ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

SUMÁRIO

1 Introdução

Este trabalho explora teorias e abordagens alternativas para o gerenciamento de projetos de desenvolvimento de software. Esta exploração é feita avaliando abordagens que foram consideradas influenciadoras nos resultados positivos de projetos (como é o caso das metodologias ágeis de desenvolvimento de software) e princípios e práticas que geraram grandes saltos de produtividade em outros setores industriais (como por exemplo, a produção enxuta cuja aplicação iniciou-se na década de 1940 pela indústria automobilística).

1.1 Motivações

A indústria norte-americana de software apresentou um significativo crescimento, atingindo um volume de vendas de 227 bilhões de dólares e empregando aproximadamente 1,13 milhões de pessoas no ano de 2002. Além disso, é ainda mais significativo o fato dela ocupar a segunda colocação na contribuição em valor agregado para a economia norte-americana, contribuindo com 125,5 bilhões de dólares e com um crescimento de 70% no período de 1997 até 2000 (NATHAN, 2003).

Segundo Nathan (2003), estes números colocam a indústria de software à frente dos setores tradicionais da economia norte-americana como a manufatura e os serviços, perdendo apenas para o setor de escritórios e clínicas médicas. O gráfico contido na Figura 1, apresenta os números de cada segmento da economia estudado, no período compreendido entre os anos de 1997 e 2000.

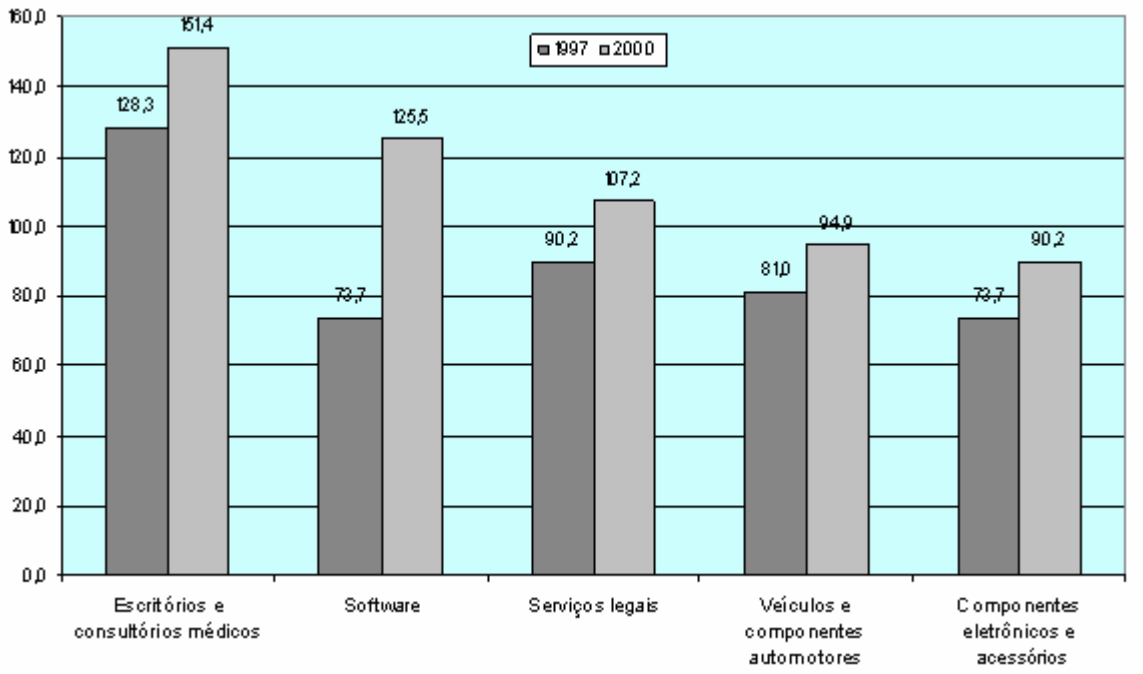


Figura 1: Valor agregado para economia dos EUA por setores da indústria em bilhões de dólares (NATHAN, 2003).

Esses dados, mesmo se restringindo à economia norte-americana, servem para ilustrar a importância da indústria de software para a economia mundial. Apesar da grande relevância desse setor para a economia norte-americana, seu dinamismo pode estar sendo prejudicado pela incapacidade de obter ganhos sustentáveis de produtividade nos últimos anos.

Os sintomas da necessidade de abordagens alternativas para o gerenciamento de projetos podem ser observados pelos números levantados por Poppendick (2004), onde os ganhos de produtividade de outros setores tradicionais da economia, como por exemplo, a manufatura e o varejo, vêm crescendo progressivamente desde 1986, enquanto que a indústria de software, após níveis de crescimento mais expressivos que esses setores, passa por um período de estagnação e declínio.

O gráfico apresentado na Figura 2, ilustra a produtividade dos setores de varejo, manufatura e software da economia norte-americana, no período de 1987 até o ano de 2001. Através da curva em azul, que representa a indústria de software, é possível identificar um declínio de produtividade após o ano de 1997.

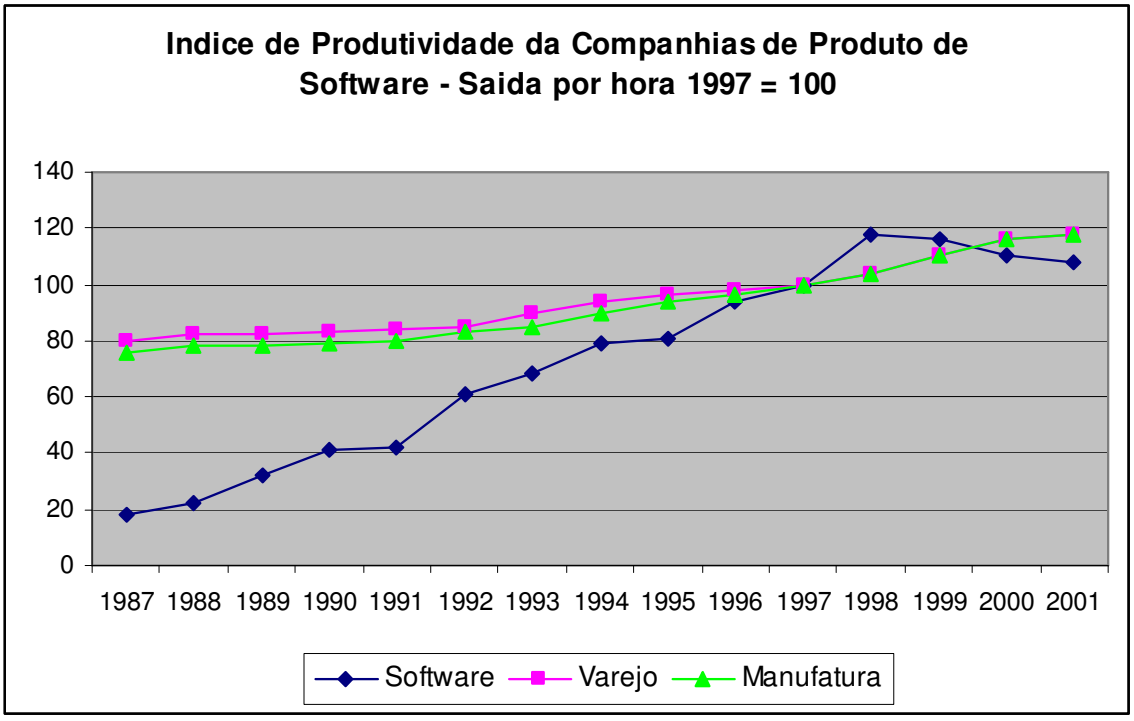


Figura 2: Dados do *US Bereau of Labor Statistics* (POPPENDIECK, 2004).

Para diversos autores, essa estagnação é justificada pela inadequação da forma de organização do trabalho utilizada por diversas empresas. Essas empresas adotaram, em muitos casos, o modelo de desenvolvimento cascata (*waterfall*), proposto por Royce (1970), que se mostra inadequado para as necessidades dos projetos de software conduzidos no cenário de negócios atual (caracterizado pela alta competitividade, pela volatilidade dos requisitos e prazos menores).

O modelo de desenvolvimento cascata, apesar de proposto na década de 1970, é ainda hoje adotado por muitos projetos de desenvolvimento de software (NEILL & LAPLANTE, 2003). Apesar de sua ampla utilização, muitos problemas foram identificados e documentados ao longo das últimas décadas.

Neste contexto, surgiram diversas propostas de novos modelos, metodologias de desenvolvimento e formas de organização do trabalho para o desenvolvimento de software buscando solucionar essa situação.

1.2 Objetivo

O objetivo deste trabalho é a proposição de um modelo de gerenciamento de projetos baseados nas metodologias ágeis de desenvolvimento de software e nos princípios da produção enxuta.

1.3 Justificativa

Este trabalho explora formas alternativas de gerenciamento de projetos de desenvolvimento de software, com o intuito de obter ganhos de produtividade e reverter o cenário de declínio apresentado na Figura 2. Esta exploração ocorre através da proposição de um modelo de gerenciamento que combina duas metodologias ágeis de desenvolvimento de software: *Scrum* e *Extreme Programming* (XP). Esta combinação reune os pontos fortes de cada uma delas e contorna suas respectivas deficiências: falta de especificação de como realizar o desenvolvimento de software (*Scrum*) e a definição superficial dos aspectos de gerenciamento de projeto (XP).

Por se tratarem de metodologias novas, quando comparadas com outras existentes, o XP e o *Scrum* sofrem grandes críticas e são constantemente questionadas quanto ao seu domínio de aplicação. Para contornar a falta de credibilidade herdada das metodologias selecionadas pelo modelo proposto, são utilizados os princípios da produção enxuta, originados na manufatura automobilística japonesa. A transferência de conhecimentos entre setores industriais distintos (manufatura e software), fortalece o embasamento teórico da proposta, utilizando princípios que proporcionaram grandes ganhos de produtividades para a indústria automobilística.

O modelo proposto também explora teorias alternativas relacionadas ao gerenciamento de projetos. Em um trabalho apresentado por Koskela e Howell (2002), foram identificadas anomalias nas teorias de projetos e nas teorias de gerenciamento que compõem o PMBoK (*Project Management Body of Knowledge*). Estes autores apresentaram teorias alternativas que poderiam ser exploradas para contornar estas anomalias. Estas teorias são apresentadas e descritas de forma sucinta na Tabela 1, agrupadas de acordo com cada tópico (projeto e gerenciamento).

Tabela 1: Teorias subjacentes do gerenciamento de projetos
(KOSKELA & HOWELL, 2002).

Tópicos de teorias		**Teorias**	**Descrição**
Projeto		Transformação	Consiste na decomposição hierárquica do trabalho a ser realizado, minimizando o custo de cada tarefa individualmente.
		Fluxo	Adicionam considerações de tempo, variação e satisfação do cliente para a concepção do projeto.
		Geração de valor	
Gerenciamento	Planejamento	Gerenciamento por planejamento	No nível operacional, o gerenciamento consiste na criação, revisão e implementação de planos.
		Gerenciamento por organização	Adiciona a idéia da atividade humana inerentemente situacional (respondem as situações atuais), focando na estruturação do ambiente de trabalho para contribuir com as ações executadas.
	Execução	Teorias clássicas de comunicação	Assume que as tarefas planejadas podem ser executadas simplesmente pela notificação do inicio por uma autoridade central.
		Perspectiva linguagem e ação	Conceitua a comunicação e o comprometimento de duas direções.
	Controle	Modelo termostato	Consiste em três elementos: existe um desempenho padrão, o desempenho é mensurado através dos resultados obtidos e caso hajam desvios, ações corretivas são tomadas para aproximar os resultados do padrão pré-definido.
		Modelo de experimentação cientifica	Foca na descoberta das causas das variações nos resultados obtidos e nas ações aplicadas para corrigir e eliminar os desvios.

1.4 Estrutura do Trabalho

Este trabalho é estruturado da seguinte forma:

- **Capítulo 1 - Introdução**: este capítulo faz uma breve descrição sobre o presente trabalho, contemplando objetivos, motivações, justificativas e a estrutura do trabalho.
- **Capítulo 2 - História da Indústria de Software**: este capítulo descreve o surgimento e a maturação da indústria de software, contemplando a evolução de tecnologias, organização do trabalho e mercado. Neste capítulo também é feita uma descrição do modelo cascata, apresentando o contexto do seu surgimento e alguns de seus problemas levantados a partir de uma revisão bibliográfica.
- **Capítulo 3 - Metodologias Ágeis**: este capítulo apresenta a história do movimento de desenvolvimento ágil de software e descreve duas metodologias relevantes para o trabalho: *Extreme Programming* (XP) e *Scrum*.
- **Capítulo 4 - Desenvolvimento Enxuto de Software**: este capítulo faz um breve histórico da produção enxuta, descrevendo seus princípios e valores, construindo uma base teórica, para depois fazer a transposição dos conceitos da manufatura para o desenvolvimento de software.
- **Capítulo 5 - Modelo de Gerenciamento Proposto**: este capítulo traz a descrição do modelo proposto, combinando as duas metodologias ágeis selecionadas (*Extreme Programming* e *Scrum*) e os princípios e valores da produção enxuta.
- **Capítulo 6 - Estudo de Caso**: este capítulo ilustra a aplicação do modelo proposto em um estudo de caso, juntamente com análise qualitativa dos resultados obtidos. Em seguida é feita uma comparação do modelo proposto com o cascata, identificando suas vantagens e escopo de aplicação.

- **Capítulo 7 - Conclusões**: este capítulo contém as considerações finais e as contribuições deste trabalho realizado, finalizando com propostas de trabalhos futuros a partir dos resultados obtidos.
- **Referências Bibliográficas**: esta seção contém uma listagem, em ordem alfabética dos autores, dos trabalhos referenciados ao longo deste documento.
- **Bibliografia Complementar**: esta seção contém uma listagem, em ordem alfabética dos autores, dos trabalhos pesquisados mas não referenciados ao longo da elaboração da pesquisa apresentada neste trabalho.
- **Anexo A - Produto do Estudo de Caso**: esta seção do trabalho apresenta uma descrição sucinta do produto resultante da realização do estudo de caso.

2 História da Indústria de Software

A indústria de software pode ser considerada nova quando comparada a outros setores industriais (como por exemplo, a manufatura). Apenas em meados da década de 1950 foi evidenciada a necessidade de programação e, somente na década de 1960, o termo "software" foi popularizado. Quando surgiu, o termo era utilizado em contextos para descrever qualquer tipo de serviço que não era fornecido pelos próprios fabricantes de hardware.

Somente na década de 1970, dez anos após seu surgimento, o termo "software" obteve o significado que possui nos dias atuais: "sistema de processamento de dados de um computador" (HOUAISS, 2004).

Durante a década de 1960, as empresas realizavam grande parte do desenvolvimento de software internamente, apesar de dependerem fortemente dos sistemas operacionais e ferramentas de programação fornecidas pelos fabricantes de computadores. O conceito de softwares de prateleira (*COTS – Commercial Off The Shelf*), ainda era desconhecido e muitas das empresas pioneiras da indústria de software eram sustentadas por contratos militares e governamentais.

O trabalho realizado nas empresas de software era muito mais parecido com as atividades atualmente realizadas por empresas de contabilidade, consultoria e fornecedores de computadores.

O que impulsionou o surgimento de fornecedores independentes de software (ISV – *Independent Software Vendors*), foi a crescente procura por soluções sob demanda para as necessidades específicas de cada cliente. Essa demanda não era atendida pelos fabricantes de computadores, que insistiam no fornecimento de soluções genéricas.

Ainda na década de 1960, as empresas de software começaram a construir bibliotecas de programas compostas de rotinas reusáveis e a adquirir experiências, passando a ser capazes de realizar projetos sob demanda e superaram a capacidade dos fornecedores de computadores.

Aliada à crescente necessidade de soluções sob demanda, está o fato de que no fim da década de 1960, a complexidade exigida para construir sistemas de informações demandava habilidades de programação e projeto de sistemas raramente encontrados nos

departamentos de processamento de dados corporativos, aumentando ainda mais as oportunidades para as empresas fornecedoras de software.

Esta oportunidade influenciou no aumento do número de empresas fornecedoras de software, o que atraiu a atenção de investidores. Estes investidores enxergavam nessas empresas uma oportunidade de grande retorno sobre investimento (ROI – *Return Over Investment*). Entretanto, no início da década de 1970, esta promessa de retorno não se concretizou, e muitas das empresas que receberam estes investimentos nunca tiveram lucro.

Neste cenário de crise, os softwares de prateleira surgiram como uma, dentre muitas outras tecnologias, como uma alternativa para a programação de aplicativos por parte das corporações. Para ilustrar a mudança que esta nova abordagem acarretaria, uma pesquisa foi realizada pelo *Wall Street Journal* dos EUA em 1969 (HAIGH, 2002), onde foram entrevistados por volta de 800 executivos, e constatou-se que 44% das empresas realizavam todo o desenvolvimento de aplicativos internamente e 38% delas realizam grande parte dele internamente.

A adoção de softwares de prateleira representava uma grande mudança no paradigma da época, onde o desenvolvimento do software passaria a ser realizado por terceiros. Esta mudança foi acelerada pela introdução de novas tecnologias para fabricação de computadores (*System*/360), onde para adequar os sistemas em operação nas empresas para a nova plataforma, exigiria um esforço bastante alto. Esta transição de tecnologia, também favoreceu e facilitou a adoção de softwares de prateleira.

Esta mudança de paradigma exigiu uma mudança no modelo de negócio das empresas de software, por exemplo, os clientes passaram a pagar por funcionalidade de aplicações, as correções de defeitos eram gratuitas e apenas era cobrada uma pequena taxa para atualizar as regras de negócios utilizadas pelo software. Através da adoção de softwares de prateleira, as empresas esperavam reduzir drasticamente os custos de manutenções.

Na década de 1980, a demanda por softwares de prateleira começou a ser atendida, e surgiram as primeiras empresas lucrativas que forneciam este tipo de solução, dando origem à estrutura atual da indústria de software.

Outra representação da história da indústria de software foi apresentada por Raccoon (1997), onde a história do setor é dividida em seis ondas com duração de aproximadamente uma década cada uma (Imatura, Função, Programação Estruturada, Módulo, Objeto e Programação por Padrões).

A Tabela 2 ilustra esta representação, sumarizando a evolução da indústria de software ao longo dos 50 anos de sua existência, no período que compreende o ano de 1945 até a especulação de como será o cenário em 2010. Esta tabela apresenta cronologicamente o progresso sobre diversos aspectos, como por exemplo: hardware, organização, ambiente de programação, necessidade do projeto, modelo e participação do usuário.

Tabela 2: Aproximação dos 50 anos de progresso na engenharia de software, adaptado de Raccoon (1997).

	Imatura 1945 - 1955	Função 1956 - 1966	Programação Estruturada 1967 – 1977	Módulo 1978 - 1988	Objeto 1989 - 1999?	Programação por Padrões 2000? - 2010?
Economia de hardware	Mainframes de pesquisa	Mainframes comerciais	Mini computadores comerciais	Computadores pessoais		Internet
Organização	Declaração	Funções		Módulos	Objetos	
Ambientes de programação		Editores e compiladores		Ferramentas de propósitos gerais	Ferramentas de domínios específicos	
Necessidade do projeto		Útil	Documentado	Correto	Utilizável	
Modelos			Modelo cascata		Modelo espiral	Modelo caótico
Participação do usuário	Nenhuma		Uma vez		Periódica	Durante

Através da análise da Tabela 2, é possível perceber a rápida evolução num período relativamente curto de tempo, onde a economia de hardware mudou de computadores de grande porte voltados a pesquisa (*mainframes*), utilizados única e exclusivamente para fins militares e científicos, para computadores pessoais em menos de 40 anos.

As duas ondas que correspondem ao período em que ocorreu o maior crescimento, difusão e maturação do modelo cascata, compreendido entre os anos 1967 e 1988, são as que Raccoon (1997) nomeou de onda de Programação Estruturada (1967-1977) e onda do Módulo (1978-1988).

Na onda de Programação Estruturada, a principal necessidade dos projetos de software era a documentação. Já na onda seguinte (Módulo), a necessidade principal era a de que o programa estivesse correto, ou seja, atendesse as obrigações contratuais e estivessem corretamente documentadas.

Durante as ondas Programação Estruturada e Módulo, a participação do usuário no projeto restringiu-se a um único momento: durante a contratação e especificação de requisitos. Nas ondas seguintes (Objeto e Programação por Padrões), a participação dos usuários era periódica ou constante durante a realização do projeto.

2.1 O Primeiro Modelo Estruturado

O modelo conhecido popularmente por cascata, ou *waterfall*, foi proposto na década de 1970 por Winston Royce no artigo intitulado "*Managing the development of large software systems*" (ROYCE, 1970). Ele surgiu da necessidade de controlar grandes projetos personalizados de software, como por exemplo, os softwares militares e governamentais norte-americanos dos anos 70. Ao contrário dos modelos *ad hoc* extensivamente adotados na década de 1960, o modelo cascata é extremamente estruturado, seqüenciado e direcionado para manter relatórios de todas as decisões tomadas e atividades realizadas durante o desenvolvimento.

Royce (1970) identificou duas grandes etapas atendidas em todos os projetos de software: a análise e a codificação. Utilizar apenas essas duas etapas pode ser extremamente útil para projetos pequenos e com baixa complexidade. Para projetos de maior complexidade, Royce desenvolveu e expandiu seu modelo, que também conta com as etapas de análise e codificação, porém sucedidas e precedidas por outras etapas bem especificadas.

A Figura 3 ilustra uma representação gráfica do modelo cascata, apresentando as etapas que compõem o modelo (requisitos, análise, projeto, codificação, testes e operação) e as interações entre elas.

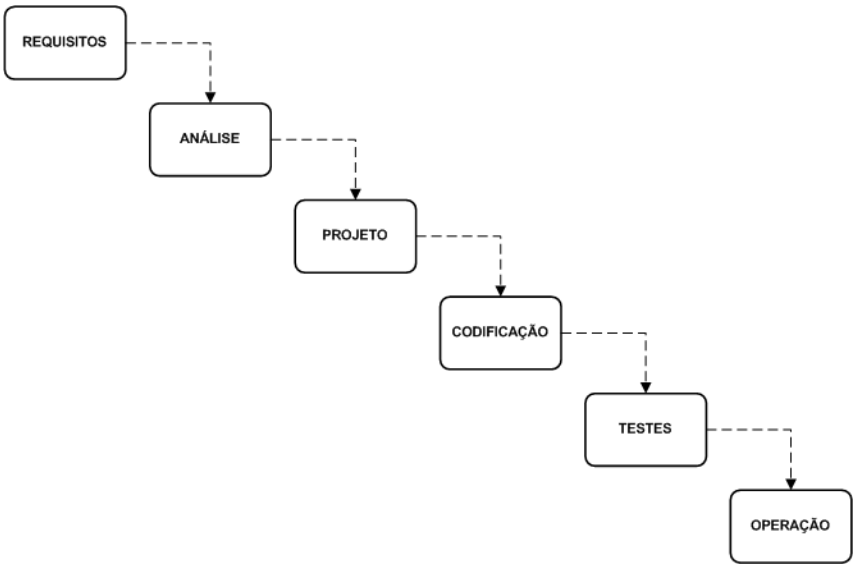


Figura 3: Modelo cascata, adaptado de Royce (1970).

Nesse modelo, o desenvolvimento de um software parte de uma rígida especificação dos requisitos de sistema, onde são levantadas as necessidades do cliente e as funcionalidades que o software precisa executar. Em seguida, estes requisitos são enumerados e acordados entre os envolvidos. Isso se deve ao fato de que o software faz parte de um sistema maior que precisa ser delineado.

A partir do levantamento inicial dos requisitos e após acordado o escopo do projeto, inicia-se o aprofundamento e o detalhamento dos requisitos de sistema inicialmente levantados. No fim da etapa de requisitos, a especificação do software deverá ser clara e sem ambigüidades para guiar as etapas seguintes.

O principal propósito da etapa de análise é transformar as informações levantadas referentes aos requisitos, critérios do usuário e previsões do projeto em uma especificação estruturada. Esta atividade envolve a modelagem do ambiente do usuário através de diagramas de fluxo de dados, entidade-relacionamento, transição de estados, classes e outras ferramentas. Além do modelo do sistema, geralmente ao final da etapa de análise, é preparado um conjunto de orçamentos e cálculos de custo e benefício mais cuidado e detalhado.

Essas duas primeiras etapas são de fundamental importância para a compreensão total e definição do escopo do projeto que levará a benefícios, como a maior aceitação do software desejado entre envolvidos (cliente, desenvolvedores, etc.), melhoria no

cumprimento de prazos e orçamento, melhoria da qualidade e melhor desempenho dos testes.

Com as etapas de requisitos e análise concluídas, inicia-se o projeto do software, que se concentra em quatro atributos distintos do programa: estrutura de dados, arquitetura do software, detalhes procedimentais e caracterização da interface.

Essa etapa traduz todas as exigências definidas nas etapas anteriores, em representações de software que podem ser avaliadas antes que a codificação se inicie. Estas representações, permitem que programadores de diversas localidades realizem a codificação, sem a necessidade de um vasto conhecimento do projeto, apenas dos trechos do programa a eles atribuídos. A etapa de projeto gera grande parte dos documentos que guiarão o desenvolvimento do software.

A etapa de codificação é aquela intuitivamente associada com a tarefa de programação e envolve a geração de linhas de código para cada uma das funcionalidades e módulos especificados na documentação das etapas anteriores. Em última instância, significa traduzir o projeto em forma identificável para sistema computacional do qual o programa fará parte.

Após a codificação, o software passa então pela etapa de testes (ou validação). Nesta etapa é feita a verificação dos aspectos lógicos internos do software (garantindo que todas as instruções foram corretamente codificadas), aspectos externos (verificando se o software encontra-se apto para a tarefa à qual foi preparado) e se atende aos requisitos solicitados e especificados em contrato.

