detalhes, visando a compreensão detalhada do domínio das informações do cliente, bem como a função, as interfaces e o desempenho requeridos. Todos estes dados são documentados e revistos com o cliente, gerando um relatório de análise.

. Projeto

O engenheiro de *Software* recebe o relatório de análise e planeja como atingir aos objetivos definidos na etapa anterior, planejando as seguintes dimensões:

. A arquitetura do *Software* ;

. A caracterização da interface ;

. A estrutura dos dados ; e

. Detalhes procedimentais .

O objetivo do projeto é possibilitar a avaliação da qualidade do projeto antes que a codificação tenha iniciado. Quanto mais detalhado o projeto, mais mecânica torna-se a fase subsequente. Este projeto é documentado e torna-se parte da configuração do *Software*.

. Codificação

O programador recebe os projetos da fase anterior, e traduz a especificação para linguagem compatível com o computador.

. Testes

Após a tradução do desenho em código, devem ser realizados testes do *Software*. Devem ser feitos testes lógicos internos, verificando todas as instruções do código, e aspectos externos, verificando se diante de determinadas entradas de dados se recebe resultados presumidos.

. Manutenção

Reaplica todas as etapas anteriores sobre o *Software*, visando adaptá-lo para as mudanças exigidas pelo cliente.

3.2.2. Problemas do processo de desenvolvimento de *Software*

Pressman (2006) cita que mesmo os primeiros defensores deste modelo declararam sua fragilidade e apontam os seguintes problemas:

. Projetos reais raramente se adaptam às fases propostas ;

. Muitas vezes o cliente não consegue declarar os requisitos inicialmente ;

. Não consegue acomodar incertezas no início do processo ;

. Não consegue acomodar mudanças no decorrer da execução ;

. Versões do programa só estarão disponíveis ao final do cronograma ;

. Erros detectados quando o programa está pronto são fatais .

Rubem Melendez Filho (1990) detectou os seguintes problemas:

. Processos dos usuários são muito dinâmicos, sujeitos à alterações diárias ;

. Fornece pouco suporte para a melhoria de sistemas existentes ;

. Não da suporte a consistência de redundâncias ;

. “Gera documentação extremamente longa, que nem gerentes nem usuários lêem”;

. Linguagem técnica diferente da do cliente ;

. Eficiência dependente da capacidade de comunicação escrita do analista ;

. “Usuário é obrigado a aprovar especificações sem muita firmeza”;

. “As especificações são congeladas durante o projeto. De um lado o usuário pode mudar sua visão das necessidades de informações à medida que passa a pensar mais no sistema. Do outro, o analista se sente apreensivo porque não tem certeza de que entendeu as necessidades reais do usuário”;

. “Especificações demasiadamente detalhadas ensejam a existência de erros.”

. “Existem provas estatísticas de que erros de especificação lógica sejam maiores do que erros de programação.”

Eduard Yourdon (1988) é taxativo dizendo que a metodologia de análise estruturada, como é chamado o processo cascata na época, não atinge os objetivos, e que técnicas como prototipação sejam muito mais eficientes.

Tom DeMarco (2002) pede desculpas por ter incentivado o uso do modelo cascata (DeMarco, 1975), e explica que utilizou uma única vez para um sistema de controle de processos em telefonia, onde funcionou bem. Os problemas apontados por DeMarco são:

. Documentação sobre o projeto todo são somente fonte de retrabalho ;

. Quebrar a análise em partes menores é essencial para diminuir a complexidade ;

. Um processo definido nunca será universal para desenvolvimento de *Software* .

Chris Gane (1988) apresenta os seguintes problemas:

. O grau de detalhe das documentações gera retrabalho ;

. Desestímulo dos programadores por não se sentirem parte pensante do processo .

Edward Yourdon (1989) observa que apesar do processo ser ultrapassado, até o final da década de 80 ainda estava sendo ensinada no currículo das universidades, e que outras técnicas como prototipação atenderiam muito melhor a comunidade.

Rubem Melendez Filho (1990) acrescenta que a quantidade de documentos gerados pela fase de levantamento de requisitos e análise de requisitos é insustentável. Não se consegue usar a documentação ou mesmo mantê-la atualizada.

Quando o levantamento é feito segundo o preconizado no processo, o volume de informação pode ser tão grande que inviabiliza as fases de análise e projeto.

A medida que as fases se sucedem a documentação das fases anteriores torna-se desatualizada.

Segundo o autor é muito frequente que os sistemas entrem em manutenção logo assim que terminam de ser construídos porque as necessidades dos clientes mudam durante o desenvolvimento.

Ainda segundo Rubem Melendez Filho (op. cit.), ao participar de um processo o usuário aprende e muda sua percepção rapidamente. Ao fornecer informações ele percebe as implicações da automatização dos processos e imagina novas soluções. Segundo este autor, ocorre o fenômeno descrito como o princípio da incerteza no desenvolvimento de *Software*: “A solução do problema altera o problema”.

Segundo o autor são necessários processos mais ágeis, que viabilizem acompanhar as mudanças necessárias ao *Software* para atender aos anseios dos clientes, como por exemplo a prototipação.

Em 1995, uma empresa de pesquisas chamada Standish Group, publicou um relatório chamado “Chaos” (STANDISH-GROUP, 1995), descrevendo projetos de *Software* que falharam e levantando características dos projetos e evidenciando possíveis causas. Segundo este relatório, as taxas de sucesso em projeto de *Software* são muito baixas, podendo se acompanhar o resultado da época e suas atualizações na Figura 4.

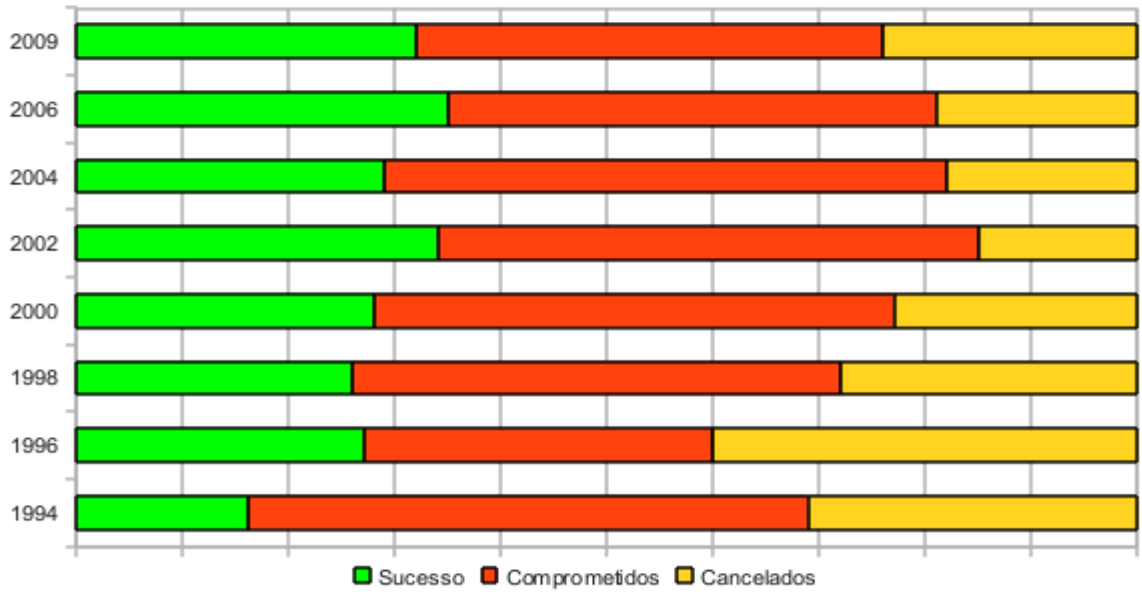


Figura 4: Projetos de Software baseado no Standish-Group (2010)

Este relatório apresenta os problemas mais comuns encontrados em projetos que falham (Tabela 2). Segundo este relatório, projetos são considerados comprometidos caso não atinjam os objetivos de custo, prazo e escopo.

Entretanto na Figura 4, pode-se observar que apesar de estar razoavelmente estável o percentual de projetos com sucesso, ainda apresenta um percentual baixo e preocupante, em torno de 30%. Ainda mais alarmante o percentual de projetos cancelados vem crescendo em um ritmo estável, atualmente 24%.

Tabela 2: Problemas comuns em projetos que falham (STANDISH-GROUP, 1995)

	Critério de Sucesso	Pontuação Ponderada
1	Requisitos Incompletos	13,1 %
2	Falta de envolvimento do Cliente	12,4 %
3	Falta de Recursos	10,6 %
4	Expectativas não realistas	9,9 %
5	Falta de apoio executivo	9,3 %
6	Mudanças nos requisitos	8,7 %
7	Falta de Planejamento	8,1 %
8	Tornou-se Desnecessário	7,5 %
9	Falta Gestão de TI	6,2 %
10	Falta de Conhecimento Técnico	4,3 %
11	Outros	9,9 %
	Soma	100,0 %

Segundo Womack e Jones (1998), os critérios 1 e 2, Requisitos incompletos e Falta de envolvimento do cliente, deveriam ser o ponto principal para uma organização enxuta, pois o cliente deve ser tratado como um parceiro, visando que este sinta confiança e satisfação em cada relação comercial. A satisfação do cliente é essencial e o processo enxuto deve entregar o valor correto, na quantidade correta, na hora correta, no local correto, com a qualidade máxima e custo enxuto.

Estes problemas teriam sido resolvidos pelo primeiro princípio do pensamento enxuto, Identificar valor, conforme visto no tópico 2.5.4 – 1° passo.

O critério 3, Falta de Recursos, pode ser contornado pelo pensamento enxuto tendo em vista a grande economia de recursos que ocorre em um processo enxuto. Várias operações podem ser enxugadas, e o processo em lote unitário, torna-se mais factível por depender de uma equipe menor.

O critério 4, Expectativas não realistas, é um ponto abordado pelo pensamento enxuto através dos conceitos de simplicidade e da cadeia de agregação de valor. Segundo Ohno (1997), deve-se fazer somente operações que agreguem valor ao cliente, sendo qualquer outro esforço um desperdício.

O quinto critério que atinge negativamente projetos de *Software* é a falta de apoio executivo. Entretanto, em um empreendimento enxuto para *Software*, este apoio é intrínseco, porque os trabalhadores devem ser valorizados. Em um processo enxuto, os desenvolvedores tem autonomia e poder de decisão técnica. Os executivos sabem que dependem da equipe técnica, de seu treinamento, de seu bem estar, senão o trabalho não terá qualidade. O respeito ao ser humano faz parte do pensamento enxuto, o que por si mitiga este problema.

O sexto critério, Mudanças nos requisitos diz respeito à natureza do pensamento enxuto. Um processo precisa necessariamente ser responsivo a mudanças, sendo esta a característica mais forte do STP, e a base de um processo enxuto para o desenvolvimento de *Software*.

O critério de falta de planejamento, sétimo, reflete a forma de planejamento inconsistente atualmente aplicada no processo de desenvolvimento de *Software*. Pode-se comparar o planejamento no desenvolvimento de *Software* à um planejamento MRP (Material Requirement Planning). A Engenharia de Produção já percebeu os riscos do uso deste tipo de planejamento, pois um erro de fornecimento ou atraso em processo durante o MRP ocasiona o escorregamento e fatalmente a falha em prazos. O pensamento enxuto resolve este problema com extrema simplicidade e facilidade usando sistemas puxados pela demanda como o *Just-in-Time*, previsto no quarto passo do pensamento enxuto.

O oitavo critério, Tornou-se desnecessário, sinaliza que o *Lead Time* do processo de desenvolvimento de *Software*, trabalhando em grandes lotes, não consegue entregar todas as funcionalidades para o cliente dentro do período de necessidade. A soma do tempo necessário para produzir todas as necessidades acordadas com o cliente inviabiliza a entrega. Entretanto o pensamento enxuto resolve facilmente este problema através de lotes unitários, com entregas imediatas, da análise da cadeia de valor, eliminando operações desnecessárias, e pelo fluxo, eliminando esperas e demoras.

O nono critério, Falta de Gestão de TI, sinaliza que o processo de desenvolvimento orientado para gestão funcional, ou vertical, é de fato mais difícil de ser gerenciado. As métricas de gerenciamento de projeto baseadas em linhas de código equivalem a medir quantidade de parafusos apertados em uma planta de produção.

Denning e Riehle, discutem o problema da área de engenharia de *Software* não apresentar os requisitos para ser considerada uma engenharia de fato, justamente pela falta de métricas adequadas para o controle de processos. Estes autores sugerem que devem ser praticada o controle do processo apoiado por *Software* dedicado ao controle, que possibilite medir e controlar melhor o processo.

Segundo o ponto de vista desta dissertação, a mudança da orientação do processo de vertical, com gestão funcional para horizontal, praticando a gestão por processos, possibilita aplicar as métricas de controle de fluxo utilizadas amplamente pelo STP e pela engenharia de produção para controlar processos de serviço e manufatura.

O décimo critério, falta de conhecimento técnico decorre do estilo gerencial funcional, onde o desenvolvedor não tem autonomia e não é valorizado. Nas indústrias Ford, em 1904, com o apoio de Frederick Taylor, se idealizou um paradigma batizado de Administração científica, na qual os funcionários devem somente especializar-se unicamente no conhecimento relativo a sua função.

Os trabalhadores americanos diferem dos japoneses justamente por não se preocuparem com o que acontece a montante do processo. Os japoneses sempre realizam seu trabalho pensando no valor gerado para as os clientes internos e externos, nas operações a montante do processo.

Para tanto, os trabalhadores devem ser respeitados, incentivados a se capacitar, a serem os melhores naquilo que fazem, e a entender o processo como um todo, visando inclusive a melhoria do processo. Não que seja necessário contratar os melhores, mas cada um deve buscar melhorar.

O último critério, Outros, apesar de não se poder discutir, cabe lembrar que o pensamento enxuto e o STP foram o primeiro processo produtivo a implementar a melhoria contínua, que visa a busca pela perfeição, que nunca é atingida, mas que mantém a equipe, o processo e os produtos em constante aproximação desta meta.

Outro dado apresentado diz respeito à funcionalidades entregues que no entanto não se fazem necessárias no uso diário, configurando desperdícios (Tabela 3).

Tabela 3: Uso de funcionalidades (STANDISH-GROUP, 1995)

Uso de funcionalidades	%
Nunca	45
Raramente	19
Algumas vezes	16
Normalmente	13
Sempre	7
Soma	100

Pode-se observar que somente 36% destas funcionalidades são mais utilizadas, dentre as entregues (Figura 5). Seguindo o pensamento enxuto, as funcionalidades Nunca ou Raramente utilizadas (64%) do que foi entregue, nunca teriam sido produzidas.

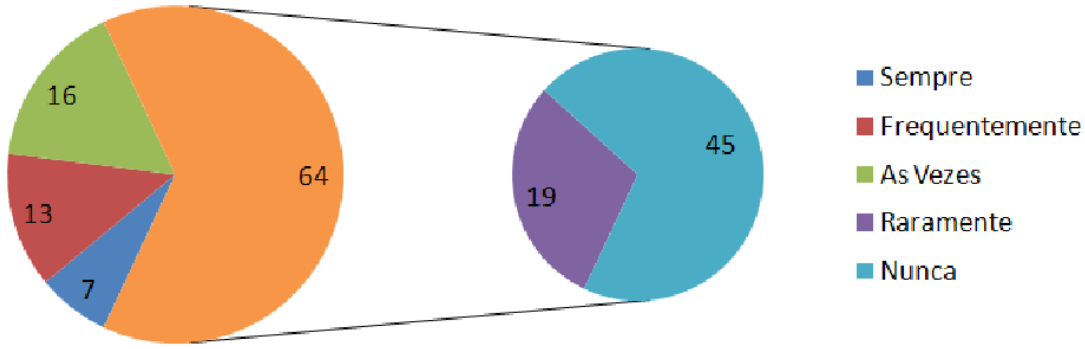


Figura 5: Relação entre funcionalidades entregues (STANDISH-GROUP, 1995).

Alguns autores discutem a validade destes dados por falhas em critérios metodológicos para a obtenção e tratamento de dados Eveleens e Verhoef (2010), no entretanto estes dados vem sendo utilizados como fonte de informação e não como valores nominais para estudos empíricos.

Barry Boehm (2000), discute que o término de projetos antecipado não significa que necessariamente os projetos tenham falhado.

3.2.3. Característica central

Cabe esclarecer que em todos os processos até este momento descritos, um ponto ainda parece não ter sido compreendido pela engenharia de *Software*. A cada momento da evolução nos processos de desenvolvimento de *Software*, normalmente motivado pelo número de falhas constatadas, os novos processos tornam-se menos prescritivos, deixando à mercê do engenheiro de *Software* as adaptações ou decisões que adequem o conjunto de ferramentas às suas necessidades.

Entretanto, dentre os conselhos e princípios remanescentes, as boas práticas e recomendações, permanecem com um ponto central ainda não percebido pela

engenharia de *Software*: Os preceitos da administração científica, da organização funcional e vertical e o trabalho em lotes, conforme discutido no tópico 2.3.12.

3.2.4. Falhas nos projetos de *Software*

Apesar de todas as críticas este modelo fez muito sucesso durante os anos seguintes e povoou o pensamento coletivo e as disciplinas de cursos, como um modelo que deveria funcionar, apesar de mostrar-se ineficiente por diversos motivos. Além disso, manteve-se presente e adaptado nas diversas metodologias vindouras, através de adaptações, suposições e mal entendidos dos praticantes que conheceram no passado o modelo clássico em cascata.

Autores como Charette (2005) vem observando um sem número de projetos que falham em atingir o escopo, os prazos, os custos ou mesmo a qualidade esperada. Este fenômeno já havia sido detectado na década de 60 e é conhecido como a crise do *Software* (DIJKSTRA, 1972).

O estado da arte em processos para o desenvolvimento de *Software* conforme a visão da Engenharia de *Software* é o conjunto de especificações da IEEE ISO/IEC 15504-5 (ISO-IEC, 2006) (SPICE) e CMMI.

A diferença entre estas propostas é que o processo CMMI (Capability Maturity Model Integration) (CHRISSIS *et al.*, 2003) mais conhecido propõe estágios de maturidade, sendo chamado também de processo estagiado. A proposta SPICE (*Software* Process Improvement and Capability dEtermination) (ISO/IEC, 2006) especifica um processo não estagiado, propondo a evolução contínua da maturidade.

Segundo (SALVIANO, 2006), ambas as especificações de processo, CMMI e SPICE, receberam pouca atenção da academia, e a especificação SPICE não foi adotada na prática, enquanto a CMMI teve grande sucesso no mercado. Ainda segundo este autor, a área de processos tem sido impulsionada pela prática de mercado, pois não recebe atenção acadêmica suficiente.

Entretanto, para utilizar o processo CMMI e usufruir de suas vantagens a empresa precisa investir pesadamente em cursos de capacitação e treinamento visando a certificação, e o próprio exame de certificação que também é pago. Desta forma, pequenas e médias empresas teriam dificuldade de adotar tais práticas.

Além disso, Burge e Brown (2002) citam como problemas a alta rotatividade da equipe de desenvolvimento de *Software* associada a longos períodos de trabalho que sujeitam a equipe a perda de desenvolvedores durante o processo de desenvolvimento.

As falhas na implantação de processos tradicionais normalmente acusam os seguintes fatores (ROCHA *et al.*, 2005; DEBOU & KUNTZMANN-COMBELLES, 2000):

- Falta de apoio da gerência ;
- Falta de infraestrutura para a implantação dos processos ;
- Estabelecimento de cronogramas impossíveis de serem atingidos ;
- Mudança da cultura organizacional ocasionada pela implantação de processos ;
- Falta de conhecimento em engenharia de *Software* por parte dos membros da organização ;
- Pressão no cronograma dos projetos que provocam o ab*Andon*o do processo por parte dos membros da equipe (isto também é reflexo da falta de apoio da gerência);
- Demora em visualizar os resultados .

A seguir será apresentado o modelo de maturidade CMMI do Instituto de Engenharia de *Software* (SEI, 2006)

3.3. CMMI

Os padrões de maturidade de *Software* CMMI e ISO/IEC 14406, mantidos pelo Instituto de Engenharia de *Software* (SEI), são um conjunto de melhores práticas utilizadas para o desenvolvimento e manutenção de *Software*, e tem como objetivo auxiliar as organizações na melhoria de seus processos.

O SEI foi criado em 1984, dentro da universidade Carnegie Mellon, pelo governo federal americano, com a missão de pesquisar práticas para suportar a produção de *Software* no prazo, custo e qualidade especificados, pois se entende que *Software* seja estratégico e crítico para o Departamento de Defesa Norte Americano (DoD – Department of Defense).

Watts Humphrey, um dos mais antigos contribuidores da IBM, junta-se à SEI em 1989 e produz o CMM (Capability Maturity Model) em 1998, que visa mapear o grau de maturidade e orientar o desenvolvimento de organizações no que diz respeito ao desenvolvimento, manutenção e aquisição de *Software*.

Entretanto foram criados vários CMM para áreas diferentes como Sistemas, *Software*, Hardware, Especificação e Aquisição, que geraram um custo elevado para sua implantação. A proposta do Capability Maturity Model Integration (CMMI) é uma evolução dos CMM integrando modelos para facilitar sua implantação. Os CMM de Engenharia de Sistemas e Engenharia de *Software* foram integrados criando o CMMI de Desenvolvimento.

A criação do CMM se baseou na premissa de que a qualidade de um produto ou serviço seja altamente influenciada pelo processo. Cabe observar que, infelizmente, os autores não conhecem a origem deste conceito, desenvolvido por Shingeo Shingo, e originada no conhecimento sobre o STP (ANTUNES *et al.*, 2008).

No texto introdutório do CMMI (SEI, 2010, p.5) observa-se a citação aos seguintes autores:

. Walter Shewhart (1931) - *Economic Control of Quality of Manufactured Product. New York: Van Nostrand*

. Phillip Crosby (1979) - *Quality Is Free: The Art of Making Quality Certain. New York: McGraw-Hill*.

. Edwards Deming (1986) - *Out of the Crisis. Cambridge, MA: MIT Center for Advanced Engineering*.

. Joseph Juran (1988) - *Juran on Planning for Quality. New York: Macmillan*.

Entretanto, além do controle estatístico sobre as operações proposto para os níveis mais altos do CMMI, somente nos níveis 4 e 5, não há evidências ao longo de todo o relatório, que se estende por mais de seiscentas páginas, sobre contribuições dos outros autores citados, sendo este comentário a única citação durante todo o texto.

Além disso, dentre as técnicas abordadas neste trabalho, citadas como boas práticas, pode-se observar a influência profunda da Administração Científica, da desconfiança no trabalho dos empregados, na delimitação de fronteiras para

funções e habilidades, na investigação e monitoramento da qualidade de execução do trabalho de cada trabalhador.

Na obra de Deming, citada (DEMING, 1986), o autor descreve a diferença entre a gestão baseada na Administração Científica e a gestão em busca pela qualidade, citanto em todos os capítulos de seu livro exemplos baseados na administração japonesa.

O próprio trabalho descreve sua atualização no que diz respeito a eficiência de processos:

… “Today, CMMI is an application of the principles introduced almost a century ago to this never-ending cycle of process improvement. ” … (SEI, 2006, p. 5).

Cabe esclarecer ainda que a indústria não veio descobrir o que é um processo recentemente, apesar das teorias sobre processo propostas pela Engenharia de *Software*.

O autor do CMMI, Watts Humprey em seu artigo “Why big *Software* project fail: The 12 key questions” (HUMPREY, 2005), destaca que a causa de falhas em projetos grandes é a falta de uma pesada enfase em planejamento.

Segundo Humprey (2005), um projeto grande deve implementar:

. Pesada enfase em planejamento ;

. Maior precisão no planejamento ;

. Decisão e controle centralizada, baseado nas estruturas militares ;

. Acompanhamento e controle sobre o estado do trabalho de cada desenvolvedor ;

. Saber o quanto um projeto já andou e quanto falta para terminar ;

. Saber o que cada um tem feito, examinar cautelosamente o que tem produzido ;

. Seguir a risca os planos e acabar o trabalho dentro de cada fase .

Humphrey afirma que os desenvolvedores não sabem planejar, não sabem em que ponto do projeto estão, e que portanto gerentes devem monitorar e controlar. Como solução o autor sugere que os desenvolvedores devem aprender a planejar precisamente o que fazem e seguir este planejamento.

Entretanto este mesmo autor cita que dois grandes projetos no qual trabalhou, o CCPDS-R das forças armadas do EUA, com 100 desenvolvedores, e o IBM

OS/360 com 3000 desenvolvedores, obtiveram sucesso com as seguintes características:

. Desenvolvimento evolucionário ;

. Entregas múltiplas ;

. Motivar os Desenvolvedores e oferecer Premiações ;

. Quebrar o *Software* em muitas pequenas partes ;

. Deixar o planejamento ser feito por cada parte .

Assim, paradoxalmente, tais características entram em conflito com os pressupostos por ele descritos em 2005 (HUMPREY, op. cit.) e se alinham às diretrizes do STP.

CMMI – Desenvolvimento

O CMMI-Desenvolvimento tem como objetivo auxiliar organizações na melhoria de seus processos de desenvolvimento de *Software*. Ele busca apresentar indícios de como atingir maturidade no desenvolvimento e manutenção de *Software* através de melhorias no processo.

Para tanto o CMMI documenta um conjunto de práticas em um framework no qual se pode escolher quais as práticas mais adequadas para aparelhar seu processo, observando as necessidades e capacidades, visando construir ou adaptar um processo que será guiado para a maturidade.

Segundo a SEI (2010) um processo une 3 dimensões críticas em um modelo (Figura 6)

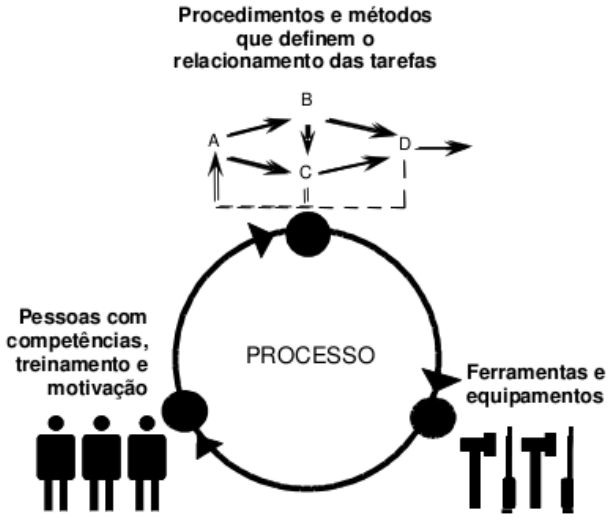


Figura 6: Três dimensões críticas (SEI, 2010)

Os modelos construídos pela SEI são agrupados portanto em Constelações que preveem também material de treinamento e de avaliação. Atualmente a constelação de CMMI para desenvolvimento (CMMI-Dev) é formada por dois modelos, a saber com e sem IPPD (Desenvolvimento Integrado de Processo e Produto). Outras constelações como a CMMI-Services e CMMI-Acquisition não são foco deste trabalho.

Representações

Existem também duas representações para os conceitos deste modelo, a representação Contínua e a Estagiada.

A representação estagiada, citada como a mais utilizada no mercado, é descrita em níveis de maturidade que propõe conduzir o processo para a maturidade enquanto atender aos requisitos de cada nível. Cada nível de maturidade associa um conjunto pré-definido de áreas, e o grau de maturidade do processo.

A representação contínua, pouco utilizada, é descrita em níveis de capacidade, permitindo a adoção de quaisquer área de processo isoladamente, visando simplificar o aprendizado e a implantação das práticas propostas. Os níveis de capacidade representam a melhoria associada dentro de cada área de processo.

Para comprovar formalmente a aderência aos níveis de maturidade ou capacidade empresas precisam pagar por um procedimento chamado certificação, baseado na norma SCAMPI (. Esta certificação necessariamente deverá ser precedida por uma consultoria que capacite, treine e implante os processos para o nível desejado.

O volume de recursos financeiros necessários para atingir os níveis do CMMI inviabilizam que empresas de pequeno e médio porte busquem tais certificações. Para resolver este problema a SOFTEX no Brasil implementa um programa de melhoria do processo de *Software* (MPS.Br), que visa ser compatível com o modelo CMMI, no qual oferece mecanismos que facilitam a implantação para empresas de pequeno e médio porte. Entretanto, mesmo assim os custos ainda são altos.

Representação Contínua

Esta representação foi assimilada pela SEI, e incorpora as características de processos prevista pela especificação ISO/IEC 15504, visando manter a compatibilidade com esta noma anterior.

Esta representação apresenta para cada área de processo o seguinte conjunto de níveis de capacidade: Incompleto, Executado, Gerenciado, Definido, Gerenciamento Quantitativo, Em Otimização. Desta forma cada processo pode atingir um nível de capacidade diferente.

Categorias

Categorias são representações de áreas de processo que se inter-relacionam. Estas representações visam ajudar as organizações a escolherem um ponto de partida parar empenhar esforços para sua capacitação e melhoria dos processos.

São previstas quatro categorias na representação continuada: Suporte, Gestão de Projeto, Gestão de Processo e Engenharia.

Um exemplo de adequação dos processos aos níveis de capacidade pode ser visto na Tabela 4.

Tabela 4: Exemplo de adequação de processos à níveis de capacidade.

			NÍVEIS DE CAPACIDADE					
Categoria	Área de Processo		0 Incompleto	1 Executado	2 Gerenciado	3 Definido	4 Gerenciado Quantitativamente	5 Em Otimização
Gestão de Processo	Foco nos Processos da Organização	OPF	■	■				
	Definição dos Processos da Organização +IPPD	OPD + IPPD	■	■				
	Treinamento na Organização	OT	■	■				
	Desempenho dos Processos da Organização	OPP	■	■	■			
	Implantação de Inovações na Organização	OID	■	■				
Gestão de Projeto	Planejamento de Projeto	PP	■	■	■	■	■	
	Monitoramento e Controle de Projeto	PMC	■	■	■	■		
	Gestão de Contrato com Fornecedores	SAM	■	■				
	Gestão Integrada de Projeto +IPPD	IPM +IPPD	■	■				
	Gestão de Riscos	RSKM	■	■	■			
	Gestão Quantitativa de Projeto	QPM	■	■	■	■	■	
Engenharia	Gestão de Requisitos	REQM	■	■	■	■		
	Desenvolvimento de Requisitos	RD	■	■	■	■		
	Solução Técnica	TS	■	■	■			
	Integração de Produto	PI	■	■	■	■	■	
	Verificação	VER	■	■	■	■	■	■
	Validação	VAL	■	■	■	■	■	■
Suporte	Gestão de Configuração	CM	■	■	■	■		
	Garantia da Qualidade de Processo e Produto	PPQA	■	■	■	■	■	■
	Medição e Análise	MA	■	■				
	Análise e Tomada de Decisões	DAR	■	■	■			
	Análise e Resolução de Causas	CAR	■	■	■			

Cada categoria tem uma visão básica e uma visão avançada, que serão descritas a seguir

Níveis de Maturidade

Qualquer empresa não certificada deve iniciar no nível um. Este nível não exige processos, constituindo simplesmente de um patamar conceitual para enquadrar a maturidade das organizações. Portanto, segundo o CMMI, as empresas devem buscar implementar inicialmente o nível 2.

Na representação estagiada a evolução de cada nível de maturidade é considerado como uma camada que representa a base para a melhoria contínua. Diante disto pode-se discutir que haja uma diferença sensível entre o conceito de melhoria contínua especificada no CMMI e para a Engenharia de Produção.

Na Tabela 5 podem ser vistas as áreas de processo nos níveis de maturidade a partir do qual são exigidos.

Tabela 5: Níveis iniciais de maturidade na Representação Estagiada

Maturidade Inicial	Categoria	Área de Processo	Sigla
2	Engenharia	Gestão de Requisitos	REQM
2	Gestão de Projeto	Gestão de Contrato com Fornecedores	SAM
2	Gestão de Projeto	Monitoramento e Controle de Projeto	PMC
2	Gestão de Projeto	Planejamento de Projeto	PP
2	Suporte	Garantia da Qualidade de Processo e Produto	PPQA
2	Suporte	Gestão de Configuração	CM
2	Suporte	Medição e Análise	MA
3	Engenharia	Desenvolvimento de Requisitos	RD
3	Engenharia	Integração de Produto	PI
3	Engenharia	Solução Técnica	TS
3	Engenharia	Validação	VAL
3	Engenharia	Verificação	VER
3	Gestão de Processo	Definição dos Processos da Organização +IPPD	OPD + IPPD
3	Gestão de Processo	Foco nos Processos da Organização	OPF
3	Gestão de Processo	Treinamento na Organização	OT
3	Gestão de Projeto	Gestão de Riscos	RSKM
3	Gestão de Projeto	Gestão Integrada de Projeto +IPPD	IPM +IPPD
3	Suporte	Análise e Tomada de Decisões	DAR
4	Gestão de Processo	Desempenho dos Processos da Organização	OPP
4	Gestão de Projeto	Gestão Quantitativa de Projeto	QPM
5	Gestão de Processo	Implantação de Inovações na Organização	OID
5	Suporte	Análise e Resolução de Causas	CAR

No capítulo 4 (próximo tópico 4.2) serão discutidas características dos modelo de processos. No próximo tópico serão vistos os métodos ágeis, que visam dinamizar o processo de desenvolvimento, tornando-o mais responsivo as mudanças, gerando *Software* o mais próximo possível do desejado pelo cliente e dando retorno rápido ao cliente.

3.4. Métodos Ágeis

Os métodos ágeis são um conjunto de práticas assimiladas de diversas áreas da administração, física, biologia, buscando o melhor desempenho, eficiência e qualidade para o processo de desenvolvimento de *Software*.

Os métodos ágeis concentram boas soluções trazidas de diversas contribuições recentes de várias áreas do saber, como administração, engenharia de produção, física, biologia entre outras.

Entretanto uma consideração prática, no caso da engenharia de produção talvez passe desapercebida por alguns praticantes da engenharia de *Software*: a

questão de que um processo na verdade depende do conhecimento acerca do fluxo de trabalho (*Workflow*) associado, e de um conhecimento mais amplo, sobre sistemas produtivos, suas características, sua evolução, seu manejo para que se possa propor ou manter um processo produtivo (SMITH & REINERTSEN, 1992).

Os métodos ágeis propõem que o desenvolvimento de *Software* seja um processo responsivo as necessidades de mudança solicitadas pelo cliente.

Segundo Boehm (1988) um dos maiores problemas do processo de desenvolvimento de *Software* é o planejamento exaustivo no início do processo. Os processos tradicionais entendem que o planejamento inicial possibilita estabelecer um cronograma de recursos e custos que serve de base para a execução do projeto.

3.4.1. Origens

Segundo Craig Larman e Victor Basili (2003), os métodos ágeis são derivados de práticas que se originaram do Desenvolvimento Iterativo e Incremental (IID) utilizadas na década de 1950. A prática de desenvolvimento de *Software* nesta época não era documentada formalmente. Os desenvolvedores adotavam o IID, reproduzindo o conhecimento por tradição quase oral, com poucos trabalhos científicos conhecidos, por exemplo Robert Glass (1969).

A partir da década de 1970, um trabalho de Winston Royce (1970) foi mal interpretado gerando um novo paradigma para o desenvolvimento de *Software*. Na verdade, Royce tenta expor a necessidade de se praticar o desenvolvimento pelo menos em duas etapas, chegando a um produto intermediário que seria avaliado pelo cliente, e retornaria *feedback* para melhorias em uma segunda etapa. Segundo Larman e Basili (2003), este artigo foi mal interpretado e considerado pela comunidade de engenharia de *Software* como um novo processo, em uma única passagem, gerando vários documentos para promover a transferência de conhecimento entre as fases, guardando bastante semelhança com outros processos de engenharias como os processos da engenharia civil por exemplo, que lida com escopos mais estáveis e menos variações durante a execução.

Cris Gane, Tom DeMarco, Yourdon, entre outros desenvolveram e expandiram o ciclo de vida estruturado de *Software* (SDLC), tornando-o um padrão da indústria e levando o IID ao esquecimento, sendo considerado por alguns como prática informal (LARMAN e BASILI, 2003).

A seguir será feito um breve resumo de algumas destas práticas.

3.4.2. *Scrum*

O processo *Scrum* [Rising2000] se baseia no processo de desenvolvimento de novos produtos da engenharia de industrial (TAKEUCHI e NONAKA, 1986) para trazer luz ao ciclo de vida de desenvolvimento de *Software* (SDLC).

O nome *Scrum* vem da formação no jogo de Rugby onde uma equipe age de forma coesa e compacta para forçar uma jogada. Esta técnica foi criada para gerenciar o processo de desenvolvimento e manutenção de *Software* visando a responsividade às constantes mudanças nos requisitos que dão origem ao *Software*. A produção de produtos de *Software* entregáveis é baseado em: requisitos, tempo, competitividade, qualidade, visão e recursos. Também é levado em consideração a natureza evolucionária do método.

A técnica *Scrum* é baseado em princípios:

- Produtos deverão se tornar subprodutos gerenciáveis.
- O progresso pode ocorrer mesmo com requisitos instáveis.
- Tudo tem que ser visível para todos.
- A comunicação interfere positivamente na qualidade.
- A equipe é responsável solidariamente por tudo que é feito no processo, compartilhando sucesso, fracasso e autoria de tudo até o fim.
- Clientes e usuários devem participar ao vivo do processo, interferindo e sabendo como funciona e com está o desenvolvimento.
- O relacionamento e o conhecimento devem ser ampliados durante o processo e que deve ser estimulada a expectativa pelo sucesso do projeto.

Todo o processo descrito deve ser de conhecimento de toda a equipe que conta com um elemento viabilizador e condutor do processo chamado de *Scrum Master* ou SM. Seu papel é de conduzir a equipe para as regras do *Scrum*, viabilizando e facilitando o processo.

Esta técnica parte da intenção de atender à requisitos utilizando soluções em *Software*. Tais requisitos são descritos através de estórias. Estas estórias que serão desenvolvidas são de responsabilidade do contratante ou administrador por ele designado, aqui chamado de dono do produto (*Product Owner* ou PO).

O cliente precisa estar também representado por especialistas designados que deverão participar intensivamente do processo, na descrição do domínio do problema e do problema, na especificação de testes junto a programadores, na definição de telas e relatórios, na resolução de dúvidas e na execução e validação dos produtos.

As equipes devem ser formadas de profissionais multi capacitados. Deverão ser capazes de realizar todas as tarefas dentro do desenvolvimento. O cliente ou especialista designado é considerado também um integrante da equipe.

No início do processo o cliente apresenta suas necessidades que são formatadas como estórias, que precisam de :

. Um responsável – Alguém que será usuário da funcionalidade

. Um objetivo – O que a estória deverá realizar

. Um resultado objetivo – Qual a expectativa do responsável após a execução da estória

. Critérios de aceitação – Quais riscos existe durante a execução da estória, quais problemas podem acontecer durante a produção, quais erros podem ser gerados, quais prejuízos podem ocorrer, quais resultados devem ser apresentados, qual o comportamento esperado após a execução da estória.

Estórias devem ser pequenas, coesas, simples, e gerar resultado útil sob o ponto de vista do cliente.

Cada estória definida pelo cliente é analisada pela equipe e são definidas tarefas essenciais para atingir aos objetivos das estórias. O conjunto das estórias e suas tarefas será chamado de Product Backlog.

A equipe deve discutir cada estória, definir tarefas necessárias, e estimar um esforço ou tempo de desenvolvimento para cada estória.

O PO deverá, após as estimativas da equipe, dar importância as estórias (priorização) de forma a que as estórias que apresentam maior risco ou importância e que mais agreguem valor para o cliente sejam desenvolvidas primeiro e as menos importantes fiquem para o final do processo de desenvolvimento.

Esta priorização feita pelo PO poderá mudar constantemente durante o desenvolvimento em função do aprendizado e da mudança de expectativas do cliente. Mudanças no escopo das estórias, na priorização ou mesmo o aparecimento de novas estórias são esperados e bem vindas. O PO deverá aprender durante o

processo, e desta forma deverão emergir novas soluções, nova visão de prioridade, mudança de conceitos, gerando o que pode-se considerar um aumento na qualidade do produto, por isso estas mudanças são bem vindas.

As estórias e tarefas tanto pendentes quanto as em produção deverão estar agrupadas e ser fisicamente visíveis para todos, preferencialmente em um quadro *Kanban*, utilizando notas e cores.

O processo é realizado em ciclos ou iterações chamados *Sprint*s, que normalmente duram entre uma e 4 semanas, preferencialmente uma semana, onde toda a equipe se responsabiliza solidariamente pelo desenvolvimento de uma ou mais estórias, seguindo a prioridade especificada pelo PO.

No início de cada *Sprint* é feita uma reunião de planejamento. O PO deverá explicar quais as características de cada estória que deseja ver desenvolvida naquele *Sprint*.

A equipe deve discutir e planejar novamente cada estória, definindo as tarefas necessárias, e gerando uma estimativa de tempo ou esforço para cada estória. Este esforço deve ser determinado pela equipe e não será negociável. Isto configura a qualidade interna do processo. O PO somente pode interferir na qualidade externa, mudando as características e prioridade do que deseja.

A partir das estimativas de cada estória o PO poderá mudar o escopo da estória e repensar a prioridade atribuída às estórias. Serão então definidas quais estórias poderão ser executadas no *Sprint* em função do tempo e da capacidade da equipe. Equipes já experientes calculam uma velocidade baseada na quantidade de estórias executadas em *Sprint*s anteriores.

Diariamente serão conduzidas reuniões em pé, buscando a sincronia da equipe e trabalhos realizados onde deverão ser relatados o mais brevemente possível os trabalhos finalizados, os impedimentos encontrados e trabalhos futuros.

O processo de desenvolvimento devera preferencialmente ser especificado com a participação do cliente. O usuário será responsável pela execução e validação dos produtos desenvolvidos pela equipe.

Cada *Sprint* deve entregar um ou mais módulos ou funcionalidades executáveis, que agregue valor sob o ponto de vista do cliente, e que possam entrar em produção para o cliente.

O processo termina quando o PO estiver satisfeito.

3.4.3. eXtreame Programming (XP)

O mote que embasa a programação extrema, segundo Beck (1999), é que as boas práticas conhecidas para o desenvolvimento de *Software* devem ser praticadas intensamente, ou em suas palavras, ao eXtremo. Se planejar é bom, devemos planejar ao eXtremo, toda semana. Se testar é bom, devemos testar ao eXtremo, todo dia. Se submeter à apreciação do usuário é bom, devemos fazê-lo no mínimo toda semana.

O desenvolvimento não é considerado com um esforço único para entrega de um produto ao final do ciclo de desenvolvimento. Na verdade, são previstas várias entregas parciais, em ciclos rápidos e curtos, objetivando gerar produtos para que o cliente possa dar *feedback* sobre a adequação das soluções, visando a eliminação das incertezas existentes em qualquer projeto de desenvolvimento.

A técnica de XP é baseada em quatro valores básicos (BECK, 2004, p. 45):

- Comunicação – Diversas falhas em projetos estão relacionadas a comunicação entre cliente e equipe de desenvolvimento ou entre os desenvolvedores.
- Simplicidade – No XP se busca encontrar a arquitetura mais simples possível para que o objetivo seja atingido, eliminando detalhes, beleza, até que funcione. Melhorias devem passar por refatoração.
- *Feedback* – O sistema deve responder sempre seu estado atual. Os clientes devem receber código sempre que possível, sua opinião é essencial.
- Coragem – Se o código estiver fora de controle, jogue fora. É melhor recomeçar do que tentar corrigir. Se alguma ideia valer a pena, invista nela.

Para atender à estes valores são definidos os seguintes princípios (BECK, 2004):

- *Feedback* Rápido – *Feedback* muito tempo depois da ação não gera aprendizado. Aprender a fazer deve trazer *feedback* imediato para fixar o conhecimento.
- Simplicidade Presumida – Tratar todos os problemas com a solução mais simples possível, somente adicionar complexidade através de refatoração se for necessário.