Finalmente, o software é entregue ao cliente e posto em operação. Em grande parte dos casos, o software demandará alterações posteriores e outros tipos de serviços, tais como adaptações a mudanças no sistema da empresas ou mesmo acréscimo de funcionalidades. Esse tipo de serviço pode ser incluído no contrato, mas não é previsto no modelo cascata.

A passagem de cada etapa deste processo para a próxima é marcada por uma verificação e um encerramento formal. Teoricamente, todos os passos da etapa anterior são concluídos antes que a próxima inicie. Por isso, o ciclo de vida de um projeto que segue o modelo cascata é marcado por fases bem distintas, ao contrário de outros modelos de ciclo de vida que adotam abordagens iterativas e incrementais.

Na prática, raramente esse método é seguido pelos programadores devido à sua extrema rigidez e devido à dificuldade de conhecer, ainda nos passos iniciais todos os requisitos de projetos, freqüentemente complexos e em alguns casos desconhecidos até mesmo pelos clientes.

Assim, esse modelo tende a se adequar a projetos em que os requisitos do software são bem conhecidos pelos clientes e envolvendo procedimentos maduros.

2.1.1 Características do Modelo

É importante notar que apesar do modelo cascata ter sido muito criticado pelos formuladores dos modelos alternativos de desenvolvimento de software, ele contribuiu e ainda contribui significativamente para o desenvolvimento de outros modelos, como por exemplo: espiral, evolucionário, prototipação etc. Além disso, o surgimento de novos modelos só foi possível pela evolução paralela das linguagens de programação, dos compiladores e dos programas de apoio, como as atuais ferramentas CASE (*Computer-Aided Software Engineering*).

Um exemplo de como estas novas técnicas e ferramentas influenciaram na evolução dos modelos de desenvolvimento de software, era a dificuldade enfrentada pelos desenvolvedores na década de 1960, de programar usando uma linguagem estruturada, como FORTRAN ou COBOL, para executar um projeto onde os requisitos do software eram alterados com grande freqüência.

Sem a capacidade de compilar módulos e integrá-los com facilidade e sem estruturas de apoio aos desenvolvedores, qualquer um dos modelos modernos de desenvolvimento sofreria sérios danos no seu desempenho ou, em alguns casos, teriam sua existência comprometida.

Em virtude das características, citadas anteriormente em relação ao modelo cascata, os procedimentos como reuso de código, estimativa de prazos e manutenção, se davam de forma totalmente diferente ao que é comum nos dias atuais. A estimativa de prazos era feita através de métodos não formais e uma única vez, no inicio do projeto.

No contexto dos projetos que predominava no período em que o modelo cascata foi proposto, não se usava projetos anteriores para auxiliar na estimativa de prazos e custos para novos projetos. A manutenção também era vista como um prejuízo ou perda e

para evitá-la, punha-se uma forte ênfase na especificação, projeto e codificação para evitar qualquer não conformidade futura.

Vale, por fim, destacar as partes consideradas cruciais para o bom desenvolvimento de um projeto, segundo Royce (1970): projeto, documentação, experimentação, teste e envolvimento do cliente. O primeiro ponto, projeto, necessita de uma boa proposta de software para que o desenvolvimento ocorra sem percalços.

Já a documentação, requer gerenciamento para manter a uniformidade do projeto ao longo das etapas. A experiência da equipe é fundamental para estimar prazos (devido à ausência de metodologias específicas) e projetos totalmente originais precisam ser simulados.

E por fim, diferentemente de como se costuma apresentar o modelo cascata, Royce (1970) estimula a experimentação, através do desenvolvimento iterativo, e a participação do cliente no projeto em diversas etapas: antes de iniciar o projeto, ao apresentar os requisitos do sistema; após o projeto preliminar do software; após o projeto do software e após os testes. Em muitos casos, estas duas recomendações (experimentação e envolvimento do cliente), foram esquecidas ou desvirtuadas.

2.2 Problemas Evidenciados na Indústria de Software

Em um relatório publicado por Johnson et al. (1999), foi identificado que 74% dos projetos tiveram de ser cancelados ou tiveram que sofrer algum tipo de alteração em relação a prazo, escopo ou orçamento para poderem ser concluídos e, apenas 26% dos projetos analisados, foram concluídos conforme as estimativas iniciais.

Em um outro estudo realizado por Taylor (2001) foram analisados os fatores que levaram os projetos de tecnologia de informação (TI) ao fracasso. A pesquisa abrangeu 1.027 projetos no Reino Unido. Este estudo identificou que apenas 13% dos projetos não falharam e a gerência de escopo, utilizando as práticas da abordagem cascata, foi considerada o fator mais influenciador dos casos de fracassos, sendo citado por 82% dos entrevistados e considerado o fator determinante do fracasso por 25%.

Uma evidência significativa do fracasso da aplicação do modelo cascata vem de um dos usuários mais freqüentes e assíduos do passado: o Departamento de Defesa dos EUA (DoD – *Department of Defense*). A grande maioria dos projetos realizada pelo DoD

até o final da década de 1980, adotava o ciclo de vida do modelo cascata (era exigência do padrão DOD-STD-2167). Em 1994 foi publicado um novo padrão, o MIL-STD-498 que sugeria a utilização de abordagens iterativas e incrementais no lugar do modelo cascata (LARMAN, 2003).

Em um outro estudo, realizado com uma amostra de 6.700 projetos, foram identificadas duas características que contribuem para o fracasso de projetos de software, e que estão associados ou são agravados pela utilização do modelo cascata. Uma delas é a inabilidade de lidar com mudança de requisitos e a outra é o problema enfrentado com integrações tardias (JONES, 1995).

Outro índice que merece atenção é a relação apontada pelo trabalho realizado por Johnson et al. (1999), onde foi identificado que 20% das funcionalidades entregues no produto final são utilizadas sempre ou com freqüência, 16% são utilizadas algumas vezes, e a grande maioria das funcionalidades, 64%, raramente ou nunca são utilizadas.

Estes números mostram que grande parte do investimento realizado nos projetos de software não gera valor para os clientes finais. Estes números são influenciados pela necessidade de fazer o levantamento dos requisitos no inicio do projeto, momento que o cliente não tem conhecimento das suas necessidades reais.

Um estudo publicado em 2001 sumarizou os resultados de uma pesquisa realizada com mais de 400 projetos, num período de 15 anos (COHEN; LARSON; WARE, 2001). Este estudo identificou que menos de 5% do código produzido era útil ou utilizável e que grande proporção dos códigos não utilizáveis, era reduzida quando utilizado ciclos iterativos e evolucionários com períodos menores (reduzindo o tempo de lançamento de seis meses para duas semanas).

Entre os motivos apontados na pesquisa para estes resultados, pode-se ressaltar: usuários não puderam fornecer comentários antes da entrega, mudanças nas regras do negócio depois de finalizada a fase de levantamento de requisitos e as operações de negócios eram desconhecidas.

Os problemas apresentados nesta seção explicitam a necessidade de uma abordagem alternativa ao modelo cascata. As seções a seguir apresentam as teorias alternativas que serão utilizadas para compor o modelo de gerenciamento de projetos proposto.

3 Metodologias Ágeis

O termo "ágil" vem sendo usado freqüentemente sem critério, gerando desconfiança nas metodologias ágeis de desenvolvimento de software junto a comunidade cientifica e empresas que desenvolvem software. Para compreender estas metodologias é necessário antes definir o que é agilidade e porque ela é importante no contexto da indústria de software.

O conceito de agilidade, como qualquer outro conceito complexo, possui uma grande variedade de definições. A seguir são apresentadas duas destas definições:

> *"Agilidade é a habilidade de criar e responder a mudanças como forma de manter a lucratividade num ambiente turbulento de negócios."* (HIGHSMITH, 2002).

> *"Agilidade é dinâmica, específica do contexto, e orientada pelo crescimento. Não é sobre crescimento de eficiência, redução de custos etc. É sobre sucesso e vitória: obter sucesso em áreas emergentes e competitivas, ampliando a lucratividade, a participação de mercado e clientes no centro do distúrbio de competitividade que muitas empresas temem atualmente."* GOLDMAN et. al (1994).

O conceito de agilidade tem uma longa história no setor manufatureiro norte-americano. Em resposta às fábricas japonesas que utilizavam eficientemente as práticas de produção enxuta durante a década de 1980 e, a crescente preocupação do congresso norte americano a respeito da queda da lucratividade da indústria americana, criou-se um programa federal para desenvolver o que mais tarde tornou-se a base do conceito de manufatura ágil.

Um conjunto abrangente e integrado de estratégias de negócios, modelos, melhores práticas e estudos de caso foi desenvolvido durante seis anos de pesquisas estratégicas conduzidas pelo *Agility Fórum* do *Iacocca Institute*, entre os anos de 1991 e 1998. Mais de 150 empresas líderes nos setores onde atuavam, iniciaram programas de agilidade, incluindo: *Boeing*, *Goodyear*, *IBM*, *Kodak*, *Texas Instruments* e *Xerox*. (GOLDMAN et. al, 1994).

No ano de 2001, o movimento ágil foi formalizado na indústria através do "Manifesto Para o Desenvolvimento Ágil de Software", também conhecido como "Manifesto Ágil" (MANIFESTO, 2001). Este documento foi elaborado e assinado em um encontro em *Snowbird*, no Estado de Utah, nos EUA. Entre os signatários estavam representantes das metodologias *Extreme Programming* (XP), *Scrum*, *DSDM*, *Adaptative Software Development*, *Crystal*, *Feature Drivem Development*, *Pragmatic Programming* e outros defensores da necessidade de formas alternativas de desenvolver software no lugar dos modelos tradicionais, como por exemplo, o modelo cascata.

O "Manifesto Ágil" corresponde a um documento que descreve um conjunto de princípios e valores, onde todos os participantes e signatários chegaram ao consenso de sua relevância. Esses conceitos evidenciam melhores formas de desenvolver software, onde passam a valorizar:

- Indivíduos e interações **mais** que processos e ferramentas
- Softwares funcionais **mais** que documentação
- Colaboração com o cliente **mais** que negociações de contrato
- Responder a mudanças **mais** que seguir um plano

Uma observação importante e comumente mal interpretada é de que esta corrente de metodologias desconsidera a importância de processos, ferramentas, documentações, contratos e planejamento. Ao contrário dessas críticas, elas apenas valorizam **mais** indivíduos e interações do que processos e ferramentas, softwares funcionais **mais** do que a elaboração de documentações, colaboração com o cliente **mais** do que a negociação de contratos e a resposta a mudanças **mais** do que a conformidade aos planos.

Duas metodologias que fazem parte do movimento ágil foram selecionadas para compor o modelo proposto, são elas: *Scrum* e *Extreme Programming* (XP). Estas duas metodologias foram selecionadas devido à sinergia entre elas e por serem complementares nos aspectos de gerenciamento de projetos (*Scrum*) e no desenvolvimento de software (XP).

O *Scrum* não explicita um processo de desenvolvimento, ele é comparado a um *framework* de gerenciamento de projetos, enquanto que o XP aprofunda e detalha a

questão do desenvolvimento de software, não contemplando alguns aspectos gerenciais, o que acabou trazendo um pouco de descrédito à metodologia.

Ao combinar estas duas metodologias num modelo de gerenciamento de desenvolvimento de software, espera-se utilizar as vantagens de cada uma delas para obter ganhos de produtividade e melhoria da qualidade do produto final, contornando as deficiências destas metodologias.

As seções a seguir descrevem detalhadamente as metodologias selecionadas, abrangendo seus papéis e responsabilidades, as práticas, o processo e o escopo de utilização de cada metodologia.

3.1 Scrum

A primeira referência na literatura ao termo "*Scrum*" aplicado ao desenvolvimento de produtos vem de um artigo escrito por Takeuchi e Nonaka (1986), onde foram sumarizadas as melhores práticas comuns utilizadas por empresas japonesas inovadoras. Neste trabalho é descrito um processo adaptativo, rápido e auto organizado. Esta idéia foi posteriormente aprimorada e refinada, por estes autores, em um outro trabalho (TAKEUCHI; NONAKA, 1995).

A abordagem *Scrum* foi desenvolvida para gerenciar o processo de desenvolvimento de sistemas de software. Ela consiste em uma abordagem empírica, aplicando idéias da teoria de controle de processo industrial para desenvolvimento de software, introduzindo idéias de flexibilidade, adaptabilidade e produtividade (SCHWABER & BEEDLE, 2001).

O *Scrum* não define uma técnica específica para o desenvolvimento de software durante a etapa de implementação, ele se concentra em descrever como os membros da equipe devem trabalhar para produzir um sistema flexível, num ambiente de mudanças constantes.

A idéia central do *Scrum* é que o desenvolvimento de sistemas envolve diversas variáveis (ambientais e técnicas) e elas possuem grande probabilidade de mudar durante a execução do projeto (por exemplo: requisitos, prazos, recursos, tecnologias etc.). Estas características tornam o desenvolvimento do sistema de software uma tarefa complexa e imprevisível, necessitando de um processo flexível e capaz de responder às mudanças.

O *Scrum* ajuda a melhorar práticas de engenharia de software existentes dentro da organização (por exemplo: práticas de testes etc.), para isso ele possui atividades de gerenciamento freqüentes com o objetivo de identificar deficiências e restrições no processo de desenvolvimento bem como nas práticas utilizadas.

3.1.1 Papéis e Responsabilidades

Existem seis papéis no *Scrum* que possuem tarefas e propósitos diferentes durante o processo e suas práticas: *Scrum Master*, Responsável pelo Produto, Equipe *Scrum*, Cliente, Usuário e Gerente. A seguir, estes papéis são descritos de acordo com as definições de Schwaber and Beedle (2001).

- **Cliente**: participa das tarefas relacionadas à definição da lista de funcionalidade do software sendo desenvolvido ou melhorado, elaborando os requisitos e restrições do produto final desejado.
- **Gerente**: é encarregado pela tomada das decisões finais, utilizando as informações visuais disponibilizadas graficamente pelos padrões e convenções a serem seguidas no projeto. Ele também é responsável por acordar, junto aos Clientes, os objetivos e requisitos do projeto.
- **Equipe *Scrum***: é a equipe de projeto que possui autoridade de decidir sobre as ações necessárias e de se organizar para poder atingir os objetivos pré-estabelecidos. A Equipe *Scrum* é envolvida, por exemplo, na estimativa de esforço, na criação e revisão da lista de funcionalidade do produto, sugerindo obstáculos que precisam ser removidos do projeto.
- ***Scrum Master***: é responsável por garantir que o projeto esteja sendo conduzido de acordo com as práticas, valores e regras definidas no *Scrum* e que o progresso do projeto está de acordo com o desejado pelos Clientes. O *Scrum Master* interage tanto com a Equipe *Scrum*, como com os Clientes e o Gerente durante o projeto. Ele também é responsável por remover e alterar qualquer obstáculo ao longo do projeto, para garantir que a equipe trabalhe da forma mais produtiva possível.
- **Responsável pelo Produto**: é oficialmente responsável pelo projeto, gerenciamento, controle e por tornar visível a lista de funcionalidade do produto.

Ele é selecionado pelo *Scrum Master*, Clientes e Gerente. Ele também é responsável por tomar as decisões finais referentes às tarefas necessárias para transformar a lista de funcionalidades no produto final, participando na estimativa do esforço de desenvolvimento necessário e é responsável pelo detalhamento das informações referentes à lista de funcionalidade utilizada pela Equipe *Scrum*.

3.1.2 Práticas

O *Scrum* não provê nem exige um método de desenvolvimento de software específico. Ao invés disto, ele estabelece algumas práticas utilizadas nas várias fases do *Scrum*, propondo uma organização do trabalho adequada para situações caracterizadas por incertezas e complexidades (SCHWABER, 1995).

A seguir, são descritas brevemente as práticas do *Scrum* conforme Schwaber e Beedle (2001).

3.1.2.1 Elaborar e Revisar o *Backlog* do Produto

O *backlog* do produto define todas as funcionalidades e características que devem ser contempladas no produto final, de acordo com o conhecimento atual dos envolvidos no projeto. Desta forma, esta lista define os requisitos e o trabalho que deve ser realizado no projeto.

O *backlog* do produto contempla uma lista priorizada e constantemente atualizada dos requisitos técnicos e de negócio do sistema que está sendo construído ou aperfeiçoado. Os itens do *backlog* podem incluir, por exemplo: novas características, funcionalidades, correções, pedido de melhorias, atualizações das tecnologias utilizadas etc. Também são incluídas na lista de *backlog* restrições que devem ser resolvidas antes que outros itens possam ser levados em consideração. Múltiplos atores podem participar na geração do *backlog* do produto (Cliente, Equipe *Scrum*, Responsável pelo Produto, Gerente etc.).

Esta prática inclui tarefas para criar a lista de *backlog* do produto e controlá-las de forma consistente durante o processo através da adição, remoção, especificação, atualização e priorização dos itens da lista. O Responsável pelo Produto é encarregado por manter o *backlog* do produto atualizado e priorizar os itens que o compõem.

3.1.2.2 Estimar o Esforço

A estimativa de esforço é realizada iterativamente, onde a quantidade de trabalho necessária para transformar os itens da lista de *backlog* em incrementos do produto, é refinada de acordo com a disponibilidade de informações que são descobertas e/ou fornecidas sobre o item ao longo do projeto.

O Responsável pelo Produto e a Equipe *Scrum* são responsáveis por estimar o esforço necessário para implementar cada um dos itens que compõem o *backlog* do produto.

3.1.2.3 Desenvolvimento Iterativo (*Sprint*)

Corresponde ao ciclo iterativo onde é realizado o desenvolvimento do software e que auxilia na adaptação para as mudanças nas variáveis ambientais (requisitos, tempo, recursos, conhecimento, tecnologia etc.). A Equipe *Scrum* organiza-se para produzir novos incrementos de produtos executáveis a cada s*print*. As práticas e ferramentas utilizadas pela equipe dentro de cada *sprint* são:

- Reunião de planejamento do s*print*;
- *Backlog* do s*print*;
- Reunião diária do *Scrum*;
- Reunião de revisão do *sprint*.

A Figura 4 apresenta um fluxograma relacionando o *sprint* com suas práticas, entradas e saídas.

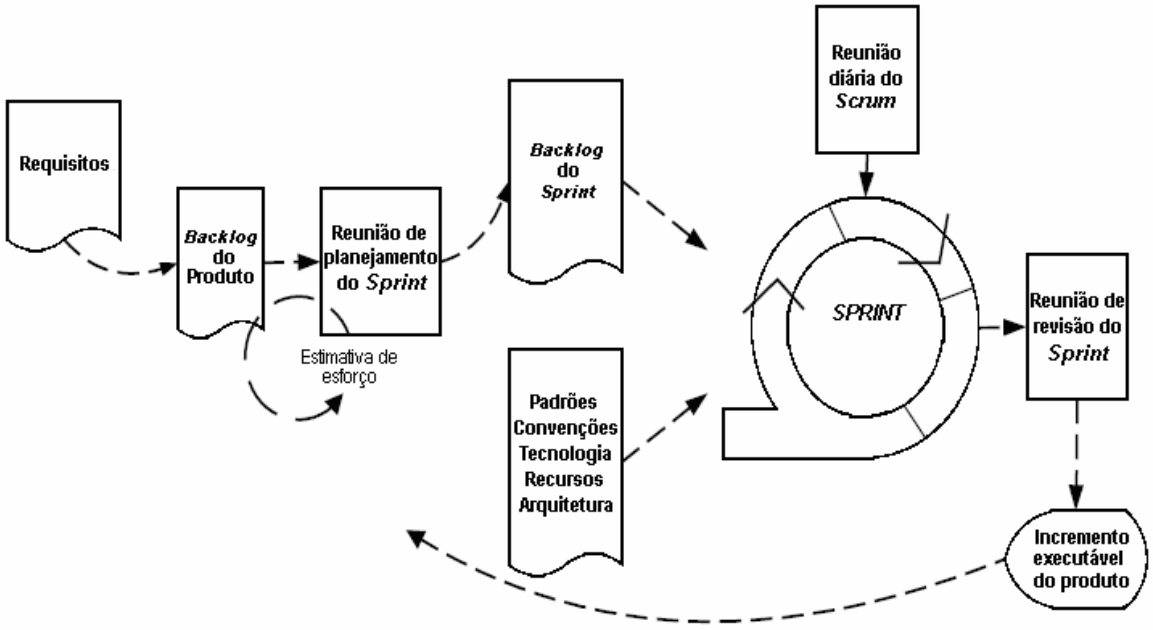


Figura 4: Ciclo de desenvolvimento proposto pelo *Scrum*
(SCHWABER, 1995).

3.1.2.4 Reunião de Planejamento do *Sprint*

A reunião de planejamento do s*print* é uma reunião de duas fases organizada pelo *Scrum Master*. Os Clientes, o Responsável pelo Produto e a Equipe *Scrum* participam da primeira fase da reunião para decidirem sobre os objetivos e funcionalidades do próximo s*print*.

A segunda fase da reunião é conduzida pelo *Scrum Master* e pela Equipe *Scrum* focando em planejar como o incremento do produto será desenvolvido ao longo do próximo s*print*.

3.1.2.5 Elaborar o *Backlog* do *Sprint*

O *backlog do sprint* é o ponto de partida de cada *sprint*. Ele corresponde a uma lista de itens selecionados a partir da lista de *backlog* do produto que devem ser desenvolvidos no próximo *sprint*.

Os itens são selecionados pela Equipe *Scrum*, juntamente com o *Scrum Máster*, o Responsável pelo Produto e o Cliente, na reunião de planejamento do *sprint*. Esta seleção é feita de acordo com os itens priorizados e com os objetivos definidos para o s*print*.

Diferente do *backlog* do produto, o *backlog* do *sprint* é estável até o final do *sprint* (iteração). Quando todos os itens do *backlog* do *sprint* estão implementados, um novo incremento funcional do produto é entregue aos clientes.

3.1.2.6 Reunião Diária do *Scrum*

As reuniões diárias do *scrum* têm o objetivo de monitorar continuamente o progresso da Equipe *Scrum*. Elas também servem como reuniões de planejamento, identificando o que foi feito desde a última reunião e o que precisa ser feito até a próxima.

Os problemas e outras questões relevantes, também são discutidos e controlados nestes pequenos encontros diários (onde a duração sugerida é de quinze minutos aproximadamente). Nestas reuniões, qualquer deficiência e/ou obstáculo ocorridos durante o desenvolvimento do produto, são identificados, analisados e removidos para melhorar o andamento do projeto.

O *Scrum Master* é responsável por conduzir estas reuniões, que não devem ser utilizadas para resolver problemas isolados. Os problemas levantados são tratados após a reunião, por subgrupos relevantes da Equipe *Scrum*. Durante a reunião, cada membro da Equipe *Scrum* responde a três perguntas.

1) O que foi realizado ontem?
2) O que será realizado hoje?
3) Existe algum obstáculo/impedimento?

3.1.2.7 Reunião de Revisão do *Sprint*

No final de cada s*print*, a Equipe *Scrum* e o *Scrum Master* apresentam os resultados obtidos ao final da iteração (o incremento funcional do produto) para os Clientes, Gerente e para o Responsável pelo Produto na reunião de revisão do *sprint*.

Geralmente estas reuniões têm o formato de uma apresentação do incremento funcional do produto desenvolvido no *sprint* encerrado. Os participantes avaliam as funcionalidades novas do produto e tomam decisões a respeito das atividades seguintes, de acordo com os objetivos estabelecidos na reunião de planejamento do *sprint*.

A reunião de revisão do *sprint* pode acrescentar, remover ou alterar itens do *backlog* do produto, bem como pode mudar o rumo do produto que está sendo desenvolvido.

3.1.3 Processo

O *Scrum* define um *framework* iterativo de gerenciamento de projeto de desenvolvimento de software. Nele o período sugerido de cada iteração (denominado *sprint*) é de sete a trinta dias. O *framework* do *Scrum* é composto de três fases denominadas: pré-game, desenvolvimento e pós-game. O software é desenvolvido através da realização de sucessivos *sprints*.

A Figura 5 ilustra um ciclo de vida da metodologia *Scrum*, apresentando suas fases, práticas, ferramentas e as interações entre elas (com entradas e saídas).

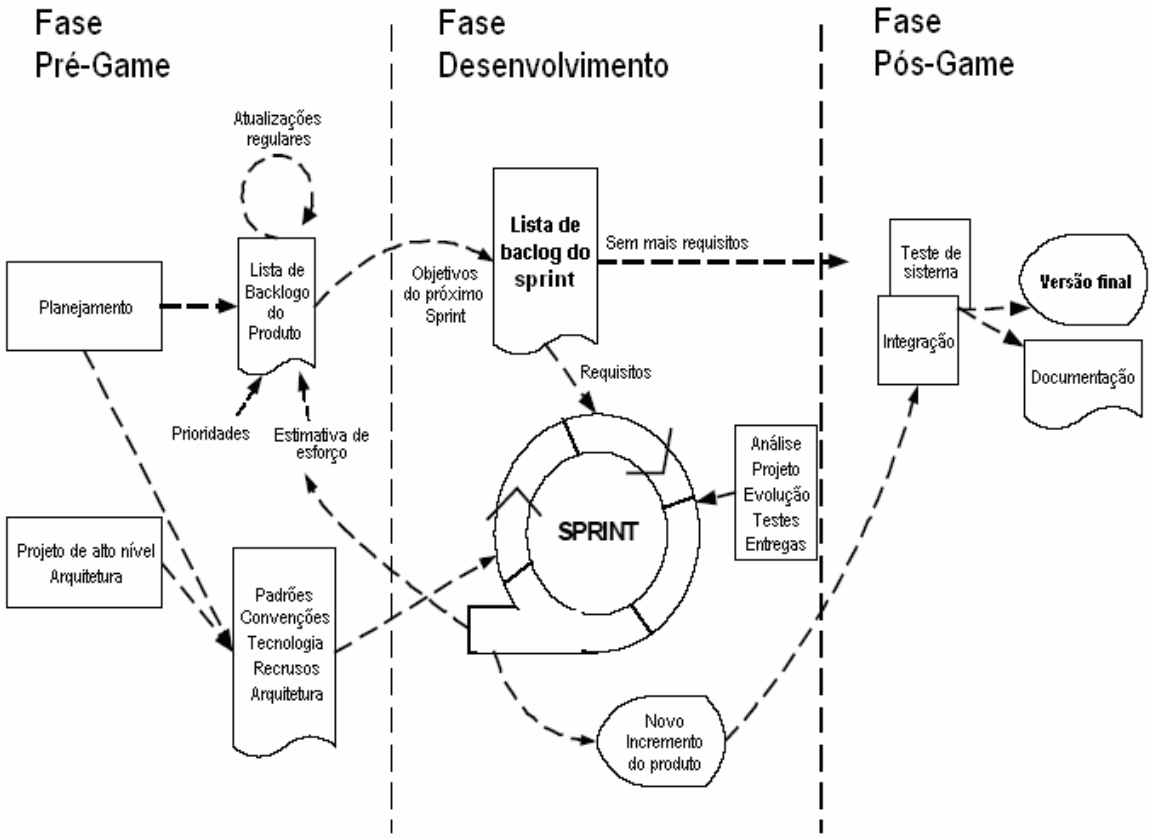


Figura 5: Visão geral do ciclo de vida do *Scrum* (SCHWABER, 1995).

O trabalho inicia-se com o levantamento dos requisitos e a elaboração de uma lista de *backlog* do produto. Juntamente com o *backlog* do produto, são estabelecidos

padrões e convenções referentes a tecnologia, recursos e arquitetura que são utilizadas ao longo do projeto.

Em seguida são estabelecidos objetivos, de acordo com as características do produto, para o próximo *sprint*. Após definido o objetivo, são selecionados itens do *backlog* do produto que constituirão o *backlog* do *sprint*, estes itens são trabalhados no período definido inicialmente (que pode variar de sete a trinta dias) e ao final da iteração, será entregue aos Clientes um incremento funcional do produto. Este ciclo se repete até serem concluídos todos os itens que compõem o *backlog* do produto.

Nas seções a seguir, as fases Pré-*Game*, Desenvolvimento e Pós-*Game* do *Scrum* são descritas de acordo com Schwaber (1995) e Schwaber e Beedle (2001).

3.1.3.1 Pré-*Game*

A fase denominada Pré-*Game* possui duas atividades principais: planejamento e projeto de alto nível da arquitetura.

- O planejamento inclui a definição do produto a ser desenvolvido. É criada a lista de *backlog* do produto contendo todos os requisitos conhecidos até o momento. Os requisitos podem ter sido originados pelos Clientes, Responsável pelo Produto, *Scrum Master*, Gerente ou pela própria Equipe *Scrum*. Os requisitos são priorizados e o esforço necessário para implementá-los são estimados. A lista de *backlog* do produto é constantemente atualizada com novos itens, mais detalhes, estimativas mais precisas e novas ordens de prioridades. O planejamento também inclui a definição da equipe de projeto, ferramentas e outros recursos, avaliação de riscos, variáveis de controle, necessidades de treinamento e verificação da aprovação de gerência. Em cada iteração (*sprint*), a lista de *backlog* do produto é atualizada e revisada pela equipe, para desta forma, obter o comprometimento dos envolvidos para a próxima iteração.
- O projeto de alto nível da arquitetura é realizado de acordo com os itens presentes na lista de *backlog* do produto. No caso de uma melhoria em um sistema já existente, são identificadas as mudanças necessárias para implementar os itens do *backlog* junto com os problemas que estas alterações podem ocasionar. Uma

reunião para avaliar as propostas de implementação é realizada e planos preliminares para definir os conteúdos dos incrementos do produto são preparados.

3.1.3.2 Desenvolvimento

Esta fase corresponde à parte ágil da metodologia *Scrum*. Ela divide a codificação dos requisitos em iterações (*sprints*), onde as incertezas são tratadas e mitigadas. As diferenças ambientais e diferenças técnicas (por exemplo: qualidade, requisitos, recursos, tecnologias de implementação e ferramentas e até mesmo métodos de desenvolvimento) identificadas no *Scrum*, que podem mudar durante o processo, são observadas e controladas através de várias práticas durante os s*prints* (reuniões de planejamento do *srpint*, reunião de revisão do *sprint*, reuniões diárias etc.).

Ao invés de levar em consideração estas características apenas no início do projeto de desenvolvimento de software, o *Scrum* estabelece o controle constante para poder adaptar de forma flexível às mudanças.

Dentro da fase de desenvolvimento, o sistema é construído em s*prints* (iterações). *Sprints* são ciclos iterativos onde as funcionalidades são desenvolvidas ou aprimoradas para produzir novos incrementos. Cada s*print* inclui as etapas tradicionais do desenvolvimento de software: requisitos, análise, projeto e entregas evolutivas. A arquitetura e o projeto do sistema evoluem durante o desenvolvimento do sistema.

3.1.3.3 Pós-*Game*

Esta fase é iniciada quando é acordado que as variáveis técnicas e ambientais (como por exemplo, os requisitos) estão completos, e o sistema ou o incremento do sistema está pronto para a utilização.

O fechamento de cada *sprint* contempla práticas, como por exemplo, integração, teste de sistema, reunião de revisão dos resultados alcançados e documentação.

3.1.4 Escopo de Utilização

O Scrum é uma metodologia destinada a pequenas equipes com menos de dez pessoas. Schwaber e Beedle (2001) sugerem que a equipe seja composta de cinco a nove

integrantes, se mais pessoas estiverem envolvidas no projeto, devem-se formar múltiplas Equipes *Scrum*.

3.2 Extreme Programming

O *Extreme Programming* (XP) surgiu como uma tentativa para solucionar os problemas causados pelos ciclos de desenvolvimento longos dos modelos de desenvolvimento tradicionais. Ele é composto por práticas que se mostraram eficientes nos processos de desenvolvimento de software nas últimas décadas. Depois de aplicado e obtido sucesso num caso real, o XP foi formalizado através de quatro princípios chaves e doze práticas (BECK, 1999).

Quando analisadas individualmente, as práticas que compõem o XP não são novas. No XP estas práticas foram reunidas e alinhadas de tal forma a criar uma interdependência entre elas, e assim, deu origem a uma nova metodologia de desenvolvimento de software. Os quatro princípios chaves do XP são:

- **Comunicação**: muitos dos problemas que ocorrem no decorrer do projeto podem ser relacionados com problemas de comunicação entre a equipe ou entre a equipe do projeto e o próprio cliente. Uma pessoa pode deixar de comunicar um fato importante à outra pessoa, um programador pode deixar de levantar uma questão importante ao cliente etc. O XP mantém o fluxo de comunicação através de algumas práticas que não podem ser realizadas sem comunicação. Exemplos disto são: testes de unidade, programação em pares e estimativa do esforço de cada tarefa.
- **Simplicidade**: deve-se sempre selecionar a alternativa mais simples que possa funcionar. O XP se baseia no fato que é mais barato fazer algo mais simples e alterá-lo conforme as necessidades forem surgindo do que tentar prever as necessidades futuras, introduzindo uma complexidade que possa vir a não ser necessária no futuro.
- **Realimentação**: todo problema é evidenciado o mais cedo possível para que possa ser corrigido o mais cedo possível. Toda a oportunidade é descoberta o mais

cedo possível para que possa ser incorporada de forma rápida ao produto que está sendo construído.

- **Coragem**: é preciso coragem para apontar um problema no projeto, para solicitar ajuda quando necessário, para simplificar o código que já está funcionando (podendo introduzir novos defeitos), comunicar ao cliente que não será possível implementar um requisito no prazo estimado e, até mesmo, para fazer alterações no processo de desenvolvimento.

3.2.1 Papéis e Responsabilidades

Existem diferentes papéis sugeridos pelo XP para diferentes fases, práticas e ferramentas necessárias ao longo do projeto. A seguir, estes papéis são descritos de acordo com Beck (1999).

- **Programador**: escrevem testes e mantém o programa o mais simples e conciso possível. A primeira característica que torna o XP possível é a habilidade de comunicação e coordenação com outros membros da equipe.
- **Cliente**: escreve as estórias e os testes funcionais, além de decidir quando cada requisito foi satisfeito. O cliente também define a prioridade de implementação de cada requisito.
- **Testador**: ajuda o cliente a escrever os testes funcionais. Ele também realiza os testes funcionais regularmente, comunicando os resultados dos testes e mantém o conjunto de testes.
- **Monitor**: fornece a realimentação para a equipe do projeto. Ele acompanha a conformidade das estimativas feitas pela equipe de desenvolvimento (por exemplo, estimativas de esforço) e fornece comentários de quanto acuradas elas estão, para poder melhorar futuras estimativas. Ele também acompanha o progresso de cada iteração e avalia se o objetivo é viável dentro das limitações de tempo e recursos, ou se alguma mudança é necessária no processo.
- **Treinador**: é a pessoa responsável pelo processo como um todo. Um profundo conhecimento do XP é importante para este papel, pois é ele que guiará os outros envolvidos no projeto a executar o processo de forma adequada.

- **Consultor**: é um membro externo com conhecimento técnico específico necessário para o projeto. O consultor auxilia a equipe a resolver problemas específicos.
- **Chefe**: responsável pelas tomadas de decisões. Para isso, ele comunica-se com a equipe de projeto para determinar a situação atual e para identificar qualquer dificuldade ou deficiência do processo.

3.2.2 Práticas

O XP é composto por doze práticas que são descritas a seguir:

- **Jogo de planejamento**: nesta prática existe uma grande interação entre o Cliente e os Programadores. Os Programadores estimam o esforço necessário para implementar as estórias definidas pelo Cliente e este, decide sobre o escopo e duração das iterações.
- **Incrementos curtos e pequenos**: um incremento simples e funcional é gerado rapidamente pelo menos uma vez a cada dois ou três meses. Desta forma é possível ter um retorno por parte do Cliente em tempo hábil para poder incorporar mudanças e corrigir o produto sendo desenvolvido.
- **Metáfora**: é elaborada uma descrição que permite todos envolvidos no projeto (Clientes, Programadores, Gerente etc.) explicar como o sistema funciona. Ela cria uma visão comum e sugere uma estrutura de como o problema e a solução são percebidos no contexto do sistema sendo produzido. Ela também auxilia os envolvidos a compreender os elementos básicos do sistema e seus relacionamentos, criando um vocabulário comum para o projeto.
- **Projeto simples**: o sistema deve ser projetado da forma mais simples possível de acordo com as necessidades atuais do projeto. As complexidades desnecessárias são removidas assim que são descobertas.
- **Testes**: o desenvolvimento do software é dirigido por testes. Os testes de unidade são desenvolvidos antes da codificação são executados continuamente. Os testes funcionais (aceitação) são escritos pelo Cliente.

- **Reestruturação**[1]: melhoria do sistema através da remoção de duplicações de código, melhorando a comunicação, simplificando e adicionando flexibilidade.
- **Programação em pares**: dois Programadores escrevem o código em um único computador.
- **Propriedade coletiva**: qualquer Programador pode alterar qualquer parte do código em qualquer lugar do sistema a qualquer momento.
- **Integração contínua**: o sistema é integrado e são geradas versões internas, diversas vezes ao dia, sempre que uma estória é finalizada.
- **Semanas de 40 horas**: não se deve trabalhar mais do que quarenta horas por semana, isto deve ser encarado como uma regra. Nunca trabalhe mais do que isso duas ou mais semanas consecutivamente.
- **Cliente no local**: um usuário real do produto (Cliente) deve ser adicionado à equipe de Programadores. Ele deve estar disponível em tempo integral para responder as eventuais dúvidas.
- **Padrão de codificação**: os Programadores escrevem todo o código de acordo com regras que enfatizam a comunicação durante a codificação. Antes do início do projeto deve ser definido um padrão que deverá ser seguido por toda a equipe de Programadores.

As práticas do XP são agrupadas em quatro grupos de acordo com sua finalidade, começando da elipse central para a extremidade: Codificação, Equipe, Processo e Produto. A Figura 6 apresenta uma ilustração gráfica do agrupamento das práticas do XP, descritas anteriormente.

[1] Reestruturação (do inglês Refactoring) é o processo de modificar um sistema de software para melhorar a estrutura interna do código sem alterar seu comportamento externo.

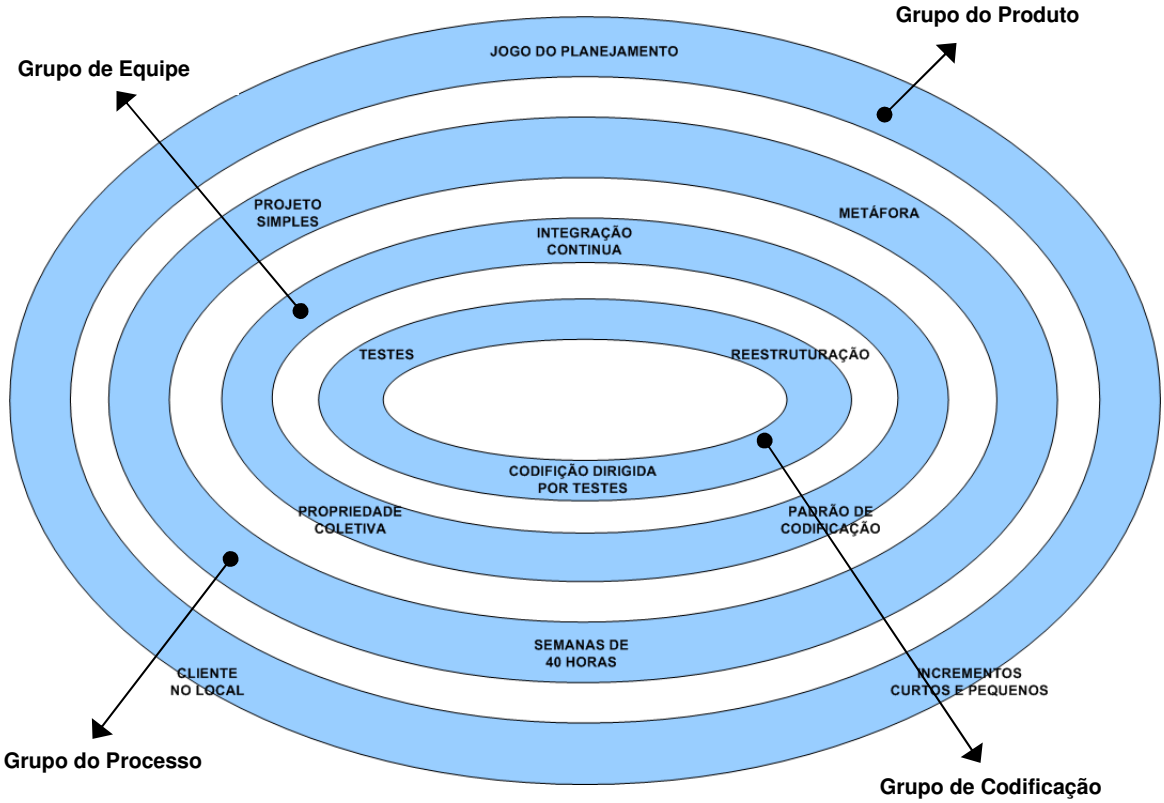


Figura 6: Agrupamento das práticas do XP (BECK, 1999).

No XP, as práticas que compõem os dois grupos internos da Figura 8 (Codificação e Equipe) são mais detalhadas e definidas que as práticas representadas pelos dois grupos externos (Processo e Produto).

Devido à falta de foco nas práticas relacionadas ao gerenciamento do projeto, o XP tem sido criticado e questionado por parte dos gerentes e diretores de empresas desenvolvedoras de software e por parte da comunidade científica.

3.2.3 Processo

O ciclo de vida do XP é composto de cinco fases: Exploração, Planejamento, Iterações para o Lançamento, Produção, Manutenção e Morte. A Figura 7 ilustra o ciclo de vida desta metodologia.

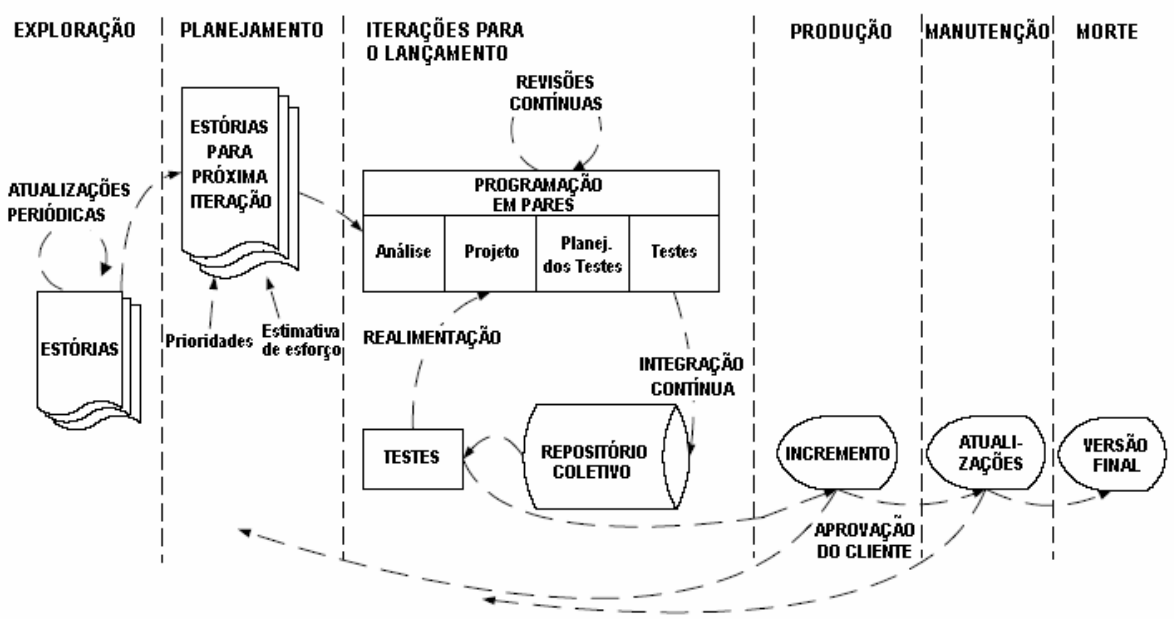


Figura 7: Ciclo de vida proposto pelo XP (ABRAHAMSSON et al., 2002)

As seções a seguir descrevem sucintamente cada uma das fases que compõem o ciclo de vida proposto pelo XP.

3.2.3.1 Exploração

Nesta fase os Clientes escrevem os cartões de estórias (requisitos) que desejam incluir no produto a ser construído, onde cada cartão descreve uma característica a ser implementada.

Ao mesmo tempo a equipe de projeto familiariza-se com as ferramentas, tecnologias e práticas que serão utilizadas no projeto. A tecnologia que será utilizada é testada e um esboço da arquitetura é criado e em seguida, validado através de experimentações.

A fase de exploração pode durar de algumas semanas a alguns meses, dependendo da familiaridade que os Programadores possuam com a tecnologia que será utilizada e com o domínio de negócio do produto a ser construído.

3.2.3.2 Planejamento

Depois de escritos os cartões de estórias, eles são ordenados por prioridade e é definido o conteúdo do primeiro incremento do produto, isto é, quais estórias serão adicionadas ao programa.

Primeiramente, os Programadores estimam o esforço necessário para implementar cada uma das estórias para depois definir o cronograma. O prazo para a entrega do primeiro incremento do produto deve ser entre dois e seis meses.

3.2.3.3 Iterações para o Lançamento

O cronograma acordado é dividido em iterações de uma a quatro semanas. Cada iteração produz um conjunto de casos de testes funcionais para cada estória agendada para esta iteração.

A primeira iteração é focada na arquitetura. São selecionadas as estórias que forçam a criar a estrutura do sistema como um todo, mesmo que seja apenas seu esqueleto. Após a primeira iteração, o Cliente passa a selecionar as estórias que serão desenvolvidas em cada iteração. Ao final da última iteração o sistema está pronto para a operação.

A transformação dos cartões de estória em incrementos ocorre através da codificação em pares que é feita paralelamente com as atividades de análise, projeto, planejamento dos testes e codificação dos testes unitários. Estas atividades são realizadas sem a necessidade de uma transição formal entre elas, o que dificulta a determinação das fronteiras entre elas.

3.2.3.4 Produção

Esta fase corresponde à fase final do desenvolvimento e implantação do incremento do produto. O ciclo de realimentação é encurtado, e no final de cada iteração os Clientes podem fornecer informações aos Programadores para corrigir eventuais desvios e inadequações. Tipicamente pode existir algum processo de validação para assegurar que o software esteja pronto para entrar em produção.

3.2.3.5 Manutenção

Depois que o primeiro incremento do produto é posto em produção, é preciso mantê-lo funcional enquanto são produzidos novos incrementos. Para isso, esta fase necessita de um esforço extra para tarefas de apoio ao Cliente, o que pode acarretar numa desaceleração na velocidade de desenvolvimento. A manutenção pode demandar a entrada de novas pessoas na equipe, bem como a mudança na estrutura da equipe.

3.2.3.6 Morte

Esta fase ocorre quando o Cliente não possui mais estórias para serem desenvolvidas e quando não existem mais alterações a serem feitas. É neste momento que é gerada a documentação necessária do sistema. Ela pode ocorrer também quando o Cliente acredita que a equipe possa não conseguir desenvolver o produto desejado (baixa qualidade, inviabilidade econômica etc.).

3.2.4 Escopo de Utilização

De acordo com Beck (1999), a metodologia do XP é apropriada para todos os tipos de projetos de desenvolvimento de software, e ainda não teve todas suas limitações identificadas. Uma restrição levantada é de que a metodologia é apropriada apenas para equipes de pequeno e médio porte, variando de três até vinte membros compondo a equipe de projeto.

3.3 Aspectos Relevantes na Elaboração do Modelo Proposto

A sinergia existente entre as duas metodologias selecionadas pode ser evidenciada uma vez que o *Scrum* não explicita práticas específicas para o desenvolvimento de software, que é realizado diariamente dentro de cada *sprint*. Ele foca nos aspectos relacionados ao gerenciamento dos resultados e na adaptação dos cenários emergentes.

O XP por sua vez, define claramente as práticas que devem ser utilizadas durante o desenvolvimento de software e não enfatiza aspectos relacionados ao gerenciamento de projetos.

O modelo proposto utiliza as práticas e ferramentas de desenvolvimento de software do *Extreme Programming* (XP), agrupadas e representadas pelos três grupos

centrais denominadas Equipe, Codificação e Processo na Figura 6, destacadas na Figura 8.

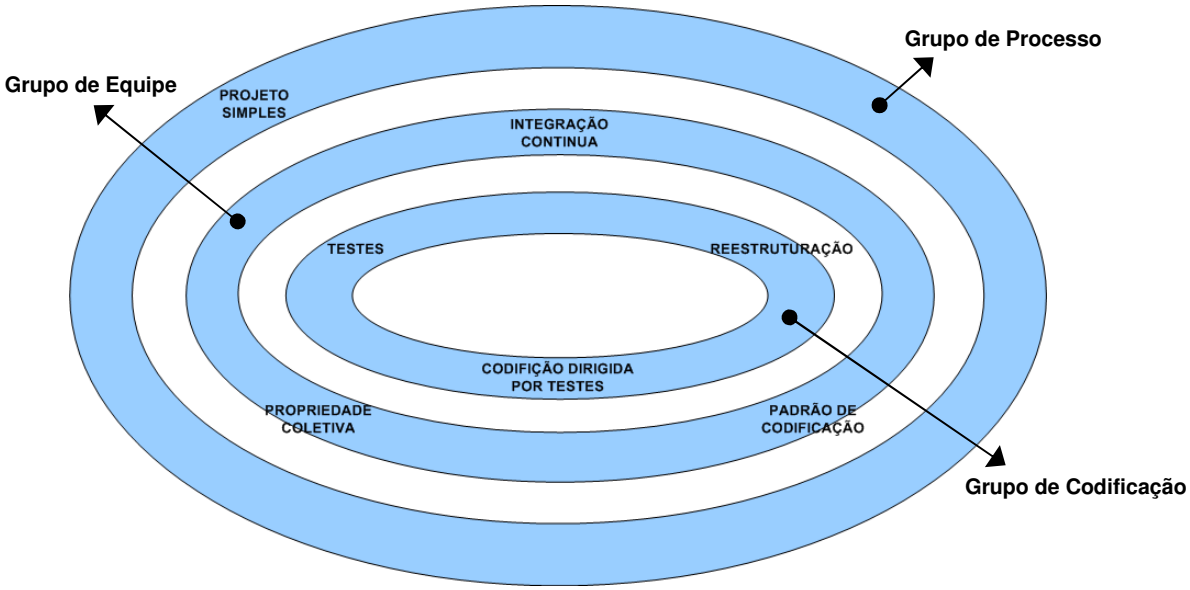


Figura 8: Representação das práticas selecionadas do XP, adaptado de Beck (1999).

Com relação ao *Scrum*, por ele consistir em um *framework* de gerenciamento de projetos e não definir como deve ser realizado o desenvolvimento de software, foram selecionadas todas as suas práticas e ferramentas, fazendo apenas modificações nas nomenclaturas de suas fases e seus papéis para compor o modelo proposto. Estas alterações são justificadas e discutidas posteriormente, na seção **5.1 Papéis e Responsabilidades**.

No modelo proposto, os papéis utilizados são herdados da metodologia *Scrum* (descritas na seção **3.1.1**), com alterações de nomenclatura para generalizar a proposta. Esta opção foi feita para preservar a nomenclatura da metodologia que compôs a maior parte do modelo proposto.