- Mudanças incrementais – Grandes mudanças de uma vez só tem grandes chances de não funcionar. Mudar incrementalmente, refatorar aos poucos mitiga o risco.
- Aceitação das mudanças – A melhor estratégia é aquela que preserva o maior número de opções enquanto resolve o problema mais urgente.
- Alta qualidade – Qualidade não é opcional, todo projeto deve ser considerado de alto risco.
- Ensinar aprendendo – Não ditar como fazer, contar a experiência e incentivar a aprender fazendo.
- Investimento inicial pequeno – não contar com muitos recursos no início, busque a justificativa primeiro.
- Jogar para ganhar – Não ser tímido, jogar para vencer, sem chance de errar e sem medo.
- Experimentação concreta – Toda decisão deve ser testada, experimentada e validada, senão torna-se um risco.
- Comunicação honesta e franca – Não deve haver receio ou meio entre os desenvolvedores, ou entre equipe e cliente, comunicação tem que ocorrer de forma eficiente.
- Trabalhar a favor dos instintos do pessoal, e não contra eles – práticas devem resolver os problemas de imediato, não dar rodeios.
- Aceitação de responsabilidades – Não comandar tarefas, aceitar responsabilidades para participar da equipe.
- Adaptação local – Não usar XP, adaptar XP para as condições locais.
- Viajar com pouca bagagem – Diminuir artefatos e usar poucos, simples e valiosos.
- Métricas genuínas – Estimativas com menores chances de falhar. No máximo 3 a 4 métricas que traduzam a realidade de forma clara para o cliente.

- Planejamento das iterações

Após o jogo do planejamento das estórias é realizado um novo jogo de planejamento para os *Sprint*s. Neste jogo usa-se cartões de tarefas para os próprios desenvolvedores, visando o planejamento de um *Sprint*.

- Gerenciamento

O gerenciamento deve funcionar viabilizando o trabalho da equipe e acompanhando as métricas. O processo deve ser monitorado com relação a execução das estórias. As métricas de estimativa devem ser comparadas com os resultados atuais e apresentados de forma visível para todos, por exemplo em um quadro de gerenciamento visual.

O gerente deve agir como um treinador e assumir a responsabilidade por decisões técnicas difíceis que a equipe não possa assumir. Apoiar aos iniciantes e fazer a interface com superiores. É muito importante que o gerente saiba se comunicar, ouvir e incentivar a equipe.

As intervenções em uma equipe XP devem ser feitas para evitar falhas do projeto ou prejuízos maiores para os desenvolvedores. Pode ser necessário dispensar um elemento que não esteja rendendo o suficiente para a equipe, antes que a equipe seja penalizada.

Um gerente não deve indicar como mudar algo, mas sim mostrar um efeito que deva ser corrigido. A equipe deve decidir como resolver o problema.

- Planning Poker

Descrita por COHN (2005) está é uma técnica de estimativa em conjunto, pelos membros de uma equipe de desenvolvimento, podendo ser feita como um jogo. Todos os membros da equipe, com a exceção do Cliente, participam de forma individual para chegar a um consenso de estimativa de esforço para a execução de uma estória.

Cada desenvolvedor recebe no início da seção um conjunto de cartas com valores possíveis de estimativa de pontos para a realização da estória.

De posse deste conjunto de cartas, cada desenvolvedor deverá, ao ser solicitado escolher e baixar na mesa sigilosamente as cartas que representem sua estimativa.

Desta forma, após todos os desenvolvedores baixarem suas cartas, sem que nenhum outro saiba sua opção, todos deverão virar as cartas simultaneamente, tornando possível uma discussão onde cada um justifique sua estimativa quando diferente da maioria do grupo.

Deve-se tentar novamente até que o grupo chegue a um consenso sobre um valor, cabendo argumentação entre os desenvolvedores na busca pela unanimidade.

- Programação em pares

Esta prática do XP é implementada diretamente no processo de desenvolvimento proposto e da suporte ao desempenho, atingi diretamente a melhoria da qualidade do produto e do processo de desenvolvimento.

Segundo Beck (2004), todo código deve sempre ser manipulado por duas pessoas em um mesmo tempo em parceria. Um deve comandar e o outro deve pensar de forma mais ampla e analisar o que está sendo escrito. Um parceiro deve pensar na implementação da solução, enquanto o outro deve pensa estrategicamente. Estes papeis devem se inverter em seções de 30 minutos. A formação de pares ainda pode variar entre a manhã e a tarde, por exemplo, pois os parceiros não deve ser fixos.

Williams e colaboradores (2000), chama a atenção para as várias vantagens da prática, afirmando que vários desenvolvedores que nunca realizaram são descrentes para os valores atingidos. Neste trabalho são apresentados dados sobre projetos onde o tempo de desenvolvimento de fato diminui, a quantidade de erros diminui, as chances de desenvolver arquiteturas incoerentes diminuem.

O trabalho em pares foi busca atingir os seguintes objetivos:

- Homogeneizar o conhecimento
- Aumentar o desempenho na realização das tarefas
- Somar esforços em problemas mais complexos
- Aumentar a qualidade das soluções
- Diminuição de risco de falhas ou mal entendidos

Uma vantagem atribuída nesta dissertação é amenizar a lei de Brooks (BROOKS, 1995), pois viabiliza o aumento de capacidade de trabalho na equipe. Brooks afirma em sua obra que adicionar pessoas em estágios adiantados de projeto aumenta o esforço de gerenciamento e treinamento, diminuindo o *Lead Time*. A programação em pares, entretanto diminui o *Lead Time* do projeto durante o aprendizado, proporcionando um aprendizado constante que produz resultados para o projeto, amenizando as consequências previstas por Brooks.

3.4.4. Testes de *Software*

Testes são importantes para garantir a qualidade de *Software* e demonstram falhas no processo. Um erro encontrado tardiamente implica em um backflow, um retorno a operação onde foi desenvolvido. Todo backflow é motivo para a parada da produção, avaliação das condições em que ele foi gerado e planejamento visando impedir que ocorra novamente.

Existem várias preocupações que precisam ser atendidas ao desenvolver *Software*. Crisping e Gregory (2009) citam as seguintes aspectos ou riscos a serem avaliados:

. Testes de unidade e integração;

. Testes de negócio;

. Testes de usabilidade e cenários;

. Testes de performance, carga;

. Testes de segurança.

Portanto, segundo o pensamento enxuto deve-se prevenir falhas mais intensamente do que procurar erros. As técnicas apresentadas a seguir visam prevenir e monitorar as falhas, gerando *feedback* imediato para a equipe de desenvolvedores, para que as falhas sejam sempre imediatamente identificadas e corrigidas, não passado para etapas seguintes do processo.

3.4.6. TDD

A sigla TDD (BECK, 2002) significa Test-Driven *Development* (Desenvolvimento Orientado a Testes), entretanto o termo teste não exprime satisfatoriamente as vantagens trazidas por esta técnica.

Apesar desta técnica ter sido descrita inicialmente visando testes, aos poucos os desenvolvedores que a aplicaram perceberam outros efeitos colaterais de sua utilização.

Apesar de não ser intuitivo, descobriu-se que:

. Diminui o tempo de desenvolvimento ;

. Auxilia na documentação do *Software* ;

. Auxilia no planejamento da arquitetura do *Software* ;

. Aumenta a qualidade do *Software* pela prevenção de erros ;

. Da confiança ao desenvolvedor, pois gera *feedback* imediato a cada anormalidade encontrada ;

. Suporta a manutenção, melhoria e refatoração do código ;

. Pode ser usado como uma métrica de qualidade é *Software* .

Kent Beck defende a teoria que os testes devem ser feitos antes do desenvolvimento do código propriamente dito. Fazer testes antes já era uma técnica que havia sido esquecida com o tempo. Na verdade outras técnicas parecidas, por exemplo DbC já existiam a mais tempo.

Segundo Beck, originalmente os desenvolvedores obtinham e ainda buscam resultados esperados para as funções que seriam desenvolvidas como forma de verificação, com o objetivo de garantir que haviam atingido os resultados certos. A técnica de TDD automatiza este processo, fazendo com que as expectativas sejam monitoradas automaticamente.

Entretanto, ao praticar TDD o desenvolvedor é cooptado para pensar na arquitetura dos *Software*, para escrever as expectativas de comportamento do *Software*. Assim se percebeu que desenvolvedores que praticavam TDD atingiam arquiteturas de *Software* melhor planejadas, que eram guiadas pelo esforço de imaginar o comportamento e as respostas do *Software* antes de começar a programar o código.

O ciclo de desenvolvimento do TDD :

1. Criar testes antes da funcionalidade, verificando o comportamento esperados ;

2. Integrar e executar todos os testes. Veja o resultado ;

3. Escrever ou modificar o código fonte para eliminar as anormalidades apresentadas ;

4. Integrar e executar todos os testes. Veja o resultado ;

5. Caso existam anormalidades voltar ao passo 3 (escrever código);

6. Caso sejam necessários mais testes adicionar testes e voltar ao passo 4 (integrar e testar);

7. Caso a funcionalidade ainda não esteja pronta, crescer a funcionalidade e ir ao passo 4;

8. Caso a funcionalidade esteja pronta, refatorar para melhorar a arquitetura e ir ao passo 4;

9. Terminou a funcionalidade

- Refatoração

O princípio que leva a refatoração é que não se deve programar para o ótimo. Um programa deve emergir paulatinamente, atingindo objetivos simples de forma planejada e estável. Objetivos devem ser simples e atingíveis. A construção do código é um esforço de aprendizado, portanto deve ser o mais simples possível. Durante a construção do código deve-se criar especificações que monitorem o comportamento do código. Após todos os objetivos serem atingidos, pode-se refazer o código, criando uma arquitetura mais eficiente, que suporte crescimento.

Só se deve refatorar após ter aprendido o suficiente sobre o domínio do problema, o código atingir os objetivos, e se ter um monitoramento do código que de segurança para a refatoração.

3.4.7. BDD

A técnica de Desenvolvimento guiado por comportamento (BDD) (NORTH, 2006) é considerada por Manhães (2010) um método ágil de nova geração, e visa integrar o ciclo de desenvolvimento de *Software*, desde a identificação de requisitos até a validação e aceitação do *Software*. Esta técnica surgiu em 2006 como uma evolução da técnica de TDD (BECK, 2002).

Uma das vantagens desta técnica é a automatização da especificação e validação do *Software*. Segundo esta técnica os requisitos são descritos utilizando uma linguagem ubíquoa executável, que permite o rastreamento dos critérios de aceitação dos requisitos diretamente no código fonte.

Segundo os métodos ágeis os requisitos são divididos em estórias que devem ter objetivos gerar valor mensurável para o cliente. Um requisito pode ser mapeado diretamente para uma ou mais estórias.

Cada estória deve ser descrita no método BDD sob o formato de um preâmbulo e vários cenários. O preâmbulo descreve uma funcionalidades, conforme abaixo:

Funcionalidade: <nome descritivo da funcionalidade desejada>

Como um <papel do interessado>

Eu preciso <alguma funcionalidade>

Para que <resultados esperados>

O <papel do interessado> pode ser por exemplo um vendedor, um engenheiro, um cliente, um diretor, alguém que venha a se relacionar com a funcionalidade.

O parâmetro <alguma funcionalidade> se refere à função que deverá ser providenciada.

Os <resultados esperados> especificam resultados esperados pelo interessado ao término da execução da funcionalidade.

Este preâmbulo precisa atender aos seguintes objetivos:

. Independência de outras funcionalidades ;

. Ser negociável e negociado entre desenvolvedores e cliente ;

. Gerar um valor bem definido para o cliente ;

. Ser estimável em termos de esforço para os desenvolvedores ;

. Ser pequena e não complexa, visando mitigar os riscos e diminuir o tempo de entrega ;

. Ser testável, pois é necessário que se definam critérios de aceitação que verifiquem e validem o *Software*.

Após o preâmbulo são construídos os cenários, seguindo a seguinte sintaxe:

Cenário: <nome descritivo do cenário1>

Dado que <pré-condição>

Quando <evento>

Então <resultado esperado>

Cenário: <nome descritivo do cenário2>

Dado que <pré-condição1>

E <pré-condição2>

Quando <evento1>

E <evento2>

Então <resultado esperado1>

E <resultado esperado2>

Desta forma um cenário descreve uma declaração do cliente de um comportamento esperado para a funcionalidade, dentro das pré-condições, o evento ou estímulo e os resultados que devem ser monitorados ou testados.

Pode-se observar que no segundo cenário foi utilizada a expressão lógica “E” para aumentar a quantidade de cada parâmetro informado.

Entre as características do Manhães (2010) cita:

. Ciclo *Outside-In* ;

. Estabelece uma linguagem universal ;

. Herda conceitos do TDD ;

. Passos de bebê ;

. Especificações de aceitação ;

. Especificações de unidade ;

. Expectativas de comportamento ;

. Documentação executável.

. Ciclo Outside-In

O desenvolvimento é guiado a partir dos requisitos e da visão do cliente até os artefatos do *Software* real sendo codificado e monitorado. São realizados a serie de passos abaixo (Figura 7):

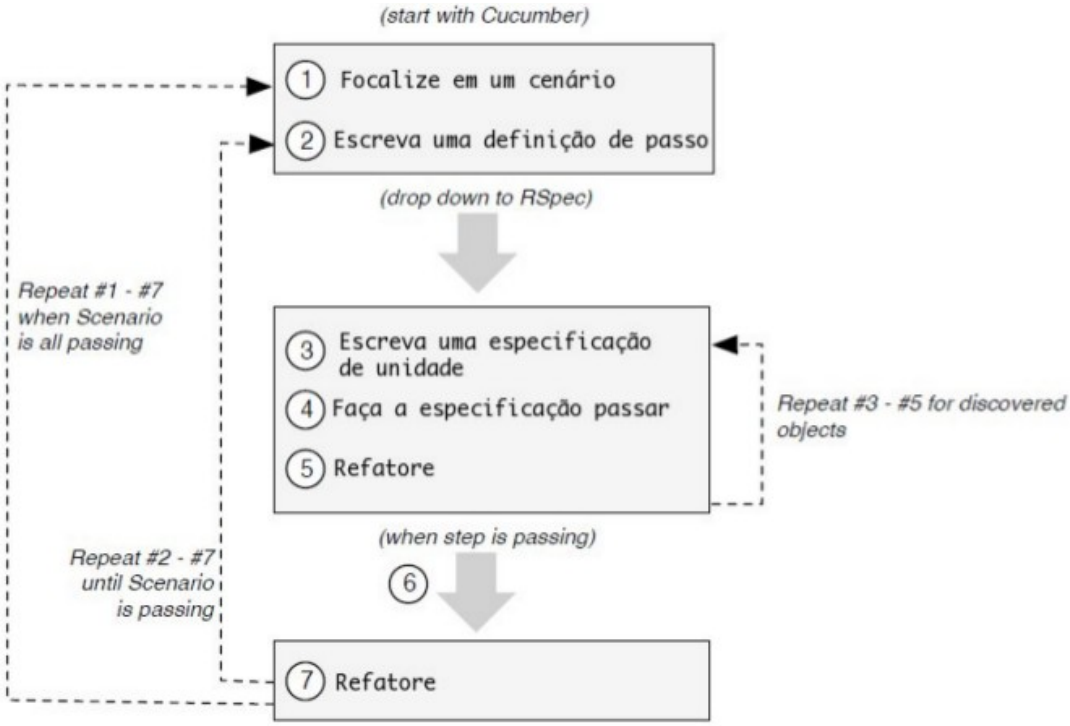


Figura 7: Ciclo Outside-In (MANHÃES, 2010)

3.4.8. BLDD

Carvalho e colaboradores (2010a) desenvolve um trabalho chamado *Business* Language Driven *Development*, que visa dar suporte ao levantamento de valor junto ao cliente. Segundo estes autores o cliente sabe bem quais seus problemas, entretanto pode não ser o melhor especialista sobre as causas destes problemas. O cliente pode precisar de um consultor externo para definir as reais causas de inoperâncias internas em sua organização, ou mesmo sugerir as melhorias para um processo interno.

Os métodos ágeis consideram que o cliente entrega o valor de negócio perfeito para a solução de seus problemas, entretanto o cliente nem sempre conhece perfeitamente o problema que tem.

Além disso, seguindo o caminho da técnica de BDD (NORTH, 2006), na automatização da validação de estórias, o NSI está desenvolvendo uma camada de abstração superior ao BDD para que se possa mudar a linguagem utilizada por outras linguagens, inclusive gráficas. A ideia é propiciar uma discussão de mais alto nível na modelagem de valor, viabilizando maior facilidade na exploração dos valores reais desejados.

Esta camada superior agregará não só uma linguagem plugável para definição de valor, mas também um ambiente de navegação gráfica que propicia andar pelo fluxo de trabalho do desenvolvimento de *Software* e executar o *Software* e os testes relacionados.

3.4.9. *Lean Software Development* (LD)

Esta técnica, considerada um método ágil, foi desenvolvida como uma adaptação dos conceitos do STP para o processo de desenvolvimento de *Software* (POPPENDIECK e POPPENDIECK, 2003).

O LD prescreve sete princípios para o desenvolvimento de *Software* enxuto, conforme o quadro abaixo (Tabela 6).

Tabela 6: Princípios e Ferramentas do LD.

Princípio
P1-Eliminar desperdícios
P2- Amplificar Aprendizado
P3- Decidir o mais tarde possível
P4- Entregar o mais rápido possível
P5- Dê poder à equipe
P6- Construir com integridade
P7- Ver o todo

P1- Eliminar desperdícios

Desperdícios são qualquer atividade que não agrega o valor desejado pelo cliente (OHNO, 1997). No LD considera-se sete desperdícios que devem ser eliminados no desenvolvimento de *Software*, vistos a seguir.

- D1- Trabalho Parcialmente Executado

Qualquer processamento de informações ou requisitos que não seja integrado, executado pelo cliente, validado e aceito é *Software* parcialmente executado.

No LD, se considera que *Software* tende a tornar-se obsoleto rapidamente, sendo qualquer trabalho não terminado um risco de obsolescência.

O trabalho só é considera pronto (Done) quando entra em produção para o cliente.

- D2- Processos Extra

LD considera a documentação um processo desnecessário, um desperdício, que tende a ficar defasado e obsoleto pela evolução do *Software*, gerando retrabalho.

A solução oferecida é omitir a documentação e esperar se alguém sente falta do documento, para então eliminá-lo.

Documentos aceitáveis são desejados pelo cliente, pequenos, de alto nível e produzidos depois do processo.

- D3- Funcionalidades Extra

Funcionalidades não solicitadas pelo cliente, são consideradas como desperdícios. Durante o processo de desenvolvimento podem ser produzidas por sobra de tempo ou por interpretação incorreta do valor desejado pelo cliente.

- D4- Chaveamento de Tarefas

Quando um desenvolvedor é associada à vários projetos, considera-se que ele estará chaveando entre tarefas distintas. Como consequência as tarefas irão demorar mais do que o necessário.

- D5- Espera

Existem várias esperas conhecidas, por exemplo no início do projeto, reuniões, documentação, revisões, aprovações, testes, e para entrega de *Software*.

O objetivo do desenvolvimento é que o cliente veja o *Software* o mais rápido possível.

- D6- Movimento

Dificuldade para resolver problemas técnicos durante o desenvolvimento, Demora para tirar dúvidas com o cliente, facilidade de acesso aos resultados dos testes, são situações que envolvem movimentação dos desenvolvedores. Para viabilizar a concentração dos desenvolvedores deve-se minimizar a movimentação física.

Deve-se diminuir também a movimentação de artefatos entre desenvolvedores.

- D7- Defeitos

Defeitos causam perda de tempo e geram impactos. Entretanto descobrir defeitos imediatamente não gera problemas, enquanto descobrir tardiamente gera problemas maiores.

Portanto deve-se testar rapidamente, integrar constantemente e entregar sempre que possível.

F2- Mapeamento da Cadeia de Valor

Para descobrir os desperdícios no processo de desenvolvimento de *Software* é aconselhado o mapeamento da cadeia de valor.

Para tanto se recomenda trabalhar junto com as pessoas que realizam as operações e anotar o que fazem e o tempo que demoram esperando e realizando as operações.

Dois exemplos são oferecidos, representando um processo tradicional (Figura 8)

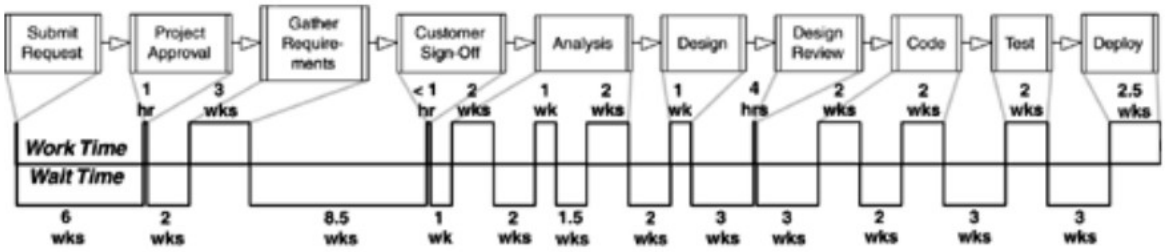


Figura 8: Mapa da cadeia de valor tradicional (Poppendieck & Poppendieck, 2003)

e o mapeamento da cadeia de valor de um processo ágil (Figura 9).

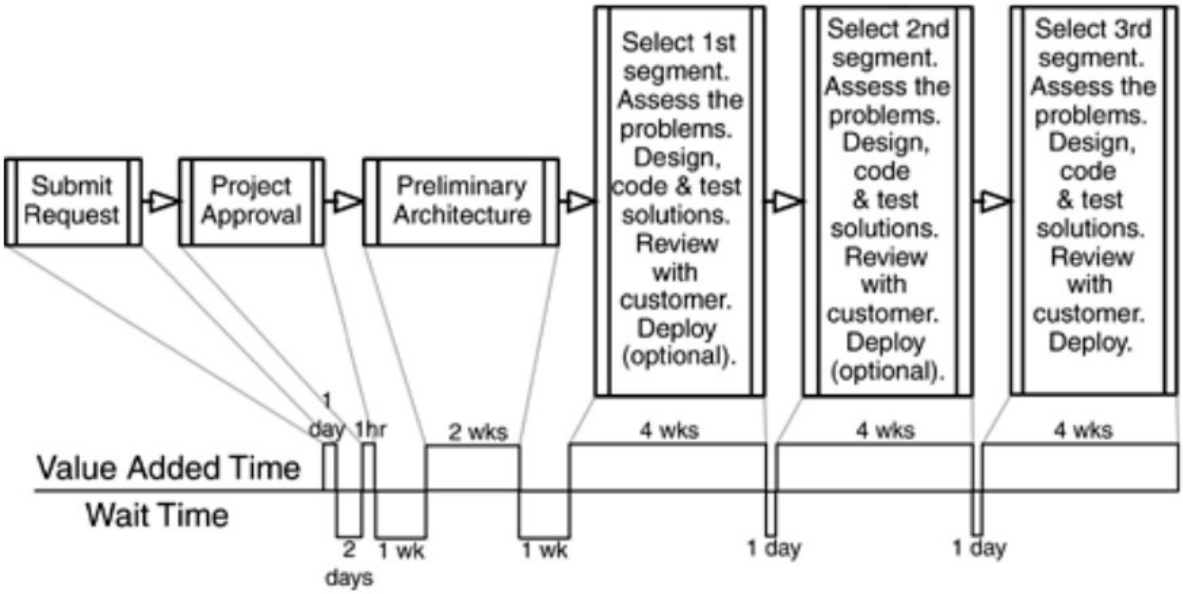


Figura 9: Mapa da cadeia de valor ágil (Poppendieck & Poppendieck, 2003)

P2- Amplificar Aprendizado

Neste princípio o LD indica que se deve aprender a aprender junto ao cliente. É enfatizada a necessidade de desenvolvimento iterativo e incremental. Desenvolver certo de uma só vez pode implicar em desenvolver errado. Desenvolver de forma crescente, gerando resultados e obtendo *feedback* do cliente minimiza o risco e aumenta a qualidade.

Desta forma são planejadas opções que podem atender aos requisitos. Ampliar a lista de opções é fácil, e descartar opções é mais barato do que mudar um *Software* que não esteja preparado para opções.

P3- Decidir o mais tarde possível

O LD sugere que as decisões de projeto devem ser tomadas o mais tarde possível. O objetivo desta técnica é acumular informações até que seja obrigatória tomar uma decisão. Para tanto não se deve planejar detalhadamente antes da hora. O planejamento detalhado deve ser imediatamente antes da ação.

Deve-se entretanto, trabalhar de forma concorrente, materializando o *Software* de forma incremental e o mais rápido possível, visando o *feedback* do cliente, que trará luz às dúvidas existentes.

P4- Entregar o mair cedo possível

O quanto mais rápido um *Software* for entregue, menor o risco de mudanças e menor o trabalho para concertar erros encontrados. Além disso, sob o ponto de vista do cliente, o Retorno sobre Investimento ocorre quando o *Software* começa a gerar resultados para o cliente.

Cabe ressaltar que, como no processo de desenvolvimento de *Software* é necessário aprender sobre o domínio do problema, este aprendizado sobre diferentes realidades pode gerar falhas de interpretação. Considera-se portanto que seja um risco a interpretação incorreta de pressupostos durante o desenvolvimento de *Software*. Entregar *Software* rápido, porém com qualidade, resolve definitivamente os pressupostos e gera o *feedback* do cliente sobre a consistência do trabalho.

P5- Dar poder à equipe

No modelo de trabalho do STP os trabalhadores recebem autonomia para decisões técnicas, assumindo parte das responsabilidades gerenciais. Neste modelo o trabalhador é valorizado e incentivado a conhecer cada vez mais sobre o seu trabalho, tornando-se o melhor naquilo que faz. O trabalhador no STP deve também se preocupar com seus clientes internos e externos, trabalhando de forma a agregar valor sob o ponto de vista destes clientes.

No processo de desenvolvimento de *Software* o desenvolvedor também deve assumir responsabilidade por tomar decisões técnicas, aumentando a qualidade do produto e do processo, e tornando o fluxo de trabalho mais rápido.

P6- Construir com integridade

O *Software* só faz sentido se agrega valor ao cliente. Portanto, a identificação do valor desejado pelo cliente é essencial para o processo de desenvolvimento de *Software*. Este conjunto de valores desejados faz sentido para o cliente dentro de um domínio de problema onde as funcionalidades interagem, os dados colaboram e o cliente sente-se confortável.

Atualmente entretanto, as técnicas de TDD, BDD e Integração Contínua (BECK, 2002) monitoram o código e dão segurança aos desenvolvedores, para que estes efeitos colaterais sejam alertados a todos os desenvolvedores no exato momento que eles ocorram. Alertar imediatamente evita que o *backflow* e retrabalho, de um erro detectado em uma operação futura do processo, ou mesmo venha a causar prejuízos ao cliente.

P7- Ver o todo

Pensar no desempenho e otimizar operações locais não garante o desempenho do processo. É necessário estabelecer políticas conscientes para a melhoria do processo, visando eliminar impedimentos ao crescimento, conforme o padrão de limite para o crescimento do Pensamento Sistêmico (SENGE, 2002).

Como otimização local não garante ótimos globais, métricas locais não são bons indicadores do desempenho do processo. Por exemplo a Linha de produção da Ford trabalhava em fluxo contínuo de forma ótima, entretanto trabalhava para gerar estoques que levaram a empresa Ford rapidamente a escassez de recursos financeiros frente a concorrência contra a GM. Em um processo produtivo, uma máquina muito rápida pode sobrecarregar o fluxo de produtos e tornar lentas as máquinas à montante.

Portanto, métricas para um processo enxuto devem observar o desempenho do processo e a satisfação do cliente. Exemplo de métricas são: *Lead Time* do processo, falhas descobertas depois do desenvolvimento, desempenho do processo, aprovação do cliente, tempo entre falhas, entre outras.

3.4.10. *Kanban*

A técnica de *Kanban*, descrita por David Anderson (2010) tem como objetivo criar um processo de desenvolvimento que seja enxuto, escalável para um grande número de desenvolvedores e estórias, estável e previsível.

Segundo Anderson a utilização de fichas *Kanban* para controlar estórias limita a quantidade de trabalho em processo, pois só se deve lidar com uma quantidade de trabalho finita, que seja compatível com a capacidade de trabalho instalada na equipe. Volume de trabalho acima da capacidade da equipe leva a diminuição do fluxo, conforme a lei de Little.

Anderson sugere os seguintes objetivos:

. Ritmo estável de desenvolvimento ;

. Gerenciar mudanças em fluxo puxado ;

. Controle da Variabilidade .

3.4.10.1. Descrição

- O ritmo de desenvolvimento

Anderson propõe o estabelecimento de um processo de trabalho no qual o trabalho flua, em ritmo constante, tornando-se previsível e estável.

Desta forma o ritmo de desenvolvimento torna-se sustentável e gera confiança nos clientes e na equipe, tornando o processo agradável para todos.

- Gerenciamento de mudanças através do fluxo puxado

Sendo o processo orientado a estórias pequenas e em fluxo, os clientes podem sugerir mudanças em estórias coesas, pois as consequências permanecerão na escala de trabalho de estórias, sendo portanto gerenciáveis.

Neste processo, as operações são puxadas pela demanda do cliente e pelas operações adjacentes no fluxo de trabalho, sendo enfatizado do fluxo de trabalho.

- Controle de Variabilidade

A variabilidade, segundo Anderson, implica em ciclos de trabalho maiores e menor qualidade dentro do processo. Processos maduros precisam controlar as fontes de variabilidade para diminuir o desperdício de trabalho e ganhar estabilidade.

As propriedades de um sistema *Kanban*:

. Gerenciamento visual do fluxo de trabalho ;

. Limitar o trabalho em processo (WIP);

. Mensurar e gerenciar fluxo ;

. Tornar claras as regras do processo ;

. Usar modelos para reconhecer oportunidades de melhorias .

- Gerenciamento visual

Conforme apresentado no tópico 2.3.3, a ferramenta *Kanban* do STP provê uma visão do trabalho em processo e do estado da produção, possibilitando a visualização tanto do fluxo normal de trabalho quanto a identificação de gargalos.

- Limitar o trabalho em processo (WIP)

A limitação de trabalho em processo serve para otimizar o fluxo de trabalho no quadro *Kanban*. Quando a quantidade de trabalho em processos estiver acima da capacidade de trabalho, segundo a lei de Little, a velocidade de fluxo deverá diminuir. Este efeito é sentido na prática, e vem acompanhado de baixa na qualidade de trabalho, falhas em produção, demoras, entre outros problemas.

Muito trabalho em processo torna o *Lead Time* maior, enquanto pouco trabalho em processo faz com que se termine muito cedo, gerando muita folga.

- Mensurar e gerenciar fluxo

A folga de trabalho, chamada de *Slack*, pode ser uma ótima ferramenta para otimizar o processo se bem configurada. Claramente folga demais torna o fluxo ineficiente, entretanto um pouco de folga propicia integração entre os desenvolvedores. Os desenvolvedores precisam também de tempo para se organizarem, aliviar a tensão e pensar. A equipe precisa de tempo para realizar o trabalho com precisão e atenção. Segundo Anderson, um pouco de folga é essencial para a melhoria contínua.

Segundo Anderson, o gerenciamento do fluxo não depende da priorização do trabalho, sendo esta na verdade um desperdício do processo, pois perde-se tempo realizando estatísticas desnecessárias. Entretanto pode-se pensar que o uso de ferramentas de *Kanban* e controle de processo possam automatizar este processo fornecendo estas informações para os desenvolvedores automaticamente, gerando um *feedback* gerencial tanto para os desenvolvedores quanto para os clientes.

- Tornar claras as regras do processo

Em um processo com gerenciamento funcional, baseado em comando e controle, várias regras são impostas ao gerente, ao processo e aos desenvolvedores. Entretanto estas regras não são nem ótimas nem são verificadas quanto a seu impacto no processo. Ocorre frequentemente que estas regras perdem-se em um emaranhado de políticas e perdem mesmo a possibilidade de contribuição para a qual elas tenham sido originalmente concebidas.

Na implantação de um método *Kanban*, conforme preconizado por Anderson, deve-se buscar cancelar ao máximo possível estas regras, e consolidar as regras que restarem em políticas claramente declaradas e estabelecidas no mural do quadro de gerenciamento visual, junto ao *Kanban*, discutindo com a equipe sobre o porque e os objetivos a serem alcançados com cada uma delas.

- Usar modelos para reconhecer oportunidades de melhorias .

Operações padronizadas, conforme o STP as chama, são modelos que visam atingir uma qualidade de trabalho mínima. Entretanto, conforme já discutido, os modelos estabelecidos não são estáticos, devem servir como Benchmark para a proposição e substituição por modelos melhor elaborados, propostos e aceitos por todos.

As melhorias propostas, entretanto, precisam ser medidas no processo e verificadas quanto a sua real contribuição.

3.4.10.3. Métricas

Ao executar um processo é necessário saber, por exemplo, o estado do processo, se o processo e estável, se é predizível, se o fluxo está otimizado, se os desenvolvedores estão produzindo com qualidade esperada. Todas estas questões precisam ser inferidas através de dados do processo.

- WIP

Para avaliar o funcionamento do *Kanban*, Anderson aconselha manter atualizado um gráfico de Fluxo Acumulativo (Figura 10)

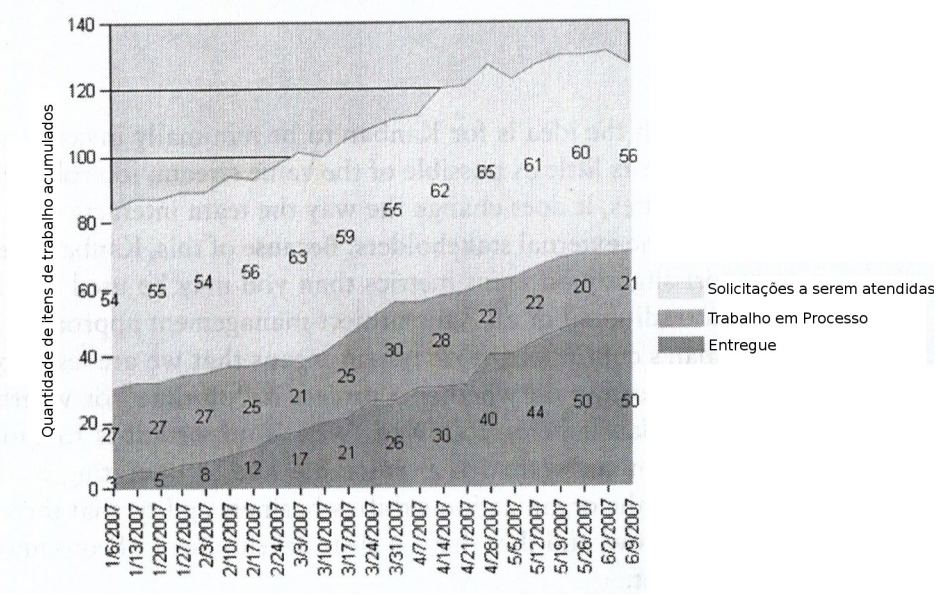


Figura 10: Gráfico de fluxo acumulativo (ANDERSON, 2010)

A partir deste gráfico pode-se perceber áreas cujos volumes representam cada a carga de operação do processo ao longo do tempo. As operações estão apresentadas na legenda, referem-se à itens de trabalho solicitados, trabalho em processo e trabalhos entregues, neste caso representa um quadro *Kanban* bastante simples, com somente estas três colunas.

A quantidade de trabalho em processo pode ser medida no eixo vertical, em cada operação estão inscritos acima das datas, dentro das áreas. O *Lead Time* pode ser estimado ao percorrer uma reta na horizontal.

Pode-se portanto acompanhar a variação de WIP por operação, visando estimar futuras operações e avaliar a estabilidade atual.

- Lead Time

Uma das variáveis controladas é o *Lead Time*, tempo de porta a porta. Através do *Lead Time* pode-se dizer se o sistema está atendendo aos clientes de forma estável. Entretanto é necessário tratar estas variáveis ao longo do tempo, para entender seu comportamento a sua variação. Deve-se buscar evidências de que o processo esteja melhorando continuamente.

A variação em *Lead Time* define a previsibilidade da empresa entregar aos clientes. Caso se trabalhe com classes de serviço ou operações prioritárias, estas deverão ser monitoradas separadamente, sob pena de distorcerem as estatísticas globais.

Pode-se para esta avaliação produzir um gráfico da variação de *Lead Time* em torno da média e avaliar os outliers do *Lead Time*.

- Desempenho do processo (*Throughput*)

O número de itens que sai por tipo e por medida de tempo. Anderson aconselha a informar um número real, que possibilite informar claramente tanto aos desenvolvedores quanto à clientes. Segundo Anderson o desempenho no *Kanban* não deve ser usado para estimar entregas, para isso deve-se usar o *Lead Time*. O desempenho deve ser observado e monitorado para indicar a estabilidade e quanto tem se feito de melhoria no processo.

- Impedimentos e Itens bloqueados

Um item bloqueado é quando não se pode passar para a próxima coluna por já estar no limite do WIP. Os impedimentos representam problemas em uma operação que impedem de continuar, ficando portanto o operador parado esperando ajuda.

A quantidade destes evento traduz a estabilidade do processo e é portanto importante ser monitorado. Um gráfico pode ser traçado e espera-se que a ocorrência tenda a diminuir em um processo que evolui.

- Eficiência do Fluxo

Este indicador é a razão do tempo bloqueado sobre o tempo efetivamente trabalhado. A forma de acompanhar este indicador é medindo e anotando sempre que algum impedimento ocorrer, quanto tempo ele durou. Estes eventos também precisam ser minimizados.

- Qualidade inicial

Quantas falhas são detectados após a operação onde foram criadas ainda dentro do processo. Este problema, que deve causar a parada do processo visando sua prevenção, deve ser monitorado e deve ocorrer cada vez menos. Esforços deve ser feitos usando meios de prevenção como o *Jidoka*, já visto no tópico 2.3.4.

- Quantidade de Falhas

Quantos erros são solicitadas correção por falhas encontradas em produção. Este é um dos mais importantes números para detectar a qualidade no processo. Todos os esforços devem ser feitos para impedir que estes eventos ocorram, pois pode causar danos ao cliente, prejuízos financeiros ou até de vidas humanas.

3.4.11. *Scrum*ban

*Scrum*ban é uma contração entre as palavras *Scrum* e *Kanban*, representando uma adaptação proposta por Corey Ladas (2008), visando otimizar o processo ágil *Scrum* com os princípios do pensamento enxuto e sua visão de Engenharia de *Software*.

Ladas entende os conceitos de *Lean Software Engineering* sejam mais amplos e capazes de responder as seguintes perguntas:

. O que seria um fluxo contínuo de desenvolvimento de funcionalidades?

. O que deveria ser feito e em que ordem?

. O que é um fluxo ideal?

. Qual é o melhor fluxo dada uma certa capacidade?

. Como melhorar na direção do fluxo ideal?

A natureza do *Kanban* não são as cartas, são os limites impostos à quantidade de trabalho em processo, visando estabilizar o fluxo, eliminar desperdício, otimizar o processo.

Em uma implementação *Kanban*, deve haver um *Workflow* padrão que sera incorporado, depois otimizado continuamente. Portanto Ladas incorporou o *Workflow* do *Scrum* e começou a aplicar os conceitos do Pensamento Enxuto neste *Workflow*.

Entretanto, no workflow proposto pelo Scrum não oferece estados compatíveis com tarefas do desenvolvimento, sendo possível apenas observar se uma estória está ou não em desenvolvimento. As operações que uma determinada equipe realizem não são transparentes para quem observe o quadro, que deveir proporcionar o gerenciamento visual do fluxo de trabalho. Impedimentos que aconteçam durante alguma etapa também não são visíveis, muito menos em que operação tenham ocorrido.

Como solução Ladas sugere a criação de colunas que representem operações do fluxo de trabalho da equipe, prevendo que a partir do gerenciamento visual a própria equipe estabeleça um aprendizado sobre as consequências da arquitetura de seu processo, possibilitando assim a o monitoramento de gargalos de produto e a melhoria contínua da arquitetura do processo.

Com consequência dos impedimentos, o fluxo de trabalho terá sua mobilidade limitada, chegando à um ponto em que pare completamente. Os desenvolvedores

devem prestar atenção ao quadro *Kanban* e assim que um impedimento apareça o primeiro desenvolvedor livre deve ajudar a resolver o problema prioritariamente, antes de se comprometer com outra estória.

Após resolver o problema, deve-se buscar identificar uma forma de prevenir que o problema volte a acontecer, melhorando assim o processo.

Como foi implementada uma especialização, e portanto um *Hand-Off* entre desenvolvedores, que antes não existia, faz-se necessário definir regras claras para a comunicação entre os desenvolvedores especializados. Estas regras compõe padrões operacionais que devem ser definidos pelo consenso entre desenvolvedores. Estes padrões operacionais não são estáticos, devem ser mantidos enquanto houver consenso, e devem evoluir em operações *Kaizen*.

Enquanto o foco do *Scrum* é no gráfico de *Burn-Down*, o foco do *Scrum*ban passa a ser o *Cycle Time*, que representa neste caso o tempo de engenharia da estória.

Se os WIP estiverem bem configurados, o processo começa encontrar um fluxo natural, e o *Cycle Time* tende a estabilizar. O *Cycle Time* estabilizando o gráfico de *Burn-Down* deixa de ser necessário.

- Diferenças entre Scrum e Kanban

Algumas diferenças entre os métodos podem ser vistas na Tabela 7.

Tabela 7: Diferenças entre Scrum e Kanban

Característica	*Scrum*	*Kanban*
Natureza	Processo	Princípios e Ferramentas
Product Owner	Sim	Não, Cliente
Workflow	Simples, definido	Qualquer um
Prescritividade	Pouca	Menos Prescritivo
Ciclo	Tempo fixado	Tempo livre
Item trabalho	Estória	Valor do Cliente
Orientação	*Software*	Satisfação do Cliente
Controle	Iteração	Sistema Puxado
Quadro Visual	Sem granularidade	Apresenta Processo
Análise	Dentro Desenvolvimento	Operação separada
Kanban Reset	A cada iteração	Não

Fluxo	Começa/Para	Contínuo
Limita WIP	Por Tempo	Por operação

- Natureza

O *Scrum* é um processo enquanto *Kanban* é uma técnica associada à princípios.

- Product Owner

Só existe no processo *Scrum*.

- Workflow

No *Scrum* existe um *Workflow* bem simples, porém definido, enquanto na técnica *Kanban* deverá ser utilizado o *Workflow* preexistente no ambiente onde a técnica for implementada.

- Prescritividade

O processo *Scrum* prescreve papéis como o *Scrum Master*, o *Product Owner*, o *Coach*, e o *Developer*. Na técnica de *Kanban* não há prescrição de papeis, pois não há processo, somente princípios e a ferramenta *Kanban*.

- Ciclo

O ciclo do processo *Scrum* é feito em iterações que podem durar entre uma e quatro semanas, por exemplo. A técnica *Kanban* enxerga o processo e o trabalho em processo. Existem os limites humanos para o trabalho, portanto o desenvolvimento deve acontecer durante o tempo que for necessário. As unidades de trabalho devem ser pequenas, o menor possível buscando a diminuição da complexidade e do *Cycle Time*.

- Item de trabalho

No processo *Scrum* o item de trabalho é chamado de estória e tem como objetivo gerar valor concreto para o cliente, entretanto deve se enquadrar ao tamanho do *Sprint* definido pela equipe, por exemplo em uma semanas. No *Kanban*, um item de trabalho pode ser uma funcionalidade, o tamanho e a complexidade devem ser pequenos, entretanto não são limitados.

- Orientação

Scrum preconiza que a equipe de desenvolvimento seja orientada para o *Software*, pois a equipe técnica deixa o foco no valor para o *Product Owner*,

enquanto a equipe se preocupa com a qualidade interna ou técnica. Na técnica *Kanban* se mantém o foco no valor desejado pelo cliente.

- Controle

Scrum mantém controle sobre cada iteração. As iterações passadas saem do foco gerencial. O *Kanban* não tem janela de tempo, portanto o controle sobre o processo nunca se reinicia, as variáveis de controle do processo continuam influenciando as médias e estatísticas.

- Quadro visual *Kanban*

O quadro *Kanban* do processo *Scrum* apresenta somente uma coluna de produção, não oferecendo visibilidade para o estado atual do processo dentro da produção. No quadro *Kanban* as operações do processo atual são mapeadas para colunas, possibilitando identificar o estado do processo, o fluxo do processo e sua evolução contínua.

- Análise

No processo *Scrum* a operação de análise é feita pelo mesmo desenvolvedor que faz a implementação cancelando o *Hand-Off*, o que aumenta a qualidade do processo.

No técnica de *Kanban*, a análise e codificação podem ser feitas pelo mesmo desenvolvedor, entretanto considera-se que a especialização na função traga vantagens para a melhoria da operação.

Cabe lembrar que a técnica de *Kanban* não prescreve um processo. O processo natural de cada organização será otimizado pela técnica *Kanban*, que oferece condições de monitorar e gerenciar o fluxo de trabalho apresentando as deficiências e possibilitando o Benchmark do processo e das melhorias implementadas.

- Kanban Reset

No processo *Scrum* o quadro *Kanban* é reiniciado a cada iteração, enquanto na técnica *Kanban*, os itens de trabalho saem individualmente somente algum tempo depois de estarem em produção no cliente. Os desenvolvedores devem considerar um período de tempo para monitoramento dos processos que entraram em produção, visando alguma manutenção necessária.

- Fluxo

O processo *Scrum* não apresenta fluxo, pois as operações entram em execução sem etapas descritas e terminam. A técnica *Kanban* apresenta fluxo entre operações, com influência direta de uma operação sobre as outras.

- Limitação do WIP

No processo *Scrum* o trabalho é limitado pelo tempo de um *Sprint*, enquanto na técnica *Kanban* se recomenda que cada desenvolvedor somente trabalhe um item de trabalho de cada vez. Cada operação do processo também deve ser configurada para receber um número máximo de operações dentro do processo ao mesmo tempo. O processo como um todo também é limitado para um estoque interno máximo.

4. Mapeamento do Pensamento Enxuto para o Desenvolvimento de *Software*.

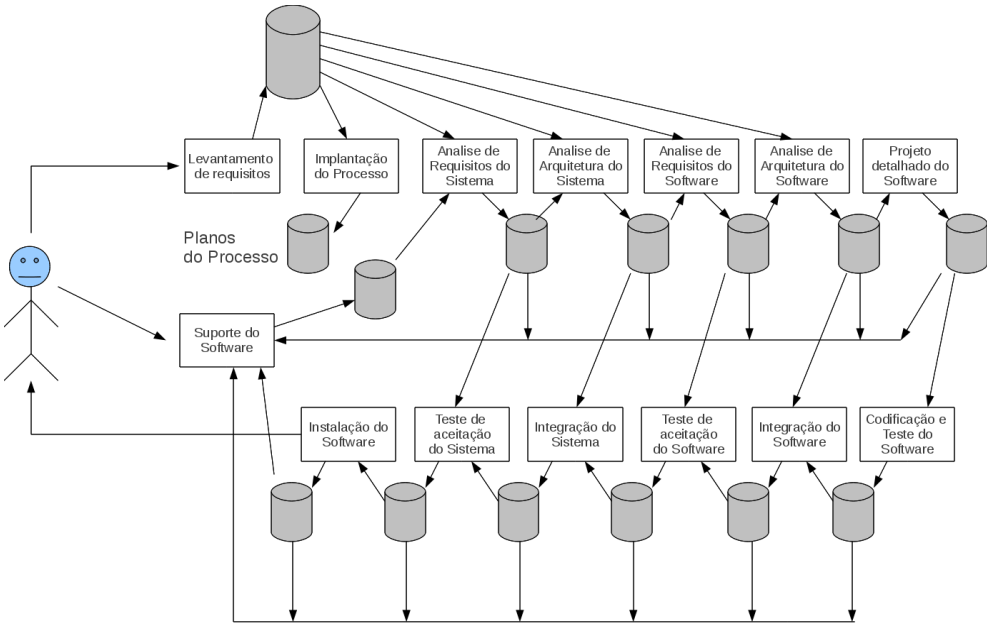
Este capítulo visa trazer como contribuição o mapeamento do pensamento enxuto, das práticas e ferramentas do STP para melhorar o processo de desenvolvimento de *Software*.

Dentre os setores produtivos, os bens intangíveis como *Software* são os de maior peso na balança comercial dos países desenvolvidos, portanto a especificação de processos coerentes e sua otimização baseada em princípios estudados pela Engenharia de Produção, que domina esta área de conhecimento, são uma importante contribuição para a ciência e para a sociedade.

No contexto da gestão de produção são administrados recursos de transformação para gerar produtos e/ou serviços, por exemplo os processos utilizados para produção de bens intangíveis como o *Software*.

4.1. Exemplificação do processo tradicional

Para embasar a discussão abaixo, é necessário deixar claro o que se entende pelo processo tradicional. Será apresentado um exemplo gráfico e discutidos os aspectos deste processo, que será referenciado ao longo do texto (Figura 11).



ISO/IEC 12207 (Processo de Desenvolvimento) Obs.: foram omitidas 4 revisões e 2 auditorias

Figura 11: Processo tradicional

Conforme discutido no tópico 3.2, os métodos tradicionais, aqui representados pelo processo de desenvolvimento da norma ISO/IEC 12207, tem como princípio o acoplamento funcional, onde se agrupam atividades semelhantes em lotes, provavelmente visando a economia de escala, conforme preconizado por Ford na produção em massa.

Para tornar mais claro o funcionamento dos processos tradicionais, cabe esclarecer que é inicialmente feito um contato com o cliente no qual se identificam os requisitos e funcionalidades desejadas pelo cliente. Dentro do processo estes requisitos são processados em lotes por cada operação dentro do processo em um arranjo funcional. A cada operação as informações são semi processadas em lotes, não sendo terminadas, e entregues para a fase subsequente (clientes internos) somente ao terminar o processamento do lote, visando dar continuidade no processamento ainda inacabado.

O cliente externo somente recebe o *Software* ao final do processo, após todos o processamento ter acabado, todas as verificações feitas, quando emite *feedback* sobre sua satisfação e gerando novas demandas de adaptação.

Dentre outros assuntos será discutida a implantação de lotes unitários para a eliminação de estoques e a melhoria contínua, através da autonomia dos desenvolvedores e do gerenciamento visual.

Um exemplo do mesmo processo em lote unitário pode ser visto na Figura 12.

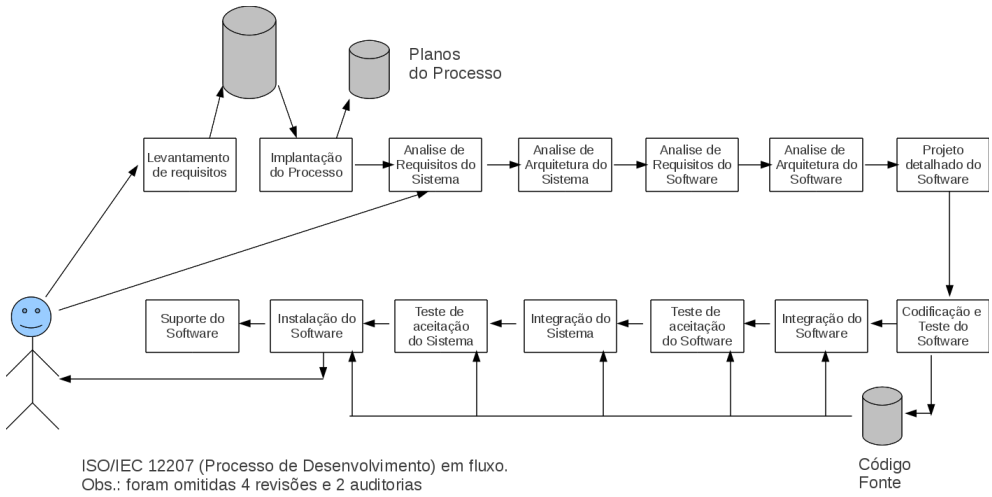


Figura 12: Alterações no processo tradicional para implementar lote unitário.

Nesta adaptação do processo, os requisitos são levantados e priorizados, causando que estes sejam desenvolvidos individualmente em fluxo contínuo pelos desenvolvedores livres.

4.2. Histórico de Sistemas de Produção

O processo produtivo do desenvolvimento de *Software* conforme descrito nos métodos tradicionais é um processo de produção que entrega valor no final da produção. Segundo o pensamento enxuto este processo pode ser considerado como uma prestação de serviço, com vários ciclos de produção e entregas de valor durante o processo.

Segundo o pensamento enxuto a investigação dos valores do cliente a cada ciclo de produção leva o cliente a participar do desenvolvimento, aprendendo e emitindo *feedback* a cada ciclo, a partir de produtos prontos que resolvam algum problema real. Ocorre portanto neste processo um consumo imediato do serviço, diferentemente do processo tradicional no qual o contato entre o consumidor e o processo produtivo se da na solicitação e na entrega somente.

Para tanto são necessários trabalhadores capacitados em planejamento, análise, projeto, desenvolvimento e testes de *Software* entre outras habilidades. Segundo o Pensamento Enxuto estes trabalhadores precisam ser multifuncionais, pois devem se preocupar com o impacto do que fazem em todo o processo, principalmente com a operação seguinte no fluxo.

Nos métodos tradicionais se considera a utilização de profissionais especializados em uma única tarefa, visando atingir a otimização do processo a partir da otimização de cada operação do processo individualmente.

Segundo métodos tradicionais o *Software* será entregue ao cliente ou patrocinador no final do processo. No pensamento enxuto o *Software* será apresentado a cada ciclo de produção, e entregue em períodos acordados, chamados entregas de *Release*s, sendo que cada ciclo de produção deve entregar uma funcionalidade pronta para ser testada pelo cliente, e que possa ser utilizado em produção caso necessário.

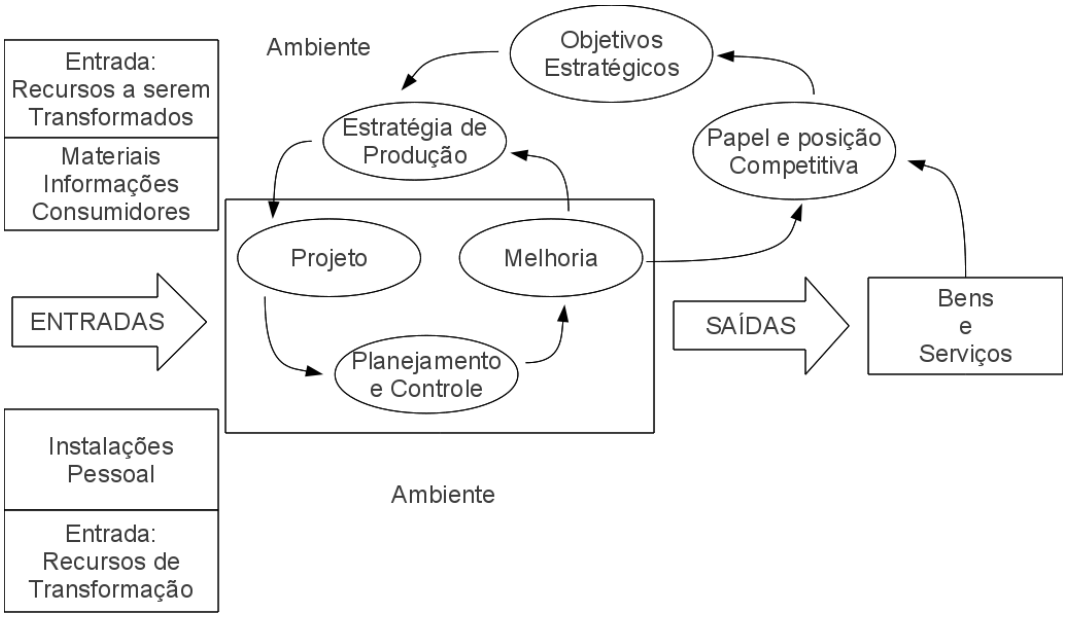


Figura 13: Modelo do Processo de Produção (adaptado de Slack et al., 1999)

Portanto um sistema de produção que representa o desenvolvimento de *Software* segundo *Slack et al.* (1999) pode ser visto na Figura 13.

A seguir serão discutidos os principais sistemas produtivos conhecidos e a relação do desenvolvimento de *Software* com estas formas de produção.

4.2.1. Produção Artesanal

O desenvolvimento de *Software* gera uma oportunidade única para desenvolvedores talentosos. Ocorre que os desafios de desenvolver *Software* provocam o ânimo de profissionais de grande capacidade, levando-os a escrever *Software* segundo sua percepção de qualidade técnica.

Sob um ponto de vista de engenharia, um processo produtivo precisa apresentar repetibilidade. Em busca da repetibilidade do processo, alguns gestores e engenheiros de *Software* acabam por coibir a criatividade de desenvolvedores.

Apesar desta constatação, na indústria em geral se busca meios de incentivar a criatividade e inovação, porque a criatividade pode render vantagens competitivas, lucros e até novos negócios.

Segundo Sommerville (2003), antes do termo engenharia de *Software* surgir em 1968 a produção de *Software* era artesanal.

Larman e Basili (2003) citam que antes de 1970, o processo de desenvolvimento de *Software* apesar de ser pouco documentado academicamente era iterativo e incremental e era passado por tradição oral entre os desenvolvedores

de *Software*, até que Royce (1970) publica a ideia de que se deveria fazer o ciclo de desenvolvimento de *Software* hoje conhecido como Cascata (*Waterfall*).

Pfleeger (2004) considera o desenvolvimento de *Software* uma atividade artesanal por depender da criatividade humana.

A produção de *Software* artesanal é portanto mais flexível, sendo mais indicada para problemas pequenos de alta complexidade.

Middleton e Sutton (2006) considera que a produção artesanal não é adequada para o desenvolvimento de *Software* porque os produtos tendem a evoluir e necessitar de manutenção tanto corretiva quanto evolutiva, sendo necessária a utilização de técnicas arquiteturais avançadas que possibilitem a evolução do código.

Entretanto, cabe ressaltar que dependendo da empresa, do problema, da tecnologia utilizada e da expectativa de vida do *Software*, cabe o bom senso de utilizar técnicas que empreendam mais ou menos cuidados, recursos, custos e prazo, para a produção do *Software*.

4.2.2. Partes intercambiáveis

No desenvolvimento de *Software* existe a noção prática e teórica da necessidade de facilidade na interligação das partes. Inicialmente o esforço de codificação, mesmo para desenvolvedores mais experientes, pode gerar códigos monolíticos.

Um sistema monolítico é dito quando este é único, contendo várias partes inclusive fracamente relacionadas. Segundo Pressman (2006) no final da década de 60 os desenvolvedores perceberam que havia vantagens em dividir o código em módulos mais coesos. Em seguida, se percebe as desvantagens do acoplamento ou dependências entre os módulos. O próximo passo foi identificar objetos que escondem seus dados.

Neste ponto se entendeu que a comunicação entre módulos poderia ser feita através de interfaces bem definidas.

Atualmente utiliza-se também soluções arquiteturais como arquitetura orientada a serviços, arquitetura orientada a componentes que possibilitam integrar sistemas mesmo utilizando linguagens de programação diferentes.

Para a construção do código existem inúmeras soluções técnicas que possibilitam simular módulos inexistentes, normalmente conhecidas como Dubles de código (PINHEIRO & REZENDE, 2009), que possibilitam o trabalho de desenvolvimento desde as condições iniciais até a manutenção evolutiva do *Software*.

Atualmente, no paradigma ágil de desenvolvimento no qual se entrega *Software* funcionando para o cliente a cada semana, todos estes conceitos são necessários em conjunto e são aplicados extremamente. Segundo Beck (1999), se arquitetura é importante para a qualidade do *Software* deve-se utilizar estes recursos extremamente.

Ao praticar métodos ágeis o próprio processo de desenvolvimento de *Software* que leva o *Software* a emergir das necessidades do usuário, depende de arquitetura extremamente modular e expansiva, sem o que o *Software* não poderia crescer.

Praticas como refatoração, na qual se pretende melhorar um código que já atinga os resultados esperados, também depende de uma infraestrutura que possibilite manter a intercambialidade de peças, sendo este portanto um conceito conhecido apesar de pouco utilizado.

4.2.3. Produção em Massa

Software pode ser produzido e entregue após a produção como um bem, em processos similares ao da produção em massa.

Segundo a Engenharia de *Software*, na maioria dos processo documentados até então, os requisitos identificados servem de base para o planejamento da produção de *Software*. A partir destes requisitos o gerente de desenvolvimento estima os recursos necessários e o tempo de desenvolvimento para cada requisito, confeccionando um plano contendo prazo e custo para todas as tarefas de todas as fases necessárias ao desenvolvimento.

Os processos de desenvolvimento tradicionais se apoiam na necessidade de uma estrutura organizacional grande o suficiente para controlar problemas complexos, de alto risco, com grande volume de requisitos, que deverão ser tratados com ampla documentação, vários processos de fiscalização em lotes de trabalho grandes, por trabalhadores com fronteiras funcionais bem definidas.

Este plano, caso aceito, será implementado em várias fases de produção. Cada fase deverá processar um grande lote com todos os requisitos, gerando resultados semi-processados para a fase seguinte. Após a produção do *Software* este produto será entregue ao cliente ou patrocinador do desenvolvimento.

Cada uma destas fases trata todos os requisitos ou funcionalidades, normalmente recebendo lotes de informações semi-processadas das fases anteriores. Para realizar estas operações, os desenvolvedores no processo tradicional são normalmente organizados em departamentos funcionais, descrevendo uma clara organização vertical.

Outro aspecto que contribui para esta avaliação é o planejamento de produção detalhado, que submete tantos os desenvolvedores quanto os clientes em ter limitadas as possibilidades de mudança no escopo do produto final que será entregue. Ao falhar este planejamento são impactados prazo e custo, toda a produção precisa acelerar e perde-se a qualidade no produto. Desta forma o processo prefere limitar ou impedir quaisquer mudanças de escopo.

4.2.4. Taylorismo (Administração Científica)

Segundo Poppendieck e Poppendieck (2011), a definição de tarefas especializadas fazia sentido para que se pudesse acelerar o processo, mas também para substituir pessoas com facilidade. No desenvolvimento de *Software* tradicional, ao desconsiderar a capacidade criativa dos desenvolvedores, busca-se um processo onde possa-se substituir um desenvolvedor por outro mais barato, ou assim que necessário.

Portanto, as relações de trabalho segundo o pensamento enxuto visam treinar e capacitar cada vez mais os desenvolvedores, para que estes possam ser os melhores especialistas no que fazem, diferentemente do preconizado por Taylor e de práticas baseadas no comando e controle, em gerenciamento funcional vertical e cobrança do trabalhador, cobrança de desempenho por tempo ao invés de por qualidade, ainda hoje tão comuns.

Jim Highsmith (2002), discute que a estrutura de trabalho no desenvolvimento de *Software* tradicional segue os princípios da administração científica, com especialização de trabalho, controle em diversos níveis, gestão funcional e organização vertical.

Os problemas deste tipo de gestão, conforme já discutidos, são vários, diminuindo a criatividade, o bem estar do trabalhador, focando os problemas no trabalhador ao invés do processo, entre outros.

Como no STP as equipes de trabalhadores são semiautônomas, assumindo responsabilidades gerenciais e conhecimento multifuncional, assim também deveriam ser as equipes de trabalho em um processo enxuto. No STP equipes recebem autonomia para gerar comprometimento, criatividade, inovação, e desempenho na manufatura. Equipes que trabalham em métodos ágeis aplicam este mesmo princípio.

4.2.5. MRP – Produção Empurrada

Atualmente os processos de desenvolvimento de *Software* tradicionais incorrem nos problemas citados por Womack e Jones (1998), relacionados ao uso do MRP. O planejamento no início do processo de desenvolvimento de *Software* propõe uma previsão de todas as atividades e seu tempo de execução durante todo o processo. Entretanto, o desenvolvimento de *Software* é composto por diversas atividades intelectuais, ocorrendo desde o aprendizado de novos conceitos acerca do domínio do problema até a pesquisa exploratória sobre soluções de melhor qualidade e desempenho. Como consequência este planejamento normalmente e naturalmente falha, tendo como consequência as enormes taxas de falhas e comprometimento de projetos de *Software* já citadas no tópico 3.2.4.

Poppendieck e Poppendieck (2003) cita que o uso de cronogramas tendem a falhar, sendo mais adequado a utilização de processos.

Entende-se que processos são ortogonais a cronogramas, pois um cronograma estipula atividades encadeadas no tempo como se fossem atividades únicas, enquanto um processo percebe o fluxo de operações e busca avaliar o tempo de cada repetição do processo.

O planejamento de tarefas no tempo, para desenvolvimento de *Software* força a execução de cada tarefa dentro de janelas de tempos que podem ser irreais, forçando a qualidade para níveis mais baixos.

Enquanto o atraso de uma operação em um cronograma provoca o atraso das operações seguintes, o atraso em um processo normalmente pode ser compensado pela variação de outras operações.

A solução encontrada pelo pensamento enxuto é a produção em lotes unitários e o trabalho sobre demanda (*Just-in-Time*). Neste sistema considera-se que priorizando os critérios de qualidade, os prazos das operações sejam tão enxutos quanto possível, e planeja-se usando o *Cycle Time* do processamento completo, que possibilita margens de erros proporcionalmente menores que a margem de erro de cada operação.

4.3. Pensamento Enxuto

O Pensamento Enxuto, conforme descrito por Womack e Jones (1998), cria um modelo conceitual sobre o que ocorreu no STP.

O trabalhador neste contexto assume um sentimento de equipe, na qual apesar de conhecer profundamente, melhor do que o normal do mercado, suas funções, ele também precisa conhecer o fluxo de trabalho como um todo.

No pensamento enxuto trabalha-se em fluxo. A gestão é horizontal, não funcional. Quando ocorre algum gargalo no fluxo de trabalho os outros trabalhadores podem fazer um enxame para ajudar o trabalhador sobrecarregado. Em casso mais graves o processo deve ser rebalanceado, conforme proposta similar da TOC (GOLDRATT e COX, 2002).

O conhecimento dos trabalhadores deve ser repassado aos outros membros da equipe, corroborando com o trabalho em pares dos métodos ágeis (BECK, 2004).

A participação do cliente no processo de desenvolvimento provoca o seu aprendizado. Conforme o cliente aprende ele pode sugerir mudanças do planejamento mais frequentes, o que atualiza o valor do produto.

Com a evolução das técnicas, tornou-se possível usar os conceitos de peças intercambiáveis para que o *Software* possa sofrer mais mudanças impactando menos o processo. O método ágeis consideram portanto o objetivo principal do processo de desenvolvimento a satisfação do cliente ou patrocinador, portanto possibilitam que o cliente ou um representante atue e tenha um papel importante dentro da equipe durante o desenvolvimento.

O pensamento enxuto sugere que o trabalho em lotes unitários possibilita o planejamento individualizado no último momento responsivo, mantendo abertas as possibilidades até o momento em que realmente se precisa planejar para executá-lo.

O planejamento no último momento, para uma unidade única de trabalho garante a menor complexidade do planejamento, diminuindo assim o risco. Garante também que a quantidade de informações disponíveis para avaliar o problema seja máxima, em contraste com o planejamento inicial dos métodos tradicionais.

Para executar um lote unitário também surge a necessidade real de priorização, já que não será processado um lote inteiro antes da entrega.

Conforme já descrito, são propostos 5 passos para otimização de um processo de trabalho, seja este de serviços ou manufatura, visando executar somente operações que gerem o valor demandados pelo cliente ou patrocinador, sem interrupções ou desperdícios.

A aplicação destes princípios no processo de desenvolvimento de *Software* pode ser implementada em uma mudança radical (*Kaikaku*), ou em mudanças suaves incrementais, através do aprendizado e melhorias constantes (*Kaizen*).

Portanto, cabe esclarecer que o pensamento enxuto pode começar com a situação atual de qualquer processo, seja ele ágil, ou mesmo um processo tradicional de desenvolvimento de *Software*, para que ao longo do estudo dos princípios enxutos se possa evoluir, gerando maior desempenho, menor custo, mais satisfação para o cliente e mais estabilidade ao processo.

Primeiro é necessário se discutir os desperdícios durante o desenvolvimento de *Software*.

4.3.1. Eliminação de desperdícios

São analisados os desperdícios na manufatura, em serviços, em *Software*, a compilação destes princípios e comentários a seguir. O objetivo da eliminação de desperdícios é melhorar o processo continuamente, visando que se estabeleça um fluxo contínuo de objetos de trabalho pequenos e simples, de forma a manter o processo estável, predizível e melhorando continuamente.

Excesso de produção (Lotes não unitários e produção para estoque)

A superprodução nasce do medo de não ter recursos no futuro, da falta de tempo para realizar o trabalho, e da falta de controle dos erros.

A experiência dos gerentes de desenvolvimento de *Software* indica que é melhor sobrar tempo do que faltar. Como os contratos são fixados por escopo x tempo x custo, ocorrem os seguintes fenômenos:

. Cobrança de prazo sobre cada atividade ;

. Repreende-se o trabalhador pelos atrasos do projeto ;

. Ao faltar tempo, diminui-se a qualidade .

A superprodução na manufatura é considerada como a produção dissociada da demanda real. No desenvolvimento de *Software* a quantidade de trabalho simultânea. No desenvolvimento de *Software* a superprodução pode ser vista como produzir o que não se tem certeza se será necessário, adicionando funcionalidades ainda não solicitadas, estimando que serão necessárias.

Ocorre também a execução de tarefas não programadas e desnecessárias, muito comum em projetos, sendo citada por Goldratt (1998) como Síndrome de Parkinson, descrita no tópico 2.6.3.

A superprodução no desenvolvimento de *Software* pode indicar um contato inadequado com o cliente. É comum o contato com o cliente somente no início do processo, e pouco ou insuficiente durante o processo, permitindo a produção de funcionalidades desnecessárias.

O fato mais importante com relação a este desperdício é compreender o valor de uma boa arquitetura. A arquitetura do *Software* deve permitir que a qualquer momento possa-se adicionar novas funcionalidades, portanto deve-se esperar até o último momento possível para adicionar funcionalidades, até se ter certeza absoluta da real necessidade, solicitada pelo cliente.

A insegurança em mudar o projeto durante o andamento leva os processos tradicionais à planejar tudo que for possível, mesmo o não solicitado, desde o início do desenvolvimento, criando um planejamento de atividades granular de longo prazo que fatalmente não se concretiza.

Espera ou atraso

As esperas durante o processo de desenvolvimento tradicionais são comuns, toleradas e até planejadas. Ocorre na verdade em diversos formas de serviço que as esperas são toleradas com naturalidade.

Entretanto, como o pensamento enxuto não produz em lotes, produz unitários, qualquer desperdício de tempo é sentido pelo cliente e pelo sistema. Pode-se esperar e tolerar demoras quando se esconde atrás de uma produção de batelada, pois o tempo total é muito maior. Entretanto quando se produz em lotes unitários este tempo previsto de resposta não suporta esperas e demoras.

Tais esperas também representam desperdícios de mão de obra, de estoque, de recursos apropriados para a execução da tarefa que está aguardando.

O tempo esperando faz com que o desenvolvedor fique impaciente, perca o foco sobre o trabalho, queira começar a fazer outra tarefa, e possa esquecer o contexto mental do trabalho que ainda não se tornou disponível.

Atrasos são considerados em processo em fluxo como anormalidade no processo, e precisam ser tratados como tal, investigados e eliminados.

Outras fontes conhecidas de atrasos são atividades desnecessárias como burocracia excessiva, documentação não desejada pelo cliente, reuniões demoradas, por exemplo. Em cada caso, deve-se perguntar que atividades realmente contribuem para gerar o valor que é pago pelo patrocinador. Caso o patrocinador não veja valor na execução da atividade, esta será uma forte candidata a ser eliminada.

Processo Gordo (Retrabalho e atividades desnecessárias)

Este desperdício ocorre quando ocorrem erros, que são resolvidos através de retrabalho, ou quando ocorre um *Hand-Off*, que implica em reaprender sobre um domínio que alguém já estudou e conhece bem.

O retrabalho é um desperdício notável e amplamente discutido, talvez o mais conhecido. No desenvolvimento de *Software*, pode ocorrer retrabalho desnecessário pela falta de contato com os especialistas do domínio do problema a ser desenvolvido. Por isso os métodos ágeis aconselha a inclusão do cliente durante o desenvolvimento da funcionalidade. Para cada ciclo semanal necessário para o desenvolvimento de uma funcionalidade, aconselha-se a solicitação de um especialista que deverá esclarecer aos desenvolvedores, aprender sobre as possibilidades de informatização, ajudar a definir a solução, os testes e aceitar ou não o resultado atingido.

Outra forma de desperdício é o retrabalho gerado pela ocorrência de erros. Após detectados erros serão encaminhados para desenvolvedores recuperar o contexto mental daquele *Software* específico, daquela funcionalidade para que este possa investigar as causas do erro, busque soluções, implemente tais soluções e teste sua nova solução.

Royce (1970), em seu trabalho sobre processo para o desenvolvimento de *Software*, faz uma declaração fundamental, corroborando o pensamento de Ohno, afirmando que os passos que realmente agregam valor para o produto sob o ponto de vista do usuário, são a Análise e a Codificação.

Entretanto, cada processo é único, diante dos recursos disponíveis, e dos requisitos apresentados pelo cliente, devendo ser reconhecidas que algumas operações, apesar de não agregar valor diretamente, podem ser necessárias ao processo, consideradas como dependências transitórias do processo, e que deverão ser otimizadas em versões futuras do processo.

Estoque de Funcionalidades Inacabadas

Os estoque no desenvolvimento de *Software* são derivados do objeto de trabalho deste processo. O processo de desenvolvimento de *Software* recebe como entrada informações e conhecimentos passados por especialistas, visando prestar um serviço de aprendizado e manufatura de um produto que automatize a solução dos problemas.

Segundo Poppendieck & Poppendieck (2003), estoques em *Software* são trabalhos inacabados. O objetivo do desenvolvimento de *Software* enxuto deve ser começar o desenvolvimento de uma única funcionalidade que agregue valor para o cliente, executar cada uma das etapas somente desta funcionalidade, testar, submeter ao aceite do cliente, integrar ao *Software* e entregá-la pronta, sem esperas ou demoras durante o processo, em fluxo contínuo.

Desta forma espera-se conseguir reduzir a quantidade de rascunhos, estoques de conhecimento, reaprendizado, perda de informações durante o desenvolvimento.

Pode-se observar que alguns documentos são necessários para o cliente, para que outras equipes consigam manter o *Software*, normalmente para o futuro após o término do desenvolvimento. Entretanto outros documentos são utilizados

como formas de comunicação visando a passagem de bastão durante o desenvolvimento. Estes documentos e a passagem de bastão em si devem ser eliminados, pois são um fonte de risco, mal-entendidos e perda de informação durante este processo.

Ohno (1997), afirma que estoques de material semi-processado encobrem defeitos e inadequações nos produtos, que somente serão detectadas no momento em que estes produtos forem novamente movimentadas para sua utilização, o que ocorre exatamente no processo de desenvolvimento tradicional ao aguardar após o término de todo o desenvolvimento para realizar-se testes.

Movimentação do desenvolvedor

A movimentação em manufatura representa se esforçar para chegar a algum lugar, ou buscar alguma peça ou ferramenta.

A movimentação no processo de desenvolvimento de *Software* é considerado como o chaveamento de contexto. Quando um desenvolvedor começa a trabalhar em uma funcionalidade e, antes de ter a funcionalidade pronta passa para outra funcionalidade ele precisa escrever em algum rascunho ou documento o rascunho do que já foi feito e do que falta ser feito. Neste momento ele perde tempo salvando seu contexto mental para que em um momento futuro ele ou outro desenvolvedor possa recuperar o contexto mental daquela funcionalidade já trabalhada, porém inacabada.

Goldratt (1998), em sua teoria para o gerenciamento de projetos, a Corrente Crítica, já citava os problemas advindos do chaveamento de contexto como :

- Diminuir a produtividade ;
- Aumentando seriamente o Risco de falhas ;
- Diminuindo a qualidade do trabalho.

O chaveamento de contexto estará presente sempre que houver um *Hand-Off*, ou passagem de bastão. O desenvolvedor de *Software* quando começa a pensar em uma nova funcionalidade precisa estudá-la para carregar seu contexto mental, ambientando-se com as informações do novo domínio ou contexto.

Para assumir uma tarefa inacabada o desenvolvedor precisa encontrar as informações básicas daquele contexto, a arquitetura da solução, as últimas mudanças realizadas naquele *Software*. As informações recuperadas sobre a nova

tarefa podem não ser completas, aumentando o risco de que a atividade não seja concluída ou de que tenha menor qualidade. O tempo para recuperar o contexto também soma como um acréscimo ao tempo total de trabalho, conforme pode ser observado no exemplo da Figura 14.

Na indústria isto se chama tempo de *Setup*, ou preparação para o processamento. Este tempo precisa ser evitado ou diminuído ao máximo. Sendo contra producente diminuir o tempo de estudo para o desenvolvimento, é mais coerente deixar que uma mesma pessoa possa lidar com o problema do início ao fim.

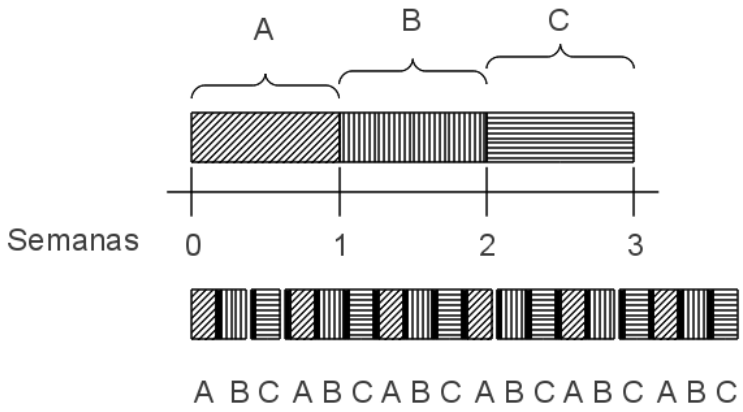


Figura 14: Influência do tempo de preparo no chaveamento de tarefas

Conforme pode ser visto na Figura 14, o tempo de preparo sempre causa impacto no tempo total de execução das tarefas, forçando neste exemplo 3 tarefas A, B e C que poderiam ser executadas em três semanas a ultrapassar este prazo.

Defeitos

Segundo Taiichi Ohno (1997) os defeitos devem ser prevenidos, pois são uma das maiores formas de desperdício e refletem deficiências do processo. A solução segundo o autor é a utilização dos conceitos de *Jidoka* em especial o *Poke-Yoke*.

As perdas em produção ocasionadas por falhas em *Software* podem colocar em risco a sobrevivência de empresas (CHARETTE, 2005). O autor cita o trabalho de Schulmeyer (1990), que trata do estudo de *Poke-Yoke* para a prevenção e análise do contexto dos erros no desenvolvimento de *Software*.

Danovaro e colaboradores (2008) também trata da autonomação no contexto de desenvolvimento de *Software*. O autor relata suas experiências empíricas e analisa resultados do uso de uma ferramenta para controle de testes de unidade e da técnica de TDD (BECK, 2002).

Beck (2002) propõe que o processo de codificação seja precedido pela especificação de testes em código fontes, visando monitorar intensivamente o código fonte e alertar ao desenvolvedor sobre erros e anormalidades, corroborando exatamente com Ohno e seus dispositivos *Poke-Yoke*.

North (2006) propõe uma expansão da técnica de TDD, visando não somente o teste, mas automatizar a validação das estórias escritas nos processo ágeis. De fato, a técnica de TDD oferece segurança com relação a eficiência de se atingir os resultados previstos pelo programador e elimina erros de construção do projeto. Entretanto a ferramenta de North busca validar se o *Software* atende aos valores especificados pelo cliente, sendo portanto uma ferramenta de monitoramento da qualidade externa do *Software*.

Erros de programação ocorrem em circunstâncias normais, por distração humana, ou por interferências internas ou externas (GOBBO e VACCARI, 2008), ou ainda por entendimento incorreto de alguma informação necessária ao desenvolvimento.

Para viabilizar as melhorias do *Software*, o crescimento incremental e a qualidade interna do *Software*, é aconselhável a utilização de técnicas *Poke-Yoke*, que visam monitorar o *Software* verificando anormalidades e alertando instantaneamente o desenvolvedor. Atualmente são disponíveis técnicas como DbC (MEYER, 1992), TDD (BECK, 2002), BDD (NORTH, 2006) entre outras, amplamente utilizadas nos métodos ágeis.

Segundo Poppendieck & Poppendieck (2011) deve-se definir semanalmente alguns dos desenvolvedores para serem responsáveis pela manutenção de problemas do *Software*, em um rodízio semanal. Como consequências os desenvolvedores conseguem manter o conhecimento sobre o *Software* após a manutenção, e se preocupam em não deixar erros porque elas terão que ser responsáveis pelos erros posteriormente.

Transporte

O transporte no STP significa a movimentação de mercadorias a longas distâncias, normalmente por estradas, navios ou aviões. Faz sentido pensar que o transporte do objeto de trabalho por longas distâncias é um desperdício, desde que Kiichiro Toyoda disse que um processo produtivo eficiente deveria ter todos os insumos e ferramentas próximas a mão (OHNO, 1997).

No caso do desenvolvimento de *Software*, estima-se que o transporte de informações já aprendidas por alguém para outra pessoa seja um transporte caro, que implica em perda de tempo e risco de perda de sentido, de semântica e de conteúdo. Takeushi e Nonaka (1986) falam sobre a dificuldade em transmitir e assimilar conhecimento tácito que é difícil de ser formalizado, transmitido, expresso, capturado, sendo necessária uma vivenciação para seu entendimento.

Quanto mais transporte o conhecimento sofre, maiores as chances de degradação do conhecimento. Logicamente, deve haver o repasse dos conhecimentos, entretanto para desenvolver *Software* considera-se que o risco aumenta na medida em que as pessoas se distanciam.

Idealmente uma pessoa entender um problema pequeno e desenvolver a solução. Caso seja necessário repassar o conhecimento, o ideal é o contato direto, usando gráficos, exemplos concretos, buscando facilitar e otimizar a troca de conhecimentos.

Neste caso, segundo o pensamento enxuto, ocorre o risco do *Hand-Off*. O repasse de conhecimentos para terceiros visando o desenvolvimento é considerado um risco. Repassar tais conhecimentos em um lote de requisitos aumenta ainda mais a complexidade do problema e diminui o foco.

Segundo o pensamento enxuto, deve-se identificar funcionalidades pequenas o suficiente para que um mesmo desenvolvedor possa responsabilizar-se pelas tarefas de análise, desenvolvimento de testes automatizados, projeto da arquitetura, desenvolvimento do código, sem precisar repassar para terceiros durante este processo.

Caso a funcionalidade seja complexa e envolva mais especialistas, estes deve trabalhar em conjunto e focados em uma única funcionalidade, visando não criar mais complexidade do que o necessário e com isso criar o maior foco de atenção possível no problema em pauta.

Finalmente existe também um tempo necessário para outro desenvolvedor entender o domínio do problema no qual está entrando. Este tempo de *Setup* impacta o processo como um desperdício originado pelo transporte de informações.

Mais uma vez a chave para a diminuição do risco, da complexidade, do tempo de *Cycle Time*, e aumento da velocidade, do foco e da qualidade, sugerido pelo pensamento enxuto é a diminuição do tamanho do lote, se possível para lote unitário, e do escopo do trabalho, trabalhando com estórias tão pequenas quanto possível, em fluxo contínuo e sob demanda.

Não ouvir o empregado

Considerando que na indústria manufatureira o respeito ao trabalhador já considera um risco e um desperdício não ouvir ao empregado, no desenvolvimento de *Software* considera-se também importante receber preparar o processo para receber as contribuições de quem realmente agrega valor para o cliente. As considerações que quem está programando podem trazer a luz da realidade às estimativas, aos planejamentos que falham e a otimização do fluxo do processo.

O caminho para obter inovação, desempenho e comprometimento dos elementos da equipe é a comunicação associada ao poder de cooperação e intervenção dos desenvolvedores no processo do qual fazem parte.

4.3.2. Cinco passos do Pensamento Enxuto

Identificação de Valor

No desenvolvimento de *Software* tradicional a divisão do processo em fases suscita a criação de um estoque dos requisitos elencados para serem operados a cada fase do desenvolvimento. O processamento em lotes, proposto pelo planejamento inicial, define fases que deverão processar todos os requisitos. Em um desenvolvimento estruturado, que entrega um produto ao final do processo não faz diferença quando um dado problema será resolvido, porque o produto será único somente no final do processo, forçando que o cliente a espere até o final quando recebe um produto que contempla todos os resultados desejados de uma só vez.

Um cliente que demanda por um serviço de desenvolvimento de *Software* pode não saber o que deseja por não conhecer que recursos computacionais e

soluções técnicas estão à sua disposição para resolver seus problemas, tudo o que o cliente conhece é o seu problema. Ao ser apresentado às soluções computacionais o cliente pode aprender e imaginar modificações que aumentem ainda mais o valor apresentado, mas isso é um processo de aprendizado e exploração, que traz resultados através da iteração entre o cliente e o desenvolvedor.

Ao mesmo tempo, as necessidades dos clientes são fruto de sua relação com o mercado. Como o mercado é dinâmico as necessidades podem e devem ser instáveis, causando mudanças nos requisitos levantados inicialmente.

Portanto, durante o desenvolvimento de *Software* o *feedback* do cliente sobre o *Software* é um fator essencial para reduzir o risco de erros nos requisitos, aumentando a qualidade externa e a certeza sobre a aderência do *Software* às necessidades do cliente. Desta forma, trabalhar próximo ao cliente é uma estratégia que precisa ser levada em consideração, buscando maximizar o *feedback* durante o desenvolvimento.

Por vezes, para identificar o valor desejado pelo cliente no desenvolvimento de *Software*, pode ser necessário também o mapeamento da cadeia de valor do cliente, na busca pelo melhor entendimento do contexto do valor desejado. Deve-se entender a cadeia de agregação de valor do cliente para entender como o valor a ser criado será utilizado, qual a participação do valor criado na cadeia de valor, quais as características mais importantes do valor esperado pelo cliente.

Apesar de buscar identificar um estoque de valores a serem desenvolvidos no início do processo, esta lista visa a priorização dos valores mais importantes para o cliente, para que desta forma se diminua o risco de insucesso do projeto através da geração priorizada dos valores mais importantes e que agreguem mais valor desde o início do projeto.

O valor no desenvolvimento de *Software* precisa ser algo pequeno, que gere retorno ao cliente e que possa ser desenvolvimento em curto prazo. A objetivo é de desenvolver sempre partes pequenas que possam ser avaliadas pelo cliente, gerando *feedback* e diminuindo o risco de mal entendidos quanto ao valor esperado pelo cliente.

O processo de desenvolvimento de *Software* deve gerar como valor as funcionalidades desejadas pelo cliente. O pensamento enxuto, ao propor lotes

unitários, visa gerar a cada ciclo do processo uma funcionalidade útil para o cliente. A integração destas atividades deve ocorrer em tempos planejados junto com o cliente, entretanto a cada trabalho realizado deve culminar com uma nova funcionalidade pronta para uso pelo cliente. Dividir o desenvolvimento de *Software* em pequenas partes ajuda a diminuir o risco nos projetos pelo aumento do *feedback* entre do cliente para o desenvolvedor.