A partir da definição inicial do *Scrum*, foram feitas adaptações para incorporar as responsabilidades de desenvolvimento de software definidas pelo XP (descritas na seção **3.2.1**).

- **Treinador:** este papel herdou a nomenclatura do XP e as responsabilidades do *Scrum.* Ele é responsável por garantir que o projeto esteja sendo conduzido de

acordo com as práticas, ferramentas, valores e regras do modelo proposto e que o progresso do projeto esteja de acordo com o desejado pelos Clientes. O Treinador interage tanto com a Equipe, como com os Clientes e o Gerente. Ele também é responsável por remover e alterar qualquer obstáculo identificado, garantindo que a Equipe trabalhe da forma mais produtiva possível.

- **Líder de Projeto:** este papel herdou as responsabilidades do papel Responsável pelo Produto, definido pelo *Scrum.* No modelo proposto, sofreu apenas a alteração de nomenclatura, para generalizar seu significado para projetos sob demanda e desenvolvimento de produtos.
- **Equipe:** no modelo proposto, este papel assume as responsabilidades do *Scrum,* ou seja, ele possui autoridade para decidir sobre as ações necessárias e de se organizar para poder atingir os objetivos de cada *sprint.* Ela é envolvida na estimativa de esforço, na criação e na revisão dos itens *backlog* e por sugerir os obstáculos que devem ser removidos. Este papel também herdou as responsabilidades dos papéis Testador e Programador do XP, onde auxiliam o Cliente a escrever os testes funcionais, pela manutenção dos mesmos e por executarem periodicamente os testes. A Equipe também é responsável por projetar, codificar e manter o programa o mais simples e conciso possível.
- **Cliente:** papel e responsabilidades herdadas do *Scrum.* No modelo proposto não sofreu alteração.
- **Gerente**: papel e responsabilidades herdadas do *Scrum*. No modelo proposto não sofreu alteração.

A seguir, são apresentadas as práticas e ferramentas que são compartilhadas pelas duas metodologias selecionadas (XP e *Scrum*) e que facilitam a sua combinação:

- **Iterações**: todo o trabalho é feito iterativamente, com o cliente sendo capaz de direcionar e corrigir eventuais desvios do projeto a cada iteração.
- **Incrementos**: toda iteração produz um incremento funcional do produto, contendo os requisitos de maior prioridade definidas pelo cliente. Se

desejado, os clientes podem solicitar que a equipe do projeto coloque estas funcionalidades em produção a qualquer momento.

- **Emergência**: apenas as funcionalidades selecionadas pelos clientes para a próxima iteração são consideradas e codificadas. Os clientes não pagam por funcionalidades que não selecionaram e os desenvolvedores não precisam codificar, depurar e manter códigos irrelevantes.
- **Auto-organização**: os clientes definem o que desejam, a equipe do projeto define o quanto ela pode produzir durante uma iteração e quais atividades cada membro desempenhará.
- **Colaboração**: os clientes e a equipe do projeto colaboram para definirem as melhores alternativas para construir o produto e definem o que deve ser incorporado ao produto a cada iteração.

As metodologias XP e *Scrum* definem práticas complementares, porém algumas delas se sobrepõem, como por exemplo, no Jogo do Planejamento (XP) e no Planejamento do *Sprint* (*Scrum*).

Ambas encorajam valores e princípios semelhantes que surgiram na manufatura enxuta e foram posteriormente formalizados na indústria de software através do Manifesto Ágil (MANIFESTO, 2001).

Combinadas, elas fornecem uma estrutura onde o cliente pode participar ativamente durante o planejamento e o desenvolvimento para que seja produzido um produto que melhor atenda as suas necessidades, implantando funcionalidades incrementalmente, para tomar vantagem das oportunidades de negócios emergentes.

4 Desenvolvimento Enxuto de Software

O Desenvolvimento Enxuto de Software é uma abordagem inicialmente desenvolvida por Bob Charette (HIGHSMITH, 2002). Ela evoluiu da experiência de gerenciamento de riscos e dos princípios e valores da manufatura enxuta, abordados por Womack, Jones e Roos (1991). Charette aborda a agilidade como tolerância a mudanças desenvolvendo uma abordagem de três camadas para transformar às mudanças numa vantagem competitiva. O conceito chave é o de "risco empreendedor", definido como a habilidade de transformar os riscos em oportunidades.

Esta abordagem pode ser considerada muito mais uma estratégia de negócios e de gerenciamento de projetos do que um processo de desenvolvimento de software. Ela não é específica para uma prática de desenvolvimento de software, apenas descreve um conjunto de princípios, valores e ferramentas que devem ser utilizados para tornar o desenvolvimento enxuto, seguindo os conceitos desenvolvidos na manufatura pela empresa japonesa Toyota de automóveis.

O trabalho de transferência dos princípios e valores da manufatura para o desenvolvimento de software iniciado por Charette, foi pouco divulgado. Pela falta de publicações, o presente trabalho se baseia na publicação feita por Poppendieck, M. e Poppendieck, T. (2003).

4.1 Produção Enxuta

A história da produção enxuta iniciou-se no Japão com a família Toyoda e a empresa *Toyota Motor Company*, fundada em 1937. Após ter passado por períodos difíceis no final da década de 1930, época em que foi obrigada pelo governo a produzir caminhões militares, com métodos em grande parte artesanais, a Toyota resolveu ingressar na fabricação em larga escala de carros e caminhões comerciais, porém deparou com uma série de problemas.

- Mercado interno limitado, demandando grande variedade de produtos (carros de luxo, caminhões de grande porte, caminhões de pequeno porte e carros pequenos).

- Força de trabalho nativa do Japão já não estava mais propensa a ser tratada como custo variável ou peça intercambiável.
- Grande concorrência dos produtores estrangeiros ávidos pelo mercado japonês e dispostos a defenderem seus mercados contra exportações japonesas.

A produção em massa era a forma mais barata de produzir carros, mas significava produzir um grande número de carros iguais e o mercado japonês não era suficiente para consumir uma quantidade grande de veículos iguais. A Toyota precisava produzir carros em pequena quantidade, mas ao mesmo tempo manter o custo tão baixo quanto ao da produção em massa.

A partir deste dilema, na década de 1950, surgiu o Sistema Toyota de Produção (TPS – *Toyota Production System*) para formar a base de uma nova forma de pensar sobre manufatura, logística e desenvolvimento de produtos. Os grandes responsáveis pela criação e formalização desta nova abordagem foram Taiichi Ohno (engenheiro chefe da Toyota) e Shigeo Shingo (engenheiro consultor que contribuiu com a melhoria do TPS). Ela está centrada em um princípio fundamental da abordagem enxuta: eliminar perdas ao longo do processo de produção.

Tudo que não agregava valor para os clientes era definido como perda. Entre os exemplos de perdas, destacam-se: fazer alguma atividade que não é necessária no momento, movimentação, transporte, espera, processamento extra etc. Ohno e Shingo não copiaram os princípios e valores da produção em massa, seu ideal era produzir e entregar o produto imediatamente após o pedido ter sido feito pelo cliente. Eles acreditavam que era melhor aguardar o pedido ao invés de construir um estoque para atender aos pedidos futuros.

Na década de 1970, a indústria automobilística do ocidente encontrava-se em um quadro generalizado de desaceleração do crescimento econômico, quando comparada com o elevado desempenho que as empresas automobilísticas japonesas apresentavam e mantinham desde a metade dos anos de 1960.

De fato, como mostra Womack, Jones e Roos (1991), a indústria automobilística foi palco, mais uma vez, para profundas transformações na produção industrial no último quarto do século XX. Esses novos conceitos de produção referem-se a um conjunto de

inovações organizacionais que a Toyota vinha desenvolvendo desde a metade da década de 1950. Esta nova abordagem criada na Toyota foi mais tarde batizada de *lean production* (produção enxuta) por Krafcik (1988), em oposição a *buffered production* que, caracterizava a produção em massa.

A abordagem comumente utilizada até então nos processos fabris era aquela fornecida pela produção em massa, ou seja, a fabricação em grandes volumes de produtos padronizados para um mercado de amplas dimensões.

Devido à existência de grande diferença quantitativa entre as demandas dos mercados americano e japonês, a Toyota tinha a necessidade de produzir de forma competitiva, uma maior variedade de modelos em pequenas quantidades. Sob este aspecto, o problema estava em alcançar a eficiência e a redução de custos, não mais com base em economias de escala, mas em outros elementos da produção manufatureira.

De acordo com Ohno (1997), na equação simples de produtividade (quantidade de produto total dividido pela quantidade de trabalho aplicado), a forma tradicional de se conseguir o ganho de produtividade é pelo aumento da escala de produção. Entretanto, em períodos de lento crescimento econômico, a eficiência deve ser alcançada mediante a redução da quantidade de trabalho empregada na produção. Isto requer uma racionalização do processo de trabalho, diferente daquela da produção em massa.

Ohno (1997) menciona que, antes mesmo de seu ingresso na Toyota, ouvia-se no Japão comentários de que, em média, os trabalhadores americanos eram mais produtivos que os operários japoneses. Na sua avaliação, essa maior eficiência não era devido aos trabalhadores americanos possuírem mais força física que os japoneses.

A baixa produtividade da força de trabalho japonesa residia, segundo Ohno (1997), em formas inadequadas de trabalho que levavam ao desperdício. Se fosse possível eliminar de todos os tipos de perdas, então a produção de automóveis poderia ser economicamente viável no Japão.

Conforme Levy (1997), entre os principais objetivos da produção enxuta destacam-se: entregas no tempo certo (JIT – *Just in Time*), estoques reduzidos, defeito zero, produção flexível e cooperação tecnológica entre os fornecedores.

Na constituição do sistema JIT (produzir apenas quando necessário), contribuiu ainda o método de vendas de produtos aos consumidores praticado nos supermercados

americanos. Ohno (1997) relata que ao final da década de 1940 ele procurou adaptar a idéia do supermercado à produção de automóveis.

Nesse tipo de empresa comercial, os clientes se dirigem somente quando for preciso e para adquirir apenas os produtos e as quantidades específicos para atender suas necessidades de consumo momentâneas.

Cabe ao supermercado ir repondo as mercadorias à medida que são retiradas das prateleiras. Enquanto a retirada não ocorre, não há motivo para colocar produtos adicionais nas gôndolas. Esse procedimento inspirou e influenciou uma das ferramentas básicas do sistema JIT – o *kanban*, implantado na empresa em 1953.

4.1.1 Organização do Processo de Trabalho

A concepção do processo de trabalho sob o sistema de produção enxuta é radicalmente diferente daquele da produção em massa. Muda-se a perspectiva de visão sobre como o trabalho deve ser organizado.

No sistema de produção em massa, o processo de trabalho é concebido para empurrar a produção para fora da fábrica (*push*), cabendo ao departamento de vendas a responsabilidade de encontrar demanda para o produto que está saindo da linha de produção.

Sob o sistema de produção enxuta, a produção sai da empresa como se fosse puxada (*pull*). A fabricação é iniciada a partir de uma demanda pré-existente, solicitando materiais ao longo do processo produtivo em sentido inverso ao da produção em massa, ou seja, partindo dos pedidos em direção aos componentes e depósito de matérias primas.

Ohno (1997) descreve o cliente da indústria enxuta como parte integrante da sua equipe e o coloca no começo do ciclo de produção e não no fim deste ciclo, como nos modelos tradicionais de produção em massa. As necessidades dos clientes, suas sugestões, queixas e problemas são considerados com seriedade por todos e em toda a linha de produção. Este sistema evita produção excessiva e atende à demanda dos clientes no começo do processo.

A produção enxuta e a produção em massa representam conceitos totalmente opostos de "puxar" (*pull*) e "empurrar" (*push*). A filosofia enxuta é concebida com o objetivo de evitar perdas para conseguir aumentos de produtividade e reduções de custos.

Ohno (1997) identificou sete tipos de desperdícios: superprodução, espera, transporte, excesso de processamento, estoque, movimentação de trabalhadores e produção de produtos defeituosos.

Segundo Ohno (1997), para alcançar a eficiência, o processo se organiza sob dois pilares básicos: a "autonomação" e o JIT. O vocábulo "autonomação" é uma combinação de duas palavras: autonomia e automação. Ele procura representar a idéia do dispositivo inventado por Sakichi Toyoda (fundador da Toyota) que permitia a parada automática (*poka-yoke*).

As vantagens desse mecanismo é que um mesmo operador pode monitorar várias máquinas simultaneamente, além da redução do desperdício de matérias-primas e peças defeituosas que se consegue detectar com antecedência, durante a produção e não apenas através de inspeções ao término do processo.

O JIT corresponde à concepção do processo de produção onde o material a ser trabalhado deve chegar à linha de montagem, apenas no momento em que está sendo demandado e somente na quantidade necessária.

Assim, se um posto de trabalho produzir componentes em quantidade maior ao que à seção seguinte consiga processar, estoques intermediários de trabalho não finalizados irão se acumular ao longo da linha de montagem, desperdiçando matéria-prima, trabalho, espaços e capital investido. A efetividade do sistema JIT é alcançada mediante a utilização de algumas inovações técnicas e de procedimentos de produção, destacando-se o *kanban*.

O *kanban* é um cartão que circula no chão de fábrica, verticalmente e horizontalmente, em caixas ou carrinhos contendo instruções do trabalho a ser realizado, e permite passar informações entre os postos de trabalho. O cartão indica a quantidade e os tipos de materiais que cada posto de trabalho está solicitando em sentido inverso que será utilizado.

A concepção do *kanban* é de que o posto de trabalho corrente dirija-se ao posto precedente, e assim sucessivamente, e demande ou retire apenas a quantidade de materiais ou componentes que necessite (o principio do supermercado mencionado anteriormente). Segundo Ohno (1997), a implantação do *kanban* na Toyota não foi uma tarefa simples, pois ele contrariava a sabedoria convencional, que visava maximizar a

produtividade aumentando a quantidade de bens produzidos. A implantação do *kanban* na Toyota estendeu-se por cerca de uma década e sua introdução permitiu atuar sobre uma das principais fontes de desperdício: o estoque.

O objetivo de só produzir aquilo que está sendo demandado (*pull*) evita tanto o acúmulo de peças trabalhadas e não finalizadas entre os postos de trabalho, quanto à formação de estoques de produtos finais indesejados.

Outras iniciativas adotadas na Toyota contribuíram para a redução de estoques, uma delas refere-se à delegação de responsabilidade aos trabalhadores pelo controle de qualidade da produção, onde foi solicitado que produzissem corretamente desde o início, tendo autonomia para devolver a peça defeituosa ou interromper o processo de produção.

A necessidade de a Toyota ter que produzir baixos volumes de modelos diferenciados, levou à criação de novas práticas manufatureiras, como por exemplo, o nivelamento da produção. Isso contornava a necessidade de manter a configuração das máquinas por longos períodos, para ter sua eficácia maximizada, como na produção em massa.

Nivelando a produção evitam-se as flutuações abruptas e consegue-se produzir lotes pequenos de modelos diferentes. Ohno (1997) ilustrou esse feito relatando que as trocas de matrizes de prensas na Toyota, que levavam cerca de três horas na década de 1940, tiveram o tempo reduzido para menos de três minutos ao final da década de 1960.

A operacionalização desse sistema de produção requereu outras práticas manufatureiras, tais como o *andon*. A não existência de estoques intermediários exigiu que se estabelecessem procedimentos de correção rápida de problemas surgidos na linha de produção. Assim, delegou-se aos operários a responsabilidade de parada da linha através do uso de controles visuais, o *andon*, colocados em lugares visíveis ao longo da linha de montagem: a luz verde indicava que a produção fluía normalmente, o acionamento de luz amarela revelava a necessidade de auxílio, e a vermelha sinalizava a parada total da produção.

O *kanban* e o *andon*, que permitem uma "administração visual", requerem um trabalhador com um mínimo de escolaridade, capaz de ler e entender instruções, transmitir informações e participar do processo produtivo. Por fim, a rotação nos postos

de trabalho (a chamada flexibilidade interna ou funcional) e o trabalho em equipe são outros traços característicos desse arranjo organizacional.

Na produção enxuta, a racionalização do trabalho é caracterizada pela não especialização dos trabalhadores, transformando-os em operários polivalentes e multifuncionais. Assim, a mão-de-obra não está restrita a um único posto de trabalho, podendo circular entre as diferentes tarefas estabelecidas ao grupo de trabalho a que pertencem.

4.1.2 Automação e "Autonomação"

Como citado anteriormente, a "autonomação" é um dos pilares para alcançar a eficiência. Para atingir este ponto de maturidade, as fábricas precisaram passar por dois estágios de amadurecimento:

- Automação que significava a mecanização do processamento, assim como a fixação e remoção da peça sendo trabalhada. Mesmo assim, máquinas automatizadas não estavam dotadas das funções do cérebro humano. Isso significava que os trabalhadores tinham de estar próximos, até mesmo daquelas máquinas que haviam sido automatizadas.
- "Autonomação" é atingida quando as máquinas foram providas de uma função de cérebro humano, ou seja, a capacidade de detectar anomalias de forma autônoma e não necessitavam mais da supervisão humana.

A transferência do trabalho dos homens às máquinas abrange duas questões: como transferir o trabalho físico às máquinas e como transferir o trabalho intelectual (inteligente) às máquinas. O sucesso inicial ao efetuar essa importante mudança é uma característica significativa do Sistema Toyota de Produção.

Esta distinção será importante posteriormente, para definir as fronteiras de contribuição deste trabalho.

4.2 Princípios do Desenvolvimento Enxuto de Software

No trabalho publicado por Poppendieck, M. e Poppendieck, T. (2003), é feito o mapeamento dos sete princípios da manufatura enxuta para o desenvolvimento de software. Neste trabalho, os princípios são definidos como idéias que guiam uma estrutura de trabalho e discernimentos sobre uma disciplina, enquanto que as práticas são definidas como ações necessárias para executar os princípios.

Os princípios são universais, enquanto que as práticas fornecem guias específicos de um domínio do que deve ser feito, e desta forma, precisam ser adaptadas para cada contexto de utilização. Neste ponto de vista, Poppendieck, M. e Poppendieck, T. (2003) sugerem que os problemas que surgem quando se tenta aplicar técnicas e conhecimentos de outras disciplinas (como por exemplo, a manufatura) para o desenvolvimento de software, geralmente são resultantes da tentativa de transferir as práticas, ao invés de transferir os princípios.

A seguir, são abordados os sete princípios enxutos contextualizados para a indústria de software e que auxiliam a tradução das práticas utilizadas pelas metodologias que compõem o movimento ágil de desenvolvimento de software. Os sete princípios são:

- Eliminar Perdas;
- Amplificar o Aprendizado;
- Tomar Decisões o Mais Tarde Possível;
- Fazer Entregas o Mais Rápido Possível;
- Tornar a Equipe Responsável;
- Construir Integridade;
- Visualizar o Todo.

4.2.1 Eliminar Perdas

Perda é algo que não adiciona valor ao produto (valor percebido pelo cliente). No pensamento enxuto, o conceito da perda é um grande obstáculo. Quando uma planta de fábrica produz mais do que é necessário imediatamente, isto corresponde ao desperdício.

Quando um desenvolvedor implementa mais funcionalidades do que são necessárias, isto também corresponde a perdas.

O ideal é descobrir o que o cliente deseja e entregar somente o necessário, no menor intervalo de tempo possível. Qualquer obstáculo à rápida satisfação das necessidades do cliente é considerado uma perda.

4.2.2 Amplificar o Aprendizado

O desenvolvimento é um exercício de aprendizado, enquanto que a manufatura é um exercício de redução de variação. Por esta razão, as práticas da abordagem enxuta para desenvolvimento de software são diferentes das práticas utilizadas na manufatura. A melhor abordagem para melhorar o ambiente de desenvolvimento é através da amplificação do aprendizado.

4.2.3 Tomar Decisões o Mais Tarde Possível

Práticas de desenvolvimento que suportam tomadas de decisões tardias são eficientes em domínios que envolvem incertezas, porque elas dão suporte a uma abordagem baseada em opções. Num cenário de incertezas, muitos mercados econômicos desenvolvem opções como uma forma dos investidores evitarem tomadas de decisões errôneas e precipitadas, até o futuro estar próximo o suficiente e mais facilmente previsível.

É de grande valor retardar as tomadas de decisões, uma vez que as melhores decisões são tomadas baseadas em fatos, não em especulações. Uma estratégia chave para retardar as tomadas de decisões ao desenvolver sistemas complexos é projetá-los e construí-los de forma a facilitar mudanças futuras.

4.2.4 Fazer Entregas o Mais Rápido Possível

Até recentemente, a rápida entrega de software não era valorizada, a estratégia de não cometer erros era vista como mais importante. Porém, o desenvolvimento rápido do software tem diversas vantagens: sem ele não é possível obter uma realimentação de informações confiável e o ciclo de descoberta é crítico para o processo de aprendizado

(projetar, implementar, realimentar, melhorar – quanto menor este ciclo, maior o aprendizado).

A velocidade de desenvolvimento também ajuda atender às necessidades atuais do cliente, além de permitir adiar a tomada de decisões para quando for acumulado conhecimento suficiente para tal. Uma das lições aprendidas da linha de manufatura enxuta é a de comprimir o fluxo gerador de valor ao máximo, para eliminar o desperdício no processo.

4.2.5 Tornar a Equipe Responsável

Envolver os desenvolvedores nas decisões de detalhes técnicos é fundamental para atingir a excelência. Quando dotados com a experiência necessária e guiados por um líder, eles tomarão decisões técnicas e de processos, melhores que qualquer outra pessoa poderia tomar por eles.

Pelo fato das decisões serem tomadas tardiamente e a execução ser conduzida de forma rápida, não é possível gerenciar as atividades dos trabalhadores através de uma autoridade central. As práticas enxutas utilizam as técnicas de produção puxada (*pull*) para agendar o trabalho e possuem mecanismos de sinalizações locais, de forma a permitir que outros trabalhadores identifiquem o trabalho que necessita ser realizado.

No desenvolvimento de software enxuto, o mecanismo da produção puxada (*pull*) corresponde ao acordo de entregar versões refinadas e incrementais do software em intervalos regulares. A sinalização local é feita através de gráficos visuais, reuniões diárias, integrações freqüentes e testes automatizados.

A Figura 9 ilustra um quadro que pode ser utilizado para distribuir as funcionalidades a serem realizadas em cada iteração, semelhante ao *kanban* do Sistema de Produção da Toyota. Ele também é utilizado para nivelar e controlar o fluxo de produção, definindo a quantidade de trabalho a ser realizado em cada iteração.

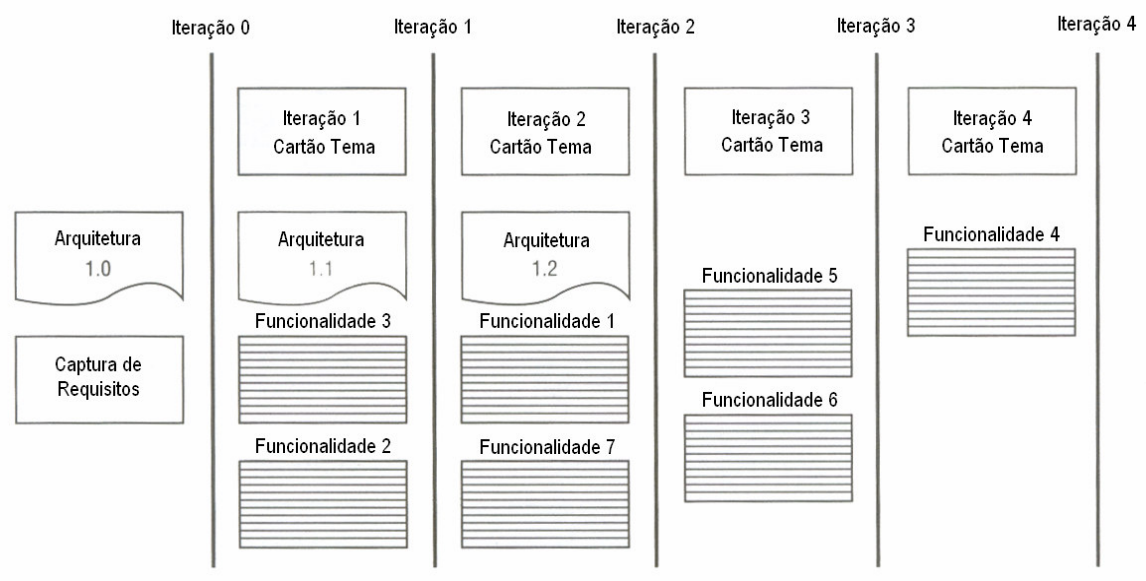


Figura 9: Quadro de cartões de funcionalidades
(POPPENDIECK, M.; POPPENDIECK, T., 2003).

As colunas representam a segmentação do trabalho a ser realizado através das iterações. Em cada coluna é acrescentado um cartão que corresponde ao objetivo da iteração (Cartão Tema), e abaixo deles são colocados cartões correspondentes aos requisitos a serem implementados. Estes cartões podem ser relacionados à arquitetura ou às funcionalidades do produto.

O nivelamento da produção é feito através da quantidade de trabalho a ser realizado para transformar os requisitos descritos nos cartões em incrementos do produto; o esforço alocado para cada iteração deve ser homogêneo, não podendo haver grandes variações entre elas.

A Figura 10 ilustra um exemplo dos controles visuais utilizados para acompanhar a execução do projeto. A curva azul representa a estimativa de esforço necessário para finalizar as tarefas remanescentes na iteração atual e a linha amarela corresponde à quantidade de funcionalidades remanescentes para serem codificadas e incorporadas ao incremento do produto a ser entregue no fim da iteração.

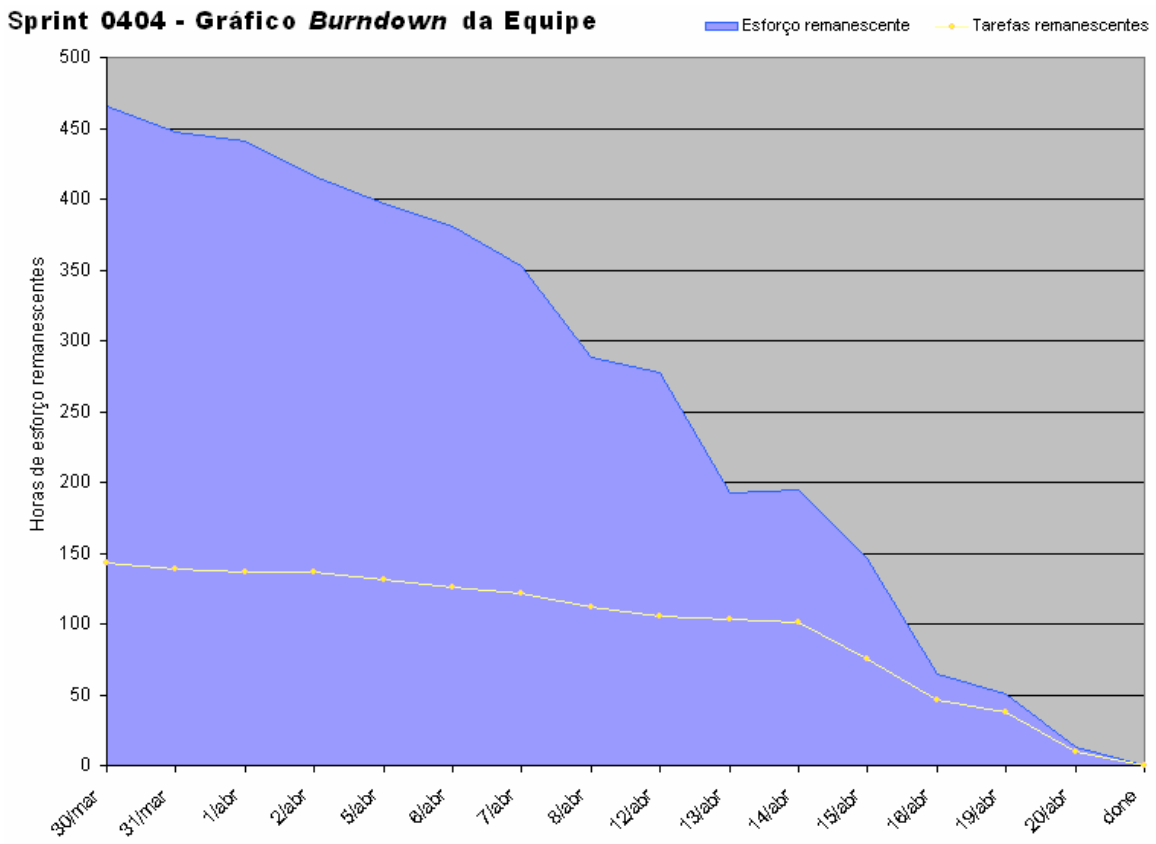


Figura 10: Exemplo de gráfico do tipo *burndown* (HARTMANN, 2004).

A Figura 11 apresenta um outro exemplo de controles visuais elaborado por DeLuca, Coad e Lefebvre (1999), utilizados para planejar e acompanhar o andamento do projeto. Esta figura ilustra o exemplo do desenvolvimento de uma ferramenta de CRM (*Customer Relationship Management*), que inclui duas subáreas de negócio (Gerenciamento de Vendas e Marketing) e sete atividades, representadas como caixas dentro de cada uma das subáreas de negócios.

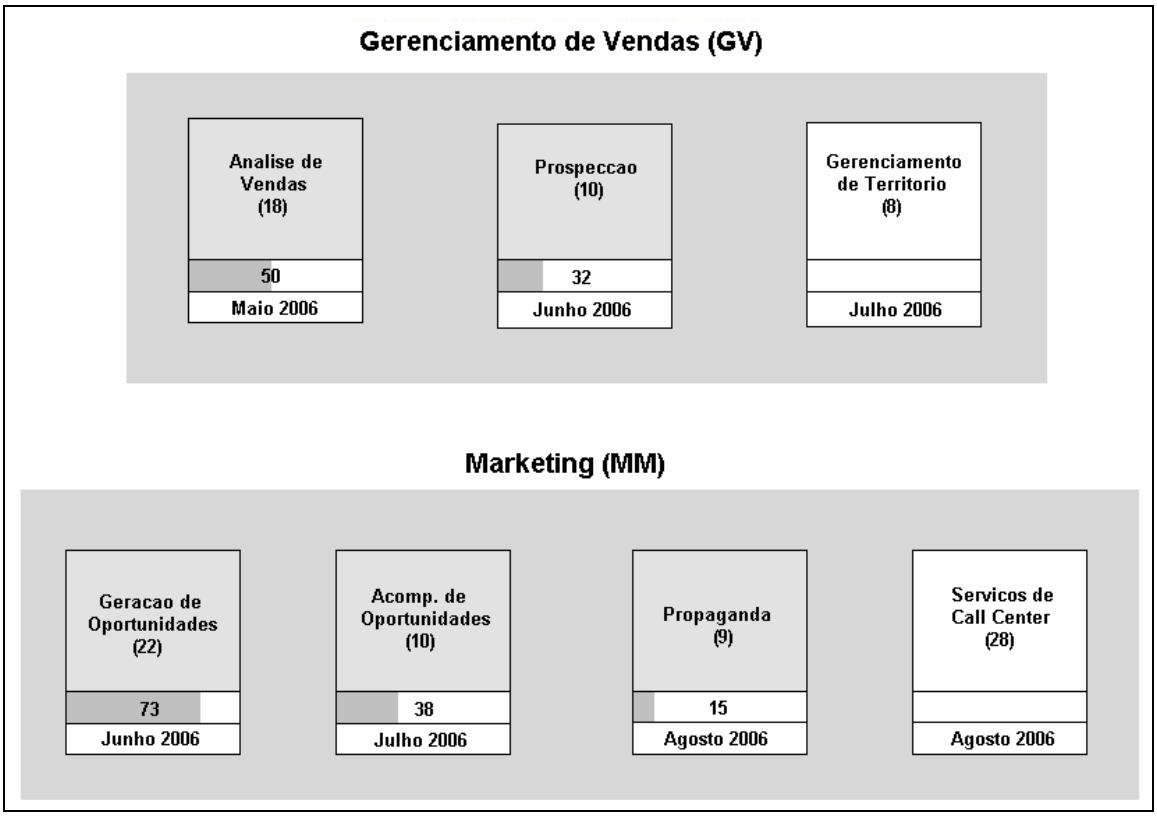


Figura 11: Controle visual de planejamento do tipo Parquímetro (*Parking lot*) (ANDERSON, 2004).

Os números entre parênteses, abaixo dos nomes das atividades, representam o número de funcionalidades identificadas em cada atividade. As datas, na parte inferior das caixas referentes às atividades de negócio, indicam as estimativas de término e entrega delas.

Em cada uma das caixas das atividades, existe uma faixa que indica o percentual do progresso do desenvolvimento de cada uma das atividades. Por exemplo, a atividade "Acompanhamento de Oportunidades" tem 38% das funcionalidades identificadas concluídas.

4.2.6 Construir Integridade

As organizações enxutas sempre buscam produzir produtos de alta qualidade; elas não conseguem consistentemente tomar decisões o mais tarde possível e realizar entregas rápidas, fazendo produtos de baixa qualidade. Uma vez que estas organizações focam na entrega de valor aos clientes finais, elas precisam desenvolver um conhecimento

profundo do que representa o valor para eles e constantemente ajustar os procedimentos internos, de forma a empregar todos os recursos disponíveis (tempo, pessoas, equipamentos etc.) na criação de valor para os clientes.

A parcela de mercado é um esboço da medida da integridade percebida dos produtos, uma vez que ela mede a percepção do cliente ao longo do tempo. A integridade conceitual significa que os conceitos centrais do sistema, quando reunidos, funcionam de forma regular, coesa e é um fator critico na criação da integridade (BROOKS, 1995). O software necessita de um nível adicional de integridade e precisa manter sua utilidade ao longo do tempo. O software que possui integridade possui uma arquitetura coerente, facilidade satisfatória de uso, atende aos propósitos para o qual foi proposto, manutenível, adaptável e extensível.

Pesquisas indicam que a integridade é resultado de uma sábia liderança, experiências relevantes, comunicação efetiva e disciplinas saudáveis. Processos, procedimentos e métricas não são substitutos adequados para eles (POPPENDIECK, M.; POPPENDIECK, T., 2003).

4.2.7 Visualizar o Todo

Para obter a integridade nos sistemas de grande complexidade é preciso um conhecimento profundo de diversas áreas. Um dos problemas não tratados no desenvolvimento de produtos é que especialistas de cada área têm a tendência de maximizar o desempenho na parte do produto onde são especialistas, ao invés de focar no desempenho do produto como um todo.

Quando indivíduos e organizações são medidos de acordo com a contribuição especializada e pontual ao invés do desempenho global, o resultado provável é que ocorram otimizações pontuais, podendo acarretar em uma degradação no desempenho do processo quando analisado como um todo (GOLDRATT, 1997).

4.3 Desperdícios no Desenvolvimento de Software

Shingo (1981), um dos engenheiros do Sistema de Produção da Toyota (TPS), identificou sete tipos de desperdícios na manufatura. Tomando como base os desperdícios

identificados na Toyota, Poppendieck, M. e Poppendieck, T. (2003) propuseram o mapeamento ilustrado na Tabela 3 para o desenvolvimento de software.

Tabela 3: Mapeamento do desperdício da manufatura no desenvolvimento de software (POPPENDIECK, M.; POPPPENDIECK, T., 2003).

Manufatura	**Desenvolvimento de software**
Estoque	Trabalho parcialmente finalizado
Processamento extra	Processo extra
Excesso produção	Funcionalidades extras
Transporte	Chaveamento de tarefas
Espera	Espera
Movimento	Movimento
Defeitos	Defeitos

4.3.1 Trabalho Parcialmente Finalizado

Trabalhos parcialmente realizados têm a tendência de se tornarem obsoletos e podem obstruir o desenvolvimento de outras funcionalidades que precisam ser feitas. O grande problema com os trabalhos parcialmente realizados é que eles não permitem verificar se eles irão eventualmente funcionar.

Para verificar o correto funcionamento é preciso integrá-lo ao restante do ambiente e até mesmo colocá-lo em produção, uma vez que não é possível verificar se ele atende aos problemas de negócios, sem iniciar sua utilização.

O desenvolvimento parcialmente realizado consome recursos de investimentos que ainda precisam produzir resultados. No desenvolvimento de software, estes investimentos, algumas vezes, são capitalizados e a depreciação inicia quando o software entra em produção.

Quando o sistema não entra em produção é preciso descartar o investimento, e por este motivo os trabalhos parcialmente realizados podem carregar grandes riscos

financeiros. Minimizar a quantidade de desenvolvimentos parcialmente realizados é uma estratégia de redução de riscos, bem como de redução de fontes de desperdícios.

4.3.2 Processo Extra

Muitos processos de desenvolvimento de software requerem diversos documentos para os clientes assinarem, para prover rastreabilidade ou obter aprovação de mudança de escopo.

O fato de a documentação ser considerada um artefato necessário e um produto a ser entregue, não significa que ela adicione valor. Se for necessário produzir documentos que agreguem pouco valor para os clientes, é recomendável que eles sejam breves, de alto nível, e de preferência, sem utilizar os desenvolvedores para elaborá-los.

Os sistemas de segurança crítica frequentemente são regulamentados e geralmente necessitam de requisitos escritos que são rastreáveis até o código. Neste caso, organizar os requisitos, de forma que eles possam ser facilmente avaliados e verificados quanto à completeza, pode ser qualificado como uma atividade que gera valor.

Uma forma de avaliar o valor agregado pela documentação é verificar se existem pessoas aguardando sua conclusão para iniciar o trabalho. Se um analista preencheu um modelo, fez uma tabela ou escreveu requisitos que outros membros da equipe estão aguardando para codificar, testar ou escrever manuais de treinamento, provavelmente estes documentos agregam valor.

Mesmo assim, deve haver uma busca constante para a maneira mais eficiente e efetiva de transmitir a informação. Por exemplo, devem-se escrever os testes de aceitação no lugar dos requisitos.

4.3.3 Funcionalidades Extras

Os desenvolvedores podem querer adicionar novas funcionalidades ou capacidades técnicas apenas para verificar como elas funcionam. Apesar de esta prática parecer inofensiva, ela é um desperdício, pois precisará ser testada e acrescentará códigos que outros desenvolvedores precisarão compreender e manter.

Cada linha de código no sistema precisa ser rastreada, compilada e testada toda vez que é feita alguma alteração, além de precisar ser mantida durante toda a sua vida útil.

Outra interferência de funcionalidades extras no sistema é que elas aumentam a complexidade e acrescentam um possível ponto de falha. Há uma grande possibilidade de que o código extra se torne obsoleto, antes mesmo dele ser utilizado.

4.3.4 Chaveamento de Tarefas

Atribuir pessoas a múltiplos projetos é uma fonte de desperdícios. Toda vez que um desenvolvedor precise alternar entre tarefas, é preciso um tempo de preparação (similar ao tempo de configuração da Toyota), onde é necessário concentração na nova atividade, consumindo tempo (DEMARCO; LISTER, 1987). Pertencer a múltiplos projetos geralmente causa um número maior de interrupções e assim um número maior de chaveamento entre tarefas.

A forma mais rápida de completar dois projetos que utilizam o mesmo recurso é realizá-los sequencialmente. Se dois projetos estão estimados para durar duas semanas, conduzindo-os em paralelo e compartilhando um recurso, os projetos provavelmente durarão mais de cinco semanas (GOLDRATT, 1997). O trabalho será realizado de forma muito mais rápida através do conceito de *pipelines*, que não é utilizado em sua capacidade máxima, utilizando a teoria de filas para gerenciar a capacidade.

4.3.5 Espera

Uma das maiores fontes de desperdício no desenvolvimento de software geralmente são as esperas de ocorrência de eventos. Atrasos no inicio do projeto, no recrutamento da equipe de projeto, atrasos acarretados pela documentação excessiva dos requisitos, nas revisões e aprovações, nos testes e na instalação do sistema são considerados fontes de perdas. Estes atrasos são comuns nos processos de desenvolvimento de software e não parece intuitivo considerá-los como fontes de perdas.

Os atrasos impedem os clientes de perceber o valor o mais rápido possível. Quando uma necessidade crítica do cliente é entregue à empresa de desenvolvimento, a

velocidade com que ela consegue responder está diretamente relacionada com os atrasos sistêmicos do seu ciclo de desenvolvimento.

Para alguns ambientes, os atrasos podem não parecer tão problemáticos quando comparados aos outros problemas. Entretanto, quando o desenvolvimento é realizado num domínio em expansão, os atrasos representam um problema sério e determinam qual empresa será bem sucedida em um mercado altamente competitivo.

A Figura 12 exemplifica a cadeia de valor e os atrasos no modelo de desenvolvimento cascata. A linha do tempo está dividida em duas regiões: a de trabalho (que agrega valor) e a de espera (que não agrega valor). Na parte superior da figura, estão apresentadas as etapas do modelo cascata (Requisitos, Análise, Projeto, Codificação, Testes e Operação) e outras atividades de interação com o cliente (Negociação, Aprovação, Início e Revisão).

Nesta figura é possível observar que existe um período grande de espera quando comparado com o período de trabalho. Um dos motivos para a existência destes períodos de esperas são as transições entre cada etapa, que exige uma verificação dos resultados obtidos e um encerramento formal.

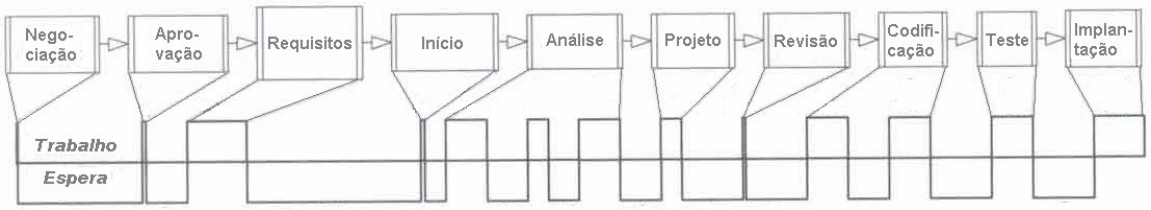


Figura 12: Mapa da cadeia de valor do modelo cascata
(POPPENDIECK, M.; POPPENDIECK, T., 2003).

A Figura 13 exemplifica a cadeia de valor e os atrasos presentes no desenvolvimento enxuto. Da mesma forma que a Figura 12, a linha do tempo está dividida em duas regiões: a de trabalho e a de espera. Nesta figura é possível observar que os tempos de espera são menores e menos freqüentes, e os períodos de geração de valor (trabalho) são maiores.

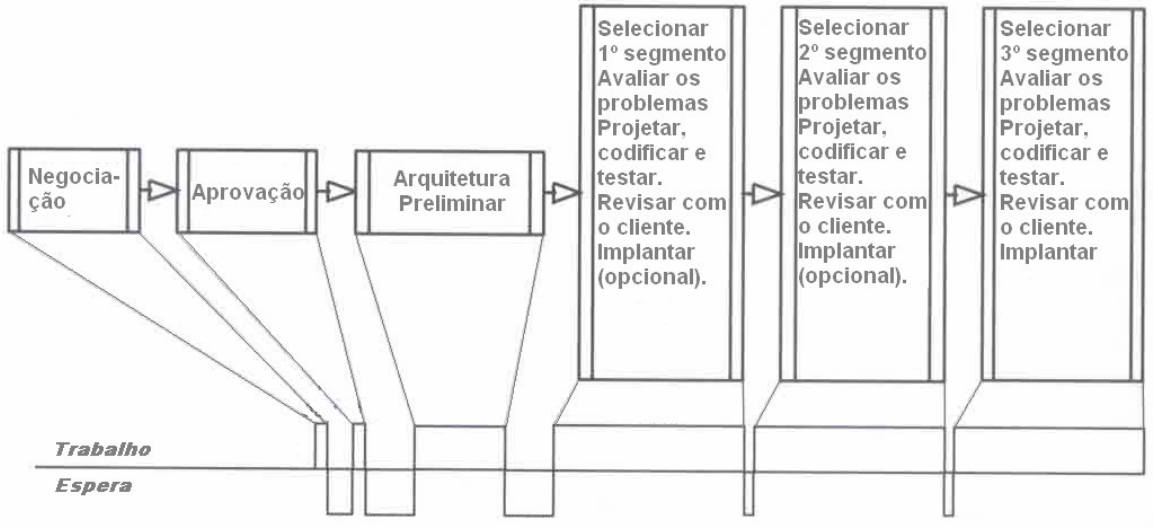


Figura 13: Mapa da cadeia de valor do desenvolvimento enxuto de software (POPPENDIECK, M.; POPPENDIECK, T., 2003).

Diferentemente do modelo cascata, o desenvolvimento enxuto não caracteriza formalmente cada uma das etapas de análise, projeto, codificação e testes. Estas etapas ocorrem dentro de cada iteração e geralmente são desempenhadas pelo mesmo grupo de pessoas, não exigindo um encerramento formal.

4.3.6 Movimento

O desenvolvimento de software é uma atividade que requer grande concentração, por isso, as movimentações dos desenvolvedores para obter informações necessárias podem tomar muito mais tempo do que parecem.

O desenvolvedor pode levar mais tempo para retomar a concentração na atividade inicial do que para resolver uma dúvida. Por esse motivo, as práticas de desenvolvimento ágil geralmente recomendam que a equipe trabalhe num ambiente único, que facilite a comunicação entre desenvolvedores, testadores e clientes.

As pessoas não são os únicos recursos que se movimentam, muitos artefatos também se movimentam. Os requisitos podem se movimentar dos analistas para os projetistas, os documentos de projetos depois são enviados para os programadores, o código depois é entregue aos testadores e assim por diante.

Cada movimentação de artefatos está repleta de oportunidades de desperdícios, o maior desperdício nestas movimentações é que os artefatos não possuem todas as

informações que a próxima pessoa que o utilizará precisa para conduzir seu trabalho. Grande parcela do conhecimento tácito é mantida pelo criador do artefato e não é entregue ao receptor.

4.3.7 Defeitos

A quantidade de desperdício causada por um defeito é o produto do seu impacto pelo tempo que decorre sem ele ser descoberto. Um defeito crítico que é detectado durante o desenvolvimento não é uma grande fonte de desperdício. Um pequeno defeito que é detectado meses depois que o sistema já está em operação é uma fonte de desperdício muito maior. A forma para reduzir o impacto dos defeitos é identificá-los precocemente, realizando testes imediatos, automatizados, integrando frequentemente e colocando versões em produção o mais cedo possível.

4.4 Aspectos Relevantes na Elaboração do Modelo Proposto

O pensamento enxuto é utilizado para fortalecer a base teórica do modelo proposto, fornecendo valores e princípios que influenciam a forma de organização do trabalho.

O pensamento enxuto aperfeiçoa o desenvolvimento de software, eliminando as perdas identificadas (abordados na seção **4.3 Desperdícios no Desenvolvimento de Software**), colocando as pessoas envolvidas no projeto como o recurso mais flexível no processo, por exemplo, disciplinando os envolvidos para fazerem as tomadas de decisões o mais tarde possível (conforme o princípio descrito na seção **4.2.3**).

Para clarificar e explicitar as inter-relações entre o pensamento enxuto e o modelo proposto, foi elaborada a Tabela 4. Nela são mapeados os princípios descritos anteriormente na seção **4.2 Princípios do Desenvolvimento Enxuto**, com as práticas compartilhadas entre as duas metodologias selecionas e que compõem o modelo proposto (apresentadas na seção **3.3**).

Tabela 4: Mapeamento dos princípios do desenvolvimento enxuto de software e práticas das metodologias selecionadas.

Princípios do desenvolvimento de software enxuto	**Práticas e ferramentas compartilhadas pelas metodologias selecionadas (*Scrum* e XP)**
Eliminar perdas	– Auto-organização – Colaboração – Emergência – Incrementos – Iteração
Amplificar o aprendizado	– Colaboração – Incrementos – Iteração
Tomar decisões o mais tarde possível	– Emergência
Fazer entregas o mais rápido possível	– Incrementos – Iteração
Tornar a equipe responsável	– Auto-organização – Colaboração
Construir integridade	– Colaboração – Incrementos
Visualizar o todo	– Iteração – Incrementos

5 Modelo de Gerenciamento Proposto

Neste capitulo é apresentado o modelo de gerenciamento de projetos proposto, combinando as metodologias XP e *Scrum*, apresentadas e discutidas anteriormente no capítulo **3 Metodologias Ágeis**, juntamente com os princípios e valores do Desenvolvimento Enxuto de Software (apresentado no capítulo **4**), utilizados para fortalecer a base teórica desta proposta.

A Figura 14 apresenta um mapeamento da utilização do *Scrum* para realizar o gerenciamento de projetos (região amarela), o XP para realizar o desenvolvimento de software (região azul), e o pensamento enxuto fornecendo princípios e valores que influenciam ambas as metodologias (região verde).

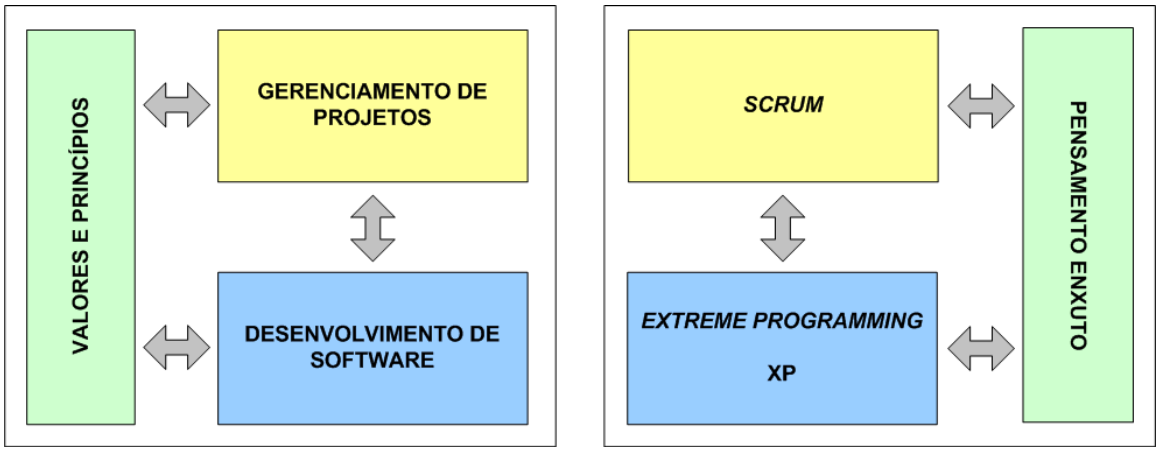


Figura 14: Mapeamento das metodologias selecionadas, suas interações e as áreas de aplicação.

A interação entre as duas metodologias (*Scrum* e XP) ocorre nos trabalhos realizados diariamente dentro dos *sprints*. Através da reunião de planejamento do *sprint* (*Scrum*), são definidos os requisitos a serem trabalhados na próxima iteração (*backlog* do *sprint*). Iniciado o *sprint*, cada item do *backlog* é trabalhado diariamente pelas práticas e ferramentas selecionadas do XP, para transformar os requisitos selecionados em incrementos do produto.

A Figura 15, ilustra a interação descrita, onde as práticas e ferramentas do XP são aplicadas nos trabalhos diários (circulo azul) e as definidas pelo *Scrum*, são utilizadas para realizar o gerenciamento do desenvolvimento de software (região em laranja).