A avaliação do produto pelo cliente aumenta a familiaridade do cliente com a solução proposta pelo desenvolvedor, possibilitando a confirmação ou o replanejamento da solução encontrada.

Poppendieck e Poppendieck (2001) comparam esta política sugerida com o planejamento realizado precocemente, comum em projetos de desenvolvimento de *Software*, onde considera-se que mudanças podem atrapalhar o projeto.

Cabe observar que mudanças necessárias, não detectadas ou evitadas, aumentam o risco de não atingir o valor esperado pelo cliente, gerando insatisfação ao receber um produto incompatível com suas necessidades.

Uma funcionalidade ou valor desejado pelo cliente poderá ser dividido em uma ou mais estórias. Cada estória é definida com um critério de aceitação que especifique objetivos práticos a serem atingidos ao final do desenvolvimento da estória.

O planejamento do desenvolvimento é confexionado de forma que possa reunir estórias em um período de desenvolvimento pequeno, entre uma semana e um mês, chamado de iteração.

Uma funcionalidade deve ser pequena o suficiente para que utilize o mínimo possível de iterações. As estórias, de forma semelhante, devem ser planejadas para que caibam dentro de uma iteração. É recomendado que cada iteração tenha um tema que proporcione o foco ao desenvolvimento dentro da iteração.

Ao terminar a implementação das estórias que compõe a funcionalidade, o cliente poderá aceitar a funcionalidade desenvolvida, ou apontar as mudanças e melhorias necessárias.

Portanto, a iteração pode ser considerada como uma janela de tempo, que deverá ser grande o suficiente para conter um conjunto de pequenos valores que suportem atingir ao valor esperado pelo cliente.

Entretanto, pode-se considerar a lista de Funcionalidades de produtos como um desperdício, por ser um estoque. De fato, o preestabelecimento de funcionalidades, com profunda análise na forma de BDUF (big-design-up-front) constitui um trabalho antes do necessário, podendo ser necessário retrabalho no momento de sua real execução.

Idealmente as estórias deveriam ser desenvolvidas por demanda, sempre que um cliente tiver a necessidade deverá postar as suas necessidades re-priorizando as que ainda não foram executadas. O acúmulo de demandas distorce o *Lead Time* do sistema, forçando um ritmo não natural.

Identificar valor pode envolver mais do que ouvir a solicitação do cliente. Existem casos em que o próprio cliente não tem condição de diagnosticar qual o real problema que está afetando sua empresa. Neste caso o cliente deverá solicitar uma consultoria que faça um diagnóstico e avalie os processo da empresa visando desenhar uma ferramenta adequada para corrigir tais problemas.

Neste caso Carvalho e colaboradores (2010a) entende que a ferramenta BDD poderia apresentar uma interface que possibilitasse outras formas de especificação de estórias, como por exemplo gráficos de máquina de estados, que são utilizados em modelagem de processos.

Ocorre entretanto que o processo tradicional de análise de requisitos utiliza princípios bem diferentes do pensamento enxuto, conforme apresentado na Tabela 8.

Tabela 8: Comparação entre Resolução de Valor (EV) e Análise de Requisitos (ES).

Característica	Resolução de Valor	Análise de Requisitos
Foco	Prioridades do Cliente	Funcionalidades do Produto
Expansividade	Linha de Produtos Orientado para o Futuro	Produto Unitário Orientado ao Presente ou Passado
Principais Preocupações	Satisfação do Cliente Aderência ao negócio	Realidade da Organização
Nível de Detalhe	“O que” e “Quando”	“O que”, “Quando”, e “Como”
Outras Preocupações	Sistemático	Esporádico
Envolvimento do Cliente	Central	Periférico
Partes Responsáveis	Desenvolvedores	Grupo de Alto nível
Riqueza de Resultados	Domínio, reconhecidos, não reconhecidos e priorizados.	Normalmente Somente “Reconhecido”

Conforme pode ser observado na Tabela 8, ambas as técnicas são utilizadas para investigar as características do problema do cliente, com vistas ao planejamento e implementação do produto desejado.

Com relação ao foco da atividade, a Resolução de Valor (RV-EV) busca entender as prioridades do cliente, enquanto a Análise de Requisitos (AR-ES) busca a definição de funcionalidade do produto desejado.

- Valor para o negócio do Cliente (*Business Value*)

O desenvolvimento de sistemas de informação é um caso específico no qual observa-se as maiores falhas relatadas na literatura. Nos resultados podem ser observados projetos que foram considerados comprometidos, isto é, que terminaram porém não atingem algumas de suas premissas, como escopo, prazo e valor.

Outra informação apresentada diz respeito a projetos que foram cancelados, pois os interessados perceberam a tempo que o resultado final do projeto não atendia as expectativas.

Portanto, tanto o processo de desenvolvimento de *Software*, quanto o valor esperado pelo cliente tem sido causas frequentes de prejuízos em projetos de desenvolvimento de *Software*.

O Valor esperado pelos interessados no desenvolvimento de *Software* advém do valor que o *Software* pode gerar dentro das empresas. Existe um risco intrínseco e diretamente proporcional à distância entre a visão de valor de negócio e o valor do produto *Software* desenhado.

Entretanto, a especificação de valor tende a ser observada como um processo a ser exaurido no início do processo, tendo em vista utilizar esta visão de valor como uma linha mestre para o planejamento de atividades, custos, recursos humanos, recursos financeiros, aquisições e por último qualidade do projeto de desenvolvimento.

Esta visão já representa e faz parte do processo de desenvolvimento de *Software*.

Segundo esta visão, a identificação completa e exaustiva do valor desejado pelo cliente, pode ser divida em partes e agendada para ser desenvolvida, gerando um projeto perfeitamente condizente com o esperado pelo cliente originalmente.

Um erro comum neste caso é a confusão entre desenvolvimento incremental e desenvolvimento

Entretanto, vários problemas podem ser enumerados diante desta visão, tendo em vista a produção de um sistema com a totalidade dos valores observados:

1. O valor desejado pelo cliente podem se mover durante o desenvolvimento, ao esperar períodos longos de tempo, por exemplo após seis meses, ou 1 ano.

2. A visão de valor observada pelo desenvolvedor pode ser incorreta ou incompleta, sendo portanto base para produção de um produto que não gere real valor para o cliente.

3. A visão do cliente pode estar incorreta, gerando um produto incorreto que somente será descoberto no final do desenvolvimento. Portanto o processo precisa implementar *feedback* intermediários com o qual o produto possa ser validado e modificado iterativamente.

A Cadeia de Agregação de Valor (Value Stream Mapping)

O objetivo deste passo é o melhoramento desta cadeia de operações. Pode-se melhorar a cadeia melhorando operações atuais ou eliminando desperdícios. Portanto é necessário mapear a cadeia de valor atual, visando a análise das operações desta cadeia para identificação e eliminação de desperdícios, e avaliar oportunidades de otimização.

Após mapear a cadeia de valor, busca-se conhecer e caracterizar as operações atuais, anotando seu tempo médio de execução, tempo de espera, quantidade de trabalho em processo média, seu desempenho médio (tempo médio em que entrega saídas), e informações necessárias para a discussão sobre sua importância para o processo e para o consumidor.

Ocorre entretanto que torna-se difícil para desenvolvedores habituados com um processo já estabelecido ir contra paradigmas sedimentados no senso comum. Neste caso, cabe buscar um referencial externo que gere desenvolvimento para a empresa, para os clientes e para os desenvolvedores. O STP adota a satisfação do cliente como medida de qualidade de seus produtos e a Voz do Cliente como a orientação soberana com respeito a satisfação do cliente.

Neste sentido Womack e Jones (1998) descreve que as operações devem ser avalias podem ser:

1. As que agregam valor sob o ponto de vista do consumidor ;
2. As que não agregam valor mas temporariamente não podem ser eliminadas ;
3. As que não agregam valor e devem ser eliminadas .

A análise das operações neste passo do pensamento enxuto deve evidenciar a contribuição de cada operação sobre a ótica do cliente. Deve-se responder a pergunta se o cliente acha sinceramente necessária a operação, e se ele deixaria de pagar por ela.

Pode-se, por exemplo elencar atividades a serem desenvolvidas e perguntar aos clientes o quanto eles gostariam de pagar parar a execução destas atividades. As respostas mais realistas demonstrarão que atividades são essenciais e quais diminuem o desempenho do sistema e poderiam ser automatizadas.

As operações de testes não precisam ser eliminadas totalmente, mas atualmente são disponíveis técnicas que possibilitam a prevenção de erros (*Poke-Yoke*) como o BDD e TDD, já apresentados neste trabalho.

A documentação pode ser automatizada usando ferramentas de engenharia reversa para recuperação de artefatos. Desta forma pode-se eliminar o retrabalho na manutenção de documentos confeccionados prematuramente.

O uso de arquiteturas genéricas, de linguagens de mais nova geração, de frameworks, de reutilização de código, Arquitetura orientada a Serviços, componentização, são exemplos de ferramentas que viabilizam o crescimento incremental da arquitetura de *Software*, possibilitando que o processo responda melhor à evolução e mudanças dos requisitos.

Fluxo de Produção

Neste passo deverão ser avaliadas as operações do mapa de agregação de valor, construído no passo anterior, e avaliar todas as operações com relação ao tempo de espera para seu início e demoras durante a operação. Deve-se eliminar estas esperas e demoras, visando que uma vez o processo inicie ele seja capaz de começar e acabar sem esperas e demoras.

Cabe esclarecer que não se trata de ser o mais rápido possível, mas sim sem desperdícios. A velocidade chegará naturalmente, como consequência da objetividade e simplificação dos objetos de trabalho.

Operações que apresentem espera no mapa de agregação de valor construído no passo anterior, devem ser otimizadas, visando eliminar totalmente as esperas de todas as operações. Atingir fluxo em um processo depende de se remover impedimentos, esperas, demoras, retrabalho, desperdícios de uma forma geral. Um processo no qual uma atividade fica desocupada esperando o resultado de outra atividade pode ser otimizado.

Aguardar pelo documento de requisitos antes de começar a análise. A análise de um requisito pode começar enquanto alguém preenche o documento de requisitos. Segundo o pensamento enxuto não de deve trabalhar o lote inteiro, portanto pode-se separar um ou uma quantidade menor possível, visando começar a análise enquanto outro desenvolvedor documenta os outros requisitos. Esta atividade paralela otimiza o processo, eliminando a espera. Além disso, o uso de documentos somente para comunicação entre elementos da equipe deve ser eliminado pois agrega risco ao processo.

Aguardar todos os módulos serem codificados para depois iniciar os testes não faria sentido. Pode-se iniciar os testes do primeiro módulo enquanto um próximo começa a ser codificado.

Esperar a assinatura de alguém pode demorar. A autonomia controlada neste caso pode servir como dinamizadora do trabalho. No STP, os trabalhadores são considerados autoridades em questões técnicas. Os engenheiros aprendem com os trabalhadores e discutem a viabilidade e riscos com estes especialistas antes de iniciar projetos.

Esperar pela compilação de um sistema é uma atividade que pode ser automatizada, como por exemplo usando a Integração contínua, já descrita no tópico 3.6.3.

A espera em um processo deve ser vista como um gargalo ou uma diminuição na velocidade no fluxo de trabalho. Cabe ressaltar que é ideal que o fluxo de trabalho pare quando for necessário para analisar o processo, identificar melhorias, identificar motivos de falhas, prevenção de erros futuros, mas aqui se discute esperas, e não paradas.

Idealmente o processo deve fluir com cada operação trabalhando em lotes unitários, entregando seu objeto de trabalho para a operação seguinte do *Workflow*

sem esperas, demoras ou impedimentos. A ocorrência de impedimentos deve ser investigada visando sua prevenção.

O fluxo de trabalho irá definir o *Cycle Time* do processo, e portanto está submetido a Lei de Little, conforme já explicado no tópico 2.4. A sobrecarga de trabalho em processo irá diminuir a velocidade do sistema. Isto também precisa ser levado em consideração quando analisar demoras e esperas.

Cada desenvolvedor se tornará responsável pela eliminação das demoras e esperas nas operações nas quais participar. Chefes ou gerentes até podem sugerir mudanças, mas deverão ser autorizadas ou aceitas pelos trabalhadores. Sua real função é oferecer desafios aos trabalhadores, evidenciando oportunidades de melhoria.

Lideres deverão motivar os colegas na buscar e discussão das melhorias no fluxo de trabalho e eliminação de desperdícios, observando o quadro *Kanban* diariamente e colaborando uns com os outros nos empecilhos e dificuldades encontrados.

Visando o fluxo no processo de desenvolvimento de *Software* deve-se mitigar o risco da falta dos desenvolvedores. Desenvolvedores podem ficar doentes, faltar, mudar de emprego, tornando sua ausência um risco para a organização. A prática de pareamento, discutida no tópico 3.6.3 (XP), viabiliza uniformizar e homogeneizar o conhecimento sobre o *Software* e as habilidades na equipe, além de aumentar a qualidade do *Software*, incentivar inovação e diminuir o tempo de engenharia.

Estabelecimento de Demanda (Just-in-Time)

Preparar um processo para ser executado *Just-in-Time* (JIT) implica que só se deve trabalhar diante de uma demanda, que pode ser de clientes internos ou externos do processo. No caso do desenvolvimento de *Software* atender à demanda poderá ser receber uma solicitação do cliente ou perceber uma nova estória tenha sido movimentada para a fila de entrada de sua operação.

Uma demanda do cliente chega na forma de uma solicitação de funcionalidade. As solicitações de funcionalidades deverão ser dividas em estórias, orçadas com relação ao seu esforço de desenvolvimento pela equipe técnica, e por fim, priorizadas pelo cliente.

O fornecimento interno deve prever as necessidades dos clientes internos. Cada desenvolvedor, exercendo sua atividade em uma dada operação do *Workflow*, deve conhecer a rotina de trabalho das outras operações do *Workflow*, principalmente a sua vizinha próxima, visando buscar a otimização dos seus resultados para facilitar o trabalho de seu vizinho.

Para implementar o *Just-in-Time*, pode-se utilizar o quadro *Kanban* para a sinalização do trabalho em processo. Com a utilização do quadro *Kanban* e *Andon*, a equipe passa a compreender melhor o processo e sugerir melhorias.

O objetivo do JIT no desenvolvimento de *Software* é evitar a execução desnecessária de trabalhos não solicitados, e sincronizar o trabalho entre as operações do processo.

Através do JIT, se implementa o sistema de produção puxado, no qual as operações tendem a acelerar e convergir para um ritmo estável. O sistema passa a ser confiável com respeito a sua previsibilidade.

Busca pela perfeição

Este princípio se baseia no fato de que qualquer processo pode melhorar. As operações podem ser automatizadas visando liberar o ser humano de tarefas repetitivas e chamando sua atenção para tarefas que dependam da interpretação e da lógica humana.

Operações que não agreguem valor sob o ponto de vista do cliente devem ser suprimidas.

Os desenvolvedores devem ser estimulados a identificar oportunidades de melhoria no processo, nas técnicas utilizadas para o desenvolvimento de *Software* e na busca por soluções que agreguem realmente valor para o cliente.

O mapa de agregação de valor deve ser constantemente atualizado e avaliado, buscando eliminar as operações que não agregam valor sob o ponto de vista do cliente. Com o tempo, novas formas de otimização tornam-se disponíveis, possibilitando a automatização destas atividades que não agregam valor.

De tempos em tempos, por exemplo semanalmente, deve-se alocar espaço para uma reunião de avaliação dos serviços realizados. As regras do processo precisam ser confrontadas para verificar se realmente ajudam o trabalhador e amplificam o fluxo ou se se burocratizam e impedem o fluxo.

Os procedimentos devem ser padronizados visando estabelecer um desempenho mínimo para o processo, entretanto devem ser revistos periodicamente, visando identificar melhores formas de trabalho, que amplifiquem o valor desejado pelo cliente e aumentem o fluxo de trabalho. Estes objetivos podem ser alcançados usando *Kaizen* suavemente e ao longo do tempo ou *Kaikaku*, reformando operações.

O objetivo do processo é gerar satisfação para o cliente, não fazendo nada que não agregue o valor desejado pelo cliente. Para tanto precisa-se atacar os desperdícios, burocracias e ineficiências continuamente.

4.4. Ferramentas do STP

4.4.1. Lotes unitários

O processo de produção com lotes gera estoques intermediários e com isso cria a espera para a disponibilidade do lote. O lote sendo gerado torna-se disponível somente após o tempo necessário para sua conclusão. Se a produção de um lote de 50 peças demora 50 minutos e uma peça demora 1 minuto para ser feita, na produção em lote haverá o desperdício de 49 minutos para a utilização de uma peça.

Este mesmo problema é abordado por Goldratt (1998) ao discutir a multitarefa nociva, no contexto da corrente crítica, conforme descrito. Segundo Goldratt, na execução de serviços em que prestador de serviço não tem foco em um único serviço, mas dá atenção à mais de um serviço ao mesmo tempo, os clientes de todos os serviços sendo prestados aguardam durante muito mais tempo do que o necessário se houvesse foco em um único serviço.

Percebe-se uma ligação íntima entre o lote unitário do Pensamento Enxuto e a Corrente Critica, que combatem a demora para obtenção de resultados e na baixa qualidade do que é produzido, pela falta de foco.

A implantação do lote unitário implica inevitavelmente na linearização do processo, com a natural eliminação de alguns estoques intermediários (documentos), pois os requisitos trabalhados não precisam esperar para serem processados mais tarde, são processados em fluxo contínuo até se tornarem funcionalidades implementadas e prontas para a produção. Cabe observar que a

comunicação deve ser direta entre os desenvolvedores, evitando a comunicação através de documentos. Documentos devem ser produzidos quando necessário depois que o a funcionalidade ou produto estiver estável.

Um processo dividido em várias operações que trabalham em lotes, deverá consumir muito mais tempo do que o necessário para a produção de lotes unitários. De fato a produção destas mesmas operações gerando lotes unitários elimina os estoques intermediários, e diminui sensivelmente o tempo de produção para uma funcionalidade para o cliente, possibilitando através do foco de atenção do desenvolvedor e portanto maior qualidade no *Software*.

No caso do lote unitário, ocorre também a possibilidade de contínuo alinhamento do processo produtivo, diminuindo gradativamente a utilização de espaços de estocagem (documentos), reforçando a comunicação direta, e suscitando automação das operações, gerando uma economia em movimentações, ao enxugar operações do fluxo. Documentos desejados pelo cliente devem ser produzidos na última operações do fluxo.

Durante a produção em lotes, o *Software* fica parado esperando para ser utilizado pelo cliente, deixando de ser produtivo para o cliente, e acumulando o risco da identificação futura de defeitos, risco de que ele se torne desnecessário.

4.4.2. *Kanban*

Uma vantagem do *Kanban* é que imagens tornam ideias mais claras. Ao ver um quadro *Kanban*, todos entendem instantaneamente o status de cada trabalho, o volume de trabalho em processo. Ao acompanhar um quadro *Kanban* durante o tempo, qualquer pessoa percebe o desempenho das operações.

No desenvolvimento de *Software* este quadro também é usado como uma ferramenta de gerenciamento visual. Ele deverá representar as operações, o fluxo de trabalho, e as estórias sendo processados no mesmo. Neste caso os objetos de trabalho são solicitações que serão transformadas em uma ou mais estórias, que por sua vez serão agendadas para entrega em um *Release*.

O quadro *Kanban*, visa implementar um sistema puxado, no qual se limita o número de estórias que podem ser desenvolvidas ao mesmo tempo. Quanto maior a quantidade de estórias menor a velocidade de fluxo do sistema, maior o estoque de trabalho em processo e maior o risco de perda de foco e qualidade.

Para utilizar o quadro *Kanban*, deve-se mapear as operações que realmente são realizadas no processo atual. A visualização do processo faz com que todos desejem participar de melhorias no processo. Cria uma imagem mental única do processo, aumentando a sinergia da equipe e possibilitando que se discuta em conjunto a evolução do processo.

A imagem das operações no *Kanban* e o fluxo vivo sendo executado torna as ideias mais claras, possibilita entender o fluxo e o comportamento de cada operação. Ao participar de um processo monitorado por um *Kanban*, os desenvolvedores motivam-se para sugerir modificações, percebem oportunidades para melhoria do fluxo e do processo como um todo.

Dependendo da configuração do processo pode ocorrer sobrecarga de operações, também chamados de gargalos. O quadro *Kanban* possibilita a visualização dos gargalos e a configuração de limites do processo para diminuir ou eliminar sua ocorrência.

O quadro *Kanban* pode ser visto como um tabuleiro de um jogo, no qual as configurações podem fazer o fluxo aumentar ou diminuir. O resultado das operações se torna visível possibilitando avaliar se o custo de cada operação é um benefício para o processo.

Cada coluna do quadro deverá ter pessoas associadas, responsáveis pela execução das operações correspondentes. Portanto cada operação poderá atender à um volume de demanda máximo, sendo cada pessoa impedida de puxar mais do que uma estória de cada vez. Para evitar que ocorra sobrecarga de demanda em cada operação utiliza-se um valor máximo de trabalho em processo para cada operação (WIP – Work in Process) .

Este valor deve ser definido de tal forma que não se sobrecarregue os desenvolvedores da operação. A sobrecarga dos operadores faz o fluxo do sistema com um todo diminuir, conforme descrito na lei de Little.

Somente desenvolvedores livres podem puxar operações no quadro *Kanban*, portanto enquanto um desenvolvedor estiver realizando uma tarefa ele não deve se preocupar com outras operações que existam na sua fila de entrada, deve somente focar no trabalho atual e como pode fazê-lo da melhor forma possível, evitando erros e retrabalho futuro.

Cada operação pode ter ou não uma coluna de Fila de Espera ou de Entrada. As colunas de fila também devem ser limitadas, por exemplo com a mesma quantidade de WIP da operação, possibilitando que haja sempre ao menos uma estória a ser processada por cada desenvolvedor da operação.

Ocorre que quando existem vários desenvolvedores trabalhando em uma operação, pode ocorrer falta de estórias a serem processadas. A sobra de tempo se chama *Slack*, e não é necessariamente ruim. Não se deve ter como objetivo manter todos ocupados todo o tempo. Este pensamento de ocupar 100% do tempo de pessoas e máquinas para justificar seus custos é oriundo da Administração Científica e é comprovadamente ineficiente. Não se garante o ótimo global pela soma dos ótimos locais.

Algum tempo de folga de processamento é importante para se poder ajudar à outros desenvolvedores com impedimentos, estudar, relaxamento e o gerenciamento do próprio trabalho que passa a ser responsabilidade do próprio desenvolvedor. O tempo de folga é recomendável, mas deve-se buscar uma configuração adequada em cada equipe.

As estórias no quadro *Kanban* podem ser de diversos tipos, sendo comum utilizar cores para diferenciar entre eles. Normalmente utiliza-se estórias para desenvolvimento de novas funcionalidades, estórias para concertos de erros e estórias urgentes solicitadas pelo cliente. A quantidade de estórias urgentes no quadro *Kanban* deve ser limitada, visando impedir que todas tornem-se urgentes.

Quando ocorre um problema em uma estória, é comum haver uma sinalização do problema no quadro *Kanban*, visando alertar aos outros desenvolvedores que existe um impedimento no trabalho da estória sinalizada.

Uma estória deve ser idealizada com um objetivo bem claro, deve ser tão pequena quanto possível, mas deve ser coesa e independente de outras estórias. Cada estória deve se referenciar à um personagem que seja seu cliente, à um objetivo claramente declarado e verificável, ter todos os seus resultados esperados verificáveis, além de expectativas sobre resultados não desejados também verificáveis. Os objetivos e resultados, desejáveis e indesejáveis, serão monitorados por ferramentas como BDD e TDD.

Outra preocupação do desenvolvedor é saber o que seu cliente interno precisa, para que seu trabalho atenda adequadamente ao seu vizinho no fluxo. Na

verdade todos os desenvolvedores devem se preocupar também com o real valor desejado pelos clientes internos e externos. Isso possibilita a criação de uma simulação mental do processo, o que ajuda a sua otimização.

A diminuição de WIP faz com que a velocidade do processo seja mais constante, mais estável. Quando ocorre um problema em alguma operação, pode implicar na parada do fluxo imediatamente, revelando problemas existentes no processo, exatamente como o barco sobre as rochas. Quando o sistema para, os desenvolvedores tem a oportunidade de praticar um enxame para ajudar o desenvolvedor parado. Este tipo de parada no fluxo pode ser muito bom, sendo altamente recomendado, pois possibilita a investigação dos motivos da parada, visando a melhoria do processo e a prevenção de sua repetição.

O aumento de WIP em filas de entrada para uma operação pode fazer que a velocidade do sistema diminua, conforme pode ser visto na Lei de Little.

O problema de gerenciamento do processo de desenvolvimento de *Software*, quando linearizado pela demanda e pelo fluxo unitário passa a ser um problema de filas, cunha engenharia oferece métricas e soluções como simulação para modelagem, gestão e controle.

4.4.3. Autonomação (*Jidoka*)

O uso de autonomação para o desenvolvimento envolve a liberação do desenvolvedor de tarefas repetitivas, a automação de tarefas necessárias porém que não agregam valor sob o ponto de vista do cliente, e a prevenção de erros.

Ao discutir a cadeia de agregação de valor o objetivo é identificar todas as operações presentes na versão atual sendo executada no processo de desenvolvimento. Dentre as operações encontradas, as operações que não representam valor para o cliente, ou que o cliente preferiria não pagar por elas, mas que a equipe de desenvolvimento achar que são necessárias, são grandes candidatas para a Autonomação.

Outras atividades que não dependam da inteligência de uma pessoa, devem também ser automatizadas, preocupando-se em transferir a inteligência humana para o processo, contabilizando as ocorrências e chamando atenção dos desenvolvedores quando for necessário.

Casos comuns são a recuperação de diagramas a partir de código fonte, TDD e BDD, integração contínua, rastreamento de requisitos, *Kanban* automatizado e *Andon*.

A recuperação de documentos a partir de código fonte, também chamada de Engenharia Reversa é necessária porque, apesar da modelagem ser uma necessidade indiscutível para o desenvolvimento de *Software*, a documentação antes do produto se tornar estável é uma fonte de retrabalho certa. Documentar o *Software* antes do mesmo dele estar pronto envolve sempre diversos retrabalhos após as mudanças que ocorrem, sendo mais coerente utilizar a modelagem provisoria para o planejamento, comunicação direta e discussão entre os membros da equipe. A comunicação entre desenvolvedores nunca deve ser através de documentos.

Após a estória desenvolvida, aprovada pelo cliente, caso o cliente deseje, a cadeia de agregação de valor pode prever uma operação final para confeccionar a documentação manualmente ou preferencialmente através de *Software* de engenharia reversa, desonerando os desenvolvedores do retrabalho intensivo e burocrático durante o processo.

O TDD e BDD, preferencialmente BDD, são consideradas ferramentas de documentação, projeto e de prevenção de erros. Conforme já descritos nos tópicos respectivos (3.6.6 e 3.6.7), estas técnicas possibilitam especificar os resultados esperados do *Software* e os possíveis problemas que podem ocorrer durante seu uso. Quanto mais riscos forem monitorados por estas técnicas melhor.

Além de propiciar a documentação do *Software*, estas técnicas também monitoram o *Software* quanto às anormalidades documentadas (CARVALHO E SILVA et al., 2010). Quanto mais extensa a cobertura das especificações, ou seja, quanto menos partes do sistema existirem sem especificações, maior será a qualidade do produto.

Diz-se qualidade do produto porque o *Software* totalmente monitorado possibilita que sejam feitas melhorias, mudanças, na medida da necessidade do cliente, e de acordo com as melhores práticas descobertas pela equipe, visando a qualidade do *Software*. Esta segurança se reflete também na diminuição sensível de erros quando o *Software* entrar em produção.

4.4.3.1. Andon

A palavra *Andon* em Japonês significa Quadro Luminoso. Este recurso representa mais uma contribuição para os operadores visando a melhoria do processo como um todo. Sua atuação reforça a eficiência do gerenciamento visual. Os dispositivos *Andon* apresentam luzes e sinais úteis aos operadores para sinalizar problemas na linha de produção, e mostradores luminosos que revelam o estado de máquinas e o cumprimento dos procedimentos padrão. Caso alguma anormalidade seja detectada, os quadros *Andon* notificam e informam sobre o ocorrido.

Conforme o tamanho do projeto do *Software* cresce, a complexidade para os desenvolvedores e engenheiros de processo também cresce, podendo tornar-se difícil receber o *feedback* sobre anomalias encontradas. Para dar transparência às variáveis de controle de processo e a comunicação entre os operadores e/ou engenheiros de processo, é feita uma sinalização para os desenvolvedores organizada em painéis de fácil visualização, normalmente colocados em lugares altos, para que todos possam visualizá-los.

Segundo Shingo, nem todo o problema precisa parar a produção. Alguns problemas podem ser configurados para avisar ao operador mas continuar o processamento, como por exemplo erros detectados pelo BDD ou TDD. Alguns problemas podem também ser acionados manualmente pelo desenvolvedor através de sinalização visual no quadro *Kanban*, como os cartões vermelhos para solicitar ajuda com impedimentos.

Ao se detectar um problema mais grave, o *Andon* será ativado para sinalizar à máquina para que os engenheiros de processo investiguem o caso, como por exemplo a parada do *Kanban* por gargalo de processamento. Caso não existam problemas ele sinalizará o estado atual do desempenho do processo e as suspeitas de mal funcionamento que ele tiver conhecimento como quantidade de sinais amarelos e vermelhos na integração contínua.

Cabe ressaltar o caráter visual e simples da técnica, e que o principal objetivo é a melhoria do processo, não do problema local.

No processo de desenvolvimento tradicional as informações de controle do processo são utilizadas para cobrar desempenho dos desenvolvedores, sendo a falta de transparência nestas métricas uma possível causa de falhas de comunicação e ineficiência gerencial.

A transparência das métricas do processo de desenvolvimento enxuto além de gerar aumento de desempenho, gera motivação e orgulho aos desenvolvedores.

Portanto esta técnica visa apresentar e manter atualizadas todas as variáveis de controle do processo, que devem por conseguinte ser compreendidas e avaliadas pelos desenvolvedores. Tais informações são publicadas no quadro de gerenciamento visual, junto ao *Kanban* e aos gráficos de desempenho.

Outra vantagem de se utilizar quadros *Kanban* digitais é poder atualizar e publicar as variáveis de controle do processo no mesmo quadro.

Entretanto, Ikonen et ali (2010) realiza um experimento empírico no qual testa com sucesso o uso do *Kanban* para controlar o processo de desenvolvimento de *Software*. Entretanto os autores caem em um erro conceitual. Os autores declaram neste artigo que o *Kanban* serviria para eliminar os desperdícios durante o desenvolvimento, o que não é verdade. O *Kanban* fornece visualização do estado do processo e autocontrole para os trabalhadores do processo, e com isso possibilita a eliminação de desperdícios, mas não age sobre eles, apenas da transparência e publicidade aos eventos (OHNO, 1996; SHINGO, 1997; MONDEN, XXXX).

Neste trabalho os autores exploram fontes de desperdícios no processo de desenvolvimento de *Software*, em um estudo de caso controlado com 13 desenvolvedores, e reclamam que após a utilização do *Kanban* vários problemas apareceram no processo categorizados como desperdícios.

Cabe ressaltar que a mesma dificuldade ocorreu antes da publicação do pensamento enxuto (WOMACK e JONES, 1998) quando a indústria não compreendia o papel de cada ferramenta no STP, sua real função e as dependências entre as práticas utilizadas no STP.

4.4.3.2. Poke-Yoke

As ferramentas de prevenção e monitoramento de erros, discutidas no tópico 2.3.5, são uma revolução à parte para o desenvolvimento de *Software*. Estudos empíricos realizados por diversos autores têm comprovado seu impacto na estabilidade do *Software*, diminuição de erros, aumento de produtividade, orientação para o desenvolvimento, conforme pode ser visto na Tabela 9.

Tabela 9: Publicações selecionadas sobre TDD e BDD.

Fonte	Qualidade	Erros	Produtividade	Orientação	Qtd
Erdogmus *et al.* (2005)	x	x	x	x	63
Bhat&Nagappan (2006)	x	x		x	15
George&Williams (2003)	x	x	x	x	30
Maximillien&Williams (2003)	x	x 50%	x	x	45
George&Williams (2004)	x	x		x	33
Janzen&Saiedien (2005)	x	x			30

As técnicas TDD e BDD foram discutidas nos tópicos 3.6.6 e 3.6.7.

Entende-se que a academia vem dando grande foco a esta técnica, principalmente sob a ótica de produção pelos números de citações colhidos para cada um destes artigos.

Na ferramenta de indexação de trabalhos científicos Scopus foram encontradas 134 artigos indexados discutindo TDD, e o livro de Kent Beck (2002), que descreve a técnica (TDD) apresenta 1776 citações em artigos científicos.

Danovaro e colaboradores (2008) investiga a criação de uma ferramenta para implementar *Jidoka* para o desenvolvimento de *Software*. Nesta ferramenta os autores juntam a avaliação dinâmica realizada através do TDD com avaliações estáticas realizadas em uma base de dados onde são reunidos os diagramas e o código fonte do *Software*, através da execução de regras escritas em prolog. O interessante desta abordagem é que os autores geram mais sinalizações para o desenvolvedor durante o desenvolvimento, não somente sobre não conformidade, conforme o TDD e BDD, mas alertam sobre possíveis impropriedades, como fragilidades arquiteturais, falhas de projeto, indícios de práticas negativas.

4.4.4. *Takt-Time*

O conceito de *Takt-Time*, segundo Ohno (1997) visa coordenar o ritmo de produção, buscando atingir os objetivos de demanda, aproveitando a estabilidade do processo, respeitando os trabalhadores e a qualidade do trabalho.

O objetivo é casar o ritmo de produção com o ritmo da demanda. Caso o ritmo de demanda esteja temporariamente mais forte que o ritmo de produção, pode-se

ajustar a quantidade de trabalho em processo para aumentar o fluxo e definir um *Takt-Time* menor.

O *Takt-Time* é calculado pela expressão:

$$\begin{aligned}&\textit{Para atender à uma solicitação de } 10 \textit{ estórias médias}\\&\textit{com turnos de } 8 \textit{ horas}\\&5 \textit{ dias por semana}\\&\textit{com um prazo de } 2 \textit{ meses total de } 360 \textit{ horas}\\\\&\textit{cada estória deve ser produzida em } 360/10 = 36 \textit{ horas} \,(4{,}5 \textit{ dias por estória})\end{aligned}$$

Entretanto, o uso do *Takt-Time* deve ser utilizado como um balizador de decisões e um alvo de produção. Para simplificar pode-se usar uma variável *Pitch* que simplifica e torna o alvo mais fácil de ser entendido.

$$\begin{aligned}&\textit{Dado um takt time de } 4{,}5 \textit{ dias para cada estória}\\\\&\textit{pitch} = 2 \textit{ estórias} / 9 \textit{ dias} \quad ; \textit{ou} \quad 4 \textit{ estórias} / 18 \textit{ dias}\end{aligned}$$

Desta forma pode-se variar o tempo das estórias mantendo um alvo para um grupo de estórias pequeno, e portanto gerenciável.

Segundo a produção enxuta não se deve perder desenvolvedores, pois estes deverão ser treinados, receber investimento, conhecer o trabalho, ser produtivos dentro do processo. Desta forma não serão facilmente substituíveis. Quando ocorre uma variação na demanda, para mais ou para menos, precisa-se configurar o ritmo de produção.

A demanda superior a capacidade pode ser um efeito momentâneo. No momento inicial do aumento de demanda, pode-se por exemplo solicitar a equipe um esforço temporário, pagando horas extras, visando estabelecer se a nova demanda é permanente. Assim evita-se tomar decisões permanentes antes de entender o fenômeno de variação de demanda.

4.4.5. Padronização de Operações

No STP, as operações devem ser sempre melhoradas, entretanto é importante se estabelecer níveis mínimos de qualidade e eficiência de processo. A partir destes níveis estabelecidos pode-se comparar tentativas de melhoria de uma forma mais pragmática, caso contrário as tentativas poderiam diminuir a eficiência do processo.

Procedimentos padronizados são boas práticas documentadas visando a educação de novos integrantes da equipe. Estas práticas devem ser julgadas pelos desenvolvedores. Os mesmo desenvolvedores devem ser responsáveis por sua mudança e para tanto devem receber o apoio da gerência imediata.

Cabe esclarecer que o estabelecimento de procedimentos padronizados não pode limitar a avaliação dos desenvolvedores, mas sim afirmar suas convicções. Conforme o pensamento enxuto a valorização dos trabalhadores incentiva ao desenvolvimento profissional, possibilitando que o conhecimento do trabalhador gere retorno para o processo, na forma de ideias para melhorias contínuas (*Kaizen*) e para reformas radicais do processo (*Kaikaku*).

Conforme discutido nas técnicas de XP (tópico 3.6.3), por exemplo o uso de um padrão único de codificação ajuda a criar uma cultura organizacional que une os desenvolvedores e facilita o trabalho de todos ao precisar conhecer e mudar um código ainda desconhecido.

Políticas locais das equipes também devem ser fortalecidas e incentivadas como: quando se deve submeter um código, como proceder quando um erro for indicado na integração contínua.

Entretanto não se pode limitar os desenvolvedores, todas as recomendações devem visar atingir uma qualidade mínima, sendo possível ao desenvolvedor inovar e trazer mudanças e melhorias, e mesmo usar suas próprias ferramentas.

4.4.6. *Kaizen*

As melhorias contínuas no processo de desenvolvimento são essenciais para melhorar o fluxo do processo, por exemplo. Conforme descrito no tópico 2.3.8, as melhorias *Kaizen* buscam agregar o valor desejado pelo cliente e a diminuição de desperdícios.

Estas melhorias são normalmente ideias dos desenvolvedores, discutidas em grupo, e submetidas como experiências para a melhoria do processo.

O processo deve ser monitorado e comparados os resultados do processo antes e depois. Após avaliar os resultados do processo e estas melhorias podem ser documentadas e incorporadas ao processo ou abandonadas.

As variáveis WIP de controle da quantidade de trabalho em processo de cada operação podem ser alteradas, visando melhorar o fluxo de operação.

4.4.7. *Kaikaku*

Conforme discutido no tópico 2.3.9, o *Kaikaku* representa uma melhoria radical no processo, que pode representar uma reforma ou uma reestruturação no processo trazendo resultados imediatos.

Por exemplo a própria implantação de lotes unitários em fluxo contínuo é uma operação de *Kaikaku*.

Outro exemplo seria a expansão da cadeia de agregação de valor, trazendo para dentro do processo novas operações, tornando o processo mais seguro ou documentado, poderiam ser avaliadas na prática e comparados os resultados.

4.4.8. *Layout* da Área de Trabalho

O *Layout* funcional da área de trabalho nos pensamento enxuto segue o preconizado na literatura do STP, visando os benefícios em termos de comunicação entre os elementos da equipe, aumento do desempenho, possibilitando o trabalho em pares (BECK, 2004), diminuição dos problemas de *Hand-Off*s (POPPENDIECK e POPPENDIECK, 2003).

O trabalho em pares, depende de espaço físico adjacente para que dois desenvolvedores trabalhem juntos no mesmo computador, conforme descrito no tópico sobre XP (3.6.3). O desenvolvimento de *Software* depende de transferência de conhecimento entre os especialistas na área de domínio indicados pelo cliente do desenvolvimento. Todos estes aspectos suscitam um ambiente de trabalho próximo e sem barreiras.

Conforme no STP, a área de trabalho em U é utilizada também pelos métodos ágeis, e proporciona melhor contato entre os desenvolvedores, sendo uma recomendação do pensamento enxuto.

A utilização de documentação como forma de comunicação entre desenvolvedores é considerada como um desperdício e um risco. Documentos utilizados somente para comunicação não conseguem transportar o contexto de trabalho de um desenvolvedor para o próximo, sendo considerado um meio de comunicação pobre, lento e arriscado.

Como solução é aconselhada a comunicação direta entre desenvolvedores, sendo necessário portanto uma área de trabalho onde os desenvolvedores possam

conversar e trocar os conhecimentos necessários à continuidade dos trabalhos desenvolvidos, à troca de experiências e ao apoio diante dos impedimentos.

O *Layout* da área de trabalho também ajuda quando desenvolvedores precisam colaborar em operações mais complexas ou quando o volume de trabalho se acumula em uma operação.

Oportunamente, visando a compreensão de um novo requisito, torna-se necessária a presença de especialistas para discussão e modelagem. Esta modelagem pode ser feita em quadro branco, papel rascunho ou mesmo no código fonte sob a forma de testes automáticos.

Cabe ressaltar a diferença entre modelos para discussão e entendimento do problema e os documentos dos métodos tradicionais. A produção de documentos caso solicitada pelo cliente é possível dentro dos métodos ágeis, válida e aconselhável, entretanto, se não desejada pelo cliente será considerada uma operação que não agrega valor.

Na área de trabalho deve ser visível um quadro de gerenciamento visual, onde possa ser utilizado um *Kanban*, e se possível criado um *Andon* com as variáveis de controle do processo.

4.5. Proposta de Roteiro para implantação do pensamento enxuto no Desenvolvimento de Software

Segundo as técnicas e práticas hoje disponíveis, e principalmente com um olhar para outras áreas de engenharia, abrem-se uma grande flexibilidade para o processo de desenvolvimento de *Software*, que devem ser utilizadas para suportar melhorias para os problemas atualmente comprovados.

Diante dos problemas do processo de desenvolvimento de *Software* citados no capítulo 3, cabe chamar atenção para a similaridade com os problemas relatados com o gerenciamento funcional, descrito no gerenciamento de processos, também constante no capítulo 3.

A própria criação de novos modelos de processo e modelos de maturidade, confirmam a insatisfação da Engenharia de *Software* com os resultados obtidos e a falta de previsibilidade para a execução dos processos atualmente propostos.

Denning e Riehle (2009) discutem os requisitos necessários para que se possa chamar de engenharia a disciplina que trata do desenvolvimento de *Software*.

Os autores concluem que a Engenharia de *Software* ainda não contempla os requisitos demandados e que é necessário aproveitar métodos e conceitos de outras engenharias já amadurecidas para que os processos produtivos se tornem mais seguros, estáveis e repetitivos.

Entretanto, conforme discutido no tópico sobre processos (3.2), apesar de existir uma resistência ao ab*Andon*o de gerenciamento funcional, em função de ser o paradigma vigente, existem recomendações para o uso do gerenciamento por processos visando a melhoria da qualidade, diminuição de custos, melhoria de desempenho e facilidade de gerenciamento.

Enquanto no gerenciamento funcional ou vertical ocorrem várias demoras no fluxo entre departamentos e não se tem visibilidade das filas do processo, no gerenciamento por processos ou horizontal pode-se eliminar as filas usando lotes unitários. Cabe ressaltar que o gerenciamento por processos viabiliza a utilização de todo o corpo de conhecimentos gerado na indústria, visando gestão, desempenho, qualidade e otimização de processos.

Para exemplificar o mapeamento serão descritas os princípios e as ferramentas do pensamento enxuto e do STP, visando sua aplicação em um processo de desenvolvimento de *Software*.

A seguir, serão discutidos tópicos explicando como aplicar estes conhecimentos para o processo de desenvolvimento de *Software*.

4.5.1. Implantação de Pensamento enxuto para o desenvolvimento de *Software*

A utilização do pensamento enxuto no desenvolvimento de *Software* vem sendo discutido, entretanto não se encontra na literatura um roteiro para implantação destes conceitos. Em parte porque o pensamento enxuto pode ser utilizado para melhorar processos existentes. De fato, Anderson (2010) recomenda a utilização dos princípios enxutos junto aos processos tradicionais como CMMI, paulatinamente, visando aproveitar da penetração no mercado do CMMI, todos os volumosos investimentos em treinamento e certificação, e os subprocessos propostos por esta norma.

Cabe ressaltar que os princípios do pensamento enxuto devem ser aplicados aproveitando qualquer processo de trabalho desde o mais tradicional até o mais ágil, pois o objetivo é demonstrar através de princípios como otimizar um processo

preexistente, removendo os desperdícios, linearizando o processo, prevenindo os erros através dos conceitos de *Jidoka*, provendo o gerenciamento visual com *Kanban* e *Andon*, criando foco através de lotes unitários, e com isso diminuindo o risco e gerando qualidade, velocidade e satisfação para os patrocinadores e trabalhadores.

- Definição inicial de papeis dos trabalhadores

No contexto do pensamento enxuto se considera fundamental ouvir a voz do cliente. No processo de desenvolvimento de *Software* enxuto aqui descrito será declarado o papel do *Product Owner* (SCHWABER & BEEDLE, 2002) (Dono do produto), que idealmente seria o próprio cliente ou um especialista no domínio do problema por ele designado, ou no pior dos casos alguém que faça a interface do conhecimento sobre o domínio do cliente e as funcionalidades que devam ser implementadas.

Outro papel importante é a do gerente que não exerce comando técnico sobre a equipe, mas apoia nas necessidades, orienta, da suporte, monitora e informa sobre as variáveis de controle do processo de do produto.

A equipe de desenvolvimento fica responsável por sua auto-organização nos assuntos técnicos, buscando cooperar para gerar os valores declarados pelo cliente ou patrocinador. Deverão ser responsáveis pelas questões técnicas desde a arquitetura até a solução entregue ao cliente, buscando atingir ao prazos combinados.

- Definição inicial de objetos de trabalho

Uma definição essencial para se aplicar o pensamento enxuto é a definição do objeto de trabalho no processo de desenvolvimento de *Software*. Segundo discutido neste trabalho se desenvolve *Software* a partir de informações e conhecimentos recebidos do cliente ou especialistas por ele designado, com o objetivo de entregar uma funcionalidade que automatize uma solução, agregando valor assim para o domínio de trabalho do cliente.

Portanto, o objeto de trabalho do processo de desenvolvimento de *Software* será aqui designado como uma estória. Uma funcionalidade poderá ser construída a partir de uma ou mais estórias. Entretanto uma estória deverá ser algo demonstrável

ao cliente, utilizável em produção se o cliente assim desejar. Uma estória não estará “pronta” ou “terminada” enquanto não for potencialmente útil em produção para cliente. Quaisquer estórias devem sempre agregar valor ao cliente.

Entretanto, a definição de uma estória deve ter como princípio definir um objeto de trabalho o menor, o mais simples, o mais coesa e o mais independente possível de outras estórias. A cada estória deve-se pensar no objetivo que se deseja, o porque se deseja, quais os resultados são esperados, que riscos existem associados, que tipos de falhas podem ocorrer e como estas falhas devem ser prevenidas. A prevenção de falhas será descrita no tópico sobre *Jidoka* (2.3.4)

Seguindo o pensamento enxuto, não se deve fazer todo o planejamento de forma exaustiva no início do projeto. Este planejamento inicial deve ser elástico o suficiente para acomodar mudanças, enquanto no momento certo, a cada iteração de curto prazo deve-se fazer um planejamento mais preciso.

Nos pensamento enxuto, como se trabalha em lotes unitários de estórias, uma de cada vez, não se discute com o cliente o escopo completo de todas as funcionalidades desejadas de uma só vez, mas para cada estória, criando foco a cada funcionalidade discutida. A discussão rende mais, os desenvolvedores podem discutir mais profundamente o problema criando foco sobre o mesmo. O cliente cria uma expectativa de receber rapidamente somente o que foi discutido e portanto se compromete em explicar detalhadamente somente o atinente à estória específica.

- Capacidade de trabalho

Apesar de parecer desperdício de capacidade, baseado nos conceitos do pensamento enxuto e nos resultados encontrados em literatura e experimentados na prática pessoal, considera-se aconselhável a prática de desenvolvimento em pares. Esta prática entretanto não é mandatória para o processo, nem para todas as estórias a serem desenvolvidas.

Entretanto acredita-se que seja importante com o objetivo de aumentar a qualidade do desenvolvimento, diminuir riscos, acelerar o desenvolvimento, aumentar a inovação no processo e transferir conhecimento sobre as estórias e uniformizar as habilidades dos elementos da equipe.

Cada par ou desenvolvedor isolado, quando estiver livre, deve comprometer-se com alguma estória disponível na fila de entrada de sua operação.

O objetivo máximo de todos os desenvolvedores será agregar valor que de fato seja útil ao cliente.

Ao realizar sua operação o desenvolvedor deve buscar ver-se como um fornecedor para as operações seguinte no fluxo de trabalho e para o cliente ou usuário da funcionalidade sendo produzida. Para tanto, o desenvolvedor pode repensar o fluxo de trabalho, pensando como pode agregar valor para o cliente, visando a melhoria do processo.

- Convencimento e Implantação da cultura.

O convencimento das vantagens do pensamento enxuto visa levar para a organização conceitos básicos, explicar os princípios e demonstrar as vantagens de sua aplicação. Este convencimento é essencial pois a equipe será responsável pela evolução gradativa a partir do processo atual em um esforço permanente de melhoria.

Uma forma de descrever as vantagens do pensamento enxuto é descrevendo as diferenças entre gerenciamento de processos (vertical) e gerenciamento por processos (horizontal).

O gerenciamento de processos apresenta como desvantagens :

. Falta de visibilidade do processo como um todo ;

. Falta de visibilidade do problema como um todo ;

. Perda de foco sobre o objeto de trabalho ;

. *Hand-Off* constante ;

. Dificuldades de gerenciamento ;

. Falta de valorização do trabalhador ;

. Pouca ou nenhuma inovação e melhoria ;

. Baixo desempenho .

Como pode-se observar, a maioria destes problemas vem da ausência de fluxo no processo. A mudança desta forma de gestão para a gestão por processo ou por fluxo visa abrir a possibilidade de mudar este quadro e usufruir de toda a teoria de gestão de processos da indústria.

Ao adotar um gerenciamento por processos, isto é horizontal e em fluxo, pode-se utilizar várias técnicas da indústria, por exemplo TOC, *Six Sigma*, e

Pensamento Enxuto. Neste caso a abordagem por processos aplica o pensamento enxuto no processo de desenvolvimento de *Software*.

Cabe portanto apresentar as vantagens do gerenciamento usando o pensamento enxuto:

. Oferece visibilidade do processo ;

. Gera foco sobre o problema ;

. Efeitos do *Hand-Off* são mitigados pelo foco e visibilidade do problema ;

. Tornar o processo linear e mapeado facilita o gerenciamento ;

. Valoriza o trabalhador ;

. Incentiva a melhoria e Inovação ;

. Diminui o custo e o tempo de desenvolvimento ;

. Aumenta continuamente a qualidade e o desempenho .

- Obter patrocínio para a implantação do pensamento enxuto

Após a apresentação da cultura enxuta, deve-se obter o consentimento do patrocinador ou da alta administração para a aplicação do pensamento enxuto em algum projeto, de preferência um que apresente prazos apertados e alto risco, ou mesmo já comprometido em prazo ou custos e que precise de uma solução.

Segundo Womack e Jones (1998), em um projeto simples ou pequeno os trabalhadores não terão a persistência para aplicar o método, desanimando e tornando-se descrentes diante de qualquer pensamento superficial contrário.

Mesmo entre desenvolvedores que já utilizam métodos ágeis, frequentemente tem-se dificuldade de aceitação da eficiência do pensamento enxuto. Conforme já comentado, o pensamento enxuto não é intuitivo, principalmente em função do histórico do paradigma de Ford.

- Identificar as origens e a quantidade de demanda

Inicialmente, deve-se identificar onde nascem os requisitos, em que etapas do processo, ou a partir de que solicitações. Deve-se identificar quais são os fornecedores de trabalho para o processo.

Idealmente, deve-se identificar e rascunhar as fontes, os tipos de solicitação e a frequência na qual tais solicitações ocorrem.

Em processo que utilizam métodos tradicionais, deverá ser identificado a ocorrência de um levantamento preliminar exaustivo, onde serão levantados a maioria dos requisitos, e um processo de gerenciamento de escopo, onde podem ocorrer ou serem suprimidas as mudanças de requisitos.

Cabe lembrar que as mudanças de escopo devem ser incentivadas, pois representam melhor adequação do produto às reais necessidades do cliente. Estas mudanças não são necessariamente falhas de levantamento, mas pode representar também necessidades naturais derivadas da evolução da cultura organizacional, de reação a movimentos no mercado, do lançamento de novos produtos, e do aprendizado sobre o uso de *Software*, por exemplo.

- Identificar os destinos de trabalho

Um dos objetivos da identificação do destino é descobrir as fronteiras do fluxo de trabalho. Assim teremos a exata noção do que é a engenharia que estaremos controlando, possibilitando medir o tempo de engenharia ou *Cycle Time*, e o tempo de atendimento ao cliente ou *Lead Time*.

Para o processo sendo avaliado, deve-se identificar quais são os consumidores do processo, neste caso os clientes que recebem o resultado do processo, sejam módulos, sistemas, ou outras formas de *Software*.

Cabe esclarecer que patrocinador, cliente, usuário de sistema são papeis distintos, podendo ser exercidos por pessoas distintas. O patrocinador do *Software* pode não ser o consumidor, cabendo relacionamento distinto entre o processo e cada um deles.

Deve-se observar de quanto em quanto tempo (*Lead Time*) são entregues resultados pelo sistema para cada cliente.

- Identificar e quantificar a demanda de falha

Outras solicitações que podem gerar estórias para o processo de desenvolvimento de *Software* são as demandas de falha. Demanda de falha são as solicitações de correção de erros ou modificação de *Software* por inadequação à realidade do usuário. Esta forma de demanda pode caracterizar baixa qualidade no processo, na identificação do valor do usuário e precisa se tornar visível a todos, não como forma de punição, mas para o aprendizado da equipe.

A frequência de demanda de falha indica baixa qualidade no processo, indicando que o mesmo deve ser discutido buscando identificar os problemas e propostas soluções visando a melhoria do processo e prevenção de erros, vista nos tópico sobre *Jidoka* (2.3.4).

- Identificar linhas de produção ou de produtos

Cada fonte de solicitações pode ser investigada, para verificar se suas solicitações seguem para um *Workflow* único ou mais de um *Workflow* separados, a ser avaliado pelo gestor.

Pode-se utilizar linhas de produção diferenciadas por especialização, como tecnologias antigas e tecnologias de ponta, ou projetos de pesquisa e projetos tradicionais, mas que essencialmente gerem produtos diferentes e com fluxos de trabalho incompatíveis.

Um quadro *Kanban* poderá ter apenas uma tabela *Kanban* ou mais de uma tabela. Em uma tabela *Kanban* poderão existir mais de uma baias se o *Workflow* for o mesmo e se houver necessidade dos recursos em conjunto.

- Identificar as categorias dos objetos de trabalho

Observando o processo atual, pode-se perceber um objeto de trabalho do início ao fim do processo, documentando quais operações foram realizadas para cada objeto de trabalho. Neste ponto deverão ser observados diferenças de fluxo que possibilitem caracterizar objetos diferentes de trabalho. Cabe ao gerente de processo identificar os fluxos de processo diferenciados que serão linhas de produção diferentes. Cada linha de produção deverá ser avaliada individualmente.

Ao avaliar as operações do processo, é interessante tentar criar categorias para os objetos de trabalho, por exemplo por tamanhos. No estudo de casos apresentado no capítulo 6, foram criadas categorias de tamanhos (P, M, G, XL). O objetivo destas categorias é não comparar o desempenho de objetos de trabalho diferentes dentro do fluxo. Outra forma de categorização é quanto ao tipo de demanda, por exemplo desenvolvimento novo, melhoria ou concerto de erro. Outras formas de avaliar o objeto de trabalho é sua arquitetura se é um cadastro, um relatório, estrutural do framework, por exemplo.

- Identificar operações do fluxo de trabalho

Cada operação deve ser documentada, considerando como o processamento realizado na operação gera o valor desejado pelo cliente ou patrocinador.

As operações deverão ser observadas sob a ótica do cliente e deverá ser dada uma nota de 0 a 10 avaliando o quanto o cliente gostaria de pagar pela realização da operação.

As operações deverão ser caraterizadas também quanto a sua importância técnica, descrevendo se são imprescindíveis, porque e de que forma podem ser automatizadas ou eliminadas.

- Identificar a existência de filas para cada operação

Em cada operação pode-se verificar a existência ou não de filas de espera para o processamento. Estas filas devem ser analisadas durante um tempo que represente seu funcionamento geral, e anotados os tempos entre chegadas dos objetos de trabalho, os tempos de permanência na fila, e o tamanho médio da fila.

- Identificar a capacidade de cada operação

Deve ser quantificado quantas pessoas realizam cada operação, e quantos objetos de trabalho existem ao mesmo tempo sendo trabalhados. Para tanto é indicada a observação durante um período de tempo e definida uma média ou valor representativo da quantidade de trabalho em processo (WIP) de cada operação.

- Confecção do Mapa de Agregação de Valor do processo atual

O objetivo inicial deste mapa é visualizar a engenharia do processo em cada linha de produção executada.

A partir das linhas de produção que representam fluxo de trabalho diferenciados, sob o ponto de vista do gestor do processo, deve-se construir um mapa para cada linha.

Estes mapas deverão apresentar a simbologia descrita no capítulo 3, e deverão descrever a situação atual do processo, com os tempos de espera, e os tempos de processamento médios para cada categoria de objeto de trabalho.

Este mapa será interessante para que a cada operação se consiga avaliar o quanto ela contribui para atingir o valor desejado pelo cliente, e o quanto o cliente

realmente deseja e valoriza aquela operação. Idealmente este mapa deveria ser discutido com o cliente, buscando suas considerações sobre o assunto.

- Discutir o mapa junto aos desenvolvedores

Depois de confeccionado o mapa, deve-se fazer uma reunião com os desenvolvedores, para apresentar o mapa e solicitar sua colaboração para corrigir o mapa, sob o ponto de vista de quem realmente conhece as etapas do processo. Devem ser levadas em consideração os seguintes dados:

. O tamanho das filas de espera entre cada operação (WIP);

. Quanto cada operação agrega do valor esperado pelo cliente ;

. Qual a dificuldade de retirar ou automatizar cada operação .

Na verdade esta atividade deve ser feita no planejamento inicial do processo e deve ser disponível para que a qualquer momento os desenvolvedores possam sugerir mudanças que otimizem cada um destes itens, visando a satisfação do cliente, eliminação de desperdícios e coerência sob o ponto de vista de tornar o processo mais estável e previsível.

- Enxugar o processo

O mapa futuro deve tentar eliminar ao máximo possível desperdícios, como por exemplo as demoras e esperas entre operações. Estas demoras podem ser originadas por burocracia, por sobrecarga de fornecedores internos, por falta de comunicação, entre outras doenças organizacionais.

Cabe à equipe de desenvolvimento definir também quais operações podem ser suprimidas ou automatizadas dentro do processo. Entretanto a participação de clientes e gerentes, sobretudo os mais experientes deve ser estimulada e solicitada para que o novo processo possa realmente atingir os objetivos e receba o comprometimento e o apoio de todos.

Lembrando que o objetivo máximo do processo é agregar o valor desejado pelo cliente e diminuir custos e esforço da equipe, ampliando uma relação de confiança e satisfação entre todos.

Entretanto, cada melhoria no processo deve ser mensurável, produzindo resultados não só localmente mas também no resultado do processo, em termos de

Cycle Time, *Lead Time*, falhas produzidas e encontradas, satisfação do cliente, possibilitando verificar as melhorias locais e globais do processo.

- Propor um mapa de agregação de valor Futuro

Como resultado das operações deve ser proposto um mapa de agregação de valor futuro, como um alvo a ser atingido. Estas mudanças devem ser possíveis e devem trazer melhorias, portanto a avaliação negativa na implantação destas mudanças pode ser revertida para o processo anterior.

A melhoria do processo requer coragem e comprometimento de todos. Como consequências o cliente deve ficar mais satisfeito e o processo mais estável, previsível e operável.

- Implementação de lote unitário

Uma das mudanças mais importantes neste sentido é a diminuição do tamanho de lote, se possível para lotes unitários. O trabalho em lotes unitários, conforme já discutido, diminui a complexidade, cria foco, aumenta a qualidade e o desempenho do sistema, facilita o gerenciamento e possibilita a entrega de valor ao cliente mais cedo, diminuindo as incertezas e aumentando a satisfação do cliente.

Só faz sentido de fato priorizar requisitos ou funcionalidades se elas forem entregues em tempos diferentes. Caso as funcionalidades sejam entregues todas ao mesmo tempo não faz diferença o uso de priorização.

- Operações novas no processo

Neste mapa futuro também pode haver a criação de novas operações. No caso de trabalho em lotes unitários, faz sentido trazer parte do planejamento originalmente feito com antecedência, no início do processo anterior, para o início do processamento da funcionalidade a ser executada em um lote unitário, por exemplo.

Nos métodos ágeis, leva-se em consideração vários avanços da ciência e da engenharia, como por exemplo o trabalho em equipes auto gerenciadas, que traz referência das equipes de trabalho da Toyota, as reuniões semanais de Retrospectivas, que buscam estimular a melhoria da qualidade do processo, por exemplo.

- Criação do quadro Kanban

A criação do quadro *Kanban* deve prever a visibilidade do fluxo de trabalho por todos, entre desenvolvedores, patrocinadores e gerentes.

Cabe ressaltar que a utilização de *Kanban* eletrônico, apesar de não expressamente incentivada por autores como David Anderson, Corey Ladas, propiciou no estudo de casos uma boa comunicação e agilidade para o processo, apesar de também ter sido utilizado o quadro manual concomitantemente.

O quadro *Kanban*, conforme já explicado tem duas funções básicas. A primeira é de acompanhar o fluxo de cada objeto de trabalho dentro do processo. A segunda, e talvez a mais importante é de propiciar a generalização do fluxo de trabalho. Ao generalizar o fluxo de trabalho, se pode enxergar melhorias para o processo, visando tornar o processo mais eficiente em termos de custo, velocidade e qualidade.

Para montar o quadro *Kanban* deve-se ter em mente que o desenho do quadro será um constante rascunho. A cada semana é importante que os desenvolvedores, gerentes e clientes olhem para o quadro e tentem imaginar soluções melhores para enxergar melhor o fluxo de trabalho, os gargalos, os problemas que ocorrem dentro do processo.

O quadro deve ser desenhado em um rascunho, se possível em um papel A3, ou A4 deitado, usando lápis, visando fazer um rascunho que possa ser discutido com os desenvolvedores até que se chegue à um formato aceito por todos como algo prático e real.

- Colunas do quadro Kanban

Inicialmente, deve-se separar uma coluna de “Pronto para Produção”, que deve ser a primeira do quadro. O título da coluna deve ser escrito na parte superior da coluna, reservando um espaço para a quantidade de trabalho em processo admissível naquela coluna, a ser escrito abaixo do título.

Algumas implementações expandem o Mapa de valor agregado

Como última coluna, no final da folha, deve ser escrito o título de “Entregue”, que irá representar os trabalhos entregues naquela semana.

Caso o processo atual faça entregas de *Software* por *Release*s, pode-se criar uma penúltima coluna com o título “Pronto para Entrega”.

As operações do Mapa de agregação de valor devem inicialmente ser reproduzidas como grandes colunas no quadro *Kanban*. Cada coluna referente à uma operação pode ser novamente dividida para comportar uma fila de entrada. É importante notar que nem toda coluna precisa de uma fila de entrada. Este sentimento de necessidade deverá ser estimado inicialmente e modificado dinamicamente pelos desenvolvedores, gerentes, como estímulo a melhoria e aprendizado dentro do processo que estará sendo vivenciado e melhorado por todos.

Inicialmente, estima-se que a primeira coluna da engenharia do processo, por exemplo Análise, não precise de uma coluna de entrada, pois a primeira coluna do *Kanban*, “Pronto para Engenharia”, já servirá como fila de entrada.

Após a execução de uma operação da engenharia, o desenvolvedor que terminar sua operação deverá ter um espaço onde depositar as operações terminadas, sendo necessária uma coluna de fila de entrada para as próximas operações de engenharia.

- Evolução do quadro Kanban

Cabe observar que as colunas que serão apresentadas no quadro *Kanban*, são operações que mereçam atenção, operações que possam causar um gargalo no processo ou que precisem de apoio.

Ocorre que a eliminação de desperdícios do processo é uma necessidade conceitual, entretanto durante a execução do processo podem ocorrer desperdícios eventuais, que também devem ser monitorados e eliminados. O quadro *Kanban* permite a visualização de desperdícios como gargalos, demoras, impedimentos, que por vezes não são visíveis para a equipe, pois todos estão envolvidos com seus afazeres.

Segundo o pensamento enxuto, os próprios trabalhadores devem responsabilizar-se pelo fluxo no processo e pelo ataque a qualquer desperdício no processo. Obviamente gerentes também devem interferir e apoiar a eliminação dos desperdícios

Operações que ocorram rápido demais não se considera necessário o rastreamento de sua execução. Cabe lembrar que o modelo inicial deve refletir a realidade do processo, e deve evoluir continuamente, no tempo que a equipe

perceber necessário, mas deverá ter liberdade para evoluir e incentivar sua evolução. Na verdade o quadro *Kanban* deverá representar o fluxo do processo vivo.

- Desenvolvimento em Lotes não unitários

O trabalho em lotes no desenvolvimento de *Software* é considerado quando se analisa, projeta, desenvolve e testa vários requisitos ao mesmo tempo.

Caso se deseje realizar um conjunto de requisitos e funcionalidades correlatas, o que é natural e até provável durante o projeto.

A implementação do pensamento enxuto visa o amadurecimento do processo. O processo inicial pode ser em lotes grandes de requisitos e funcionalidades, e experimentar a diminuição de lotes paulatinamente.

O uso do *Kanban*, por exemplo, possibilita visualizar o fluxo de trabalho mesmo em processos tradicionais, apesar de que a visualização individual dos requisitos e do desmembramento dos requisitos individualmente ser uma forma de gerenciamento muito mais rico.

- Desenvolvimento de Software em Lotes unitários

Uma das contribuições do pensamento enxuto mais importante é a busca por lotes unitários.

No pensamento enxuto se preconiza a diminuição da quantidade de requisitos ou funcionalidades sendo desenvolvida ao mesmo tempo visando diminuir a complexidade, criar foco e qualidade, e aumentar a velocidade da entrega da funcionalidade pronta ao cliente, porque o *feedback* do cliente poderá confirmar sua satisfação ou indicar a demanda negativa de falha na definição do real valor.

- Envolvimento dos desenvolvedores

Todos os analistas, programadores, testadores devem conhecer o processo, participar do monitoramento do processo, sugerir mudanças nas variáveis de controle do processo, sugerir mudanças nas práticas sendo utilizadas, sugerir otimizações para o processo.

As grandes contribuições para a otimização do processo emergem de quem trabalha, executa, vivencia diretamente o processo.

Ocorre portanto a necessidade de valorização de todos os desenvolvedores, repassando as decisões técnicas para os desenvolvedores, diminuindo ou eliminando o comando técnico.

- Monitoramento do processo

O longo do tempo, deve-se medir o desempenho do processo com vistas a identificar o Benchmark atual do processo, com relação ao *Lead Time* junto aos cliente e em separado com relação ao *Cycle Time* de cada categoria de objeto de trabalho (requisito ou funcionalidade).

Para tanto podem ser acompanhados:

. Os tempos entre chegadas de novas solicitações dos clientes ;

. Os tempos totais de entrega para os clientes (*Lead Time*);

. O tempo para executar cada funcionalidade por categoria (*Cycle Time*);

. O desempenho total do sistema ;

. A taxa de demandas de falha .

Com o passar do tempo, a análise de demanda por categoria de estórias deverá tender a diminuir a variação em função da experiência dos desenvolvedores.

- Participação da gerência

Os gerentes tem como funções:

. Resolver os assuntos não técnicos

. Solicitar as estimativas de prazo ou esforço

. Monitorar o processo tornando visual os dados de desempenho e métricas

. Receber notificações ou identificar problemas

. Agilizar a solução dos problemas identificados e notificados

. Sugerir melhorias para o processo, junto com os desenvolvedores

Portanto, o papel da gerência continua agregando valor ao processo, porém o comando e as decisões técnicas, passam para a equipe que toma as decisões técnicas de forma autônoma, em função de seu comprometimento com as estórias que surgem no processo.

O monitoramento do processo é uma função técnica que ajuda à equipe

- Burocracia no desenvolvimento de Software

A produção de relatórios passados entre fases de um processo no âmbito administrativo é considerada por Liker (2005) como um estoque de informações e conhecimentos, baseado no paradigma da produção em massa, visando processar informações em grande escala para ganhar em produtividade. Sob este ponto de vista este estoque precisa ser mantido, pois documentos desatualizados são um risco para o processo.

De fato, corroborando indiretamente com Liker, Pressman (2006) considera sobre os riscos de documentos de *Software* ficarem desatualizados, entretanto considera ser um trade-off aceitável a manutenção destes documentos que nascem no início do processo, durante todo o ciclo de desenvolvimento.

Avaliando a utilidade destes documentos pode-se observar que os mesmos são utilizados em duas circunstâncias:

. Como meio de comunicação durante o *Hand-Off* de trabalho entre operações distintas ;

. Para a manutenção futura do sistema em consertos ou melhorias.

No caso da primeira utilização, como meio de comunicação durante um *Hand-Off* entre diferentes profissionais, o STP resolve este problema através do lote unitário, resolvendo um problema de cada vez, e incentivando a comunicação direta no trabalho em equipes. A documentação no processo de desenvolvimento de novos produtos é feita somente no final do trabalho após se ter posições definitivas sobre o projeto.

O trabalho em lotes unitários é central para resolver este problema pois quando a Engenharia de *Software* trata vários requisitos ao mesmo tempo, cria a dependências de anotações e rascunhos para salvar o contexto de cada requisito, na forma de estoques de resultados intermediários. O desenho do processo dividido em fases exige que estes resultados intermediários de requisitos, informações e regras ainda em processo, sejam repassados para outras fases, criando um *Hand-Off* com estoque de trabalho inacabado associado. O trabalho em lote unitário, executado por uma ou mais pessoas que se comunicam diretamente sobre somente um requisito, o que possibilita a discussão e modelagem do problema, minimizando a dependência destes rascunhos e mitigando os riscos do *Hand-Off*.

Para a segunda necessidade declarada da documentação, os métodos ágeis atualmente sugerem documentar o projeto utilizando documentos executáveis, na forma de código limpo (MARTIN, 2008), TDD (BECK, 2003), BDD (NORTH, 2006), UML executável (MELLOR & BALCER, 2002).

Deve-se utilizar também, ao máximo possível, engenharia reversa pois otimiza o trabalho atingindo resultados satisfatórios. Caso a documentação seja um valor para o cliente, deve-se planejar o processo com uma última operação de documentação a ser realizada após o aceita da funcionalidade ou produto, a ser executada após o objeto de trabalho ter se tornado estável.

Cabe esclarecer que os métodos ágeis atendem aos princípios do STP. São utilizados o lote unitário de trabalho ao se criar foco em uma estória somente, com a eliminação de *Hand-Off* pois trabalhadores multi capacitados devem ser responsáveis pela estória do início ao fim, e com trabalho em pares e revisão de código, favorecendo a dispersão do conhecimento pela equipe.

Entretanto, segundo o STP e o pensamento enxuto, os profissionais não precisariam assumir todas as etapas do fluxo, sendo possível o *Hand-Off* através de comunicação direta, e sendo possível também que cada desenvolvedor multi capacitado possa ser especialista e assumir uma função.

A avaliação sob o ponto de vista da produção enxuta é que documentos criados tendo em vista a comunicação configuram um desperdício e um risco pela baixa qualidade de comunicação, e que se não forem atualizados durante todo o processo se tornam ainda mais perigosos, tornando o *Software* legado (PRESSMAN, 2006).

- Jidoka no desenvolvimento de Software

Cada falha prevista para uma estória será transformada em um dispositivo *Poke-Yoke* dentro do *Software*, utilizando a técnicas de BDD (NORTH, 2006). Estes dispositivos serão executados automaticamente a cada modificação que o *Software* sofra, visando dar segurança e confiança aos desenvolvedores, para alterarem o *Software* buscando melhorias de qualidade e eficiência.

Portanto, quanto mais falhas possíveis venham a ser identificadas para a estória, menor será o risco de falhas. A identificação de falhas possíveis é um ponto

essencial ligado a qualidade do processo de desenvolvimento, pois visa a prevenção das mesmas.

Todas as falhas possíveis por menor que sejam devem ser previstas, visando aumentar a eficiência do monitoramento automático. A cada vez que uma alteração ocorrer no *Software* todos os *Poke-Yoke* (BDD) no *Software* serão automaticamente verificadas e os desenvolvedores serão alertados caso alguma falha de qualquer estória tenha sido afetada.

- Reagir a demanda

Diante das solicitações de novas funcionalidades feitas pelo Dono do produto, estas serão desmembradas em uma ou mais estórias. Neste momento poderá ser discutida a priorização de entrega para estas funcionalidades e portanto escalonadas as entregas das mesmas.

Para tanto, o *Product Owner* deverá criar as estórias e submetê-las a equipe para que se defina o esforço necessário para sua execução. Este esforço deverá ser emitido pelos desenvolvedores, que em última análise conhecem poderão defender o tempo necessário para a execução da estória.

Com o passar do tempo, a experiência em separação de estórias em categorias deve conduzir a uma convergência de tempos estimáveis e à um desenvolvimento estável, permitindo estimativas cada vez mais previsíveis.

Os fatores considerados essenciais para estabilizar o desenvolvimento são a simplificação e diminuição das estórias, a categorização das mesmas e o respeito à estimativa dos desenvolvedores.

- Execução das estórias

Cada desenvolvedor ou par de desenvolvedores, que esteja desocupado, deve observar no quadro *Kanban* se existe algum gargalo na produção, que pode ser reconhecido como uma coluna no limite da capacidade dentro da operação e em sua fila de entrada.

Caso não existam gargalos visíveis, o desenvolvedor deve procurar empecilhos sinalizados sobre a estória em execução de outros desenvolvedores. Um empecilho pode ser sinalizado com uma ficha *Kanban* vermelha, usada para chamar atenção e pedir ajuda à outros desenvolvedores.

Outra prioridade no quadro pode ser uma ficha cinza, simbolizando uma bala de prata, que pede atenção especial ou urgente, e deverá ser puxada pelo primeiro desenvolvedor que a observar.

Se algum erro for encontrado em alguma estória o desenvolvedor deve dar prioridade ao erro, pois este já deveria ter sido terminado e representa um retrabalho.

Caso nenhuma destas prioridades esteja presente no quadro, o desenvolvedor deve verificar a coluna de entrada para sua operação. Dentre as estórias presentes verificar a que tem a maior prioridade. Ler a estória, ler os critérios de aceitação e reposicioná-la na coluna da operação.

Cada processo apresenta critérios para as operações serem completadas. Entretanto cabe observar que estes critérios devem ser estabelecidos pelos desenvolvedores e melhorados sempre que possível, visando eliminar desperdício, aumentar a qualidade do processo e diminuir riscos.

- Parada do processo

Uma característica fundamento do pensamento enxuto é a manutenção e otimização do fluxo de trabalho. Entretanto quando algum erro for detectado, estes casos devem ser anotados visando uma reunião, não para culpar alguém, mas para buscar o aprendizado de todos e a criação de técnicas formais, padronizadas, que consigam prevenir para que estes problemas nunca mais ocorram.

É muito comum que se imagine em somente fazer um retorno da estória no *Kanban*, mas isto não representa evolução nem melhoria do processo. Portanto, o esforço de prevenir pode parecer incoerente em uma visão de curto prazo, entretanto gera mais resultados a médio e longo prazo, pois o processo precisa ser confiável.

- Sobrecarga de trabalho

Quando houver sobrecarga de trabalho em alguma operação do processo enxuto, outros desenvolvedores, ao terminarem suas tarefas, poderão se juntar na operação sobrecarregada, eliminando temporariamente o problema. Este processo se chama enxame (Swarming).

Neste caso, após resolver a sobrecarga, deve-se analisar os níveis de trabalho máximo de cada operação (WIP), buscando balancear a produção para que o trabalho não se avolume novamente.

A diminuição de WIP na fila de entrada da operação sobrecarregada leva o fluxo de trabalho para a velocidade real do processo.

Desta forma o processo deve ser avaliado visando melhorias, como a automatização, desburocratização, aumento da capacidade de trabalho na operação com a contratação de pessoal, por exemplo.

- Mapeamento de Stakeholders

A identificação de patrocinadores, clientes e interessados pode parecer uma coisa trivial, entretanto é de suma importância no pensamento enxuto. Cada stakeholder deve dar sua parcela de contribuição para a identificação dos valores a serem gerados e para a melhoria do processo. Desprezar a importância de um stakeholder é um erro conceitual grave, que não deve ocorrer em um processo otimizado pelo pensamento enxuto, pois diminui a eficiência do processo, o valor do produto e a confiança na parceria que está sendo construída.

- Identificação do valor

A identificação de valor desejado pelo cliente é o núcleo do processo, pois define sua orientação. Conforme afirmado, deve-se identificar inicialmente os stakeholders envolvidos e obter as informações utilizando os meios de comunicação mais eficientes disponíveis. A conversa pessoal é a forma mais eficiente conhecida, seguida por formas cada vez menos eficientes, como conversas telefônicas, depois por chat, depois por e-mail, e por último papéis.

Para garantir atingir aos objetivos é necessário identificar para cada valor desejado:

. Qual é o Domínio do problema ;

. Quais pessoas estão envolvidas no problema, que contribuem e/ou são afetadas ;

. Como a falta do valor prejudica ;

. Quanto o valor agrega de satisfação ;

. Quais são os riscos envolvidos ;

. Quais são as restrições para o problema ;

. Quais suposições ou premissas existem envolvidas ;

. Listar todos os possíveis resultados indesejados ;

. Quais são os comportamentos esperados para esta funcionalidade ;

. Quais são os resultados satisfatórios esperados ;

. Qual é a melhor forma de resolver o problema ;

. Quais outras soluções existem para o problema ;

. Qual é a prioridade para a solução deste problema .

- A prioridade pode depender de uma estimativa de esforço para o desenvolvimento que necessariamente deve emitida pela equipe, mas que pode ser baseada no histórico de desempenho da equipe.
- Pode depender também de uma combinação dos maiores valor agregado e risco envolvidos, pois os maiores riscos estima-se que devam ser desenvolvidos o mais rápido possível, visando eliminá-los o mais rápido possível.

. Quais especialistas podem ajudar a detalhar, testar e aprovar a funcionalidade.

. Após o desenvolvimento se deseja imediatamente a funcionalidade ou pode ser agendada uma entrega de um lote de funcionalidades.

- Derivação de estórias

A partir deste levantamento são definidas estórias necessárias para atingir a esta funcionalidade ou valor definido. O início de uma estória deve preencher as informações do padrão:

Como um <Interessado>
Desejo <Atividade>
Para <Valor esperado>
Quais os critérios de aceitação para a estória
<Critério1> ...

Os critérios de aceitação devem ser definidos junto ao cliente, mas devem levar em consideração:

. O comportamento esperado deve ser testado ;

. As suposições e premissas devem ser monitoradas ;

. Os riscos envolvidos devem ser testados ;

. As restrições devem ser verificadas ;

. Testar cada um dos resultados indesejados ;

. Testar os resultados satisfatórios ;

. Verificar se funções externas foram acionadas em ambiente simulado .

Como padrão arquitetural, cada solução listada deve ser possível em substituição para a solução ótima, sendo as outras soluções definidas pelo cliente opções em aberto, que podem ser implementadas futuramente. Portanto a arquitetura para a solução ótima deve ser plugável e substituível pelas outras, cabendo um estudo de interfaces que possibilite a modularidade das soluções implementadas.

Cada estória deve obrigatoriamente gerar um resultado real, mensurável e útil ao cliente. Não sendo válidas ou possíveis estórias que não gerem valor sob o ponto de vista do cliente.

- Trabalho do cliente ou especialistas designados

A participação do cliente ocorre durante vários momentos do processo.

Inicialmente se precisa do levantamento de valores desejados, que guia não só o desenvolvimento, quanto a melhoria do processo.

Quando algum desenvolvedor for realizar o planejamento das estórias deve-se ter a atenção do cliente ou especialista designado.

Quando estiver codificada, testada, integrada e validada a estória, o cliente deve dar atenção ao aceite da funcionalidade ou valor desenvolvidos.

Cabe lembrar que estas operações podem variar de processo para processo, sendo este somente uma sugestão de comportamento.

- Trabalho técnico relacionado

Após definidas as estórias junto ao cliente, os desenvolvedores devem se reunir para discutir a melhor abordagem para o desenvolvimento das estórias. Cada tarefa poderá ser dividida ou não em tarefas, que tem escopo eminentemente técnico.

Enquanto cada estória gerar valor para o cliente. As tarefas devem contribuir sob uma ótica técnica. São formas de dividir trabalho entre desenvolvedores.

Tais tarefas também devem seguir o fluxo de trabalho e devem ter um critério de aceitação, testável, verificável, e portanto um objetivo a ser atingido.

- Padronização operacional

As soluções técnicas são soberania da equipe de desenvolvedores, entretanto eles devem buscar soluções em grupo e que possam ser padronizadas a partir do consenso.

Cada solução técnica que seja consenso absoluto de todos deve tornar-se uma especificação de trabalho padronizado. O objetivo do trabalho padronizado não é prender e liminar tecnicamente, pois deverá ser revista a cada nova ideia de qualquer um dos desenvolvedores, sendo quebrada sempre que não houver consenso.

O objetivo real é otimizar as decisões e facilitar o treinamento de novos desenvolvedores.

- Entrega de Software

Uma entrega de *Software* ao cliente pode ser agendada ou pode ser solicitada a qualquer momento, portanto o *Software* deve estar sempre Integrado e livre de erros. As entregas de *Software* devem ser periódicas pois representam retorno de investimento para o cliente ou patrocinador.

O intervalo entre entregas deve ser acordado com o cliente, mas pode ser definido em termos de tempo em meses ou semanas ou em número de funcionalidades, por exemplo a cada 10 funcionalidades.

Todas as funcionalidades desenvolvidas devem ser integradas ao *Software* automaticamente, através da integração contínua, e deve haver um *Software* novo que o cliente ou um especialista por ele designado, visando o aceite da funcionalidade.

- Métricas de controle

O gerenciamento do processo de desenvolvimento de *Software* normalmente aplica métricas de controle de processo baseadas em dados técnicos referentes ao processo, e assim também age a engenharia. Entretanto o pensamento enxuto tem contribuições a este respeito.

Segundo o pensamento enxuto deve-se preocupar mais com métricas do processo do que métricas das operações. Os ótimos locais podem ser um desperdício para o ótimo global.

Ainda além do ótimo do processo, no pensamento enxuto se entende que as relações entre fornecedor e cliente devem ser pensadas à longo prazo, visando sempre ganho para ambos, para que a relação seja honesta, de confiança e duradora. Assim ocorre inclusive a transparência de ambas as cadeias de valor visando a otimização e proporcionando uma parceria de longo prazo, baseada na confiança.

Para tanto as métricas de controle de processo devem monitorar a satisfação do cliente e a real agregação do valor segundo o ponto de vista do cliente. Os valores devem ser gerados segundo as necessidades do cliente, na hora em que forem realmente necessários, no local em que forem necessários, na quantidade que forem necessários.

O processo precisa medir em três dimensões, a satisfação do cliente, a eliminação dos desperdícios, estabilidade e consequente previsibilidade do processo.

- *Lead Time* ou Tempo de Entrega

Esta variável possibilita identificar o tempo de resposta (porta à porta) das solicitações do cliente. Entretanto as solicitações podem apresentar volumes de trabalho diferenciados, sendo portanto difícil usar um *Lead Time* passado para compreender o desempenho do sistema. Normalmente, usa-se *Cycle Time* ou *Throughput* do sistema que se baseia em operações para fazer estimativas de *Lead Time*, que acaba tornando-se uma variável estimada.

- *Cycle Time* ou Tempo de Engenharia

Representa o tempo de execução do ciclo de agregação de valor do processo. Cabe esclarecer que no processo de desenvolvimento de *Software* as estórias apresentam atributos como tamanho, complexidade, risco, potencial de reutilização, que deverão influenciar no tempo de execução da estória. A criação de categorias ajuda a agrupar e analisar as estórias, possibilitando estimativas e previsões de tempo de ciclo mais ajustadas.

- *Throughput* ou Desempenho

Representa quantas estórias saem do processo por medida de tempo. Por exemplo, um processo pode ser capaz de entregar 15 estórias pequenas por semana, ou duas estórias grandes e 3 médias por semana. Representa portanto a taxa medida na saída do processo.

- Produtividade

Esta variável representa a relação entre o desempenho e a quantidade de recursos utilizados. Pode ser calculado dividindo-se a quantidade de estórias realizadas pelo quantidade de desenvolvedores envolvidos.

- Percentual de cobertura de testes

O percentual de cobertura de testes é calculado pelo número de funções com testes associados dividido pelo número de funções totais do sistema.

Este percentual de cobertura deverá se aproximar de 100%.

- Qualidade dos testes

Cabe observar que os testes de *Software* devem ser desenhados com a participação do cliente. Entretanto um bom conjunto de testes deve sempre procurar identificar e monitorar as falhas graves e riscos possíveis no domínio do problema.

Os desenvolvedores ao planejar os testes do *Software* devem tentar encontrar testes para as seguintes situações:

. Quais as falhas graves não poderiam acontecer nesta função ou nesta funcionalidade?

. Quais os sinistros deveriam ser evitados?

. Quais problemas são possíveis dentro desta funcionalidade?

. Quais problemas são comuns dentro desta funcionalidade?

. Quais problemas devem ser evitados dentro desta funcionalidade?

. Cada situação desta deve ter um ou mais testes visando forçar sua ocorrência ;

. Os limites de parâmetros numéricos devem ser testados dentro e fora ;

. A ausência de parâmetros, parâmetros ilegais e parâmetros conflitantes ;

. Quantas operações podem ser executadas em 1 minuto?

- Velocidade da equipe

A capacidade da equipe deve ser medida pela quantidade de esforço pode ser feito em uma semana.

O trabalho realizado a cada semana deve ser contabilizado em termos de pontos de estórias, visando estabelecer o número de pontos que cabem dentro de uma semana.

- Taxa de demanda (Solicitações no sistema)

Idealmente o pensamento enxuto busca estabilizar a demanda do cliente através da confiança de entregas rápidas. Deve-se evitar situações onde se compromete com várias funcionalidades ao mesmo tempo. O ideal será receber solicitações em uma quantidade estável e constante, após as entregas.

Caso o cliente deseje fazer um lote de solicitações, cada solicitação deve ser priorizadas pelo cliente. As entregas deverão ser rápidas buscando estabelecer um laço de confiança.

O cálculo da taxa de demanda deverá contabilizar o número de solicitações feitas por semana.

Idealmente o número de solicitações deverá ser inferior a capacidade de trabalho da equipe em uma semana, caso contrário estará crescendo uma fila.

- Capacidade da Equipe

A capacidade da equipe reflete quanto trabalho o sistema ou o *Kanban* do sistema suporta em um dado momento em número de estórias.

- Erros detectados depois de entrega (Nominal)

Espera-se que com o uso correto das técnicas de BDD e TDD, além da participação do cliente dentro do processo e da aceitação de cada funcionalidade, que seja diminuída sensivelmente a proporção de erros detectados durante o tempo de produção.

Entretanto, caso ocorram devem refletem que a arquitetura de prevenção de falhas não foi bem projetada. A solução do pensamento enxuto para as falhas é melhorar a prevenção para não voltem a ocorrer.