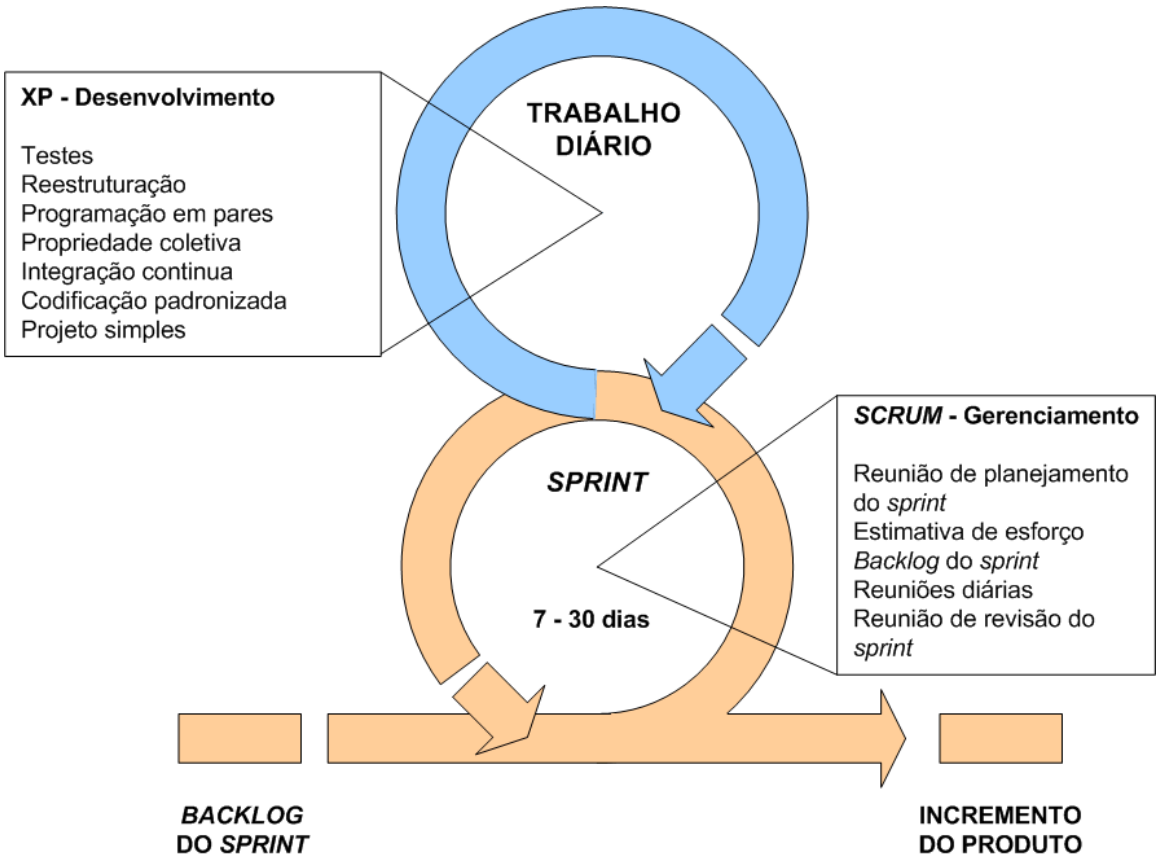


Figura 15: Interação entre o XP e o *Scrum* no modelo proposto.

5.1 Papéis e Responsabilidades

No modelo proposto, os papéis utilizados são herdados da metodologia *Scrum* (descritas na seção **3.1.1**), com alterações de nomenclatura para generalizar a proposta. Esta opção foi feita para preservar a nomenclatura da metodologia que compôs a maior parte do modelo proposto.

A partir da definição inicial do *Scrum*, foram feitas adaptações para incorporar as responsabilidades de desenvolvimento de software definidas pelo XP (descritas na seção **3.2.1**).

- **Treinador**: este papel herdou a nomenclatura do XP e as responsabilidades do *Scrum*. Ele é responsável por garantir que o projeto esteja sendo conduzido de acordo com as práticas, ferramentas, valores e regras do modelo proposto e que o progresso do projeto esteja de acordo com o desejado pelos Clientes. O

Treinador interage tanto com a Equipe, como com os Clientes e o Gerente. Ele também é responsável por remover e alterar qualquer obstáculo identificado, garantindo que a Equipe trabalhe da forma mais produtiva possível.

- **Líder de Projeto**: este papel herdou as responsabilidades do papel Responsável pelo Produto, definido pelo *Scrum*. No modelo proposto, sofreu apenas a alteração de nomenclatura, para generalizar seu significado para projetos sob demanda e desenvolvimento de produtos.
- **Equipe**: no modelo proposto, este papel assume as responsabilidades do *Scrum*, ou seja, ele possui autoridade para decidir sobre as ações necessárias e de se organizar para poder atingir os objetivos de cada *sprint*. Ela é envolvida na estimativa de esforço, na criação e na revisão dos itens *backlog* e por sugerir os obstáculos que devem ser removidos. Este papel também herdou as responsabilidades dos papéis Testador e Programador do XP, onde auxiliam o Cliente a escrever os testes funcionais, pela manutenção dos mesmos e por executarem periodicamente os testes. A Equipe também é responsável por projetar, codificar e manter o programa o mais simples e conciso possível.
- **Cliente**: papel e responsabilidades herdadas do *Scrum*. No modelo proposto não sofreu alteração.
- **Gerente**: papel e responsabilidades herdadas do *Scrum*. No modelo proposto não sofreu alteração.

5.2 Processo

A Figura 16 ilustra o modelo proposto, apresentando as práticas e ferramentas selecionadas do *Scrum* e XP, identificando suas entradas e saídas, juntamente com suas inter-relações e as fases do processo de desenvolvimento que são aplicadas (Conceber, Explorar e Adaptar e Fechar).

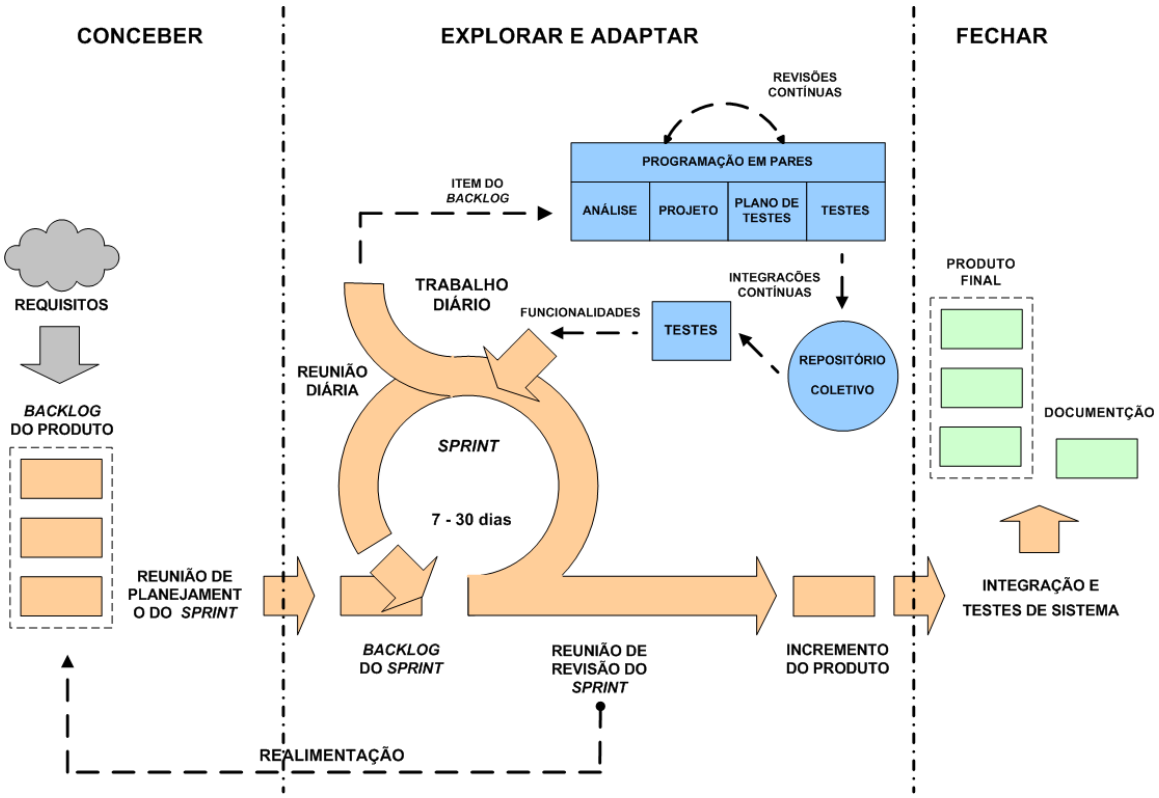


Figura 16: Representação gráfica do modelo proposto.

O modelo proposto toma como base o processo de gerenciamento de projetos especificado pelo *Scrum*, representado pela região em laranja na Figura 16, alterando a nomenclatura das fases para remover possíveis ambigüidades da tradução do inglês para o português.

A nomenclatura utilizada foi inspirada no trabalho publicado por Highsmith (2004). A Tabela 5 apresenta na coluna da esquerda o nome original definido no *Scrum*, enquanto que a coluna à direita, apresenta o nome da fase correspondente no modelo proposto.

Tabela 5: Mapeamento da nomenclatura das fases.

Scrum	**Modelo proposto**
Pré-*Game*	Conceber
Desenvolvimento	Explorar e Adaptar
Pós-*Game*	Fechar

O período sugerido para cada iteração (*sprint*) é de 7 a 30 dias, mantendo o período sugerido pelo *Scrum* (descrito na seção **3.1.3 Processo**). Esta duração deve ser definida, analisando a necessidade de cada cliente e projeto.

Em cada iteração é percorrido o ciclo definido pelas fases Conceber, detalhando os requisitos do próximo *sprint*, e Explorar e Adaptar, onde os itens do *backlog* do *sprint* são transformados em incrementos do produto e o aprendizado adquirido é realimentado ao processo após a realização das reuniões de revisão do *sprint*.

A seguir, são descritas cada uma das fases que compõem o modelo proposto, fornecendo uma visão sucinta do trabalho realizado em cada uma delas, juntamente com os artefatos produzidos e apontando quais princípios do desenvolvimento enxuto de software são aplicáveis.

5.2.1 Conceber

Esta fase vai além da iniciação do projeto. A iniciação tem a conotação de elaboração de orçamentos e cronogramas detalhados. Estes artefatos, apesar de serem necessários, precisam fluir de uma visão bem articulada, discutida e acordada por todos os envolvidos. A definição da visão pode ser expressa através de respostas a quatro perguntas:

1. Qual é a visão do produto?
2. Qual é o propósito do projeto e suas restrições?
3. Quem será incluído na comunidade de projeto?
4. Como a equipe se organizará para entregar o produto?

Para alguns projetos, esta fase pode ser realizada em poucos dias, para outros, particularmente aqueles que não passaram por algum tipo de estudo de viabilidade, pode levar mais tempo. Este aumento do tempo não necessariamente representa um aumento do esforço, este tempo pode ser necessário para difundir a visão do produto entre a comunidade do projeto.

Nesta fase também é elaborada a lista do *backlog* do produto, transformando a visão criada em requisitos. Depois de elaborada esta lista, é criado um projeto de alto

nível, contemplando uma arquitetura preliminar do software, e uma reunião é conduzida para elaborar planos preliminares dos conteúdos dos incrementos do produto de cada *sprint*.

A concepção do produto não é feita apenas uma vez no início do projeto. Antes do inicio de cada *sprint*, quando é reiniciado o ciclo Conceber, Explorar e Adaptar, a Equipe se reúne com os Clientes para planejar a próxima iteração. Nesta reunião a visão e os requisitos do produto são revisados e uma visão particular para o próximo *sprint* é criada (que corresponde a um subconjunto da visão global do produto).

Esta revisão serve para modificar a visão ou para relembrar a Equipe do propósito do empenho de cada um. Além disto, esta revisão permite adicionar, alterar e remover itens que compõem a lista de *backlog* do produto.

Durante a reunião de planejamento do *sprint* é definido o escopo a ser trabalhado no próximo *sprint*, selecionando quais itens do *backlog* do produto serão incorporados à lista *backlog* do *sprint*, que permanecerá estável ao longo de todo o *sprint*.

Antes de iniciar o desenvolvimento das funcionalidades, deve ser criado um plano iterativo baseado em entregas de funcionalidades. Este tipo de planejamento força a Equipe e os Clientes a compreenderem o produto e o projeto de forma a focar na geração de valor e no fluxo de entregas de funcionalidades, ao invés do controle e aderência aos planos iniciais.

Os princípios do Desenvolvimento Enxuto de Software aplicáveis nesta fase são:

- **Visualizar o todo**: antes do inicio do desenvolvimento do produto é concebida a visão que guiará os trabalhos futuros (por exemplo, através de um documento de visão, declaração de elevador[2], fluxogramas etc.). Além da concepção do produto é feita a definição da equipe do projeto, das

[2] Técnica que consiste em resumir o objetivo do projeto identificando: público alvo, necessidade, nome do produto, benefícios, competidores e diferenciais competitivos. A técnica tem este nome pois deve ser possível apresentar o conteúdo em um curto período, correspondendo ao transporte em um elevador. (MOORE, 1991).

ferramentas e outros recursos, a avaliação dos riscos e necessidades de treinamento.

- **Construir integridade**: a elaboração e priorização da lista de *backlog* do produto e do *sprint* é focada no valor gerado para os clientes, uma vez que eles também participam na criação destes artefatos. Este envolvimento garante que os incrementos produzidos terão utilidade e atenderão às necessidades dos Clientes, obtendo a integridade conceitual do produto.
- **Tomar decisões o mais tarde possível**: as decisões são postergadas através da segmentação da lista do *backlog* do produto. As decisões referentes ao escopo do trabalho a ser realizado na próxima iteração são feitas iterativamente nas reuniões de planejamento do *sprint*.

A Tabela 6 apresenta os artefatos produzidos durante a fase Conceber, juntamente com uma descrição sucinta.

Tabela 6: Artefatos produzidos na fase Conceber.

Artefatos produzidos	**Descrição**
Visão do produto	Elaborado no inicio do projeto e revisto durante o planejamento de cada *sprint*.
Backlog do produto	Elaborado no início do projeto e revisto durante o planejamento de cada *sprint*, podendo inserir, alterar ou remover itens da lista.
Arquitetura	Uma arquitetura preliminar do produto é definida no inicio do projeto, e é constantemente adaptada e aprimorada para atender as necessidades emergentes.
Backlog do *sprint*	Produzido durante a reunião de planejamento do *sprint* e é estável durante toda a iteração.
Visão do *sprint*	A partir da visão do produto e do *backlog* do *sprint*, é definido uma visão especifica para o próximo *sprint*.

5.2.2 Explorar e Adaptar

A exploração é a forma que a Equipe executa os projetos. Ao invés de seguir planos prescritos, ela executa o projeto através de uma série de experimentos planejados e entregas iterativas e incrementais de funcionalidade, buscando criar uma formulação concreta da visão do produto (THOMKE, 2003).

A exploração é realizada por equipes competentes, auto-disciplinadas e dirigidas por gerentes experientes que criam um ambiente auto-organizado. As equipes trabalham de forma semi-autônoma, empenhadas em atingir os planos iterativos que elas mesmas ajudaram a construir, gerenciando sua própria carga de trabalho, colaborando para criar novas idéias e utilizando conhecimentos técnicos específicos.

Monitorar e adaptar, atividades tradicionalmente chamadas de monitorar e controlar, são partes de qualquer boa abordagem de gerenciamento de projetos. Apesar de a abordagem ágil utilizar algumas práticas de gerenciamento de projetos tradicionais, ela utiliza algumas práticas diferenciadas. Por exemplo, ao invés de identificar desvios com relação ao planejamento inicial e elaborar relatórios de exceções (apesar de serem necessários em algumas situações), a abordagem ágil busca explorar os cenários emergentes.

Iterações freqüentes, que entregam funcionalidades incrementalmente, permitem que equipes de projetos realizem ajustes baseados em resultados verificáveis, ao invés de artefatos de documentação. Este cenário pode gerar situações de desconforto para alguns gerentes e também clientes, que não desejam tratar constantemente com situações de compromisso.

O desenvolvimento do produto é feito através das práticas e ferramentas selecionadas do XP, que são utilizadas nos trabalhos diários. Diferentemente do modelo cascata, as etapas de projeto, análise, codificação e teste não são caracterizadas formalmente e não exigem um encerramento formal. As funcionalidades produzidas diariamente são agrupadas, e ao final do *sprint*, são incorporadas ao incremento do produto que será apresentado aos Clientes.

Ao final de cada *sprint*, os resultados obtidos são avaliados na reunião de revisão do *sprint* de acordo com as diversas perspectivas envolvidas na criação do produto:

técnica, percepção do Cliente, desempenho da Equipe e do processo. Os resultados destas revisões são utilizados para alimentar a reunião de planejamento do próximo *sprint*, realizado na fase Conceber.

Com as informações levantadas nas reuniões de revisão do *sprint*, que ocorre ao fim de cada iteração, os Clientes e a Equipe do projeto podem aprimorar o planejamento e o produto, incorporando o aprendizado obtido ao longo da execução do projeto.

A Tabela 7 apresenta o artefato produzido durante a fase Explorar e Adaptar, juntamente com uma descrição sucinta.

Tabela 7: Artefatos produzidos na fase Explorar e Adaptar.

Artefatos produzidos	**Descrição**
Funcionalidade	Para cada item do *backlog* do *sprint* são realizadas as atividades de análise, projeto, codificação e testes para serem transformadas em funcionalidades.
Incremento do produto	Quando todos os itens do *backlog* do *sprint* foram transformados em funcionalidades, um incremento de produto funcional para ser entregue aos Clientes.

Os princípios do Desenvolvimento Enxuto de Software aplicáveis nesta fase são:

- **Amplificar o aprendizado**: o conhecimento é adquirido incrementalmente, conforme a elaboração e a entrega de funcionalidades do produto aos clientes. Ao final de cada iteração é realizada uma reunião de revisão do *sprint*, onde a Equipe discute sobre o trabalho realizado, os problemas identificados e as alterações que devem ser feitas para aprimorar o processo e o produto em construção.
- **Fazer entregas o mais rápido possível**: o trabalho é dividido em iterações, reduzindo o intervalo entre o levantamento dos requisitos e o contato dos Clientes com partes funcionais do produto. O ajuste deste intervalo é feito através da duração dos *sprints*, sugerido entre 7 e 30 dias.
- **Construir integridade**: a integridade estrutural do produto é obtida através da experiência dos envolvidos no projeto, da utilização da prática de projeto

simples (trazida do XP), que incrementalmente são detalhados e ampliados para emergir a arquitetura do produto.

5.2.3 Fechar

Similar a fase Pós-*Game* do *Scrum*, esta fase inicia-se quando é acordado que o produto está pronto para utilização e não existem mais itens na lista do *backlog* do produto a serem implementadas. É neste momento que são realizadas as atividades de integração, testes de sistema e documentação.

O objetivo principal desta fase e dos fechamentos parciais realizados ao final de cada *sprint* é gerar e incorporar conhecimento para o trabalho da próxima iteração ou transferi-lo para a Equipe do próximo projeto.

A Tabela 8 apresenta os artefatos produzidos durante a fase Fechar, juntamente com uma descrição sucinta.

Tabela 8: Artefatos produzidos na fase Fechar.

Artefatos produzidos	**Descrição**
Produto final	Contém todos os itens da lista de *backlog* do produto e representa a visão definida na fase Conceber.
Documentação	Os documentos a serem produzidos nesta fase são definidos pelos Clientes, eles definem as necessidades e que tipo de documentação agrega valor ao projeto.

5.3 Práticas

Foram selecionadas as práticas e ferramentas do *Scrum*, relacionadas ao gerenciamento de projetos, e as práticas e ferramentas do XP que são voltadas ao desenvolvimento do software e à garantia da qualidade do produto final, para compor o conjunto de práticas e ferramentas do modelo proposto.

A partir da classificação apresentada na Figura 6 e na Figura 8 (seção **3.2**), onde as práticas dos XP foram agrupadas em quatro círculos: Codificação, Equipe, Processo e Produto, as práticas selecionadas para compor o modelo proposto, foram classificadas e o resultado é apresentado na Tabela 9.

Tabela 9: Relação das praticas selecionadas com os grupos de classificação.

Grupo de Classificação	Metodologia	Práticas
Codificação	XP	– Testes unitários – Reestruturação – Programação em pares
Equipe	XP	– Integração continua – Propriedade coletiva – Padrão de codificação
Processo	XP	– Projeto simples
	Scrum	– Reunião diária – Reunião de planejamento do *Sprint* – Reunião de revisão do *Sprint*
Produto	*Scrum*	– *Backlog* do produto – *Backlog* do *Sprint*

A Figura 17 apresenta em detalhes as práticas selecionadas da fase **Iterações para o Lançamento** do XP e o inter-relacionamento entre elas. A ligação entre as práticas do XP e do *Scrum* ocorre no âmbito do trabalho diário realizado em cada *sprint*, ou seja, no dia a dia a Equipe utiliza as práticas do XP para transformar os itens do *backlog* em incrementos do produto. A codificação em pares é realizada em paralelo com as atividades de análise, projeto, plano de testes e codificação dos testes, não existindo uma transição formal entre elas.

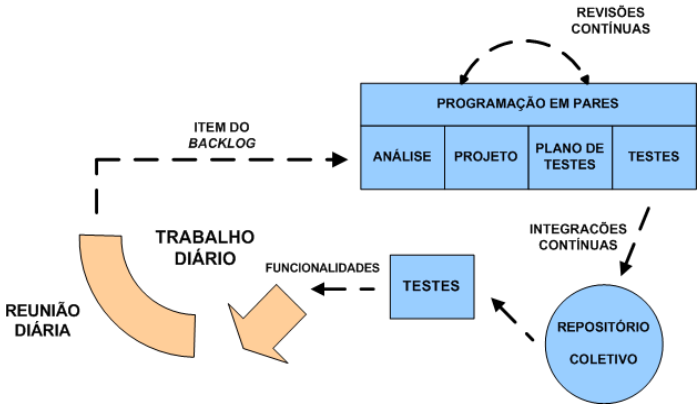


Figura 17: Detalhe das práticas selecionadas do XP.

O primeiro passo para implementar a abordagem enxuta, é aprender a identificar as perdas envolvidas no processo. Estas perdas, descritas anteriormente na seção **4.3 Desperdícios no Desenvolvimento de Software**, são classificadas como qualquer atividade que não agrega diretamente valor.

A Tabela 10 apresenta uma descrição de como as práticas e ferramentas selecionadas para compor o modelo proposto, minimizam a ocorrência das perdas no desenvolvimento de software identificadas na Tabela 3 (apresentada na seção **4.3**).

Tabela 10: Descrições de como as perdas do desenvolvimento de software são tratadas no modelo proposto.

Perdas no desenvolvimento de software	**Descrição das atividades**
Trabalho parcialmente finalizado	Os requisitos são detalhados apenas para a iteração atual. Ao final de cada iteração são entregues incrementos funcionais do produto.
Processo extra	Codificar direto das especificações contidas nos itens do *backlog*, tirando eventuais dúvidas, verbalmente, direto com os Clientes.
Funcionalidades extras	São implementados apenas os requisitos de cada s*print*.
Chaveamento de tarefas	Todos trabalham num mesmo espaço físico e cada um trabalha numa funcionalidade específica do produto (item do *backlog* do produto).
Espera	As entregas são divididas em pequenos incrementos realizados através das iterações. Por exemplo, a codificação não precisa aguardar a especificação completa e detalhada dos requisitos para iniciar.
Movimento	Todos os envolvidos no projeto, trabalham num mesmo espaço físico. Isto facilita o acesso à informação e reduz o tempo necessário para sanar dúvidas e incertezas.
Defeitos	O desenvolvimento é dirigido a testes, antes de iniciar a codificação das funcionalidades é implementado o teste unitário. São realizados continuamente testes para garantir que os incrementos realizados no repositório são consistentes e corretos, estes testes são denominados testes de integração.

A Tabela 11 apresenta resumidamente os elementos que compõem o modelo proposto, indicando as fases, as atividades, os papéis responsáveis e os artefatos produzidos.

Tabela 11: Resumo dos elementos que compõem o modelo proposto.

Fase	**Atividade**	**Responsável**	**Artefato produzido**
Conceber	Elaborar a visão compartilhada do produto do projeto	Cliente, Líder de Projeto e Equipe	Visão do Produto
	Elaborar e revisar o *backlog* do produto	Cliente, Líder de Projeto e Equipe	*Backlog* do produto
	Projeto de alto nível da arquitetura	Líder de Projeto e Equipe	Arquitetura
	Reunião de planejamento do *sprint*	Equipe, Cliente e Líder de Projeto	*Backlog* do *sprint*
	Reunião de planejamento do *sprint*	Equipe, Cliente e Líder de Projeto	Visão do *sprint*
Explorar e Adaptar	Análise, projeto, codificação em pares e testes	Equipe e Líder de Projeto	Funcionalidades
	Integrar continuamente	Equipe e Líder de Projeto	Incremento do produto
	Reunião de revisão do *sprint*	Equipe, Líder de Projeto e Clientes	Incremento do produto
Fechar	Integrar sistema	Líder de Projeto e Equipe	Produto final
	Testes de sistema	Líder de Projeto e Equipe	
	Identificar os documentos necessários	Cliente, Líder de Projeto e Equipe	Documentação
	Elaborar os documentos necessários	Cliente, Líder de Projeto e Equipe	

6 Estudo de Caso

Este capítulo apresenta a aplicação do modelo de gerenciamento de projetos proposto em um projeto concreto de desenvolvimento de um novo produto. Após a descrição desta aplicação, é feita uma análise qualitativa, determinando as vantagens de sua utilização e seu escopo de aplicação.

Para ilustrar a aplicação do modelo proposto num cenário real e analisar os resultados obtidos, foi realizado um estudo de caso no ano de 2006 em uma empresa brasileira de software de pequeno porte (com um quadro de funcionários de aproximadamente trinta funcionários).