Deve-se observar os índices de cobertura de testes e verificar quais testes existiam para o problema que não foi monitorado. As soluções possíveis são

aumentar a taxa de cobertura de testes, melhorar a qualidade dos teste e contratar um treinamento de BDD e TDD.

Para calcular a taxa de erros detectados depois da entrega, deve-se dividir o número de problemas detectados pelo número de funcionalidades entregues. Este número deverá permanecer sempre abaixo de um (1).

4.6. Comparativo de atributos entre um processo tradicional e um processo enxuto de desenvolvimento de Software

Inicialmente cabe discutir à diferença entre os processos ágeis e o processo enxuto. Enquanto os processos ágeis são bem definidos enquanto suas regras, papeis dos desenvolvedores e gerentes, práticas, o pensamento enxuto oferece princípios visando conduzir ao atendimento da vóz do consumidor, diminuição de desperdícios e melhoria contínua do processo.

Qualquer processo de desenvolvimento, seja tradicional ou ágil pode ser otimizado aplicando estes princípios. Diversos exemplos são citados na literatura como a aplicação de pensamento enxuto para otimizar processos CMMI nível 5 (ANDERSON, 2010; SUTHERLAND *et al.* 2007; JAKOBSEN & JOHNSON, 2008; JAKOBSEN & SUTHERLAND, 2009) entre outros; e otimizando métodos ágeis como o *Scrum* (LADAS, 2008).

Cabe ressaltar que entre os métodos ágeis existe também a proposta dos Poppendieck de Desenvolvimento Enxuto, propondo um método ágil com sua visão de processo já adequado. Outro método que se aproxima de otimicidade sob a ótica do pensamento enxuto é o XP, que apesar de poder ser melhorado por apresentar lotes por tempo (*Time Box*), apresenta mais proximidade e facilidade de evolução para atender aos princípios do pensamento enxuto.

Na Tabela 10 são expressas características comparadas entre os processos Tradicional e Enxuto, visando evidenciar as diferenças, vantagens e desvantagens.

Tabela 10: Comparativo de Processo tradicional x Enxuto

Atributo	Tradicional	Enxuto
Complexidade	Mapeia, documenta e testa	Simplifica, desacopla, teste mais eficiente
Simples	Custo alto, entrega demorada e burocrática	Entrega Rápida e custo baixo
Risco	Mapeia e Testa, Fé	Mitiga risco melhor, simplifica e entrega rápida
Fronteiras Organizacionais	Resolve	Envolve contratação, meios legais, quebra o fluxo do processo
Documentação Necessária	Retrabalho, comunicação inadequada	Ao final, automática, otimizada, sem retrabalho
Planejamento	Somente Inicial, independente de quem planeja: Gera produto errado ou falha	Intensivo ao longo da produção
Criatividade e Inovação	Fraco	Intensiva, incentivada
Melhoria Contínua	Baixa (Gestão Funcional) Gerencial Salvo *Lean+Six Sigma*	Intensiva: por processo, em fluxo, pelos desenvolvedores gerenciamento visual

- Quanto a Complexidade

A questão da habilidade para lidar com *Software* mais complexo é muito argumentada como uma deficiência dos processos ágeis. Apesar de considerar discutíveis as argumentações, esta discussão foge ao objetivo desta dissertação. Entretanto é necessário destacar que o pensamento enxuto é base para a grande maioria dos princípios contidos nos métodos ágeis, e precisa ser discutido como o pensamento enxuto pode impactar no processo de desenvolvimento no tocante aos atributos em destaque na Tabela 10.

Para desenvolver um *Software* cujos requisitos sejam complexos e até extensos, os métodos tradicionais confiam no levantamento inicial de requisitos e na análise exaustiva de peculiaridades para cada um dos requisitos, e no mapeamento de riscos.

Inicialmente as praticas parecem corretas, entretanto o processo por trabalhar em lotes está sujeito à perda de foco nos requisitos, diminuição do foco pelo

chaveamento de contexto. O *Hand-Off* baseado em documentos propicia a perda de conhecimento tácito, aumentando o risco de erros durante o processo.

Portanto, quanto mais complexo o projeto, estima-se que seja ainda mais adequada a utilização dos princípios enxutos, visando desenvolver as funcionalidades priorizadas pelo risco, mitigando de forma mais eficiente e obtendo o *feedback* mais rápido do usuário, confirmando ou realinhando o conhecimento da equipe de desenvolvimento para que se desenvolva o *Software* correto.

- Quanto a Simplicidade

No desenvolvimento de *Software* mais simples os processo tradicionais mostra-me extremamente burocráticos não apresentando soluções ágeis. O pensamento enxuto tem oferecido contribuição direta para os métodos ágeis, e possibilita inclusive flexibilizar os métodos tradicionais, em uma evolução suave e contínua.

Portanto, a utilização de um processo ágil pode ser considerada também um resultado do pensamento enxuto. Vem sendo discutida a melhor adequação destes métodos para o desenvolvimento de *Software* mais simples, inclusive proporcionando economia dos recursos necessários a aplicação dos métodos tradicionais.

- Risco

Projetos de *Software* que envolvam alto risco são normalmente classificados como candidatos para o desenvolvimento em processos tradicionais.

Os processos ágeis podem ser considerados ainda desconhecidos pela academia da Engenharia de *Software*, em função da ausência de trabalhos empíricos que consigam medir sua real contribuição.

Entretanto, trabalhos acadêmicos na Engenharia de Produção já discutem o STP, o Pensamento Enxuto, a aplicação do Pensamento Enxuto para serviços e o gerenciamento por processos, com resultados comprobatórios de suas contribuições.

Pela ótica do pensamento enxuto, Riscos devem ser enfrentados e eliminados o mais rapidamente possível. Riscos também devem ser prevenidos objetivamente, através de ferramental de especificação do risco e monitoramento do mesmo.

Em projetos tradicionais a identificação de risco e mitigação do mesmo conta com a experiência de cada profissional e com testes manuais depois do *Software* pronto.

A diferença entre os testes manuais e os testes automatizados foi discutida no tópico 3.6.4 (Testes de *Software*).

Para enfrentar os riscos diretamente o processo enxuto preconiza priorizar estórias pelo risco, possibilitando a rápida resolução dos problemas que possam ocorrer.

Entende-se com esta argumentação que o tratamento de riscos no pensamento enxuto seja mais pragmático e consequentemente mais eficiente.

- Fronteiras organizacionais

A aplicação de um processo em fluxo além de fronteiras organizacionais não existe uma negociação prévia. Existem suficientes evidências de processos enxutos para a gestão de cadeias de suprimentos, ultrapassando várias empresas, entretanto a faltam indícios de coordenação de desenvolvimento de *Software* usando princípios enxutos nestas condições.

Os processos tradicionais aparentam ser eficientes pois nestas condições faz-se necessário ou laços de parceria bem consolidados, como no caso do STP, ou artefatos que fixem os direitos, deveres, restrições de prazo, custo, qualidade e escopo.

- Documentação necessária

Segundo o pensamento enxuto, parte da documentação que não for considerada útil para o cliente pode ser considerada burocracia e portanto desperdício. Parte dos documentos podem ser recuperados automaticamente, sem esforço e perda de tempo pelos desenvolvedores.

Ao mesmo tempo, grande parte da documentação produzida nos processos tradicionais podem ser considerados rascunhos necessários em função do processo ser executado em lotes, podendo portanto ser eliminados. Outra parte da documentação dos processos tradicionais são constantemente reescritos, configurando retrabalho e desperdício.

Outro problema considerado crítico é o uso de documentação como forma de comunicação, que deve ser eliminado visando a eliminação do risco da comunicação de baixa qualidade, sendo aconselhada a comunicação direta entre desenvolvedores.

A documentação considerada como um valor pelo cliente deve ser gerada ao final do processo, quando a funcionalidade ou produto estiver estável e aceito pelo cliente.

Sob este ponto de vista, o pensamento enxuto pode colaborar, atingindo os objetivos propostos, melhoria da qualidade, diminuição do custo e do tempo de produção.

- Planejamento

O planejamento nos processos tradicionais é feito no início do processo, visando cobrir todas as atividades de produção necessárias para todos os requisitos levantados, normalmente sendo utilizados cronogramas de para este mapeamento e controle.

Entretanto, este planejamento acaba normalmente cobrindo um conjunto grande de atividades, em um prazo extenso, referente à um conjunto grande de funcionalidades. Este escopo de planejamento agrega riscos de diversas dimensões. O conjunto de funcionalidades pode se tornar desatualizado com relação ao desejado pelo cliente com o passar do tempo. As atividades podem demorar mais ou menos do que o estimado, provocando o escorregamento do cronograma. O prazo extenso gera uma cobrança maior por parte do cliente, por não perceber os resultados do serviço contratado.

Segundo o pensamento enxuto, o serviço deveria ser prestado realizando entregas de valor frequentes, com a participação do cliente. Para tanto o planejamento seria de um escopo menor, consequentemente menos complexo e gerando um prazo menor. Desta forma a probabilidade de variação nos prazos do cronograma seria menos representativa. O planejamento poderia portanto focar em mais detalhes sobre os riscos a serem enfrentados, identificando e mitigando-os com mais precisão.

- Criatividade e Inovação

Os processos tradicionais apresentam uma estrutura de gerenciamento funcional vertical, acoplada funcionalmente (ISO-IEC 12207). Este estilo de processo implica no gerenciamento funcional, que faz parte do senso comum, tendo sido estabelecido em todo o mundo desde 1908 pelo sucesso da Ford, conforme já discutido.

Portanto, esta forma de gestão coerciva e subreptícia tornou-se amplamente utilizada nos métodos tradicionais, diminuindo sensivelmente as formas de inovação e desestimulando a esforços para a mudanças no processo.

Atualmente os empreendimentos na busca por melhoria de processos como o CMMI aplicam *Six Sigma* na busca pela melhoria do processo. Ocorre que esta técnica realmente tem muito a contribuir na melhoria da eficiência do processo. Entretanto provou-se mais difícil obter somente com esta técnica outras mudanças não dedutíveis estatisticamente. Atualmente, a associação das técnicas *Six Sigma* e Enxuto é amplamente utilizada, justamente por serem técnicas complementares, pois o Pensamento Enxuto apresenta o potencial para mudanças organizacionais e reestruturação de processos.

- Melhoria Contínua

Esta técnica que nasceu no pensamento enxuto, se apoia na valorização dos trabalhadores, neste caso os analistas, projetistas, programadores, testadores. Nos processos tradicionais, onde estes trabalhadores não apresentam autonomia, e o processo comanda e controla os desenvolvedores, dificilmente serão obtidos resultados semelhantes aos do pensamento enxuto.

Entretanto com o vivenciamento de um processo onde os desenvolvedores são prestigiados, recebem autonomia para assumir parte da função gerencial, sobretudo autonomia técnica, emerge um sentimento de posse do processo e valorização da melhoria.

As melhorias essenciais são a busca pela eliminação de desperdícios, identificação dos valores desejados pelo cliente e adequação das operações para agregar o valor desejado pelo cliente. Ao mesmo tempo, deve-se buscar diminuir o tempo entre a solicitação do cliente e a entrega do valor.

Cabe a toda a equipe de desenvolvimento não deixar o processo cair na inércia, pois sempre existirá formas de otimizar o fluxo, automatizar ou eliminar operações, visando a melhoria do processo.

4.7. Discussão sobre Engenharia de Software e Pensamento Enxuto.

Dentre as várias engenharias como por exemplo civil, elétrica, química, entre outras, o contexto de trabalho pode variar entre dinâmico, onde se possa observar variação de contexto, e estático onde as chances de variação sejam pequenas. Um projeto de engenharia civil pode-se observar pouca variação do escopo do projeto, em função de pouca variação no contexto de trabalho. Por outro lado, projetos de investimento financeiro podem ser caracterizados como dinâmicos, por existirem em um contexto naturalmente instável.

Cabe observar que o *Software*, por sua própria natureza apresenta flexibilidade natural, sendo tecnicamente possível modificá-lo. Por outro lado a utilização de *Software* depende também da aderência do *Software* ao ambiente onde este seja aplicado, o que implica em que o *Software* perca seu valor na medida em que este se distancie das necessidades de quem precisa utilizá-lo.

Desta forma, o contexto da aplicação do *Software* sendo mais dinâmico, com mudanças constantes, implica que sua atualização também deva ser constante, sendo natural e esperável que ocorram mudanças no escopo do *Software*. Portanto, a aplicação de processos produtivos que não preveem tais mudanças de escopo conflita diretamente com a natureza e com a adequação do *Software*.

Segundo Middleton e Sutton (2006), os processos derivados de Royce (1970) apresentam fundamentais semelhanças com características da produção em massa, pois tratam em cada fase de desenvolvimento o conjunto dos requisitos de *Software* como um lote, gerando estoque de conhecimentos e produtos semiacabados

Segundo Liker (2005), os relatórios gerados pelas engenharias em processos em lotes são exemplos de desperdícios pois o processamento de conhecimento em lote, gera estoques de informações que representam passos intermediários de um processo maior. Estes estoques de informação deverão ser utilizados em outras etapas até que sejam consideradas terminadas.

Nesta dissertação se considera que o processamento de informações em lote gera rascunhos indispensáveis e inerentes ao processamento em lote. Sem estes rascunhos as informações semi-processadas seriam perdidas. Estes rascunhos podem então ser considerados estoques intermediários, que possibilitam o armazenamento para uso futuro da informação, em outra fase do processo.

A necessidades destes estoques está diretamente relacionada ao trabalho em lotes e à especialização de profissionais. Ocorre que nos processos de desenvolvimento de *Software* tradicionais faz-se a passagem de bastões entre grupos funcionais, como por exemplo relatórios de análise de requisitos produzidos por analistas, passados para programadores, exatamente como se houvessem departamentos especializados.

Para que o processo seja considerado enxuto, cada informação deveria ser tratada em uma cadeia de agregação de valor própria, em uma organização horizontal, visando gerar o valor esperado pelo cliente, executando somente operações que agreguem valor, em fluxo contínuo (sem esperas), puxada a partir do valor especificado e solicitado pelo cliente.

Segundo Liker (2005), a geração dos relatórios das engenharias, representam portanto desperdícios por estocarem informações intermediárias, não resolverem o problema e serem por isso, normalmente ineficientes com relação a sua utilização para adicionar qualidade ao processo. Segundo este autor, a produção enxuta auxilia trazendo aumento da qualidade, redução de custos e dos *Lead Time*s, em função da aplicação do conceito do fluxo unitário de produção, que neste caso implica no tratamento unitário de informação durante todas as operações que agreguem valor até que a informação seja totalmente processada e os objetivos desejados sejam alcançados.

O processo das engenharias é centrado em documentos porque, conforme já discutido, utiliza processamento de informações em lote, sendo necessário o armazenamento dos resultados intermediários para a futura recuperação e continuação do processamento, provavelmente em outros departamentos.

Segundo Womack e Jones (1998), toda a lógica de departamentos, processamento de informações em lote, são paradigmas que surgem após a implantação da produção em massa. Este paradigma é tão forte, que estabeleceu-se

como um símbolo de eficiência desde os anos do início da produção em massa até os dias de hoje.

Corroborando com isso, o chaveamento de contexto, segundo a Corrente Crítica de Goldratt (1998), é um dos males comuns na execução de atividades humanas. Em operações que dependem da inteligência humana, tipicamente em tarefas de trabalhadores do conhecimento, como é o caso do desenvolvimento de *Software*, alternar o contexto de trabalho entre mais de um problema aumenta as chances de erros, diminuindo a qualidade do trabalho e aumentando o risco em razão proporcional à quantidade de contextos de trabalho operados ao mesmo tempo.

Portanto, o trabalho cognitivo do trabalhador do conhecimento, deve ser levado a termo de uma só vez, corroborando com o pensamento enxuto que prega a eficiência do processamento em lote unitário e fluxo, com vistas a diminuição do risco, aumento do desempenho e da qualidade do processamento. Ainda considera-se como vantagem desta abordagem a diminuição do tempo de *Lead Time*, pois o valor é atingido em menos tempo do que ao alternar contexto entre mais de uma atividades.

5. Estudo de caso

Neste capítulo será apresentado o estudo de caso demonstrando um processo ágil, com claras influências do pensamento enxuto, onde serão demonstradas as características semelhantes ao STP, evidenciando a necessidade de um olhar da Engenharia de Produção para este processo produtivo.

Nos tópicos seguintes serão apresentados o ambiente do problema estudado, as características do processo, os dados colhidos neste processo e a discussão dos resultados obtidos.

5.1. Caracterização do ambiente

A unidade de análise do estudo de caso é o processo de desenvolvimento de *Software* utilizado no Núcleo de Pesquisa em Sistemas de Informação (NSI) do Instituto Federal Fluminense (IFF).

O NSI foi criado em 2002 com o objetivo de desenvolver projetos de pesquisa aplicada, capacitando alunos e gerando resultados de qualidade para a sociedade. Este núcleo já desenvolveu e desenvolve diversos projetos com empresas e organismos de governo dentro e fora do Brasil, tais como o projeto Argus desenvolvido para a PETROBRAS, os projetos da RENAPI para o MEC e o ERP5 desenvolvido em parceria com a empresa francesa NEXEDI, bem como diversos outros projetos para prefeituras da região. Para tanto o NSI tem contado com a atuação de alunos e professores do Instituto Federal Fluminense, tanto em nível técnico quanto de graduação e pós-graduação.

Desta forma, o núcleo tem crescido em importância, ampliado sua captação de recursos e assimilado um maior número de alunos de dentro e de fora do IFF. Atualmente vários alunos de outras instituições de ensino superior tem procurado o NSI para participarem dos projetos e se capacitarem.

Atualmente este núcleo de pesquisa conta com cinco professores, dos quais dois doutores, três mestres, dois técnicos de nível superior, 20 alunos de graduação, 5 alunos de mestrado, 3 agregados de empresa parceira.

Os projetos são programados com um planejamento inicial anual, com replanejamentos incrementais mensais, durando normalmente vários anos. Cada projeto envolve um quantitativo variado de desenvolvedores e professores,

apresentando casos como: a) dois professores e nenhum aluno; e b) um professor e 5 alunos. Cada projeto é gerenciado por um professor, que torna-se responsável pela definição de valores a serem atingidos e pela prioridade na qual os valores deverão ser desenvolvidos inicialmente.

Os riscos são avaliados no início e durante o processo de execução dos projetos através das reuniões de planejamento semanais, buscando os melhores caminhos de execução e trabalho.

O NSI apresenta um portfólio de projetos bem variados em escopo e complexidade, variando entre estudos de vibrações de motores visando a manutenção preventiva, até bancos de dados multi conteúdos.

O desenvolvimento no NSI conta atualmente com três frentes de trabalho a saber: gerenciamento do desenvolvimento, desenvolvimento SOA (Service Oriented Architecture) e Qualidade Ágil.

O gerenciamento do desenvolvimento é coordenado por um professor/pesquisador que conta com uma equipe de 10 alunos em média. Um dos projetos desenvolvidos nestes setor atualmente é da Biblioteca Digital, que será utilizada como estudo de casos neste trabalho.

O setor de qualidade ágil visa pesquisar novas técnicas, desbravando novas fronteiras de conhecimento, indo onde os desenvolvedores ainda não foram, com objetivo de avaliar, aprender e implantar técnicas de qualidade dentro do NSI.

O setor de serviços tem a função de pesquisar novas arquiteturas para melhorar as funcionalidades da BD, principalmente tornando as funcionalidades principais mais escalares, utilizando orientação a serviços.

5.2. O projeto estudado

Neste estudo será acompanhado o projeto da Biblioteca Digital, desenvolvida por encomenda da Secretaria de Educação Profissional e Tecnológica (SETEC) do Ministério da Educação e Cultura (MEC)

Este projeto possui ainda um subprojeto para definição e implementação de serviços assíncronos, mas que não foi alvo deste estudo.

5.2.1. Estrutura gerencial

Poderia se esperar que para um projeto do porte da BD houvesse uma hierarquia gerencial, tal qual encontra-se em diversos outros projetos, com um gerente principal e outros gerentes específicos para escopo, mudanças, tempo, por exemplo. Entretanto, no caso estudado se pratica o gerenciamento com autonomia, conforme preconizado no STP.

O coordenador do projeto cumpre o papel semelhante ao Project Owner do *Scrum*, sendo responsável pelo conhecimento do valor para os clientes, fazendo reuniões com o MEC e com os bibliotecários que utilizam a BD, trazendo as necessidades de novas funcionalidades, problemas encontrados, sugestões de mudanças e aperfeiçoamentos, e administrando e priorizando estas atividades.

Os diversos aspectos da questão técnica são discutidos em regime de Parlamento dentro da equipe de desenvolvimento, onde o parecer técnico dos desenvolvedores tem o maior peso. Os alunos desenvolvedores são incentivados a buscar e utilizar técnicas de ponta, métodos que visem o desempenho, qualidade, modularidades, flexibilidade, e portanto, sendo especialistas no que fazem, são a voz principal para a tomada de decisões técnicas.

O coordenador do projeto, entretanto, participa das decisões, colaborando também tecnicamente, mas não com intenção de assumir a decisão técnica.

5.2.2. Caracterização do projeto

O período do projeto acompanhado consiste em adaptações feitas a BD entre março de 2010 e dezembro de 2010, perfazendo um total de 33 iterações. As estórias foram executadas pelos desenvolvedores seguindo um processo baseado nas metodologias *Scrum* (SCHWABER & BEEDLE, 2002), entretanto tendeu ao longo do tempo à aproximar-se das técnicas do *Kanban* (ANDERSON, 2010), podendo ser classificado como o *Scrum*ban (LADAS, 2008), conforme explicado anteriormente.

No início deste período ocorreu uma reunião entre os bibliotecários responsáveis pela implantação da BD nos Institutos Federais no Brasil, e daí foram derivadas sugestões de melhorias a serem realizadas na BD, com objetivo de adequação de sua estrutura de funcionamento e recursos aos padrões técnicos da área de documentação e biblioteconomia.

As solicitações foram coletadas pelo coordenador de desenvolvimento, visando identificar os reais valores esperados pelos bibliotecários solicitantes. Foi desenvolvido um documento para sintetizar os valores de negócio desejados, as possibilidades de mudança e melhoria sugeridas, e os resultados esperados.

- *Layout* do ambiente de trabalho

O *Layout* funcional de desenvolvimento é apresentado na Figura 15, e busca facilitar o entrosamento entre os desenvolvedores e facilitar sua comunicação e interação.

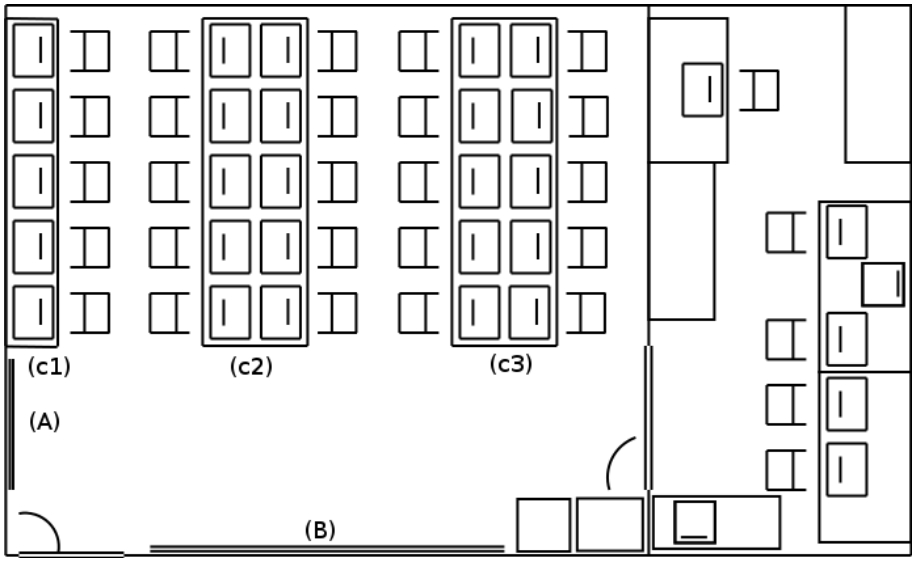


Figura 15: Layout Funcional da Sala do NSI

O desenho físico da sala do NSI foi projetado buscando o contato entre os desenvolvedores, e com os coordenadores.

Foi afixado um quadro branco grande na parede da entrada (B), para que todos os desenvolvedores possam participar de modelagem, discussões ou perceber avisos e notificações.

Na parede a esquerda, junto a porta, foi afixado um quadro radiador de informações (A), onde foi desenhado um quadro *Kanban*, compatíveis com a produção enxuta.

A sala é dividida entre desenvolvedores e professores coordenadores por uma divisória de vidro, para que o alunos possam chamar os professores quando necessário e isolamento suficiente para que os professores possa realizar outros trabalhos.

No espaço de entrada, maior a esquerda, os desenvolvedores cooperam nas bancadas em forma de U invertido, entretanto com comunicação muito próxima pelo pouco espaço da sala. Cabe observar que cada bancada (c1, c2 e c3) apresenta cinco ou dez assentos, com pleno acesso aos desenvolvedores sentados tanto na frente quanto atrás, o que proporciona espaço suficiente para discussões frequentes de 5 a 10 pessoas com facilidade.

Os requisitos especificados para uma sala de desenvolvimento, chamada de sala de guerra por Kent Beck (1999) , são:

. Espaço para que desenvolvedores sentem juntos para desenvolver em pares ;

. Espaço sobrando nas paredes para usá-las como radiadores de informação ;

. Espaço para que clientes possam circular e participar do desenvolvimento.

Segundo o pensamento enxuto o espaço deve ser compartilhado entre desenvolvedores que deverão poder sentar-se juntos com os colegas para ajudar, trocar experiências e se reunir para discussões.

5.2.3. Processo de desenvolvimento anterior

O processo de desenvolvimento de *Software* adotado originalmente já distinguia-se por não ser ortodoxo, distanciando-se dos modelos de maior desperdício citados neste trabalho como a aplicação dos conceitos de fases de desenvolvimento em lotes não unitários, com chaveamento de contexto e perda de foco e grande planejamento de escopo adiantado.

No processo de desenvolvimento anterior a equipe era gerenciada e dirigida por um professor, que assumia o papel de gerente do desenvolvimento, comandando as tarefas executadas pelos alunos. Nesta fase, vários projetos eram desenvolvidos ao mesmo tempo, com sobrecarga dos alunos mais experientes, aqui tratados de “veteranos”, que também tinham a incumbência de transferir conhecimento para os mais novos, aqui chamados de “aspiras”. Estes colaboradores, os aspiras, eram submetidos a testes para avaliação de suas habilidades, seu comprometimento e sua resiliência, e ao passo que atingiam os objetivos delegados, se tornavam aptos a assumir responsabilidades individualmente, com a colaboração dos veteranos quando necessário.

Estes projetos apresentavam resultados satisfatórios, rendendo frutos como publicações científicas, participação em congressos, e a satisfação dos clientes envolvidos.

O processo de desenvolvimento de *Software* em questão era adaptado do desenvolvimento iterativo e incremental, buscando a aplicação dos conceitos de engenharia de *Software* às características de trabalho, ao contexto dos projetos desenvolvidos no NSI, e ao bom senso dos professores e alunos que buscavam a melhor forma de desenvolvê-los.

Antes da utilização de métodos ágeis, as soluções eram discutidas com os pesquisadores e estes determinavam que tarefas deveriam ser realizadas e como. Diante do problema era discutida uma arquitetura de solução que seria seguida pelo bolsista que não possuía experiência para contestar. Os códigos eram desenvolvidos sem testes e eram verificados manualmente após o desenvolvimento. Um documento era desenvolvido no início do projeto com as soluções iniciais e durante o projeto estes planos mudavam dinamicamente a partir dos problemas encontrados.

5.2.4. Mudança do processo de desenvolvimento

A partir de 2008 o NSI passou a adaptar suas práticas de trabalho para o uso de Métodos Ágeis como o XP (BECK, 2001) e *Scrum* (SCHWABER & BEEDLE, 2002). A adaptação destas técnicas aos conceitos de Engenharia de Produção dos professores coordenadores redundou em um processo misto, semelhante ao *Scrum*ban (LADAS, 2008).

Tais métodos apresentam características muito similares e corroboram com conceitos originalmente encontrados no pensamento enxuto (WOMACK e JONES, 1998) e no pensamento sistêmico (SENGE, 2002), conforme relato pessoal do coordenador do NSI.

Ao notar as semelhanças encontradas entre o processo de desenvolvimento de *Software* resultante da aplicação da técnica de XP com o processo produtivo enxuto, o Professor Rogério Atem, estabelece uma nova linha de pesquisa e adota um orientando para que investigue o mapeamento do pensamento enxuto para o processo de desenvolvimento de *Software* sob a ótica da engenharia de produção.

O novo processo estabelecido então, atinge uma maturidade maior em 2010, quando é então monitorado para fins de avaliação científica.

5.2.5. Introdução do pensamento enxuto no ambiente

Para o projeto estimado, conforme discutido, as mudanças sugeridas não seriam muitas, tal é a adequação dos métodos ágeis ao Pensamento Enxuto.

Entretanto, algumas sugestões foram feitas:

- Manutenção de um lote unitário
- Desacoplar o desenvolvimento da criação de estórias
- Trabalho sob demanda
- Eliminação do conceito de iterações semanais
- Manutenção de um limite de trabalho em processo
- Adicionar uma coluna no quadro *Kanban* referente ao Teste de aceitação
- Deixar à cargo da equipe o comprometimento com as tarefas

- Manutenção de um lote unitário

Durante o desenvolvimento ágil utilizado no NSI, observou-se vários casos de comprometimento de desenvolvedores com mais de uma tarefa simultaneamente.

Estes casos sempre levaram a queda da qualidade do trabalho e, principalmente demora para entrega do conjunto de estórias assimiladas.

Cabe esclarecer que, após conversas explicando os fatores de risco, desempenho e qualidade envolvidos os trabalhadores retomavam a condição de execução de lotes unitários em uma única tarefa por vez.

- Desacoplar desenvolvimento e criação de estórias

Um dos problemas nos métodos ágeis é o acoplamento entre planejamento e execução de estórias. Nos métodos *Scrum* e XP, as estórias precisam ser planejadas em lotes, para serem desenvolvidas em lotes, gerando um acoplamento de fase. Para que o processo seja mais enxuto, as estórias devem ser tratadas individualmente, planejadas, priorizadas, desenvolvidas, tudo isso por demanda e em fluxo.

- Trabalho sob demanda

Recuperando o raciocínio, fazer por demanda é agir diante da solicitação do cliente, não aumentando o *Lead Time* com desperdícios. Neste ponto existe um *Buffer* de tempo imposto pelos métodos ágeis, no qual se espera pela próxima iteração. O fluxo fica portanto interrompido durante esta espera. Trabalhar em fluxo seria não haver demoras, ou desperdícios, tornando o *Lead Time* o menor possível.

- Eliminação do conceito de iterações semanais

A eliminação do conceito de iterações semanais libera o fluxo para ser executado sob demanda do cliente, sem esperas e sem limitações.

Quant. Trabalho (WIP) = Taxa Produção (N solicitações) x Tempo Percurso (Lead -*Time*)

Portanto:

Taxa de Produção = WIP / Tempo de Percurso

A baixa qualidade do trabalho citada é explicada pelo volume de trabalho ao mesmo tempo, forçando o trabalhador a acelerar seu ritmo de trabalho, perdendo foco do trabalho e possibilitando a geração de erros.

- Deixar à cargo da equipe o comprometimento com as tarefas

Várias tarefas eram atribuídas diretamente à desenvolvedores, em função do conhecimento prévio e de seu desempenho atual.

Diante desta prática decorriam alguns problemas já mapeados pelo pensamento enxuto:

- Concentração do conhecimento ;
- Gargalos de produção ;
- Baixa da qualidade do trabalho.

- Trabalho em pares

A concentração do conhecimento é o resultado do trabalho ser desenvolvido por uma pessoa solitariamente. Outros desenvolvedores ficam impedidos de participar por desconhecimento do domínio do problema e das soluções utilizadas. A solução para este problema é a formação de pares não fixos, conforme preconizado por Beck (2003), onde um especialista conhece o problema e outro aprende e tem a oportunidade de oxigenar as soluções conhecidas. Desta forma o conhecimento é repassado para outros integrantes da equipe e dificulta a criação de gargalos de execução.

Os gargalos de produção podem ser observados também pela ótica da Teoria de Filas e pela lei de Little (LITTLE, 1961).

- Mapa do fluxo de valor

O mapeamento do fluxo de valor do processo de desenvolvimento considerou como fornecedores os órgãos que irão utilizar o programa após terminado, cujas solicitações (input) formarão um estoque de mudanças a serem implementadas. Estes mesmos bibliotecários também foram considerados os clientes do *Software* que foi produzido (output). Desta forma, o mapa do fluxo de valor leva em consideração a interface entre o coordenador e estes fornecedores/clientes.

O processo para o desenvolvimento de funcionalidades foi realizado em iterações semanais, apresentando fluxo somente dentro da janela semanal. O controle da quantidade de trabalho em processo foi baseado na relação entre esforço de desenvolvimento e capacidade da equipe, sendo portanto dinâmico em função da complexidade do trabalho realizado. Respeitando esta relação trabalho x

capacidade, foi levada em consideração a velocidade estimada pela equipe, e utilizada para planejar a quantidade de trabalho compatível com a iteração.

O Mapa da cadeia de valor do processo anteriormente descrito pode ser visto na Figura 16, abaixo.

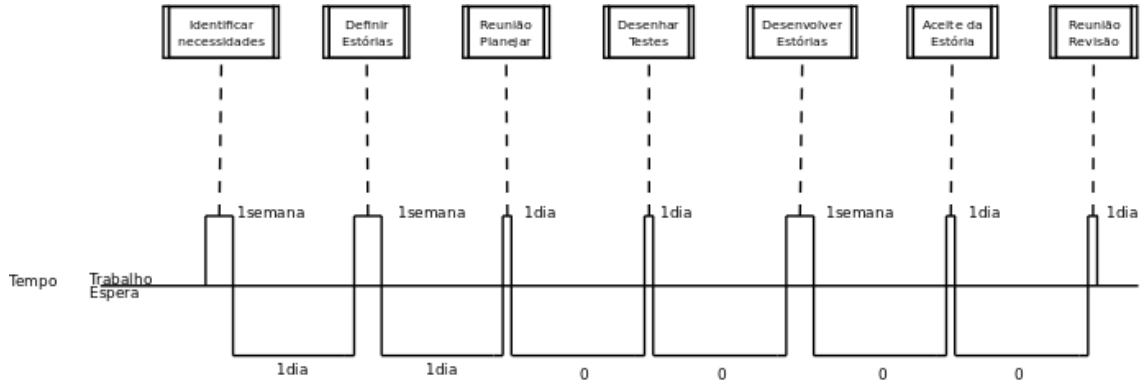


Figura 16: Mapa da cadeia de valor do processo ágil do NSI

Conforme apresentado no mapa de fluxo, existe a expectativa de que sempre seja gerado valores referentes às operações, aqui referidas como estórias, a cada semana.

Neste processo, as estórias foram planejadas pelo coordenador, com a participação da equipe de desenvolvedores, para que a cada semana as mesmas estejam finalizadas.

Cada desenvolvedor, individualmente ou em pares, buscou se responsabilizar por no máximo uma estória de cada vez, podendo obter outra estória somente após a conclusão da primeira.

- Introdução dos métodos ágeis no ambiente

Durante o ano de 2008, com a entrada de novos pesquisadores no NSI, foi recuperada a discussão sobre melhoria no processo de desenvolvimento de *Software*, motivada pelos conceitos sugeridos pela técnica de XP, que foram explicadas e avaliadas tendo em vista o início de sua utilização (CARVALHO *et al.*, 2010). As prática sugeridas nesta técnica provocavam uma reavaliação do processo produtivo porque refletiam coerência sob o ponto de vista de processos e da Engenharia de Produção.

Inicialmente foram introduzidas as práticas : entrega frequente, projeto simples, testes automatizados, refatoração, programação em pares, propriedade coletiva e padrão de programação.

Adicionalmente foi introduzido um quadro de gerenciamento visual, no qual eram dispostas as estórias a serem desenvolvidas, as tarefas desmembradas e os problemas encontrados.

Neste novo processo eram definidas estórias a serem realizadas pelos desenvolvedores, e a cada semana eram discutidos o andamento das estórias anteriores, os problemas encontrados, o escopo das novas estórias e os direcionamentos do núcleo.

Dentro de cada reunião, os problemas encontrados eram considerados por todos, sugestões emergiam pela troca de conhecimento, podendo inclusive gerar mudança de rumos e cancelamento das estórias caso fosse observado um risco maior na continuidade da execução.

Durante a apresentação das estórias novas os desenvolvedores eram cooptados a valorizar o esforço necessário para a execução das mesmas, informar sobre os riscos, dúvidas, tempo necessário para o desenvolvimento e motivação individual para o comprometimento com as estórias. Desta forma o risco era avaliado em conjunto, as estimativas eram feitas para atributar as estórias, tornando mais fácil para os coordenadores decidirem, após todas as informações disponíveis, sobre a prioridade de execução das estórias.

Após avaliar as estórias que permanecessem inacabadas da semana anterior e priorizar as estórias da semana atual, tendo em vista a capacidade de trabalho disponível, as estórias eram afixadas no quadro *Kanban*, visível a todos, para que cada um pudesse escrever seu nome e data no cartão referente a estória que pudesse se responsabilizar. As estórias assim adotadas eram repassadas para o estado de em desenvolvimento.

Para que uma estória pudesse sair do desenvolvimento para a posição de pronto, seria necessário a construção de todos os testes automatizados, entretanto alguns dos desenvolvedores ainda não haviam assimilado tais conceitos. Outros problemas que dificultaram a implantação de testes automatizados foram a falta de um ambiente de integração contínua que permitisse a todos perceber imediatamente erros no ambiente de desenvolvimento.

Outro problema inicial da equipe era a falta de padrões de programação, como o PEP-8 (PEP-0008, 2001), por exemplo. A solução para estes problemas foi a adoção de um circuito de seminários e palestras mensais, onde cada bolsista do núcleo descrevesse alguma técnica que houvesse estudado, visando homogeneizar o conhecimento entre os colegas.

A definição de tempo para a iteração causava também problemas, pois devido a complexidade dos projetos desenvolvidos, as estórias também variavam em complexidade e tamanho, tornando as iterações sempre incompatíveis com uma ou outra estória. Naturalmente os desenvolvedores ou se esforçavam demais para atingir os objetivos de suas estórias ou terminavam cedo demais, diminuindo a eficiência do gerenciamento do processo. Quando as estórias estouravam o tempo da iteração programada, sobrava tempo na iteração seguinte, gerando os efeitos conhecidos pela corrente crítica (GOLDRATT, 1998). Outro problema detectado nesta época foi que alunos se responsabilizavam por mais de uma estória ao mesmo tempo, incorrendo nos problemas advindo das Multitarefa nociva, discutida também por Goldratt (1998).

Com o tempo, a demanda pelas estórias passou a ser mais dinâmica, com a disponibilidade de estórias na posição de Backlog do quadro e os alunos puderam passar a obter estórias quase independentemente da iteração, sabendo que não iriam terminar a tempo de uma reunião, mas que poderiam apresentar os resultado obtidos na reunião seguinte.

Conforme discutido anteriormente, pode-se observar que a iterações temporais, mesmo ainda vigentes, são intuitivamente desconsideradas pois na prática forçam ritmos de trabalho não naturais, impedindo encontrar o ritmo de trabalho mais adequado.

- Quadro Kanban

O quadro de gerenciamento visual adotado era composto de 3 áreas, a saber:

- Backlog de estórias: uma lista simples de estórias a serem implementadas no futuro;
- *Workflow* do processo com as colunas: To Do, Doing, Done (Figura 17).
- Gráfico *Burn-Down* do trabalho já executado e ainda a ser executado

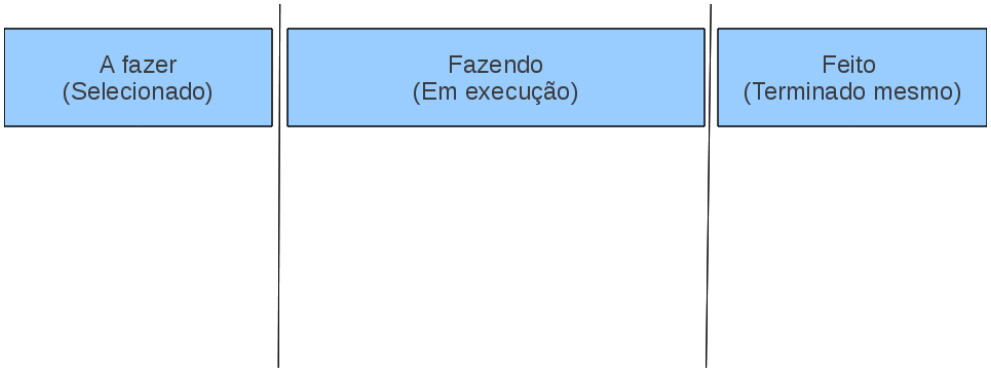


Figura 17: Quadro Kanban do projeto no NSI.

No quadro *Kanban*, a coluna “To Do” era utilizada para disponibilizar as estórias selecionadas pelo professor coordenador, enquanto ainda não houvesse um aluno comprometido com a estória, a coluna Doing representando estórias já associadas a algum desenvolvedor e portanto em desenvolvimento, e a coluna Done representando estórias já terminadas.

Infelizmente os conceitos de Done não eram atingido perfeitamente ou por todos, gerando códigos com muitos erros, normalmente não monitorados por testes, sem revisão, entre outros problemas.

- Caracterização do novo processo

O novo processo de desenvolvimento estabelecido a partir de 2010, utiliza técnicas oriundas do XP (BECK, 2001) e gerenciamento semelhante ao *Scrum* (SCHWABER & BEEDLE, 2002). Neste processo, o escopo é avaliado sob a ótica de estórias, que deverão representar valores úteis para o projeto, e que caibam em uma iteração semanal.

O professor que gerencia o projeto realiza um planejamento para cada reunião semanal, com vistas a identificação estórias que possam gerar valor para o projeto. A identificação de valor é considerada como ponto fundamental pois uma estória mal dimensionada representa risco para o projeto. As estórias são especificadas com vista aos seguintes objetivos:

1. atingir valores necessários ao projeto (objetividade, retorno de investimento);
2. de forma coesa e desacoplada de outras estórias (independência);
3. ser exequíveis pela equipe (realismo);
4. o esforço de desenvolvimento deve caber em uma semana (*Time*-Box);
5. apresentar um critério de aceitação bem definido (objetivo bem definido);

Durante o planejamento das reuniões semanais também são avaliadas as estórias que estiverem em produção atualmente. São avaliados:

- O esforço estimado inicialmente pelo desenvolvedor;
- O estado atual do desenvolvimento da estória, visível no *Kanban*;
- O tempo que a estória está em desenvolvimento (se maior que uma semana);
- A ocorrência de problemas durante a iteração em cada estória;
- A ocorrência de eventos que possam causar distúrbio de forma geral no NSI.

São então realizadas reuniões de sincronismo e planejamento com a equipe, visando acompanhar a evolução das estórias da semana anterior, a solução de problemas, o planejamento da iteração seguinte e a avaliação do processo, com a revisão do aprendizado durante a semana.

As estórias executadas pela equipe na semana anterior são apresentadas pelos desenvolvedores envolvidos. Caso tenham ocorrido problemas durante a execução das estórias, estes são recuperados e anotados para discussão ao final da reunião.

As estórias que não tiverem sido concluídas deverão ser avaliadas pela equipe e pelo professor se devem continuar durante e próxima semana, observando os problemas encontrados, a viabilidade de conclusão, as necessidades em termos de recursos, a importância da estória, e que sugestões são apresentadas pela equipe, visando mitigar o risco durante o processo.

Em seguida as estórias novas são apresentadas para que a equipe discuta o esforço necessário para a execução de cada estória. Esta discussão inicialmente mostrou uma tendência de que os desenvolvedores se comprometessem prematuramente com estórias, mas esta prática demonstrou-se prejudicial.