Esta empresa foi selecionada pela facilidade de acesso às informações, uma vez que o autor deste trabalho era funcionário, e por ela ser considerada uma das empresas líderes no setor em que atua: o setor financeiro.

Esta empresa, na época em que foi realizado o estudo, não possuía qualquer nível de maturidade de processo (por exemplo: CMM – *Capability Maturity Model*, CMMI – *Capability Maturity Model Integration*, etc.), nem utilizava alguma metodologia padronizada (por exemplo: RUP – *Rational Unified Process*, metodologias ágeis, etc.). O modelo de ciclo de vida de desenvolvimento predominante nos projetos realizados pela empresa era o modelo cascata.

O projeto selecionado para realizar este estudo foi o desenvolvimento de um novo produto (apresentado no **Anexo A - Produto Final do Estudo de Caso**), voltado para aumentar a segurança nas diversas transações financeiras realizadas pelos canais eletrônicos (ATM, internet, telefone, etc.).

A aplicação do modelo proposto contou com o apoio dos diretores da empresa, o que facilitou a realização do trabalho de pesquisa. A empresa tinha um mês para construir um protótipo do produto e apresentá-lo numa feira de exposições de tecnologia e soluções voltadas para o setor financeiro. A empresa estava atrasada em relação a seus concorrentes que estavam em estágios mais avançados no desenvolvimento de produtos similares.

6.1 Equipe do Projeto

A equipe que participou deste projeto foi composta por quatro pessoas com experiência e que trabalhavam de forma bastante integrada em um mesmo espaço físico, facilitando a troca de informações e conhecimentos (inclusive com a presença dos clientes do produto, que no caso eram os departamentos de marketing e comercial da empresa).

Dos quatro envolvidos no projeto, três exerceram o papel de membros da Equipe e um exerceu o papel Líder de Projeto e Treinador.

Estas considerações são importantes para definir o contexto onde foi realizado o ensaio e reduzir o escopo das discussões, não levando em consideração as complexidades referentes à comunicação de grandes equipes, dispersas geograficamente, e inexperiência do pessoal envolvido.

6.2 Execução do Processo

Apesar de contar com o apoio da direção da empresa para a realização do estudo de caso, o inicio do projeto foi cercado de incertezas, por parte de todos os envolvidos no projeto. O modelo proposto significou uma mudança com relação às metodologias que se tornaram norma na indústria de software, e desta forma, necessitou uma mudança na mentalidade da gerência.

O Líder de Projeto, que neste caso coincidiu com o Treinador, constantemente estava empenhado em identificar quem deveria estar fazendo o que, ao mesmo tempo em que estava construindo o ambiente de desenvolvimento adequado (ferramentas, cultura, comunicação etc.).

O projeto tinha de ser concluído em um mês. O modelo proposto, para preservar as características do *Scrum*, sugeria a duração do *sprint* de 30 dias. Esse período coincidia com o prazo para a conclusão do projeto. Para tornar o processo mais adaptável às incertezas e conseguir responder mais rápido aos requisitos emergentes, os *sprints* foram definidos com o período de uma semana (ou cinco dias úteis).

Com esta duração de cada iteração, o projeto foi estimado para ser realizado em quatro *sprints*. Neste período seriam realizados quatro Reuniões de *Sprint* (no inicio de

cada nova iteração), onde poderiam ser removidos e/ou incorporados novos itens ao *Backlog* do Produto.

6.2.1 Conceber

A Figura 18 apresenta as atividades, ferramentas e artefatos que foram utilizados, elaborados ou revisados durante a fase Conceber na realização do projeto.

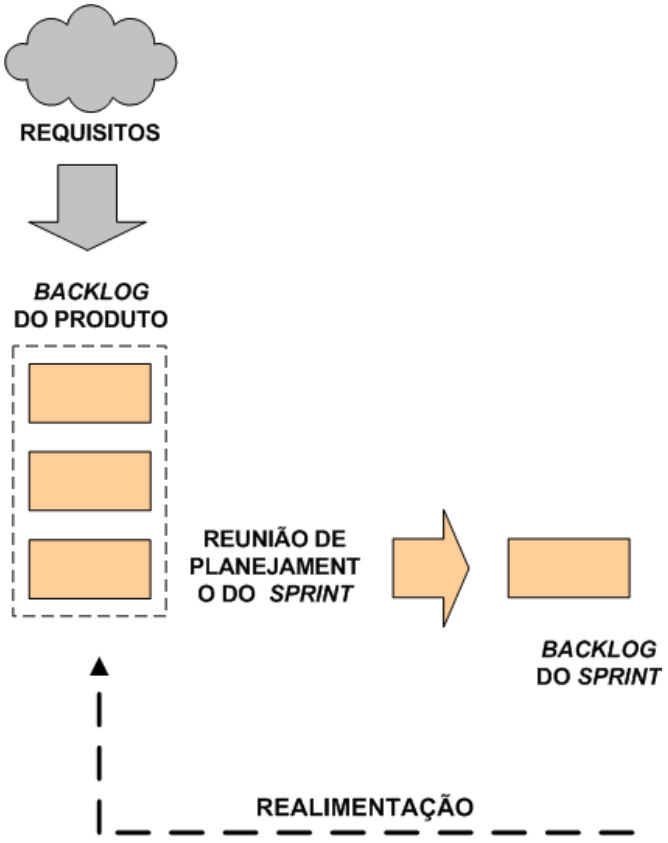


Figura 18: Ilustração da fase Conceber.

O projeto iniciou-se com uma reunião envolvendo a equipe do projeto, os clientes (neste caso representado pelo departamento de marketing e pelo departamento comercial da empresa), o gerente e o líder de projeto (neste estudo de caso também exercendo o papel do treinador).

Nesta reunião inicial, criou-se uma visão do que viria a ser o produto desejado, através da declaração do teste de elevador (descrita anteriormente na seção **5.2.1 Conceber**).

> "Voltado para bancos e empresas de grande porte que necessitam proteger informações sigilosas, transações financeiras e acessos não permitidos, o Pulso m-Security é uma solução composta por um conjunto de aplicações que transforma seu aparelho celular ou PDA num poderoso dispositivo de autenticação forte. Ao contrário das demais soluções de *tokens*, nosso produto apresenta maior flexibilidade na personalização e adequação para diferentes modelos de negócio, com um custo de aquisição muito menor."

As palavras sublinhadas segmentam e identificam as partes que compõem a declaração de elevador. Na primeira frase são destacados o público alvo, a necessidade a ser atendida e uma descrição da solução proposta. A segunda frase inicia-se identificando os concorrentes e finaliza com as vantagens competitivas da solução proposta.

Ainda na reunião inicial, juntamente com a declaração do teste de elevador, foi elaborado um fluxograma que ilustrava esquematicamente o futuro funcionamento da solução proposta, e ao mesmo tempo, este artefato identificava os módulos que comporiam o produto (Figura 19).

Figura 19: Ilustração da visão do produto.

Outro artefato produzido na fase Concepção foi o *backlog* do produto. A equipe, juntamente com o Líder do Projeto, traduziram os requisitos e as restrições fornecidas pelos Clientes para os itens do *backlog*. Neste projeto, este artefato foi produzido utilizando o auxilio de planilhas eletrônicas; o resultado é ilustrado na Figura 20.

	A	B	C	D	E
1	**Backlog do Produto**				
2	**Backlog do Produto**	**Ordem de Trabalho #**	**Descricao da Ordem de Trabalho (extraido do Backlog do Produto)**	**Descricao da Tarefa do Sprint**	**Notas do Status**
3					
4	**Tarefas do Backlog do Produto**				
5	Marketing	001	Elaboracao da apresentacao (feira)	não atribuido	Montar em flash video a ser exibido na feira
10	Servidor	002	Modelo do banco de dados (ER)	confirmar finalizacao	
13	Demo	003	Layout e design do internet banking	aguardando atualizacao	Definir com marketing a padrao e ser seguido
20	Servidor	004	Estruturar ambiente de aceite	aguardando atualizacao	Webserver, banco de dados e app server
30	Cliente	005	Teste do conceito "sandbox"	confirmar finalizacao	Necessario comprovar portabilidade do cliente
37	Servidor	006	Formulario de ativacao do app cliente	não atribuido	Relacionado ao item 017
48	Cliente	007	Inatalacao do aplicativo cliente	não atribuido	
58	Servidor	008	Bug 2		Maquina de estado na falha da autenticacao
65	Servidor	009	Mudanca de Funcionalidade 3	aguardando atualizacao	
73	Demo	010	Autenticacao	aguardando atualizacao	Finalizar integracao com a nova interface
81	Servidor	011	Bug 4	aguardando atualizacao	Ativacao nao finalizada corretamente
90	Demo	012	Produto conta corrente	aguardando atualizacao	Movimentacoes basicas de conta corrente
97	Demo	013	Autorizacao	confirmar finalizacao	
101	Demo	014	Elaboracao do extrato	confirmar finalizacao	Apresentar transacoes efetuadas no C/C
105	Demo	015	Problema com a ordenacao de transacao	analise	
108	Servidor	016	Algoritimo de criptografia dos dados	pesquisa	TripleDES ou AES
112	Cliente	017	Processo de ativacao	pesquisa	Interface de inserir as sementes
116	Cliente	018	Armazenamento das sementes	analise	Armazenar de forma segura as sementes
117	Demo	019	Logout aplicacao demo	confirmar finalizacao	Encerramento da sessao

Figura 20: *Backlog* do produto do estudo de caso.

A tecnologia selecionada para implementar o produto foi o Java; esta escolha foi feita para atender a um dos requisitos apontados pelos clientes, onde eles definiram que seria necessário instalar parte da aplicação em celulares e assistentes pessoas (PDA – *Personal Digital Assistant*).

Grande parte destes dispositivos, celulares GSM (*Global System for Mobile Communications*) e PDA modernos, possuem suporte nativo para esta tecnologia através do *Java Plataform, Micro Edition*, ou simplesmente JME.

Ainda durante a fase Conceber, após a seleção da tecnologia a ser adotada, a Equipe informou uma restrição para os Clientes que deveria ser considerada para cumprir o prazo estipulado: os celulares que utilizassem a tecnologia CDMA (*Code Division Multiple Access*) não seriam compatíveis com o novo produto porque estes utilizavam uma plataforma diferente para criação e distribuição de aplicativos: BREW (*Binary Runtime Enviroment for Wireless*) desenvolvido pela empresa *Qualcomm*.

Nesta fase também foram selecionadas as ferramentas que seriam utilizadas ao longo do projeto, para automatizar as tarefas mecânicas. Para a realização dos testes unitários e sua posterior automatização, foram utilizados os softwares livres[3] "*JUnit*" e "*CruiseControl*", respectivamente. Os testes unitários foram construídos utilizando o *framework* disponibilizado pela ferramenta "*JUnit*" e periodicamente (quando era submetida uma alteração no repositório coletivo) eram executados pela ferramenta "*CruiseControl*".

O lado servidor do repositório coletivo foi construído, também utilizando um software livre, com uma ferramenta chamada "*Subversion*". Os membros da Equipe acessavam este repositório utilizando um cliente chamado "*Tortoise SVN*".

Ao final de cada execução dos testes de integração (automatizados pela ferramenta "*CruiseControl*"), era gerado um relatório com os resultados, e quando eram identificados erros, um e-mail era enviado para a Equipe e para o Líder do Projeto.

---

[3] O software é considerado livre quando atende a quatro requisitos: livre para executá-lo, livre para estudar como ele funciona e adaptá-lo as suas necessidades, liberdade para redistribuir cópias, e por fim, liberdade para aperfeiçoá-lo.

Para a consolidação do *backlog* do produto, acompanhamento do andamento das atividades dos membros da Equipe e geração dos controles visuais foi definido que o Líder do Projeto utilizaria planilhas eletrônicas, que seriam alimentadas por ele, com as informações levantadas nas reuniões de planejamento do *sprint* e nas reuniões diárias.

No estudo de caso, o ciclo representado pela fase Explorar e Adaptar, foi repetido quatro vezes. A cada iteração (*sprint*), o Líder do Projeto criava uma visão coletiva que guiava cada membro da equipe nas decisões e priorizações. A Tabela 12 apresenta a visão de cada um dos quatro *sprints* do estudo de caso.

Tabela 12: Visão coletiva de cada *sprint* do estudo de caso.

Sprint	**Visão**
#1	*Testes de Conceitos - Viabilidades*
#2	*Módulo Cliente*
#3	*Módulo Servidor*
#4	*Aplicação Demonstrativa – Internet Banking*

6.2.2 Explorar e Adaptar

Na fase Explorar e Adaptar eram realizados o desenvolvimento dos itens do *backlog* do Produto. Ao término de cada iteração eram produzidos os incrementos do produto (conhecido como lote de produção na manufatura) de forma iterativa, ou seja, um novo incremento do produto era entregue aos Clientes ao término de cada s*print*.

No estudo de caso, conforme descrito anteriormente, foram realizados quatro iterações (*sprints*). O início de cada *sprint* era marcado pela realização da Reunião de Planejamento do *Sprint*.

Essas reuniões eram conduzidas sempre seguindo o mesmo padrão. O Cliente, juntamente com o Líder do Projeto e a equipe, selecionavam os itens do *backlog* do produto que comporiam o *backlog* do *sprint* e seriam trabalhados nessa iteração.

Além da definição dos itens que seriam trabalhados, o Líder do Projeto determinava a carga de trabalho de cada membro da equipe. O planejamento do trabalho a ser realizado, iniciava com Líder do Projeto determinando a quantidade de horas de trabalho que cada membro da equipe trabalharia no *sprint*.

A Figura 21 apresenta as atividades, ferramentas e artefatos que foram utilizados, elaborados ou revisados durante a fase Explorar e Adaptar na realização do projeto.

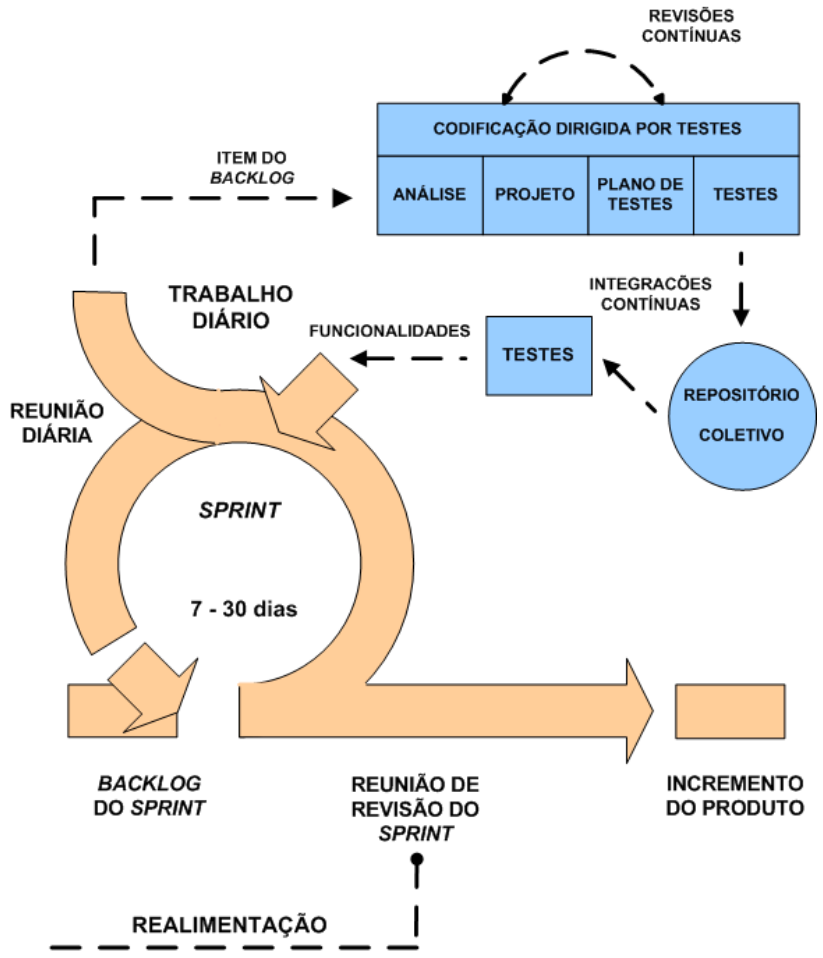


Figura 21: Ilustração da fase Explorar e Adaptar.

Esse cálculo era feito multiplicando o número de dias úteis do *sprint* pelas horas de trabalho. Em seguida o Líder do Projeto calculava a quantidade de horas que cada membro empregaria para trabalhar nos itens do *backlog*. Para isso eram determinados três fatores:

- **Fator de suporte:** definia uma estimativa do percentual do tempo de cada membro da Equipe que seria destinado para as atividades de suporte (por exemplo, auxiliando outro membro da Equipe). Esta estimativa variava de *sprint* para *sprint* e dependia do que estava sendo trabalhado.

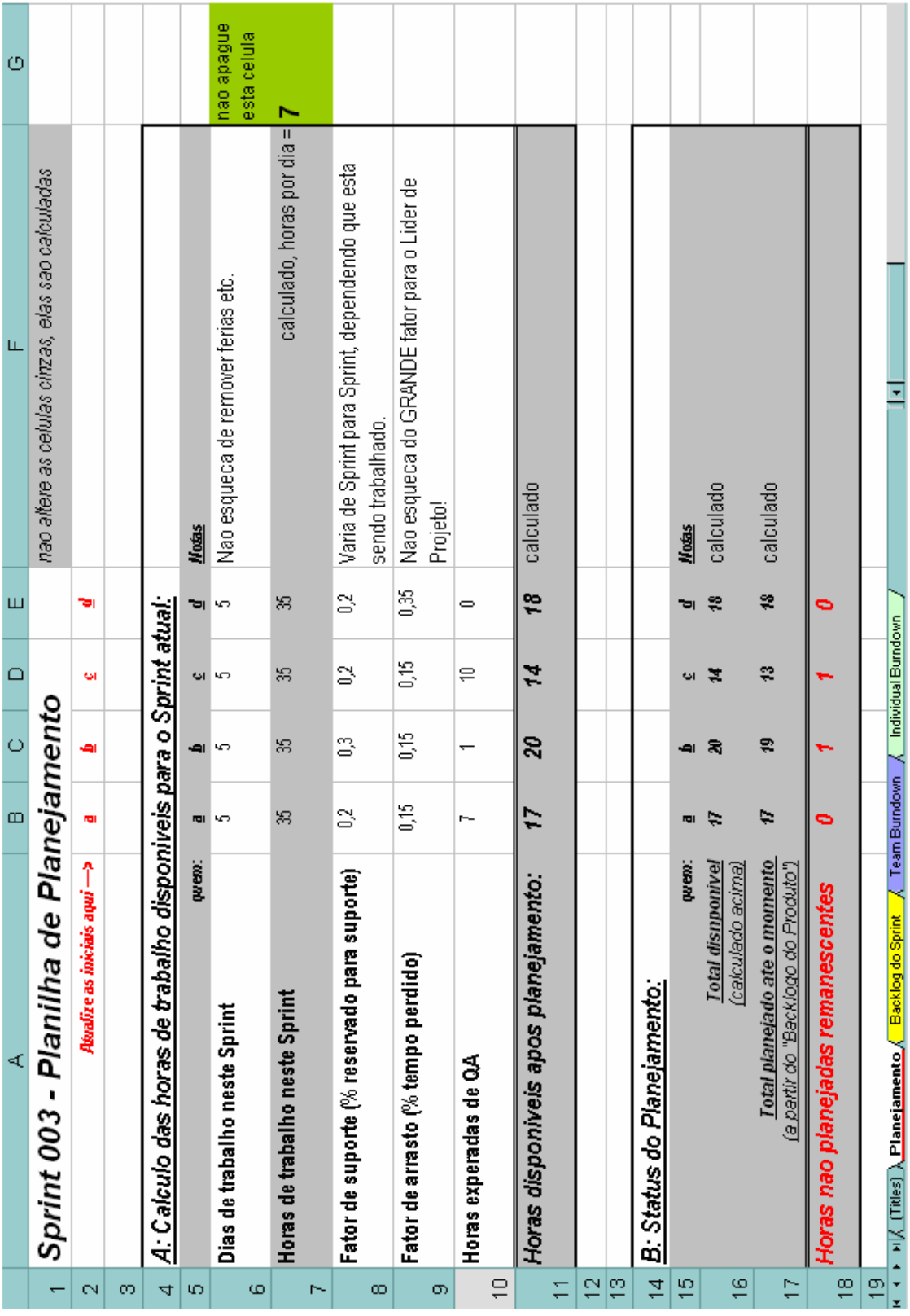
- **Fator de arrasto:** definia uma estimativa do percentual do tempo que cada membro da Equipe destinaria as atividades que não agregava diretamente valor ao Cliente (por exemplo, o Líder do Projeto atualizando as planilhas do *backlog* do produto, as reuniões diárias etc.). Este fator era sensivelmente maior para o Líder do Projeto, que além de realizar tarefas administrativas, também exercia o papel de Treinador (orientando os membros da Equipe com a aplicação do modelo proposto).
- **Horas esperadas de QA (*Quality Assurance*):** determinava a quantidade de horas que cada membro da Equipe destinaria para as atividades de garantia da qualidade.

O planejamento era realizado com o auxilio de uma planilha eletrônica, ilustrada na Figura 22. O Líder de Projeto inseria as informações de cada iteração (dias de trabalho no *sprint*, horas de trabalho por dia, fator de suporte, fator de arrasto e horas esperadas de QA), e em seguida, a planilha calculava automaticamente a carga de trabalho (em horas) disponível para o planejamento.

Depois de calculado a quantidade de horas de trabalho disponível de cada membro da equipe, iniciou-se a alocação dos itens do *backlog* que seriam trabalhados por cada pessoa. Esta atividade, na manufatura enxuta, é conhecida como nivelamento do trabalho.

A seção "B – Status do Planejamento", da Figura 22, ilustra o nivelamento do trabalho. Nesta tabela são apresentadas as quantidades de horas disponíveis para cada membro da equipe (tempo ocioso) e as quantidades de horas de trabalho (tempo alocado). A última linha em vermelho "Horas não planejadas remanescentes", indica quanto de trabalho ainda é possível alocar (quando o número é positivo) ou quanto de trabalho a mais foi alocado (quando o número é negativo).

Após este planejamento e distribuição do trabalho, encerrava-se a Reunião de Planejamento do *Sprint*. Os membros da equipe se reuniam posteriormente com o cliente para esclarecer dúvidas sobre os requisitos, ou ainda, para avaliar alternativas de desenvolvimento.



	A	B	C	D	E	F	G
1	**Sprint 003 - Planilha de Planejamento**					*nao altere as celulas cinzas, elas sao calculadas*	
2	*Atualize as iniciais aqui —>*	a	b	c	d		
3							
4	***A: Calculo das horas de trabalho disponiveis para o Sprint atual:***						
5		quem:	a	b	c	d	*Notas*
6	**Dias de trabalho neste Sprint**		5	5	5	5	Nao esqueca de remover ferias etc.
7	**Horas de trabalho neste Sprint**		35	35	35	35	calculado, horas por dia = **7**
8	**Fator de suporte (% reservado para suporte)**		0,2	0,3	0,2	0,2	Varia de Sprint para Sprint, dependendo que esta sendo trabalhado.
9	**Fator de arrasto (% tempo perdido)**		0,15	0,15	0,15	0,35	Nao esqueca do GRANDE fator para o Lider de Projeto!
10	**Horas experadas de QA**		7	1	10	0	
11	***Horas disponiveis apos planejamento:***		***17***	***20***	***14***	***18***	calculado
12							
13							
14	***B: Status do Planejamento:***						
15		quem:	a	b	c	d	*Notas*
16	*Total disponivel (calculado acima)*		***17***	***20***	***14***	***18***	calculado
17	*Total planejado ate o momento (a partir do "Backlogo do Produto")*		***17***	***19***	***13***	***18***	calculado
18	***Horas nao planejadas remanescentes***		***0***	***1***	***1***	***0***	
19							

(Titles) | Planejamento | Backlog do Sprint | Team Burndown | Individual Burndown

Figura 22: Planejamento do trabalho a ser realizado para cada membro da equipe.