O comprometimento prematuro de desenvolvedores com estórias gerou eventos em que estórias não eram executadas por outros desenvolvedores livres, por serem consideradas responsabilidade já definida de um desenvolvedor específico, gerando com isso atrasos e ineficiência do processo, pela sobrecarga de alguns desenvolvedores e simultânea disponibilidade de outros. Durante a reunião ocorrida no dia 13/13/1913 foi detectado o problema ao se discutir as estórias da semana anterior e foi alertado pelo autor que tal fato estaria relacionado a decisão

prematura (POPPENDIECK & POPPENDIECK, 2003), ao padrão da multitarefa nociva (GOLDRATT, 1998), e a sobrecarga de trabalho em processo (ANDERSON, 2010). A partir desta reunião foi decidido que cada desenvolvedor deveria se comprometer com estórias somente no momento em que tivesse terminado sua estória anterior.

Outro problema identificado nesta mesma reunião foi a falta de comunicação no momento em que ocorrem problemas, pois os problemas ocorridos durante a semana não eram participados ao professor que gerenciava o projeto. O pesquisador autor observa que o quadro *Kanban* deve sinalizar a ocorrência de problemas visualmente (ANDERSON, 2010), para que todos possam perceber imediatamente que a linha de produção está parada (SHINGO, 1996).

5.2.6. Planejamento inicial – Iteração 0

Após coletar estas informações o coordenador convoca uma primeira reunião com a equipe, tendo em vista a apresentação do contexto, informação dos valores identificados, necessidades e solicitações documentadas.

Para realizar as reuniões do desenvolvimento da BD era alocada uma sala uma vez por semana, na qual o coordenador, o pesquisador e a equipe pudessem se reunir.

Durante as reuniões, segundo as práticas ágeis e em concordância com a filosofia do pensamento enxuto, a presença de todos é necessária para que se possa apresentar os valores a serem alcançados, descrevendo o porque de sua importância, para que todos busquem a satisfação do usuário como o objetivo principal, independente das questões tecnológicas ou burocráticas da metodologia utilizada.

Com base nestes documentos foram desenvolvidas estórias a serem executadas pelos desenvolvedores da BD. Além destas estórias, formam desenvolvidas também estórias técnicas, derivadas de observação do coordenador sobre o contexto do que foi solicitado, antecipando necessidades baseadas no conhecimento adquirido dos valores desejados pelos bibliotecários, para realização de consertos de problemas conhecidos e implementação de melhorias para o ambiente.

Estas estórias foram escritas em cartões que foram posteriormente fixadas no quadro *Kanban* utilizado para o gerenciamento visual. A Figura 18 apresenta o padrão da frente do cartão utilizado para estórias.

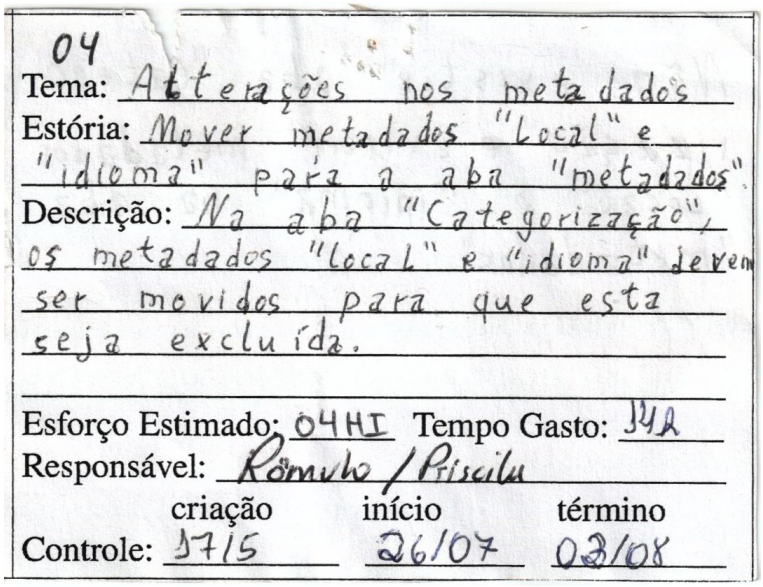

04
Tema: Alterações nos metadados
Estória: Mover metadados "Local" e "idioma" para a aba "metadados".
Descrição: Na aba "Categorização", os metadados "local" e "idioma" devem ser movidos para que esta seja excluída.
Esforço Estimado: 04HI Tempo Gasto: 34h
Responsável: Rômulo / Priscila
criação início término
Controle: 17/5 26/07 02/08

Figura 18: Cartão de Estória do Projeto BD

Neste cartão são dispostos campos necessários ao entendimento das estórias e a descrição dos eventos associados ao processamento da estória. No verso do cartão são especificados os critérios de aceitação da estória, que são declarações de ações propostas para quebrar o código associado ou a funcionalidade implementada, e mesmo validar se o valor desejado pelo cliente realmente será entregue.

Os campos apresentados na estória são descritos abaixo:

- Tema: Deve possibilitar gerar foco e orientação geral para as estórias da iteração. A utilização deste campo é oriunda de metodologias como *Scrum* e XP, não sendo de todo necessário em *Lean* ou *Kanban*. Entretanto, é interessante que os colaboradores da equipe estejam focados em problemas semelhantes para que eventuais necessidades de ajuda não gerem uma mudança de contexto muito severa.
- Estória: Denominação da estória atual. Deve descrever o problema a ser abordado de forma curta e possibilitar que a equipe e o coordenador possam se referir a ela.

- Descrição: Detalha o que se pretende com a estória, contextualiza, define objetivos, ou quaisquer outros detalhes para alertar ao desenvolvedor ou aos clientes sobre o que será realizado.
- Esforço Estimado: Neste campo os desenvolvedores em grupo devem definir a estimativa inicial de esforço para a execução da estória. Neste projeto foi adotado o uso de 2 horas ideais para cada dia trabalhado, desconsiderando-se sábados, domingos e feriados.
- Tempo Gasto: Ao término da execução da estória, ao realizar o aceite da estória, o coordenador anota as datas de término e tempo gasto em cada estória, lembrando que cada dia é considerado como 2 horas ideais.
- Responsável: Neste campo o desenvolvedor que se comprometer em executar a estória preenche seu nome, no momento que obtiver a estória, que deverá estar na posição inicial do quadro *Kanban*.
- Controle de Criação: Neste campo o coordenador escreve a data em que a solicitação obteve prioridade para ser executada. O objetivo deste campo é observar quanto tempo uma estória com prioridade leva para ser atendida. Infelizmente o nome deste campo não foi sugestivo o suficiente e não pode ser utilizado neste estudo por falta de explicação de seu valor e consistência em seu preenchimento.
- Controle de Início: Neste campo o desenvolvedor que se comprometer com a estória anota a data em que começou a trabalhar.
- Controle de Término: Neste campo o coordenador preenche a data na qual considerou aceito o término da estória. Neste dia o próprio coordenador ou algum desenvolvedor designado deverá tentar quebrar a funcionalidade desenvolvida ou modificada, utilizando pelo menos os critérios de aceitação previstos no verso do cartão.
- Critérios de Aceitação (Verso do Cartão): Estes critérios são definidos na criação da estória, atualizados ao longo da execução pelo próprio desenvolvedor comprometido, e principalmente, utilizado ao término do desenvolvimento com vistas a buscar problemas no produto. A busca por problemas e anormalidades visa manter a qualidade do produto.

– Priorização provisória das estórias

Após a criação das estórias, o coordenador precisava identificar quais as estórias deveriam ser desenvolvidas na semana. Para definir as estórias da semana o coordenador inicialmente avaliou as de maior risco, separando-as das demais. Separou adicionalmente as que gerariam maior satisfação aos clientes, conforme seu entendimento.

Avaliando estas estórias separadas, o coordenador submete-as aos desenvolvedores para obter uma estimativa de tempo ou esforço para estas estórias prioritárias.

À partir das estimativas de tempo para execução de cada estória foi possível definir quais estórias, dentre as mais importantes, foram disponibilizadas para o desenvolvimento na semana.

– Estimativa de esforço das estórias

Após a criação das estórias o coordenador reviu cada uma da estórias individualmente com a equipe de desenvolvedores. Para cada estória foi solicitada a definição de um valor de esforço, neste caso considerando cada dia como 2 horas ideais. Para cada estória os desenvolvedores em conjunto discutiram, resolveram dúvidas sobre os resultados a serem alcançados, sobre as técnicas a serem utilizadas, e decidiram em conjunto sobre a estimativa de esforço de cada estória isolada.

Durante esta atividade foram discutidos os possíveis problemas para atingir os objetivos, que soluções poderiam derivar novos problemas e quais ações de contingência poderiam ser tomadas. O tempo para desenvolvimento foi estimado tendo em vista o trabalho com qualidade. As estimativas não eram direcionadas para a produtividade de uma pessoa específica, mas para que qualquer um da equipe pudesse assumir o trabalho.

Após a estimativa de esforço produzida pelos desenvolvedores, o coordenador classificou as estórias nas seguintes ordens de grandeza:

- Minúsculas: poderiam ser resolvidas imediatamente, tomando menos de uma hora entre trabalho e verificação dos resultados. Por exemplo: Tradução de uma etiqueta de campo na tela.

- Pequena: que pudesse ser realizada entre um a cinco dias, considerando cada dia como 2 horas ideais. Por exemplo: um conserto simples de uma funcionalidade já existente.
- Média: uma estória que pudesse ser realizada em até 5 dias (10 horas ideais). Por exemplo: Implementação de uma nova funcionalidade.
- Grande: Uma estória de pudesse ser realizada em até 10 dias. Por exemplo: Uma implementação de algo difícil ou que demandasse de pesquisa, entretanto sabidamente factível pelo conhecimento da equipe.
- Extra Grande (XL): estórias que passam de 10 dias. Por exemplo: estórias consideradas grandes, que entretanto se tomaram muito maiores do que o esperado, exigindo tentativas e erros, pesquisa, conversa com especialistas sobre assuntos desconhecidos pelos integrantes da equipe.
- Desconhecido: estorias que a equipe mesmo inicialmente não fazia ideia do tempo necessário para o desenvolvimento.

Todas as estórias foram novamente estimadas na reuniões de planejamento futuras, sendo as de tamanho desconhecido reestimadas futuramente.

As estimativas minúsculas foram consideradas como 2 horas ideais (ou um dia) pois seria necessário que algum membro da equipe executasse a estória e que algum outro colaborador verificasse sua execução, sendo portanto todas as minúsculas transformadas em pequenas.

As estórias consideradas pequenas possuíam esforço estimado de até 4 horas ideais (dois dias). Tais estórias eram inicialmente relacionadas a tarefas bem simples e normalmente realizadas pelos desenvolvedores mais novos. Entretanto, as estimativas de tamanho de estória pequenas foram os maiores erros de estimativa, representando um total de 33% de erro. Dentre as noventa e sete (97) estórias pequenas realizadas somente sessenta e oito (68) foram realmente pequenas, trinta e duas (32) mudaram de tamanho ao serem executadas, sendo dezoito (18) tornaram-se médias, onze (11) tornaram-se grandes e três (3) tornaram-se muito grandes. Ocorreram ainda três estórias de outras classe que migraram para a classe de Pequenas durante a execução .

As estórias de tamanho médio eram estimadas para estórias de esforço considerável, porém que pudessem ser realizadas rapidamente. Uma estória

apresentava tamanho médio se demandasse entre 5 e 9 horas ideais, tendo ocorrido poucas estimativas dentro deste intervalo de esforço. Como resultado da execução destas estórias, observou-se que dentre as doze (12) estórias originalmente planejadas como médias, três (3) eram na verdade muito grandes (XL).

As estórias de tamanho grande eram originalmente doze (12) entretanto não foram tão problemáticas quanto as de tamanho XL.

Estórias de tamanho grande foram consideradas com esforço de execução entre 10 e 19 horas ideais. Estas estórias estiveram relacionadas a serviços grandes porém conhecidos.

A partir desta definição de esforço, o coordenador selecionava que estórias deveriam ser executadas e as priorizava, tendo em vista os critérios de risco, valor esperado, dificuldades técnicas envolvidas.

Este processo foi realizado manualmente pelo coordenador, sem utilização de artifícios, entretanto aceitando opiniões da equipe.

5.2.8. Procedimentos adotados no início e fim das iterações

A primeira atividade a cada iteração era a identificação dos problemas e impedimentos ocorridos durante a(s) semana(s) de trabalho. Surgiram dúvidas técnicas que impediram a execução das estórias, sendo necessário que algum dos desenvolvedores mais experientes fosse deslocado para ajudar na estória com problemas.

As estórias não terminadas na semana eram automaticamente mantidas e aguardadas para a próxima semana.

Cabe notar que o regime de reuniões semanais funcionou muito bem para o replanejamento e mitigação de riscos, entretanto os limites semanais para trabalho não eram eficientes.

A quantidade de estórias disponibilizadas semanalmente tem a probabilidade alta de exigirem mais ou menos esforço do que o disponível para uma semana. Em ambos os casos, falta de tempo ou sobra de tempo são consequências constantes, gerando falta de trabalho e sobrecarga de trabalho, corroborando com o conceito de desperdício (POPPENDIECK & POPPENDIECK, 2011; ANDERSON, 2010).

Após discutir as estórias terminadas, o coordenador separava por alto risco, complexidade e retorno de valor as estórias que poderiam ser executadas nesta

nova semana. Solicitava então que os desenvolvedores estimassem o esforço para estas novas estórias utilizando horas ideias.

Estas estimativas serviam também como fator componente para a decisão. A partir do esforço estimado para execução das estórias, o grau de dificuldade de cada um, do risco de não completá-la, e da estimativa de satisfação e retorno de cada estória o coordenador priorizava as estórias que poderiam ser terminadas dentro de mais uma semana de trabalho.

- Retrospectivas do trabalho

Com objetivo de assimilar conhecimento sobre o trabalho realizado, a equipe de qualidade ágil ajudava a cada reunião a fazer a retrospectiva do trabalho da semana, identificando fatos positivos e negativos sobre o trabalho e melhorias a serem implementadas.

Esta prática foi fundamental para o aprendizado e evolução da equipe, do gerenciamento e do processo.

A equipe expôs dúvidas que puderam ser discutidas, possibilitando o melhor entendimento do processo. Sem o espaço para tais discussões não se poderia garantir uma visão unificada do processo.

No STP, existe a prática de *Kaizen* ou melhoria contínua, conforme já descrito nesta dissertação. O *Kaizen* busca dar mais um passo em direção a perfeição. Para sua execução são realizadas reuniões e treinamentos para que os colaboradores tenham conhecimento e clareza do processo, e possam expressar livremente seu ponto de vista e suas ideias

Os fatos positivos foram muitos, entre eles: duvidas sanadas, aprendizado sobre aspectos técnicos antes desconhecidos, o apoio de um colega mais experiente, vencer desafios em menor tempo do que estimado inicialmente, entre outras situações.

Os fatos negativos apresentados foram, por exemplo: Dificuldade na resolução de problemas técnicas, falta de empenho de um colega pareado, doença, atraso no recebimento de bolsa de estudos (ocorreu um atraso no pagamento de todos os bolsistas).

As melhorias sugeridas servem como alvo a serem alcançados na próxima iteração. Durante a reunião é discutido o processo e como se pode efetivamente

melhorá-lo a luz dos problemas vivenciados. Como resultado devem ser sugeridas melhorias a serem implementadas.

Na Tabela 11 são descritos alguns fatores levantados nas retrospectivas realizadas.

Tabela 11: Exemplos de Resultados das Retrospectivas

BOM
Cor diferenciada para as tarefas
Apoio da equipe
Alguns testes novos criados
Testes antigos ok
Pegar tarefas - menos burocracia
Buildout funcionou
Planejamento bom
Produziu bem, por demanda
"Programador da iteração" = MJ
Apoio do pessoal mais experiente
Correr atrás individualmente
Bolsa saiu

RUIM
Muitos problemas com teste antigos
Estimativas muito otimistas das tarefas
Atraso na reunião
Prazo nas tarefas mal dimensionado
Dificuldades por inexperiência
Testes (falta BDD)

MELHORIAS
Enviar o documento completo com as histórias e descrições
Implementar o uso de cartão vermelho no quadro de taredas para Visualização e retirada de duvidas
Commitar com mais frequencia (maior possível)
Reuniões menores
Efetuar uma tarefa por vez (procurar só começar uma tarefa após terminar outra)
Retrospectiva vem primeiro na reuniao da iteraçao (olhar para trás primeiro)
Planning poker – melhorar estimativa das histórias
Funcionará por demanda (Expectativa: pelo menos uma tarefa 100% concluída até o fim da iteração)
Mudança nos pares: passar a trabalhar um membro antigo com um novo
Kanban digital
Prazo nas tarefas
Sem par nessa iteração

Obs.: Nesta tabela os alunos citam Histórias que são descritas neste texto como estórias.

Os resultados obtidos pela retrospectiva se tornaram um alvo para todos os elementos que participaram deste evento, pois todos receberam espaço para

comunicar suas opiniões, evidenciando problemas, oportunidades de melhorias e possibilitando o sincronismo de objetivos.

A execução de retrospectivas foi adequada para viabilizar a auto-organização da equipe, ocupando espaço que alguém em posição hierárquica superior ocuparia para identificar problemas, soluções, gerenciar conflitos, etc. Desta forma, o próprio grupo assumiu esta responsabilidade gerencial, dando confiança e tornando rotineira a manutenção da união.

- Rotina de trabalho após a reunião

Ao final da reunião os desenvolvedores entravam em acordo sobre o que cada um deveria fazer, cabendo sugestão do coordenador, porém sem delegação ou comando.

As estórias eram afixadas no quadro *Kanban*, na posição de selecionadas para execução, em ordem vertical de prioridade.

Cada estória era então selecionada por algum desenvolvedor, que assinava seu nome no campo responsável, anotava a data de início de execução e afixava a estória na posição “Em Execução” do quadro *Kanban*.

- Erros durante o processo

Durante a semana, caso ocorresse algum problema, dúvida ou identificação de novos riscos, o desenvolvedor ou a dupla, poderiam buscar ajuda de outros desenvolvedores na baias de trabalho vizinhas ou via internet.

Caso o problema fosse categorizado como grave o desenvolvedor chamava atenção do coordenador e solicitava sua interferência no sentido de associar formalmente outros desenvolvedores para ajudar a resolver o problema, ou para buscar informações ou para sugerir outros caminhos.

Durante o estudo realizado, os desenvolvedores esforçaram-se para desenvolver utilizando as técnicas de TDD e BDD. Entretanto este objetivo ainda não foi totalmente alcançado em função dos seguintes impedimentos:

- A plataforma na qual a BD foi construída (Plone) não oferece facilidades para aplicação de testes automatizados;

- Parte do código lida com problemas não muito comuns, como por exemplo fragmentação de vídeo, o que torna mais difícil desenho e utilização de testes automatizados;
- Grande parte do projeto já desenvolvido não possui testes;
- Grande parte das funcionalidades coberta por testes apresenta problemas com relação a recursividade de testes por falta de utilização de dubles de código, forçando a realização de testes em profundidade, ocasionando sobrecarga de processamento e lentidão nestes.

Entretanto as novas funcionalidades desenvolvidas, em sua grande maioria, receberam testes automatizados antes de serem construídas. Portanto, vários dos problemas encontrados durante o desenvolvimento já são resultados da melhoria de qualidade buscada.

Durante todo o desenvolvimento os bolsistas foram orientados a trabalhar de forma iterativa, deixando o código sempre em condições de compilação. Está prática, conhecida como passos de bebê (MARTIN, 2002), força a evolução em etapas menos complexas, guiando o desenvolvimento por etapas simples, e complementando o entendimento dos métodos ágeis.

Desta forma, ao terminar cada iteração os desenvolvedores eram orientados a realizar pelo menos uma operação de *Commit* por dia, tornando visíveis ao coordenador e aos colegas os resultados de seu trabalho. Estas políticas se complementam, minimizando o risco do desenvolvimento por caminhos mais complexos, possibilitando que o código se mantenha simples e que todos sejam capazes de entender e operar modificações quando necessário.

Ao realizar a operação de *Commit*, sistema de controle de versões local automaticamente ativa o sistema de integração contínua, que executava os testes do sistema, tornando disponível o último estado de erro/sucesso de cada um dos testes do sistema.

A integração contínua, conforme já discutido, foi um objetivo de longo prazo para o NSI, sendo hoje uma ferramenta importantíssima na garantia de qualidade de todos os projetos lá desenvolvidos.

- Entregas no prazo x entregas além do prazo

Analisando os resultados das estórias, observou-se que do total de estórias executadas, 51% foram executadas dentro da estimativa inicial, entretanto 85% das estórias terminaram com um atraso máximo de 3 dias.

Os atrasos na entrega de estórias foram atribuídas as dificuldades técnicas que os desenvolvedores mais experientes também não conseguiram resolver.

- Erros encontrados no teste de aceitação

Qualquer erro ou suspeita de erro encontrada é valiosa e deverá tornar-se uma nova estória a ser entregue ao pesquisador/coordenador, para ser priorizada.

Erros encontrados no teste de aceitação eram reportados em reunião e eram discutidos em profundidade. Tendo como exemplo o STP, tais erros demonstram não somente um problema operacional, mas sim uma falha do processo, sendo necessário portanto ações corretivas, como cobrança de mais testes, mudança dos pares, seminários e palestras sobre o assunto, maior tempo para realizar a tarefa, entre outras políticas a serem discutidas caso a caso.

Várias das estórias executadas durante o período de estudo foram reportadas pelos próprios desenvolvedores, sendo identificadas com a ajuda de TDD. A utilização da integração contínua também teve papel fundamental neste processo, possibilitando identificar os problemas ainda durante o desenvolvimento das estórias, não deixando que estórias fossem terminadas enquanto ainda houvessem erros possíveis de serem identificados automaticamente pelos testes pré estabelecidos.

Nos casos em que um erro foi encontrado durante a fase de UAT, ocorreu uma parada do desenvolvimento, o próprio testador devolveu o problema para quem o desenvolveu, e a estória voltou para a posição de desenvolvimento.

Ocorreram casos em que os desenvolvedores identificaram problemas mais graves, e chamaram outros desenvolvedores para realizar um exame em torno do problema, agregando mais cérebros e buscando uma melhor arquitetura de solução.

Ao terminar o cartão retorna à coluna do quadro *Kanban* de espera UAT.

Após realizar todos os testes possíveis, o desenvolvedor confiante da estabilidade da estória testada, anotava no cartão a data em que terminaram os testes e repassava o cartão para a coluna de pronto.

5.2.9. Término do desenvolvimento

Ao terminar o trabalho, o desenvolvedor leva o cartão da estória pela qual estava responsável, anota a data de término e repassa a estória para a coluna de teste de aceitação.

Esta estória, caso tenha sido executada utilizando-se testes automatizados, já estará livre de erros mais graves, sendo portanto o teste de aceitação uma operação muito mais intelectual e menos repetitiva do que as etapas de testes de outros processos de desenvolvimento de *Software* tradicionais.

Para validar e verificar uma estória que esteja disponibilizada na coluna de aceitação do quadro *Kanban*, algum desenvolvedor livre poderá selecionar a estória, anotar seu nome e a data em que puxou a estória para aceitá-la e posicioná-la na coluna de UAT (Unit Acceptance Test), ou testes de aceitação de unidades.

O procedimento a ser realizado nesta etapa era ler o cartão, buscando entender o que havia sido desenvolvido, ler principalmente o verso do cartão, onde já deveriam estar especificados testes a serem realizados, e tentar imaginar ainda mais outros testes que possibilitassem quebrar a funcionalidade.

O objetivo a ser alcançado é encontrar erros direta ou indiretamente relacionados com as funcionalidades em avaliação.

- Nova reunião semanal de verificação e entrega de novas estórias.

Conforme já mencionado, nas reuniões semanais eram discutidos os problemas, revistas falhas graves que porventura tenham ocorrido e repassadas novas estórias para o desenvolvimento.

5.2.10. Kanban Eletrônico

A visualização das tarefas no quadro Kanban proporcionou uma maior integração entre a coordenação do projeto e os desenvolvedores. Entretanto os principais stakeholders do projeto não tinham visibilidade do quadro físico. Outro problema era que as fichas kanban escritas em papel poderiam ser perdidas e ofereciam pouco espaço para a especificação do problema, possíveis soluções, testes desejados para a aceitação da estória, bem como a coleta de informações e estatísticos para a avaliação do processo.

Como solução para estes problemas foi utilizado um sistema Kanban eletrônico (Figura 19), que proporcionou o melhor acompanhamento pelos stakeholders e aumentou a qualidade da comunicação entre os coordenadores e desenvolvedores.

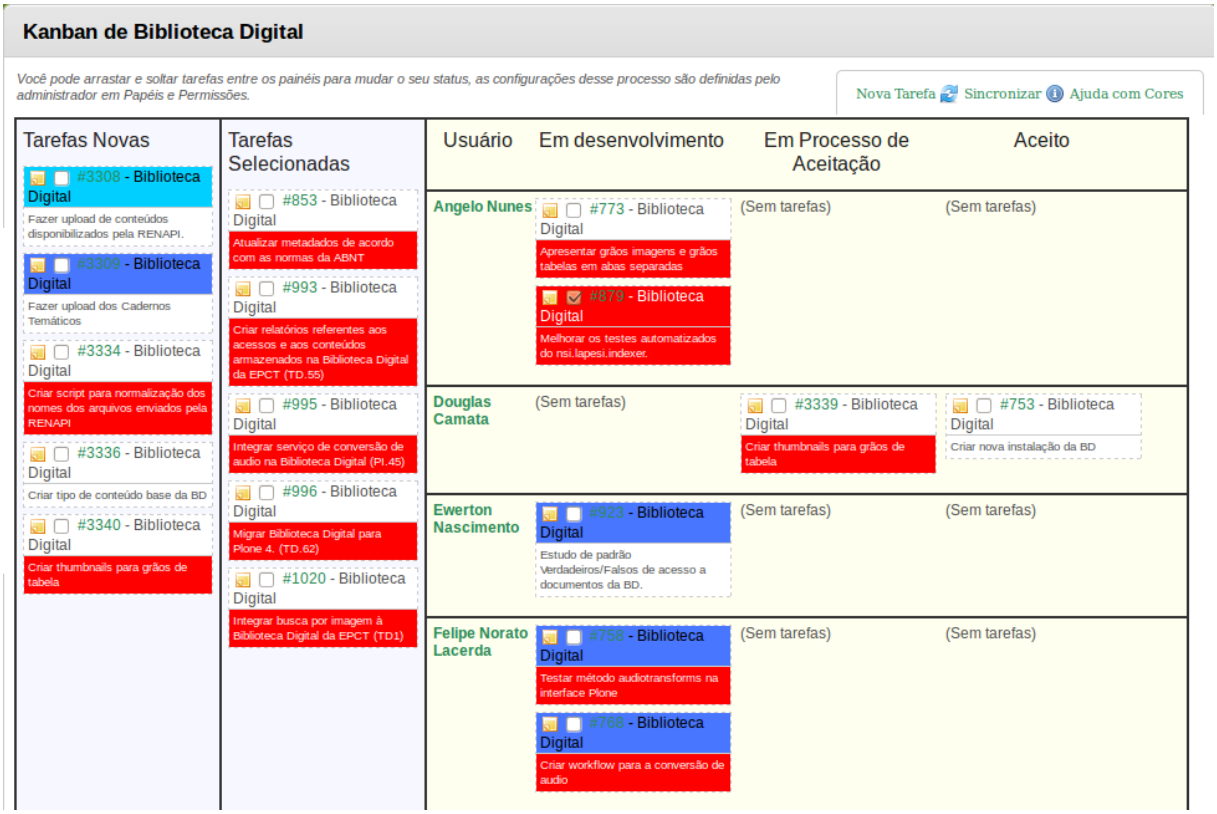


Figura 19: Velocidades das semanas

5.3. Planilhamento e Tratamento dos dados

Após o período de monitoramento foram avaliadas as variáveis do processo colhidas e compiladas observações sobre o trabalho analisado.

Cada estória realizada pelos desenvolvedores foi digitada em um planilha do sistema Calc do pacote OpenOffice versão 3.2, no sistema operacional Linux Ubuntu versão 10.10.

As estórias foram dispostas em linhas contendo as seguintes colunas:

- Inicio semana
- Fim semana
- Identificação da estória
- Tema da estória
- Nome da estória

- Descrição da estória
- Esforço Estimado
- Esforço Gasto
- Responsável
- Segundo Responsável
- Data de Criação
- Inicio Execução
- Término da Execução

O campo Inicio semana, recupera a informação de quando iniciou a semana de desenvolvimento. Nesta data ocorreu a reunião na qual a estória é apresentada e planejada junto aos desenvolvedores. Conforme discutido na apresentação do estudo de caso, no início do estudo de casos as iterações duraram 1 semana, entretanto no decorrer deste estudo, ocorreram iterações de duas semanas e ao final do estudo, as iterações tenderam a desaparecer, tornando-se um esforço realizado por demanda, puxado, mais semelhante à produção enxuta.

O campo Fim da semana, portanto, define o término formal da iteração e a data da reunião de revisão da iteração.

O campo identificação da estória faz menção ao número da requisição feita ao NSI pelos bibliotecários indicados pela SETEC/MEC, no documento de levantamento colhido pelo coordenador do desenvolvimento.

O campo Tema da estória define o foco das estórias a serem tratadas. O objetivo do tema é agrupar em um mesmo momento trabalhos correlatos, para que os desenvolvedores possam inclusive ajudar uns aos outros caso necessário.

O campo Nome da estória é uma referência para que todos possam se referenciar à uma estória univocamente.

O campo descrição da estória traz detalhes sobre o contexto, o problema, o objetivo e os resultados esperados. Este campo foi projetado também para trazer as informações padronizadas pelos métodos ágeis:

- Como um <Papel>
- Desejo <Funcionalidade>
- Para que assim eu possa <Resultados Esperados>

O campo Esforço estimado apresenta a estimativa de esforço defendida pelos desenvolvedores em conjunto, durante a reunião de planejamento no início da semana. Esta estimativa leva em consideração horas ideias, conforme descrito nesta dissertação, e considerada também que cada dia apresente 2 horas ideias.

O campo Esforço Gasto representa o número de dias totais de duração da estória vezes duas horas ideais por dia, conforme preenchido pelo desenvolvedor.

O campo Responsável apresenta o nome do desenvolvedor que executou a estória.

O campo Segundo responsável é utilizado quando as estórias são originalmente desenvolvidas em pares ou quando ao longo da execução da estória algum outro desenvolvedor se junta definitivamente para auxiliar na execução da estória.

O campo data de criação, como o nome declara, recupera a data em que a estória foi solicitada ao coordenador da BD. Seu objetivo é dar noção do custo real das operações e possibilita avaliar o *Lead Time* real das mesmas.

O campo início da execução é anotado pelo desenvolvedor e representa o dia em que o mesmo puxou a estória e com isso se tornou responsável pela mesma.

O campo término da execução é anotado pelo desenvolvedor que realiza a operação de aceite da estória, após ter testado e verificado se existe alguma forma de quebrar a funcionalidade.

A partir destes campos foram calculados os seguintes outros campos:

- Número de Ordem
- Categoria de serviço
- Tamanho Estimado
- Tamanho Real
- Esforço Real

O número de ordem é um número sequencial dado a cada estória para fins de visualização e controle.

A categoria de serviços é uma classificação criada pelo pesquisador buscando enquadrar cada uma das estórias em 3 classes, conforme descrito na Tabela 12:

Tabela 12: Classes de serviço definidas para a BD

Classe de serviço	Descrição
Erros	Concerto de alguma funcionalidade já existente
Melhoria	melhorando ou inovando uma funcionalidade já existente
Novo	Referente a criação de uma nova funcionalidade.

Estas classes seguem a proposta de classes de serviços (Class-of-Service) definida por Anderson (2010), com o objetivo de identificar o fluxo diferenciado de linhas de produção. Esta categoria foi criada por entender-se que o desenvolvimento de novas funcionalidades poderia apresentar desempenho de execução e mesmo um *Workflow* diferente do concerto de erros, por exemplo.

Anderson (2010) estende ainda este conceito para Service Level Agreements (acordos de nível de serviço), para que se possa negociar com clientes formas de tratamento diferenciadas por nível de serviço.

O tamanho estimado representa a classificação das estórias por tamanhos, conforme sugerido por Anderson (2010), utilizando os tamanhos P (pequeno), M (médio), G (grande) e XL (extra grande). Os tempos associados aos tamanhos são relacionados ao esforço para desenvolvimento da estória, segundo a Tabela 13, abaixo.

Tabela 13: Definição de tamanhos das estórias

Tamanho	Esforço
P	1-4
M	5-9
G	10-19
XL	>=20

O campo Tamanho estimado representa portanto a classe de tamanho da estória, avaliada sob a ótica da estimativa de esforço da equipe de desenvolvedores.

O campo Tamanho real, de forma similar ao tamanho estimado, representa a classe de tamanho da estória real, ou seja, calculada a partir do esforço real, calculado a partir do esforço real de execução da estória.

O Esforço real foi calculado pelo número de dias úteis entre o início e fim da execução da estória vezes duas horas úteis padronizadas por dia.

Com base nesta estrutura, os dados das estórias foram digitadas, as eventuais lacunas preenchidas após conversa pessoal com o coordenador e os desenvolvedores, e erros como inversões de datas, incoerências entre colunas localizadas e corrigidas.

A partir da tabela corrigida, foi desenvolvida uma estrutura paralela para dispor dia a dia a execução das estórias, criando uma representação similar à um gráfico de Gant. Desta forma foi possível extrair diversos dados como esforço total de cada semana, velocidade da semana, trabalho em processo (WIP), entre outras informações.

Esta planificação do trabalho também possibilitou encontrar problemas como a execução de duas estórias simultaneamente pelo mesmo desenvolvedor, o que causa o problema de multitarefa nociva, diminuindo o foco e a qualidade do trabalho e aumentando o risco de erros (GOLDRATT, 1998).

5.4. Resultados

Foram executadas portanto, entre maio e dezembro de 2010 um total de 124 estórias, sendo 52% das operações solicitadas pelos usuários diretamente. Estas operações foram também categorizadas entre novas funcionalidades, melhorias de funcionalidades já existentes, e concerto de erros. Os valores encontrados foram de 45% de melhorias, 23% de inovações e 32% de concerto de erros entre preexistentes e gerados ao longo do desenvolvimento.

As iterações foram desenvolvidas inicialmente com prazos de duas semanas, mas variando de acordo com a dificuldade das estórias envolvidas na iteração, a critério do professor coordenador, visando criar iterações na qual coubessem as estórias relacionadas. Por exemplo, a cada iteração, se as estórias priorizadas fossem orçadas pelos desenvolvedores para um tempo maior que uma semana, a iteração seria de duas semanas.

Cabe observar que, segundo o entendimento da comunidade ágil, quanto maior o tempo das iterações, menos madura a equipe. Por outro lado, mesmo praticando o desenvolvimento baseado em pensamento enxuto, podem ocorrer

estórias grandes, maiores do que uma semana, o que não pode ser julgado como um indicativo de maturidade da equipe.

5.4.1. Velocidade das estórias

A velocidade das estórias foi calculada pela soma dos esforços das estórias da semana. O gráfico a seguir (Figura 20) apresenta as velocidades calculadas das semanas.

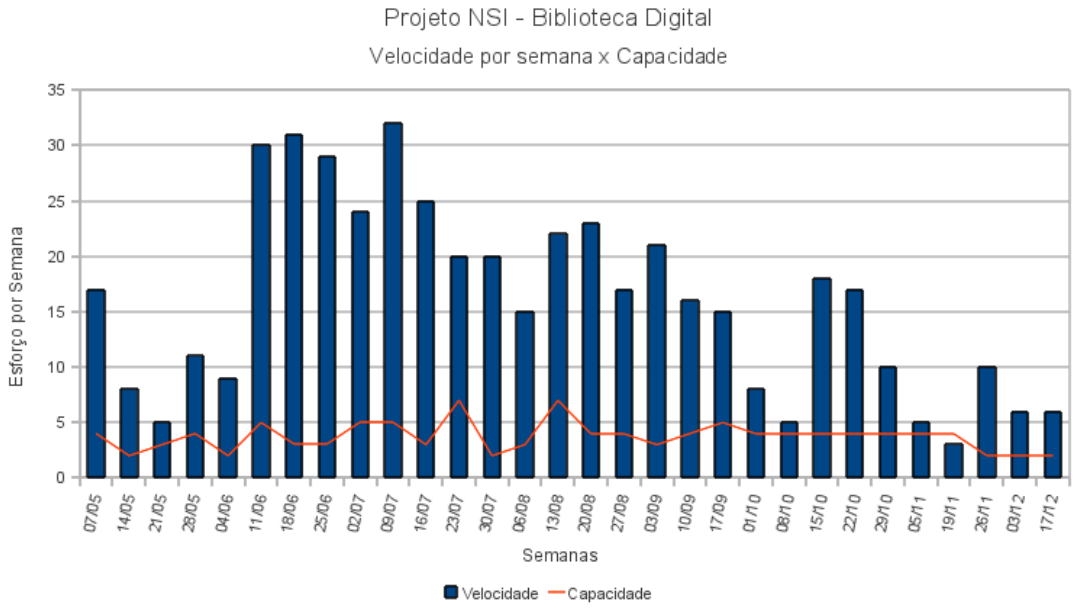


Figura 20: Velocidades das semanas

Estas velocidades apresentam uma variação considerável, em função dos seguintes aspectos:

- Os desenvolvedores neste projeto são alunos bolsistas, estando envolvidos com aulas, estudos e outros afazeres acadêmicos ;

- A própria política do processo, sugere a priorização de estórias mais importantes e de maior risco, possivelmente as maiores. Entretanto, para compatibilizar tais estórias com o período de uma iteração seria necessário que a iteração apresentasse tempos variados

- Durante a iteração, as estórias que couberam eram executadas, sobrando sempre fatias de tempo não utilizados em função da dificuldade natural em aproveitar o tamanho da iteração, ocorrendo sempre uma sobra de tempo livre considerável, fazendo que a velocidade da iteração varie.

Entretanto, a variação de velocidade apresentada, também é função dos diferentes tamanhos de estórias executadas. Em uma análise mais detalhada,

classificando as estórias por tamanho, pode-se verificar que a velocidade dentro das classes se comporta mais uniformemente (Figura 21).

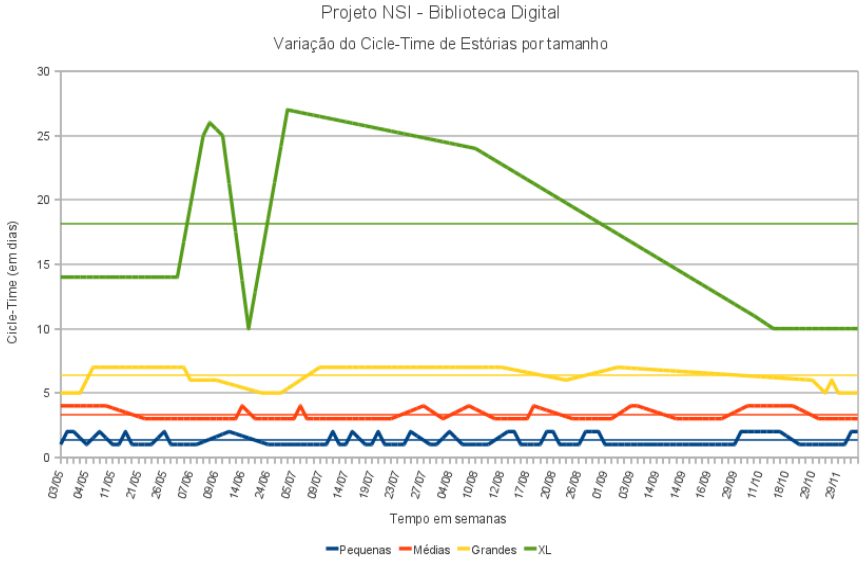


Figura 21: Variação na execução das estórias por categoria de tamanho.

A variação no formato de vale apresentada em 14/06 nas estórias XL diz respeito a estimativa sobre valorizada de um problema, que inicialmente entendia-se como de difícil solução, mas que foi solucionado rapidamente.

5.4.2. Tempo de ciclo (*Cycle Time*)

Conforme proposto pelo pensamento enxuto, o desenvolvimento não foi realizado em grandes lotes ou bateladas, mas sim em lotes unitários que foram executados do início ao fim sem interrupção. Portanto, o tempo de cada ciclo de engenharia variou conforme apresentado na Figura 20, já apresentada.

O tempo de engenharia variou durante os vários ciclos, entretanto a variação dentro das categorias de tamanho de estória apresentaram alguma convergência, com exceção da categoria XL, que engloba os problemas desconhecidos pele equipe.

Durante este trabalho pode-se observar uma baixa variação no tempo de engenharia das estórias nos tamanhos (P, M, G), sendo possível para a equipe estimar dentro de cada categoria com cada vez maior precisão.

O acompanhamento da execução das estórias, segundo os métodos ágeis, é feito através do gráfico de *Burn-Down* (Figura 22).

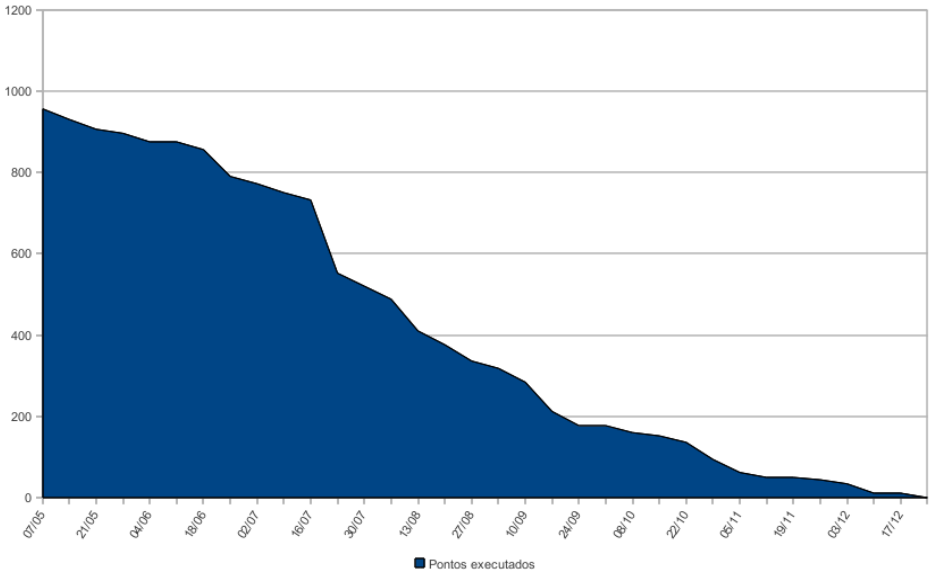


Figura 22: Burn-Down do estudo de caso

Entretanto, como pode-se observar, este gráfico apresenta o desempenho somente a velocidade em termos de pontos de estórias, não sendo possível identificar problemas nas operações.

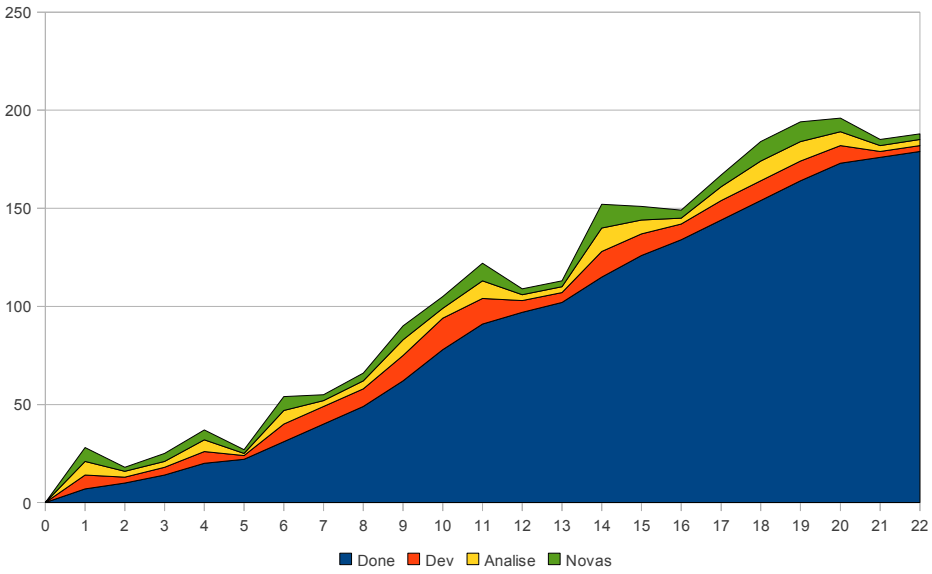


Figura 23: *Burn-Up* do estudo de caso

O gráfico proposto por Anderson (2010), apresenta cada uma das operações do fluxo de trabalho, com o volume de trabalho em processo a cada semana, sendo uma contribuição maior para o monitoramento e diagnóstico de problemas no processo (Figura 23).

5.4.3. Resultados obtidos pelo monitoramento de falhas

Durante o desenvolvimento foram utilizadas as técnicas de TDD e Integração contínua, apresentadas no tópico 3.6.6.

A técnica de TDD visa documentar e monitorar as expectativas desejadas e indesejáveis no *Software*. A técnica de integração contínua trabalha em conjunto, executando as especificações do TDD a cada vez que ocorre uma modificação no *Software*.

Estas modificações no *Software* são armazenadas em um banco de dados, contendo o nome do desenvolvedor, a data e hora, o arquivo que foi modificado, as diferenças entre a versão anterior e a atual, e uma descrição escrita pelo desenvolvedor resumindo o objetivo da modificação.

Entende-se que o uso das técnicas de TDD (visto no tópico 3.6.6) e Integração contínua agreguem valor para o processo de desenvolvimento, pois monitoram o código e oferecem um *feedback* imediato para os desenvolvedores assim que alguma modificação causa um erro no *Software*.

Por acreditar que o *Software* esteja monitorado, o desenvolvedor trabalha com mais confiança em mudar o *Software*, confiante em ser avisado instantaneamente sobre anormalidades detectadas.

Esta confiança se traduz em um estímulo para implantar melhorias e inovações, viabilizando o crescimento em um novo paradigma evolutivo além de diminuir o custo de concertos de erros detectados pelo cliente.

O tempo gasto em testes manuais, ou mesmo de outras formas de testes automatizados diminui consideravelmente.

Para medir o efeito da estabilidade no processo buscou-se traçar um gráfico que demonstrasse a variação ocasionada por erros no processo de desenvolvimento.

A expectativa foi de tentar monitorar no processo de desenvolvimento a ocorrência de três tipos de operações:

- Crescimento
- Melhorias
- Concerto

O crescimento caracteriza novas estórias, novas funcionalidades, código novo solicitado pelo cliente, essencialmente algo anteriormente inexistente.

As operações de melhoria são modificações em um código que já existia, entretanto pôde ser melhorado. A importância de monitorar melhorias é que em um processo instável, onde as mudanças no código fontes podem gerar erros em diversos ponto do *Software*, as melhorias são desincentivadas, visando mitigar riscos. Realizar melhorias afeta diretamente a qualidade do *Software*, entretanto aumenta o risco de erros e implica que todo o *Software* deverá ser novamente testado, gerando impactos em tempo e custo.

A ocorrência de erros causa um impacto mais direto no processo, pois além do tempo gasto inicialmente, será necessário mais tempo e alocação de serviços para encontrar os erros. Encontrar erros pode não ser uma tarefa trivial, podendo consumir mais recursos do que o processamento inicial.

Portanto o monitoramento gráfico realizado teve como objetivo verificar a tendência do processo em gerar erros ou crescer e melhorar.

- Coleta dos dados

Os dados foram obtidos do sistema de controle de versões utilizado durante o estudo de caso.

Estes dados foram importados para uma planilha de cálculo, onde foi necessário pouco esforço de formatação para adequá-los para trabalho.

As informações obtidas e dispostas em colunas foram:

. Número da alteração;

. Autor da alteração (desenvolvedor);

. Data e hora da modificação;

. Comentários sobre a modificação.

Foram identificadas palavras comuns utilizadas pelos desenvolvedores no campo de comentários, que possibilitaram a classificação das operações nas três categorias propostas.

A coluna de comentários foi investigada através de formulas para identificar os indícios que indicassem a categoria na qual a operação pertence.

O gráfico confeccionado pode ser visto na Figura 24.

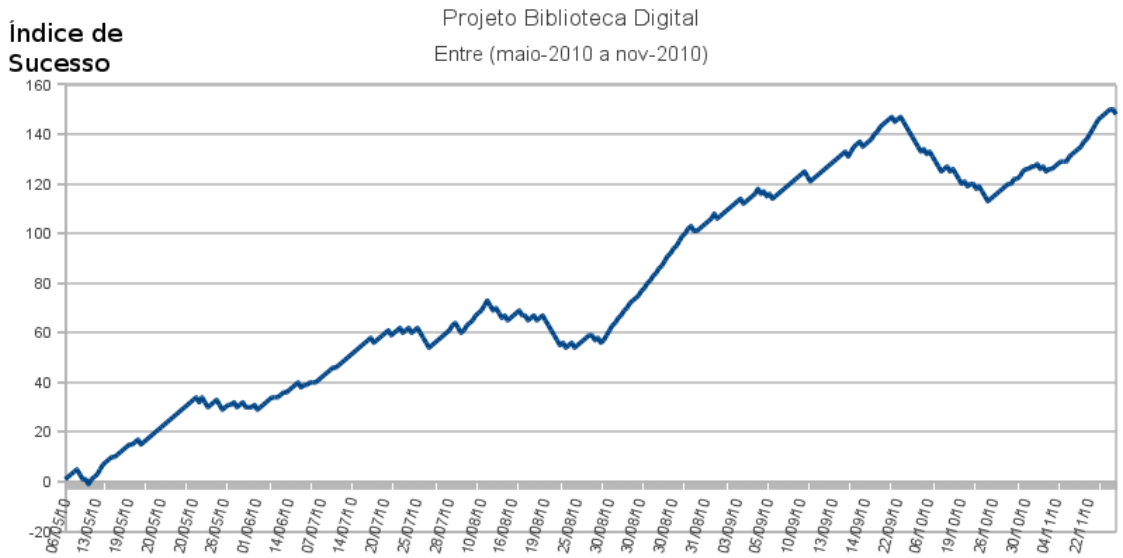


Figura 24: Evolução do Estudo de Caso (implantando TDD)

A evolução do gráfico representa o acumulado a partir de pesos atribuídos à cada categoria ao longo do tempo. Os pesos utilizados para cada categoria de operações foram:

. Crescimento (peso 2)

. Melhoria (peso 1)

. Erros (peso -2)

Estes valores foram decididos a partir de experimentação, com objetivo de observar o crescimento do *Software* durante o desenvolvimento a partir das categorias analisadas.

Neste estudo de caso foram detectados 66% de melhorias, 22% de erros detectados durante a codificação, e 12% de novas funcionalidades (Figura 25).

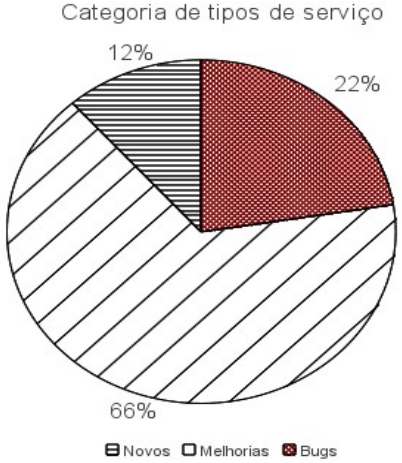


Figura 25: Concentração das categorias analisadas no estudo de caso.

O objetivo deste controle foi monitorar do processo visando observar a ocorrência erros, melhorias e crescimento do *Software* durante o estudo de caso. Os erros observados no gráfico foram detectados automaticamente, não tendo sido detectados erros nas operações seguintes do processo.

Foi observado pelos desenvolvedores que em períodos anteriores a esta pesquisa, na ausência da técnica de TDD, a ocorrência de erros foi maior, com o agravante de que vários erros eram detectados após o desenvolvimento.

Foi observado durante o estudo de caso que os desenvolvedores sentiram-se mais confortáveis e incentivados a melhorar o *Software* pela segurança oferecida pelo monitoramento automático oferecido pelo *Software*.

5.5. Trabalho em pares

Durante este estudo de caso foi observado a aprovação do trabalho em pares para o desenvolvimento de funcionalidades durante algumas semanas, porém não foi uma prática constante durante todo o período analisado.

Esta prática foi aplicada também com o objetivo de apresentar resultados mais rápidos ao detectar que uma estória específica seria mais demorada do que o estimado originalmente.

Outro motivo foi a solução de algum problema crucial ou de alto risco, para o qual não fosse adequado que um desenvolvedor trabalhasse sozinho, tendo como objetivo mitigar o risco de produzir um algoritmo de menor qualidade. Cabe esclarecer que diversas funcionalidades do projeto apresentam como requisito o desempenho elevado visando atender a demandas de escala nacional.

Como os desenvolvedores neste estudo de caso são alunos bolsistas, entendeu-se que se deveria utilizar programação em pares quando o recurso a ser desenvolvido fosse crucial para o projeto, visando portanto descentralização do conhecimento.

Percebeu-se que durante a programação em pares ocorre aprendizado para ambos os desenvolvedores, mesmo quando pareiam veteranos e novatos, pois ocorre o aprendizado ao ensinar, troca de conhecimentos técnicos, oxigenação dos hábitos e práticas e criam-se laços de amizade que ampliam as sinergias do grupo.

5.6. Conclusões deste estudo de caso

Não foi possível implementar todos os conceitos do pensamento enxuto, pois o projeto já estava em andamento e os desenvolvedores treinados para pratica *Scrum*.

Entretanto, baseado na literatura e na avaliação das práticas realmente utilizadas, entende-se que muitos dos princípios enxutos estejam presentes nas práticas observadas.

Este trabalho se posiciona como um estudo mais descritivo, não sendo o objetivo deste estudo de casos quantificar o impacto das técnicas, buscando-se somente exemplificar o mapeamento das mesmas para o processo de desenvolvimento de *Software*.

Foi possível observar as seguintes práticas:

. Produção em lotes unitários;

. Eliminação de desperdícios;

. Trabalho em equipe;

. Diminuição da cadeia de comando e controle;

. Autonomia ao desenvolvedor;

. Estimativa feita pelo próprio trabalhador;

. Ouvir a voz do cliente;

. Prevenir a ocorrência de erros.

Entretanto, não foi possível:

. Implementar experimentos empíricos, pois não havia dados históricos disponíveis com os quais se pudesse comparar os dados novos gerados durante o estudo de caso.

. A interferência no processo foi limitada, pois já havia um processo novo (ágil) sendo implantado.

. Os alunos precisam compartilhar tempo de trabalho como bolsistas e tempo de estudo, com eventos como provas, estudo, viagens, congressos e férias, apresentando portanto um rendimento limitado.

6. Considerações Finais

Este trabalho buscou mapear o Pensamento Enxuto para demonstrar sua aplicação para o processo de desenvolvimento de *Software*. Para tanto, descreveu um sistema de produção, o STP e o Pensamento Enxuto, que generaliza seus princípios, mostrando um histórico de sua evolução e correlacionando com a produção em massa.

Características dos processo de desenvolvimento de *Software* foram apresentadas, seus problemas e as soluções propostas pelo pensamento enxuto.

Demonstrou-se através do estudo de caso como aplicar tais conceitos e ferramentas.

6.1. Conclusões

Conforme demonstrado no mapeamento e no estudo de caso, o pensamento enxuto e as ferramentas do STP utilizadas contribuíram para o processo de desenvolvimento de *Software* com os seguintes resultados:

. O estudo de caso mostra um sistema de produção para *Software* que aplica os princípios descritos pelo pensamento enxuto, como eliminação de estoques, fluxo, melhoria contínua, respeito ao trabalhador, entre outros ;

. Os dados apresentados no estudo de casos são reais e referentes à um projeto importante para a comunidade, que atingirá todas as instituições públicas de ensino Federais no Brasil ;

. Foi possível detectar que a ferramenta *Kanban* gerou uma transparência essencial para o controle do processo, possibilitando identificar desperdícios e gargalos de desempenho, e possibilitando também que a equipe percebesse o processo, fazendo emergir um sentimento de colaboração e busca pela melhoria.

. O processo estudado, apesar de ainda não ter sido finalizado, apresentou pelos desenvolvedores o estímulo esperado de sugerir melhorias para o processo, provando que os próprios desenvolvedores se sentiram estimulados à melhoria;

. Apesar de o desenvolvimento de *Software* não gerar um produto físico, o gerenciamento das atividades produtivas foi beneficiado pelos mesmos princípios de trabalho em fluxo, lotes unitários e eliminação de desperdícios em operações ;

. Apesar de haverem diferenças entre os sistemas produtivos, os princípios puderam ser aplicados, gerando benefícios semelhantes aos previstos na literatura, como maior rapidez, melhor qualidade, diminuição de falhas e maior satisfação dos clientes ;

. A orientação do pensamento enxuto para a satisfação do cliente proporcionou a reengenharia do processo, tornando-o mais eficiente e enxuto, realizando mais operações que contribuam diretamente para o valor desejado pelo cliente ;

. Foi possível comprovar e aprender um sem número de conceitos, gerando um capital de conhecimentos úteis, que serão repassados para projetos, artigos, alunos e colegas professores;

. A eliminação de desperdícios proporcionou aumento do fluxo do processo proporcionando um *Cycle Time* menor ;

. Os resultados do estudo de caso evidenciaram a estabilização do processo, pois as estórias (P, M, G) apresentaram uma variação considerada pequena ;

. Foi comprovada uma diminuição do tempo total de desenvolvimento ;

. Métricas de controle do processo de alto nível possibilitando estimativas cada vez mais convergentes ;

. A produção em lotes unitários simplificou o processo de desenvolvimento de *Software* .

. A partir do estudo de caso foi possível observar que as práticas de *Jidoka* liberaram os desenvolvedores de trabalho repetitivo e de grande risco;

. Além disso o *Jidoka* preveniu a permanência de erros no *Software* ;

. A implantação do *Jidoka* gerou confiança nos desenvolvedores, tendo sido confirmado por um percentual de 66% de melhorias ;

. O pensamento enxuto para o desenvolvimento de *Software* direcionou o estudo de caso para a satisfação do cliente, e agregou valor sob a ótica do cliente e diminuiu os custos e o tempo do processo ;

6.2. Contribuições

Ocorre desde a década de 70 um esforço da Engenharia de *Software* de buscar processos e modelos de maturidade para estabilizar o processo de desenvolvimento, tornando-o predizível e reproduzível. Entretanto, mesmo hoje a Engenharia de *Software* atinge resultados insatisfatórios.

Neste trabalho foram revistos conceitos da Engenharia de produção que não foram visitados pela Engenharia de *Software*, não constando de suas referências, e que podem guardar as respostas buscadas durante tanto tempo.

Entende-se que o processo produtivo para o desenvolvimento de *Software* possa ser otimizado e melhorado continuamente usando os valores, princípios e técnicas do Pensamento Enxuto e da Engenharia de Produção.

Como os Princípios e Técnicas da Produção Enxuta garantem o desenvolvimento de processos de forma estável e repetitiva, este autor acredita estar dando uma real contribuição à Engenharia de *Software* ao apresentar uma nova visão de processo produtivo, que empiricamente, como observado no Estudo de Caso, geram resultados estáveis e que são reproduzíveis em diversas engenharias.

Este autor percebeu a necessidade da demonstração de que diversas formas de serviço obtiveram resultados consideráveis ao adotar a mudança da gestão funcional para a gestão por processos, tornando horizontal e em fluxo seus processos. Espera-se desta forma que tenha ficado claro a viabilidade de utilizar uma gestão industrial baseada em Engenharia para os processos de *Software*.

6.3. Trabalhos Futuros

Apesar dos modelos CMMI e MPS-Br preconizarem gestão funcional para o processo, sugerindo que os processos a serem adotados precisam ser executados em lotes, o autor acredita ser possível aplicar os conceitos do pensamento enxuto para a melhoria de processos de desenvolvimento de *Software* que atendam à estes modelos de maturidade.

O autor pretende portanto ampliar o escopo da pesquisa atual, investigando uma cadeia de valor mais ampla, avaliando técnicas de priorização, aplicando

controle estatístico de processos, seis sigma e simulações, dado que as operações já mostraram uma tendência a convergência dos sigmas.

Pretende-se pesquisar com mais profundidade a utilização de *Jidoka* no desenvolvimento de *Software* em função dos sensíveis ganhos observados no uso desta técnica, que atualmente ainda não foi incorporada pela Engenharia de *Software*, apesar de inúmeros trabalhos científicos, baseados em estudos empíricos já publicados em pouco espaço de tempo.

Outra possibilidade é a criação de regras integradas em um ambiente de desenvolvimento, que possibilite um monitoramento mais inteligente e autonômico do processo, visando aprender com a equipe de desenvolvimento e conduzir seu aprendizado, coletar dados para estudos e identificar padrões positivos e negativos.

Atualmente já estão em produção uma série de pequenos artigos sobre validação de *Software* associada a modelagem de negócios, publicados por Carvalho e colaboradores (2010a; 2010b; 2010c) que seguem nesta direção.

É um objetivo também realizar mais estudos de casos em outra instituições com as quais se possa estabelecer parcerias acadêmicas, visando testar os princípios e ampliar o conhecimento.

Uma hipótese também identificada durante este trabalho, tendo em vista a viabilidade do controle de desenvolvimento de *Software* utilizando práticas industriais enxutas, foi a investigação sobre o estabelecimento de uma fábrica de *Software* nos moldes do STP.

Cabe esclarecer que já foi feita uma investigação preliminar usando bases *scopus*, *sciencedirect* e periódicos da indústria que revelou que as várias tentativas de implementação de fábricas de *Software* utilizavam o paradigma industrial derivado do Fordismo, prevendo trabalho em lotes, programação da produção exaustiva, e desperdícios incontáveis. Portanto, a criação de uma fábrica enxuta de *Software* será uma tentativa inédita e ousada, entretanto com o potencial para se tornar realidade.

7. Referências Bibliográficas

ABES – Associação Brasileira de Empresas de *Software* (2007) **Bases de dados. 2007**. Disponível em <http://www.abes.org.br/templ3.aspx?id=304& sub=524> acessado em 2011/02/27.

ABES – Associação Brasileira de Empresas de *Software* (2010) **Mercado Brasileiro de *Software* – Panorama e Tendências**, 2010. Disponível em <http://www.abes.org.br/UserFiles/Image/PDFs/Mercado_BR2010.pdf> Acessado em 31/01/2011.

Alvarez, R. R. & Antunes Jr., J. (2001) ***Takt-Time*: Conceitos e contextualização dentro do Sistema Toyota de Produção**. Revista Gestão e Produção. v.8, n.1, p.1-18.

Anderson, D. J., (2010). ***Kanban*: Successful Evolutionary Change for your Technology *Business*. Blue Hole Press**. WA, Sequim. 262 p.

Antunes, J.; Alvarez, R.; Bortolotto, P.; Klippel, M. ; Pellegrin, I. (2008) **Sistemas de Produção: Conceitos e Práticas para Projeto e Gestão da Produção Enxuta**. São Paulo. SP. Artmed Editora. 326 p.

ASSESPRO, MBI e ITS. **Exportações brasileiras de TI & *Software*. Pesquisa e relatório desenvolvido pela MBI**, 2005. Disponível em: <http://www.assespro-sp.org.br/>

Beck, K. (1999). **Extreme programming explained: Embrace change**. Reading, MA: Addison-Wesley. 224 p.

Beck, K. (2002) **Test Driven *Development*: By Example**. Boston, MA. Addison-Wesley Professional. 240 p.

Beck, K. (2004). **Programação eXtrema (XP) Explicada: Acolha as mudanças**. Porto Alegre: Bookman. 182 p.

Bellinger, G. (2004) **The way of systems**. Disponível em <http://www.systems-thinking.org/theWay/theWay.htm> , acessado em 2011-02-27.

Bertalanffy, L. von. (1973) **Teoria Geral dos Sistemas**. Petrópolis: Vozes, p. 360.

Bhat, T. e Nagappan, N. (2006) **Evaluating the efficacy of test-driven *Development*: Industrial case studies**. ISESE'06 - Proceedings of the 5th ACM-IEEE International Symposium on Empirical *Software Engineering*. Volume 2006, 2006, Pages 356-363

Black, J. T. (1998) **O projeto da fábrica com futuro**. Porto Alegre: Editora Artes Médicas. 288 p.

Boehm, B. W. (1988) **A spiral model of *Software Development* and enhancement**. IEEE Computer Society. Computer. v.21 i.5 p. 61-72.

Boehm, B. W. (2000) **Project Termination Doesn't Equal Project Failure**. IEEE Computer Society. Computer. v: 33. i:9 p. 94-96.

Boyer, R.; Charron, E.; Jurgens, U.; Tolliday, S. (1998) **Between Imitation and Innovation: The Transfer and Hybridization of Productive Models in The International Automobile Industry**. Oxford University Press, Oxford. p. 416. Disponível em <http://books.google.com/books?id=LDblemSNDqEC> acessado em 2011-02-26.

Brooks, F. P. (1995) **The Mythical Man-Month**. Ed. Addison Wesley. p. 322.

Burge, J. E. e Brown, D. C. (2002) **Discovering a Research Agenda for Using Design Rationale in *Software* Maintenance**. Computer Science Technical Report, Worcester Polytechnic University, WPI-CS-TR-02-03. p. 20.

Carvalho, M. M. (1997) **QFD - Uma ferramenta de tomada de decisão em projeto**. 1997. Tese (Doutorado em Engenharia de Produção). Universidade Federal de Santa Catarina, Florianópolis, 199 p.

Carvalho, R. A. ; Carvalho e Silva, F. L. ; MANHAES, R. S. (2010b). ***Mapping Business* Process Modeling constructs to Behavior Driven *Development* Ubiquitous Language.** arXiv:1006.4892v1 [cs.SE]

4. Carvalho, R. A. Manhaes R. S., Silva, F. L. C. Filling the Gap between Business Process Modeling and Behavior Driven Development.

Carvalho, R. A. ; MANHAES, R. S. ; Carvalho e Silva, F. L. (2010a). **Filling the Gap between *Business* Process Modeling and Behavior Driven *Development.*** arXiv:1005.4975v1 [cs.SE]

Carvalho, R. A. ; MANHAES, R. S. ; Carvalho e Silva, F. L. (2010c) **Introducing *Business* Language Driven *Development*** 2010

Carvalho e Silva, F. L.; Monnerat, G. M. & Carvalho, R. A.; (2010) Autonomação na produção de software. XXX Encontro Nacional de Engenharia de Produção, 14p.

Charette,R.N.(2005) **Why *Software* fails**, IEEE Spectrum: 42-49.

Choo, C. W. (2003) **A organização do conhecimento: como as organizações usam a informação para criar significado, construir conhecimento e tomar decisões**. São Paulo: Senac.

Chou, T. (2005) **The End of *Software*: Transforming Your *Business* for the On Demand Future**. Ed. Sams Publishing, Indianapolis. 170 p.

Chrissis, M. B.; Konrad, M. e Shrum. S. (2003) **CMMI: Guidelines for Process Integration and Product Improvement**, Addison-Wesley Pub Co, p. XXXX.

Cohn, M. (2005) **Agile Estimating and Planning**. NJ. Prentice Hall. p. 368.

Corbett Neto, T. (1997) **Contabilidade de Ganhos: a nova contabilidade gerencial de acordo com a Teoria das Restrições**. São Paulo: Nobel. 190 p.

Coriat, B. (1994) **Pensar pelo avesso**. Rio de Janeiro. Ed. Revan. 209p.

Crisping, L. e Gregory, J. (2009) **Agile Testing: A Practical Guide for Testers and Agile Teams**. Ed. Addison-Wesley Professional. New York. 576 p.

Crosby, P.B. (1979) ***Quality* is free: The Art of Making *Quality* Certain**. Ed. Mentor, New York. 270 p.

Cusumano, M.A. (1985) **The Japanese Automobile Industry: Technology and Management at Nissan and Toyota** (Harvard East Asian Monographs, No. 122) Harvard University Press, Boston. 400 p.

Danovaro, E.; Janes, A.; Succi, G. (2008) ***Jidoka* in *Software Development***. In: OOPSLA'08, Nashville, Tennessee, USA:19–23.

Debou, C. e Kuntzmann-Combelles, A. (2000) **Linking *Software* process improvement to *Business* strategies: experiences from industry**. *Software* Process: Improvement and Practice 5(1): 55-64.

DeMarco, T. (1979) **Structured Analysis and System Specification**. Prentice Hall, New Jewrsey. 352 p.

DeMarco, T. (2002) **Structured Analysis: Beginnings of a New Discipline** In: SD&M Conference 2001, *Software* Pioneers Eds.: M. Broy, E. Denert, Springer 2002: 520-527.

Deming, W. Edwards (1986). **Out of the Crisis**. MIT Press., MA. 507 p.

Denning, P. J., Riehle, R. D. (2009) "**Is *Software Engineering Engineering*?**" ACM. New York. Communication of ACM v 52:3. p.24-26.

Dijkstra, E. W. (1972) **The humble programmer**. ACM. Communications of the ACM, v. 15, n. 10, p. 859-866

Dijkstra, E., HOARE, C.A., DAHL, J. (1972) "**Notes on structure programming**" in: Structured Programming, Academic Press.

Erdogmus, H.; Morisio, M.; Torchiano, M. (2005) **On the effectiveness of the test-first approach to programming**. IEEE Transactions on *Software Engineering*. 31 (3):226-237

Eveleens, L. & Verhoef, C. (2010) **The Rise and Fall of the Chaos Report Figure**. IEEE *Software* - IEEE CS. p. 30-36.

Fischmann, A. A. (1972) **Algumas aplicações de economia de empresas**. São Paulo.Tese Doutorado FEA, USP.

Forrester, J. (1968) **Principles of systems**. Cambridge: Wright-Allen. 387 p.

Fowler, M. (2005) **The New Methodology**. Disponível em <http://martinfowler.com/articles/newMethodology.html>. acesso em 2011-02-27.

Fujimoto, T. (1999) **The Evolution of a Manufacturing System at Toyota**. Oxford University Press, Oxford. 400 p.

Gane, C. (1988) **Desenvolvimento rápido de sistemas**. Editora Livros Técnicos e Científicos, Rio de Janeiro. 170 p.

Gane, C. & Sarson, T. (1979) **Structured Systems Analysis: Tools and Techniques**. Prentice Hall, New Jersey. 288 p.

George, B. e Williams, L. (2003) **An initial investigation of test driven *Development* in industry**. Proceedings of the ACM Symposium on Applied Computing 2003:1135-1139.

George, B. e Williams, L. (2004) **A structured experiment of test-driven *Development***. Information and *Software* Technology, 46(5):337-342.

George, M.L. (2004) ***Lean* Seis Sigma para Serviços: Como Utilizar Velocidade *Lean* e Qualidade Seis Sigma para Melhorar Serviços e Transações**, Rio de Janeiro: *Quality*mark.

Glass, R. (1969) **Elementary Level Discussion of Compiler/Interpreter Writing**. ACM Computing Surveys, Mar 1969, pp. 64-68.

Gobbo,F. e Vaccari,M. (2008) **The Pomodoro Technique for Sustainable Pace in Extreme Programming Teams**. XP 2008, LNBIP 9:180–184.

Goldratt, E. M. (1990). **A síndrome do palheiro: garimpando informação num oceano de dados**. São Paulo, Nobel. 304 p.

Goldratt, E. M. (1994). **Mais que sorte. Um processo de raciocínio**. São Paulo, Educator. 303 p.

Goldratt, E. M. (1998). **Corrente Crítica**. Tradução. por Thomas Corbett Neto. São Paulo, Nobel. 260 p.

Goldratt, E. M. (1999) **Theory of Constraints**, North River Press.

Goldratt, E. M.; Fox, R. E. (1992). **A corrida pela vantagem competitiva**. São Paulo, Educator. 177 p.

Goldratt, E. M.; COX, J. (2002). **Meta: um processo aprimorado contínuo**. 2a. Ed. São Paulo, Nobel. 365 p.

Goldstein , M. & FELDMAN ,Y. A. (2006) **Refactoring with Contracts** . Proceedings of AGILE 2006 Conference (AGILE'06) . p. 1-10.

Gonçalves, J. E. L. **As empresas são grandes coleções de processos**. RAE - Revista de Administração de Empresas, v. 40, n. 1, p. 6-19, jan./mar 2000

Graça, L. (1991) **As novas formas de organização do trabalho**. Universidade Nova de Lisboa.

Guerreiro, R. (1996) **A meta da empresa - seu alcance sem mistérios**. São Paulo: Atlas. 1996.

Hammer, M. **A empresa voltada para processos. HSM Management**, v. 9, n. 2, jul./ago. 1998.

Hannan, M.T.; Freeman, J. (1977) **The population ecology of organizations. American Journal of Sociology**, v. 82, n. 5, p. 929-924.

Harrington, H. J. (1991) ***Business* process improvement**. New York: McGraw Hill, 1991

Harvey, D. (1992) **A Transformação Político-Econômica do Capitalismo no Final do Século XX**. Condição Pós-Moderna. 8.ed. São Paulo: Loyola,

Highsmith, J. (2002) **Agile *Software Development* Ecosystems**. Addison -Wesley, Boston, MA.

Highsmith, J. (2004) **Agile Project Management: Creating Innovative Products**. Addison-Wesley, .

Holweg, M. (2007) **The genealogy of *Lean* production**. Journal of Operations Management 25:420–437.

Hounshell, D.A. (1984) **From the American System to Mass Production 1800-1932: The *Development* of Manufacturing Technology in the United States**. John Hopkins University Press. p. 441. Disponível em <http://books.google.com/books?id=9H3tHKUFcfsC> acessado em 2011-02-26.

Humphrey, W.S. (2005) **Why Big *Software* Projects Fail:The 12 Key Questions**. Open Forum. 25-29. disponível em <http://www.stsc.hill.af.mil> acessado em 2011-02-27.

IEEE (1990) **Standard Glossary of *Software Engineering* Terminology**, IEEE std 610.12-1990. disponível em <http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/ElectronicSupplements/glossary.pdf> acessado em 2011-02-27.

Ikonen, M.; Kettunen, P.; Oza, N.; Abrahamsson, P. (2010) **Exploring the Sources of Waste** In: *Kanban Software Development* Projects. 36th EUROMICRO Conference on *Software Engineering* and Advanced Applications. IEEE Computer Society. 827:830.

Imai, M. (1997) **Gemba *Kaizen*: Estratégias e Técnicas do *Kaizen* no piso de fábrica**. Editora IMAM, São Paulo. 332 p.

ISO/IEC 15504 (2006) **Information Technology – Process Assessment** – Part 5: An exemplar Process Assessment Model.

Jakobsen, C. e Sutherland, J. (2009) "***Scrum* and CMMI – Going from Good to Great: are you ready-ready to be done-done**?," in Agile Conference (Agile 2009), Chicago, IL, EUA. p. 333–337 .

Jakobsen, C. R. e Johnson, K. A. (2008) "**Mature Agile with a Twist of CMMI**," in Agile 2008, Toronto, ON, EUA p. 212-217.

Janzen,D. e Saiedien,H. (2005) **Test-driven *Development*: Concepts, taxonomy, and future direction**. Computer, 38(9):43-50.

Jenkins, A. M. & Naumann, J. D. (1980) **The prototype Model as a MIS Design Technique**. Indiana University.

Jeston, J.; Nelis, J. ***Business* Process Management, practical guidelines to successful implementations**. Oxford: Butterworth-Heinemann - Elsevier, 2006.

Juran, J.M. (1988) **Juran on Planning for *Quality***. New York:Macmillan. 341 p.

Kannane, R. (1994), **Comportamento Humano nas Organizações**, Editora Atlas, São Paulo.

Kruchten, P. (2000) **The Rational Unified Process An Introdution. Massachusetts**, Addison Wesley. Reading Massachusetts. 320 p.

Ladas, C. (2008) ***Scrum*ban: Essay on *Kanban* Systems for *Lean Software Development***. Editora Modus Operandi Press. Seatle. 178 p.

Larman, C. (2003) **Agile and Iterative *Development*: A Manager's Guide**. Addison Wesley Professional. 368 p.

Larman, C. e Basili, V. (2003) **Iterative and Incremental *Development*: A Brief History**. IEEE Computer vol. 36(6): pp. 47-56.

Liker, J.K. (2005) **O Modelo Toyota: 14 princípios de gestão do maior fabricante do mundo**. Editara Bookman, Porto Alegre. 316 p.

Little, J.D.C. (1961) "**A Proof for the Queuing Formula: L = λ W**," Operations Research, Vol. 9, No. 3, pp. 383-387.

Manhães, R. S. (2010) **Behaviour-Driven *Development*: Uma Abordagem Ágil para Engenharia de *Software***. Monografia de pós-graduação de Engenharia de Sistemas. Vila Velha ESAB. 69 p.

Martin, R. C. (2002) **Baby Steps**. Dr. Dobb's Journal, November. v.10, n.10, p.46-54.

Martin, R. C. (2002) **Principles, Patterns, and Practices of Agile Software Development.** Prentice Hall, 2002. 552 p.

Martin, R. C. (2008) **Cle*an* Code: A Handbook of Agile *Software* Craftsmanship** (Robert C. Martin Series). Prentice Hall PTR, August 2008.

Matos,E. e Pires,D. (2006) **Teorias administrativas e Organização do trabalho: de Taylor aos Dias atuais, influências no setor de saúde e na enfermagem**. Texto Contexto Enferm, Florianópolis, 2006 Jul-Set; 15(3):508-14.

Maximillien,E.M. e Williams,L. (2003) **Assessing test-driven *Development* at IBM**. In: Proceedings – International Conference on *Software Engineering* 2003:564-569.

Melendez Filho, R. (1990) **Prototipação de sistemas de informações: Fundamentos, técnicas e metodologia.** Editora Livros Técnicos e Científicos. Rio de Janeiro. 232 p.

Mellor,S. e Balcer,M. (2002) **UML executável: Uma fundação para a arquitetura modelo-dirigida**. Addison-Wesley. p.

Meyer, B. (1997) **Object-Oriented *Software* Construction**. Prentice-Hall, Upper Saddle River, New Jersey.

Meyer, B. Applying "**Design by Contract**". Computer, 25(10):40–51, 1992

Middleton, P. e Sutton, J. (2005) ***Lean Software* Strategies: Proven Techniques for Managers and *Developer*s**. New York. Productivity Press. 432. p.

Moreira, D.A. (2004) **Administração da Produção e Operações**. Editora Pioneira Thomson Learning, São Paulo. 619 p.

Morgan, J.M. e Liker, J.K. (2008) **Sistema Toyota de Desenvolvimento de produto: Integrando pessoas, processo e tecnologia**. Editora Bookman, Porto Alegre. 391 p.

Naur, P. e Randell, B. (1969) (Editors), ***Software Engineering*: Report on a conference sponsored by the NATO Science *Committ*ee,** Garmisch, Germany, 1968, 136 p. disponível em <http://homepages.cs.ncl.ac.uk/brian.randell/NATO>, acessado em 2011-02-27

Nonaka, I.; Takeuchi, H. (1997) **Criação de conhecimento na empresa: como as empresas japonesas geram a dinâmica da inovação.** 2a ed. Rio de Janeiro: Campus, 1997 .

North, D. (2006) **Introducing BDD**. Disponível em http://dannorth.net/introducing-bdd/ acessado em 2011/01/17.

Ohno, T. (1997) O Sistema Toyota de Produção: além da produção em larga escala. Bookman, Porto Alegre. 151 p.

Osherove , R. (2009) **The Art of Unit Testing with Examples in .NET** . Manning Publications .

Park, M. H-J.; Lim,J.W. e Birnbaum-More,P.H. (2009) The Effect of Multiknowledge Individuals on Performance in Cross-*Functiona*l New Product *Development* Teams. Journal Prod.Innov. Manag., 26:86–96.

Parkinson, C. N. (1955). **Parkinsons Law. Economist**. (Novembro-1955) disponível em <http://www.economist.com/node/14116121?story_id=14116121&Fsrc=mgttkgnwl> acessado em 2011-02-27.

PEP8 Python Enhancement Proposal 8. disponível em <http://www.python.org/dev/peps/pep-0008> acessado em 2011-02-27

Pfleeger, S. L. (2004) **Engenharia de *Software*: teoria e prática**. São Paulo: Prentice Hall, 2a edição. 558 p.

Pinheiro, D. M. & Rezende, G. G. (2009) **Uma Proposta Ferramental Para Especificação E Validação Executável De Software**. Monografia de Graduação em Tecnologia em Desenvolvimento de Software – Instituto Federal Fluminense – Campos dos Goytacazes / RJ.

Pollice, G. (2005) **Teaching *Software Development* vs. *Software Engineering***, in The Rational Edge, 5 pages, December 2005

Poppendieck, M. e Poppendieck, T. (2003) ***Lean Software Development*: An Agile Toolkit**. Addison Wesley.

Poppendieck, M. e Poppendieck, T. (2011) **Implementando o desenvolvimento *Lean* de *Software*: do conceito ao dinheiro**. Editora Bookman, Porto Alegre. 280 p.

Porter, M. E.; Millar, V. E. (1995) **How information gives you competitive advantage**. Harvard *Business* Review, July-August, pp. 1-12

Pressman, R. S. (2006) **Engenharia de *Software***. 6. ed. Rio de Janeiro: McGraw-Hill, 720p.

Roach, S. White Collar Productivity: A Glimmer of Hope? Special Economic Study, Morgan Stanley, 16. September 1988.

Rocha, A. R.; Montoni, M.; Santos, G.; Mafra, S.; Figueiredo, S.; Albuquerque, A.; Mian, P. (2005). **Reference Model for *Software* Process Improvement: A**

Brazilian Experience. In: I. Richardson *et al.* (Eds.): EuroSPI 2005, Lecture Notes in Computer Science (LNCS) 3792, pp. 130-141.

Royce, W. (1970), "**Managing the *Development* of Large *Software* Systems**", Proceedings of IEEE WESCON 26 (August): 1-9, acessível em <http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/*Waterfall*.pdf> último acesso 2011-02-27.

Salviano, C. F. (2006) **Uma Proposta Orientada a Perfis de Capacidade de Processo para Evolução da Melhoria de Processo de *Software***. Tese de Doutorado – Universidade Estadual de Campinas – Campinas / SP.

Schulmeyer, G. G. (1990) **Zero Defect *Software***. McGraw-Hill, Inc. 400 p.

Schwaber, K. & Beedle, M. (2002) Agile *Software Development* with *Scrum*. New Jersey. Prentice-Hall. 158 p.

SEI – *Software Engineering* Institute (2006) **CMMI (R) for *Development*, version 1.2**. Pittsburgh, PA. EUA. 561 p. Disponível em <http://www.sei.cmu.edu/reports/06tr008.pdf> acessado em 2011-02-26.

SEI - *Software Engineering* Institute (2010) **CMMI (R) for *Development*, version 1.3**. Pittsburgh, PA. EUA. 470 p. Disponível em <http://www.sei.cmu.edu/reports/10tr033.pdf> acessado em 2011-02-26.

Semeghini, J. (2002) **Núcleo Softex Campinas – Missão Japão 2001**. Disponível em: <http://www.cps.softex.br/relatorio.htm>. Acesso em: 24 jul. 2002

Senge,P. (2002) **A quinta disciplina: Arte e Prática da Organização que aprende**. 10 ed. Editora Best Seller, SP. 443 p.

Shewhart, W. A. (1931) **Economic Control of *Quality* of Manufactured Product**. New York: Van Nostrand. 505p.

Shingo, S. (1996) **O Sistema Toyota de Produção do ponto de vista da Engenharia de Produção**. Editora Artmed, Porto Alegre. 296 p.

Silva, C. e Magalhães, J. M. (2008) **Redução de tempos de ciclo no fabrico de protótipos de molde**. Revista Iberoamericana de Ingeniería Mecánica. Vol. 12, N.º 2, pp. 15-26.

Slack, N.; Chambers, S.; Harland, C.; Harrison, A.; Johnston, R. **Administração da produção**. Ed. compacta. São Paulo: Atlas, 1999.

Smith, P.G. & Reinertsen, D.G. (1992). **Shortening the product *Development* cycle. Research Technology Management**, 35 (3), pp. 44 – 49

SOFTEX - Associação para a Promoção da Excelência do *Software* Brasileiro (2009) **MPS.Br – Melhoria de Processo do *Software* Brasileiro – Guia Geral**, Modelo de Qualidade Versão 2.0. Brasília – DF. Disponível em <http://www.softex.br/mpsbr/_guias/guias/MPS.BR_Guia_Geral_2009.pdf> acessado em . Acessado em 2011-02-27.

Sommerville, I. **Engenharia de *Software***. 6. Ed. Editora Addison-Wesley, 2003.

Standish-Group (1995) **The Chaos Report**. Disponível em <http://www.standishgroup.com/> acessado em 2011-02-26.

Standish-Group (2010) **The Chaos Report**. Disponível em <http://www.standishgroup.com/> acessado em 2011-02-26.

Sutherland, J., Jakobsen, C., E Johnson, K. (2007) "***Scrum* and CMMI Level 5: A Magic Potion for Code Warriors!**" in Agile 2007, Washington, D.C. p. 272-278.

Takeuchi, H., Nonaka, I. (1986) **The New New Product *Development* Game**, Harvard *Business* Review, 11 p.

Tapping, D. e Shuker, T. (2010) ***Lean* Office: Gerenciamento do fluxo de valor para áreas administrativas**. São Paulo, Leopardo Editora. 186 p.

Taylor, F. W. (1990) **Princípios de administração científica**. 8. ed. São Paulo: Atlas.

Thiollent, M. (2004) **Metodologia da pesquisa-ação**. São Paulo: Cortez. 132 p.

Werkema, M. C. C. ***Lean* seis sigma: Introdução às ferramentas do *Lean* manufacturing**. Belo Horizonte: Werkema Editora, 2006. 120p.

Westbrook, R.K. (1995) **Action Research: a new paradigm for research in production and operations management**. International Journal of Operations and Production Management, Vol. 15 no.12, pp. 6-20.

Williams, L.; Kessler, R.R.; Cunningham, W. e Jeffries, R. (2000) **Strengthening the Case for Pair Programming**. IEEE *Software* . p.19 -25.

Womack, J.P.; Jones, D.T. (1998) **A mentalidade enxuta nas empresas: Elimine o desperdício e crie riqueza**. Editora Campos, Rio de Janeiro. 427 p.

Womack, J.P.; Jones, D.T.; Roos, T. (1992) **A Máquina que mudou o mundo**. Editora Campos, Rio de Janeiro. 326 p.

Yin, R. K. (2003) **Study Case Research: Design and Methods**. 3a. Edição. Thousand Oaks, Sage Publications. 200 p.

Yourdon, E. (1975) **A Case Study in Structured Programming: Redesign of a payroll System**. Proceddings of IEEE Compcon Conference, 1975, New York, IEEE.

Yourdon, E. (1988). **Administrando Técnicas Estruturadas: Estratégias para o desenvolvimento de *Softwares* nos anos 90**. Editora Campus, Rio de Janeiro. 244 p.

Yourdon, E. (1989). **Administrando a Ciclo de Vida do Sistema**. Editora Campus, Rio de Janeiro. 159 p.

Zultner, R. (1991) **Before the House; The Voices of Customers in QFD**. Zultmr & Company, Princeton NJ.