Ao longo de cada *sprint*, eram realizadas as Reuniões Diárias. Estas reuniões eram realizadas todo o inicio do dia de trabalho, e nunca levavam mais de 30 minutos. Cada membro da equipe respondia as três perguntas:

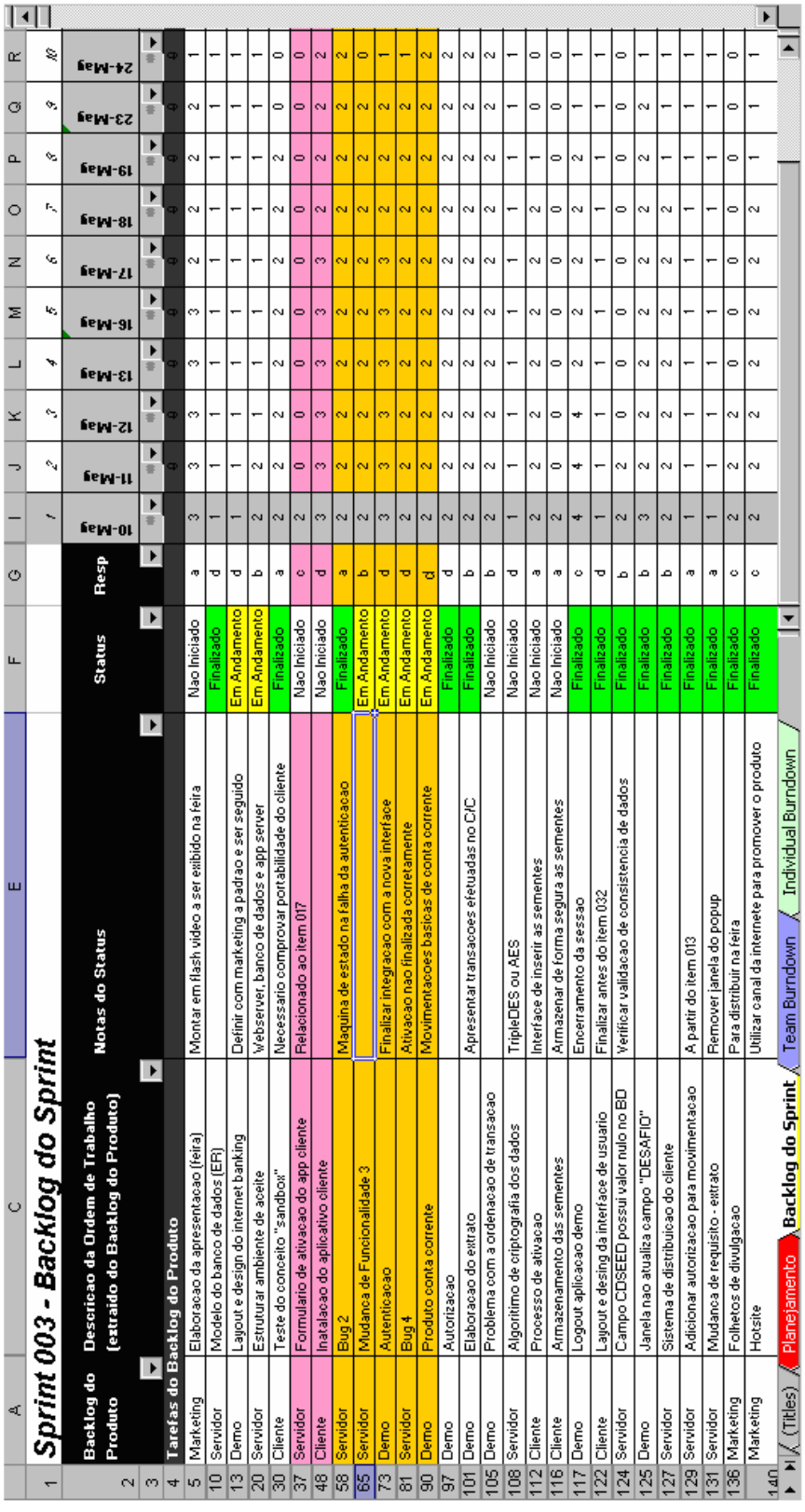
1) O que fez ontem?
2) O que você vai fazer hoje?
3) Existe algum impedimento no seu caminho?

A partir das respostas as estas perguntas, o Líder do Projeto atualizava uma planilha, ilustrada na Figura 23, que consolidava todos os trabalhos realizados e a quantidade dos trabalhos a serem realizados (itens remanescentes no *Backlog* do Produto).

Depois de atualizadas as informações, ao final de cada "Reunião Diária", eram gerados automaticamente os controles visuais. No estudo de caso foram utilizados gráficos do tipo *burndown* (individual e da equipe). Estes controles ficavam visíveis para todos os envolvidos no projeto (Cliente, Equipe, Líder de Projeto / Treinador e Gerente), para que pudessem acompanhar o progresso do projeto.

A Figura 24 apresenta o gráfico *burndown* das atividades de cada membro da Equipe. Esta ferramenta não tinha a finalidade de monitorar o trabalho realizado por cada pessoa e avaliar o desempenho individual. Mas sim, facilitar a identificação de eventuais obstáculos (quando a curva não apresenta a inclinação esperada) e quando alguma pessoa está prestes a ficar ociosa.

Ainda na Figura 24, é ilustrado o gráfico tipo *burndown* da equipe de projeto. Neste gráfico são apresentadas duas curvas: uma apresenta o esforço, em homens/hora, remanescente (representada pela curva azul) e a outra os itens do *backlog* do produto remanescente ao longo do tempo (representada pela curva amarela).



Sprint 003 - Backlog do Sprint

Backlog do Produto	Descricao da Ordem de Trabalho (extraido do Backlog do Produto)	Notas do Status	Status	Resp	10-May	11-May	12-May	13-May	16-May	17-May	18-May	19-May	23-May	24-May
					1	2	3	4	5	6	7	8	9	10
Tarefas do Backlog do Produto					0	0	0	0	0	0	0	0	0	0
Marketing	Elaboracao da apresentacao (feira)	Montar em flash video a ser exibido na feira	Nao Iniciado	a	3	3	3	3	3	2	2	2	2	1
Servidor	Modelo do banco de dados (ER)		Finalizado	d	1	1	1	1	1	1	1	1	1	1
Demo	Layout e design do internet banking	Definir com marketing a padrao e ser seguido	Em Andamento	d	1	1	1	1	1	1	1	1	1	1
Servidor	Estruturar ambiente de aceite	Webserver, banco de dados e app server	Em Andamento	b	2	2	1	1	1	1	1	1	1	1
Cliente	Teste do conceito "sandbox"	Necessario comprovar portabilidade do cliente	Finalizado	a	2	2	2	2	2	2	2	2	0	0
Servidor	Formulario de ativacao do app cliente	Relacionado ao item 017	Nao Iniciado	c	2	0	0	0	0	0	0	0	0	0
Cliente	Instalacao do aplicativo cliente		Nao Iniciado	d	3	3	3	3	3	3	2	2	2	2
Servidor	Bug 2	Maquina de estado na falha da autenticacao	Finalizado	a	2	2	2	2	2	2	2	2	2	2
Servidor	Mudanca de Funcionalidade 3		Em Andamento	b	2	2	2	2	2	2	2	2	2	0
Demo	Autenticacao	Finalizar integracao com a nova interface	Em Andamento	d	3	3	3	3	3	3	2	2	2	1
Servidor	Bug 4	Ativacao nao finalizada corretamente	Em Andamento	d	2	2	2	2	2	2	2	2	2	1
Demo	Produto conta corrente	Movimentacoes basicas de conta corrente	Em Andamento	d	2	2	2	2	2	2	2	2	2	2
Demo	Autorizacao		Finalizado	d	2	2	2	2	2	2	2	2	2	2
Demo	Elaboracao do extrato	Apresentar transacoes efetuadas no C/C	Finalizado	b	2	2	2	2	2	2	2	2	2	2
Demo	Problema com a ordenacao de transacao		Nao Iniciado	b	2	2	2	2	2	2	2	2	2	2
Servidor	Algoritimo de criptografia dos dados	TripleDES ou AES	Nao Iniciado	d	1	1	1	1	1	1	1	1	1	1
Cliente	Processo de ativacao	Interface de inserir as sementes	Nao Iniciado	a	2	2	2	2	2	2	2	1	0	0
Cliente	Armazenamento das sementes	Armazenar de forma segura as sementes	Nao Iniciado	a	2	0	0	0	0	0	0	0	0	0
Demo	Logout aplicacao demo	Encerramento da sessao	Finalizado	c	4	4	4	2	2	2	2	2	1	1
Cliente	Layout e desing da interface de usuario	Finalizar antes do item 032	Finalizado	d	1	1	1	1	1	1	1	1	1	1
Servidor	Campo CDSEED possui valor nulo no BD	Verificar validacao de consistencia de dados	Finalizado	b	2	2	0	0	0	0	0	0	0	0
Demo	Janela nao atualiza campo "DESAFIO"		Finalizado	b	3	2	2	2	2	2	2	2	2	1
Servidor	Sistema de distribuicao do cliente		Finalizado	b	2	2	2	2	2	2	2	1	1	1
Servidor	Adicionar autorizacao para movimentacao	A partir do item 013	Finalizado	a	1	1	1	1	1	1	1	1	1	1
Servidor	Mudanca de requisito - extrato	Remover janela do popup	Finalizado	a	1	1	1	1	1	1	1	1	1	1
Marketing	Folhetos de divulgacao	Para distribuir na feira	Finalizado	c	2	2	2	0	0	0	0	0	0	0
Marketing	Hotsite	Utilizar canal da internete para promover o produto	Finalizado	c	2	2	2	2	2	2	2	1	1	1

Figura 23: Planilha de acompanhamento da evolução do trabalho realizado.

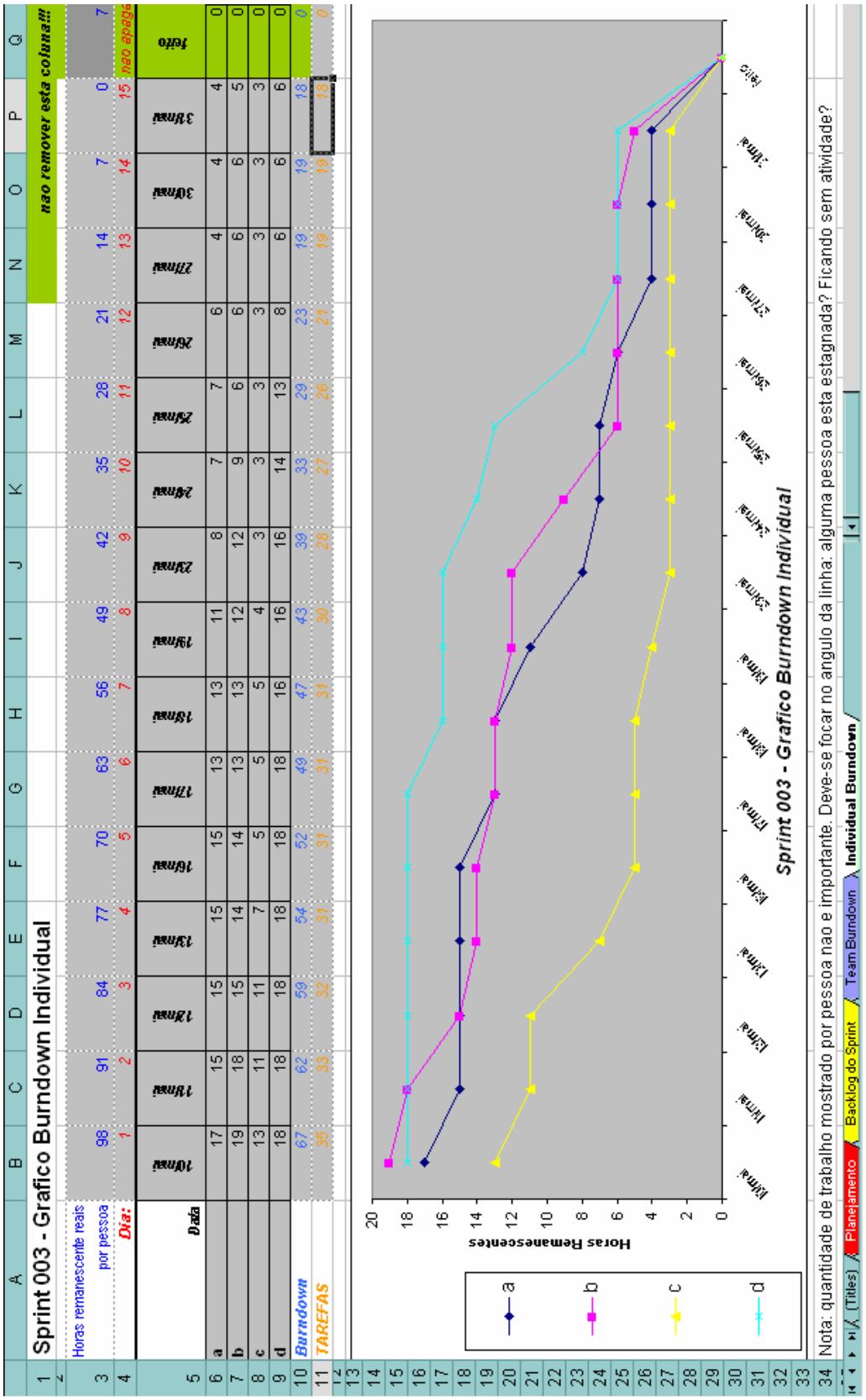


Figura 24: Gráfico do tipo *burndown* de cada membro da equipe.

O Gerente juntamente com o Cliente podiam acompanhar a produtividade da Equipe e constantemente extrair informações que facilitavam a tomada de decisões que determinavam o rumo do projeto. Um exemplo das informações contidas no gráfico da Figura 25 é extraído através da inclinação das duas curvas, dependendo da inclinação e do tempo remanescente para o final do projeto, é possível determinar a dificuldade para a conclusão do projeto.

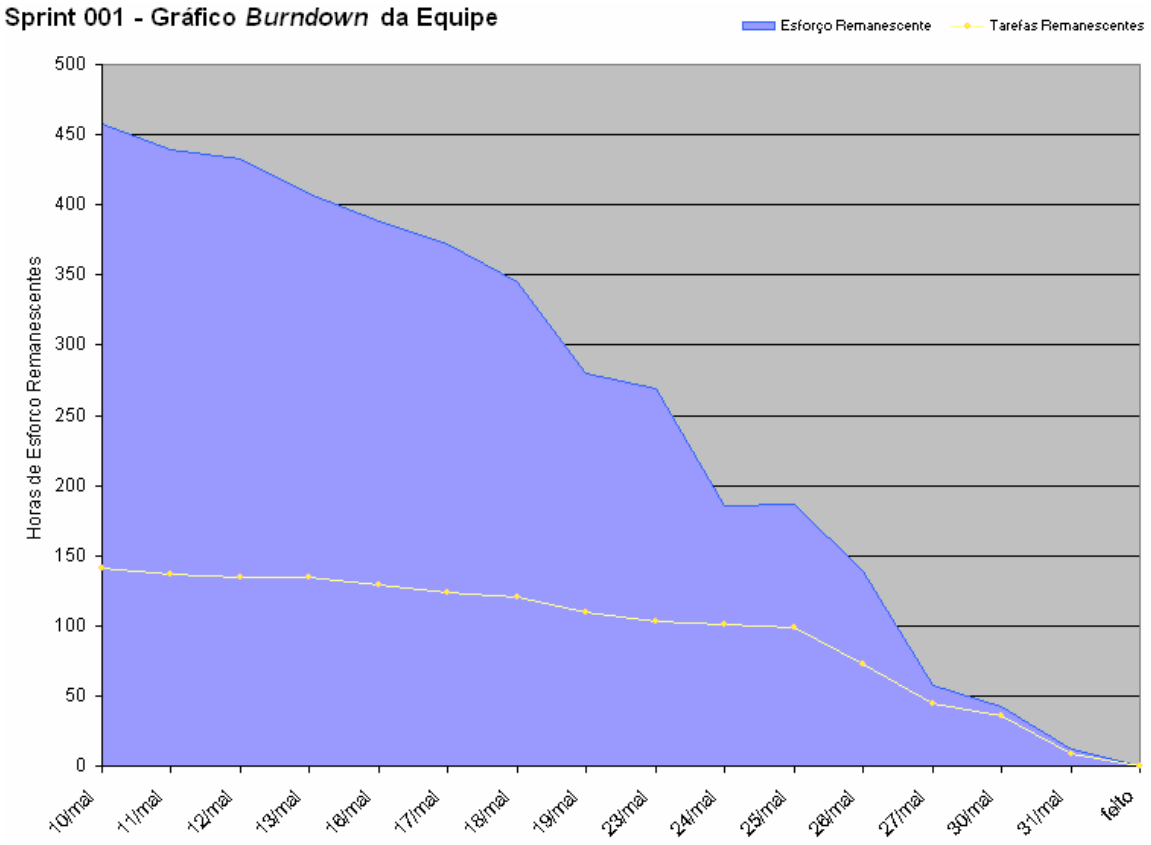


Figura 25: Gráfico tipo *burndown* da equipe.

6.2.3 Fechar

A Figura 26 apresenta as atividades (integração e testes de sistema) e os artefatos que foram utilizados, elaborados ou revisados durante a fase Fechar na realização do projeto (produto final e documentação).

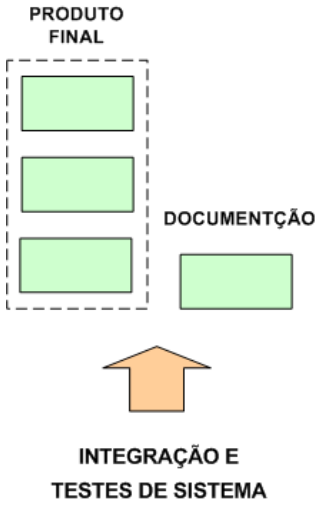


Figura 26: Ilustração da fase Fechar.

Ao final de cada *sprint* foi realizada uma reunião, com duração nunca superior à uma hora, onde foram realizados os fechamentos de cada iteração apresentado o incremento do produto. Nestas reuniões a equipe trazia informações e considerações sobre a estrutura proposta de trabalho, e o Treinador (que nesse caso também era o Líder de Projeto) ajustava o modelo proposto e o adequava para o contexto e a realidade do projeto.

Nesta reunião, realizada ao final de cada *sprint*, também era feita a apresentação do incremento do produto ao cliente e ao gerente. A apresentação era conduzida pelo Líder do Projeto, mas toda a equipe participava. Após a apresentação, o cliente podia revisar o *backlog* do Produto, acrescentando ou removendo itens.

Ao final do quarto *sprint* o projeto foi considerado concluído e o produto pronto para ser apresentado na feira de exposições. A existência de uma restrição na data de apresentação do produto final (que coincidia com a realização da feira onde seria apresentado o produto), determinou a rigidez da data final e não permitiu o prolongamento do projeto.

6.3 Análise da Aplicação do Modelo

A aplicação do modelo proposto, que combinava as metodologias *Extreme Programing* e Scrum juntamente com princípios da produção enxuta, mostrou ter influenciado significativamente a produtividade da equipe envolvida no projeto.

Ao final do projeto, a direção da empresa teve a percepção de que o modelo proposto incorporou práticas e valores à gerência, clientes e à equipe de desenvolvimento, que permitiu a criação de uma equipe unificada e disciplinada.

Do ponto de vista da sinergia entre as metodologias selecionadas para compor o modelo proposto, elas tiveram um impacto significativo na produtividade da equipe (*Scrum*) e na qualidade do produto final (XP).

6.3.1 O XP e a Qualidade do Software

A utilização das práticas de desenvolvimento de software do XP foi percebida como grande influenciadora na melhoria da qualidade do código produzido, em comparação com os projetos anteriores. Apesar de cada prática influenciar aspectos diferentes da qualidade do código, achou-se que elas afetam de forma significativa quando utilizadas em longo prazo. Isto foi percebido porque não eram incluídos códigos de funcionalidades que não seriam utilizadas e, através do padrão de codificação e da priorização pela simplicidade, a comunicação era simplificada e facilitada.

Em parte, este grande impacto foi relacionado às características incrementais que foram aplicadas, melhorando gradativamente a base de código existente enquanto trabalhando pela solução final, gradativamente convergiu-se para uma solução elegante e funcional.

6.3.2 O Scrum e a Produtividade

A aplicação do *Scrum* foi relacionada com o aumento da produtividade da equipe de projeto. Pode haver várias explicações para este aumento, mas a percepção dos envolvidos no projeto foi de que o *Scrum* permitiu que os desenvolvedores focassem apenas na entrega de funcionalidades usuais aos clientes, ao invés de artefatos de valor questionável (como por exemplo, documentações extensas, cronogramas detalhados etc.).

Os artefatos de análise e projeto eram produzidos apenas quando necessário ao longo do projeto, quando era percebido que agregavam valor. Quando não eram necessários os artefatos escritos, a equipe de projeto os ignorava.

6.3.3 Escopo de Aplicação do Modelo Proposto

As metodologias ágeis de desenvolvimento de software têm sofrido muitas criticas, como por exemplo, quanto à sua aplicabilidade em contextos onde a segurança é um fator crítico da aplicação.

Boehm e Turner (2003) identificaram que a natureza complexa do desenvolvimento de software e a grande variedade de métodos (tradicionais e ágeis) influenciam a determinação da aplicabilidade destes princípios. Neste mesmo trabalho, os autores identificaram características importantes de projetos de software, onde é possível observar diferenças entre a abordagem dirigida por planos e a abordagem ágil, que são:

- **Características da aplicação**: objetivos primários do projeto, tamanho do projeto e ambiente da aplicação.
- **Características gerenciais**: relacionamento com o cliente, planejamento e controle e comunicações do projeto.
- **Características técnicas**: abordagens para a definição dos requisitos, desenvolvimento e testes.
- **Características pessoais**: características dos clientes, desenvolvedores e cultura organizacional.

Utilizando estas características, Boehm e Turner (2003) concluíram que existem cinco fatores críticos que determinam a adequação, numa situação particular de um projeto, das metodologias ágeis e das metodologias dirigidas por planos. Estes fatores estão representados na Tabela 13.

A Figura 27 ilustra graficamente as dimensões descritas na Tabela 13. Nota-se que quanto mais próximo à intersecção dos eixos, mais adequado é para aplicar os conceitos das metodologias ágeis no projeto, e quanto mais distante, maior a necessidade de utilizar abordagens dirigidas por planejamento.

Tabela 13: Descrição das dimensões que afetam a seleção do método de desenvolvimento (BOEHM; TURNER, 2003).

Fator	**Ágil**	**Dirigido por planos**
Tamanho da equipe envolvida no projeto	Apropriado para pequenos produtos e equipes. Dependência no conhecimento tácito limita a escalabilidade.	Métodos evoluíram para adequarem-se a grandes produtos e equipes. Difícil de adequar para pequenos projetos.
Criticidade do projeto	Não testados em produtos críticos. Dificuldades potenciais com design simples e falta de documentação.	Métodos evoluíram para adequarem-se a produtos de alta criticidade. Difícil de adequar para produtos de baixa criticidade.
Dinamismo dos requisitos do projeto	Projeto simples e reestruturação contínua são excelentes para ambientes altamente dinâmicos, mas são uma fonte potencial de re-trabalho e custo em ambientes estáveis.	Planejamentos extensos e detalhados realizados no início do projeto, são excelentes para ambientes altamente estáveis, mas uma fonte potencial de re-trabalho e custo em ambiente dinâmico.
Experiência da equipe de projeto	Requer presença contínua de pessoas experientes. Existe risco em utilizar pessoas inexperientes e não conhecedoras das metodologias ágeis.	Necessita de pessoas experientes durante o planejamento do projeto, mas em estágios avançados do projeto este número pode ser reduzido.
Cultura da organização que está executando o projeto	Adequado para culturas onde as pessoas sentem-se confortáveis e responsáveis através de vários graus de liberdade (caos).	Adequado para culturas onde as pessoas sentem-se confortáveis e responsáveis quando tem uma definição clara de seus papéis através de políticas claras e procedimentos (ordem).

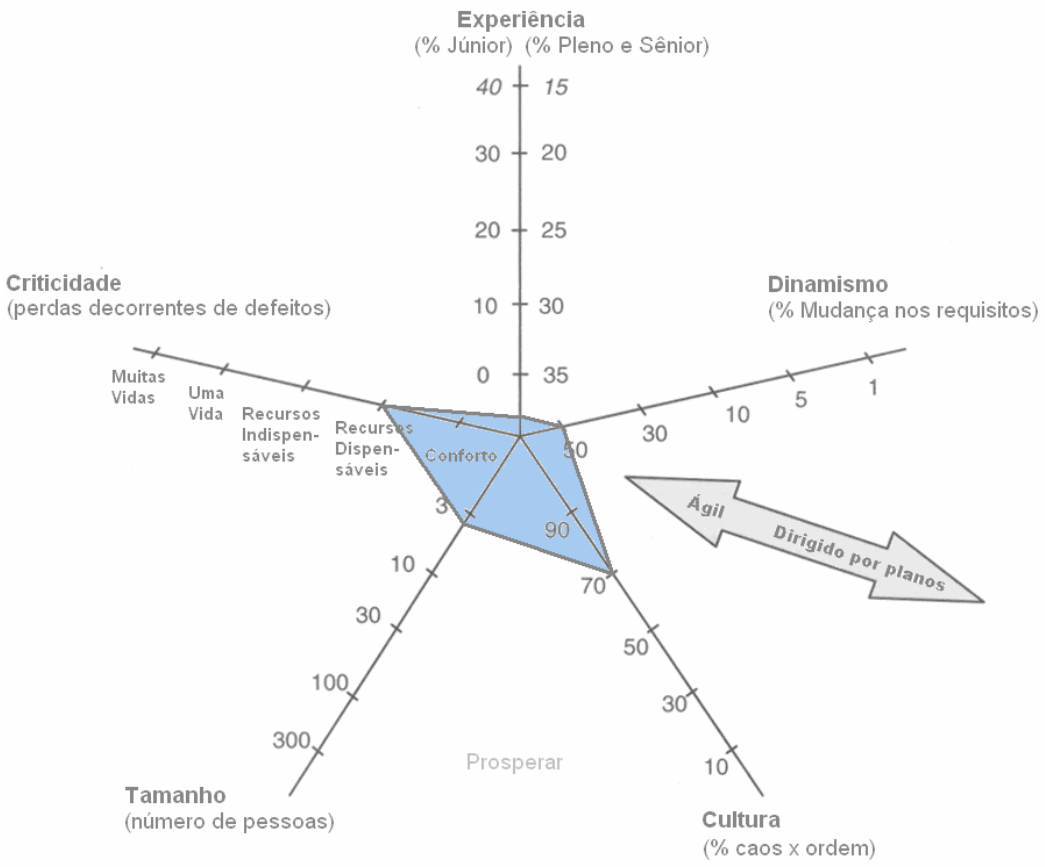


Figura 27: Dimensões que afetam a seleção do método de desenvolvimento (BOEHM; TURNER, 2003).

A região central da Figura 27, destacada em azul, corresponde ao cenário em que foi realizado o estudo de caso apresentado no capítulo **5** deste trabalho. Esta seleção facilitou a aplicação do modelo proposto e foi um elemento influenciador para o resultado positivo apurado ao final do projeto.

Embora haja muita discussão sobre o escopo de aplicação da abordagem da linha de produção para setores de serviço (onde a indústria de software se enquadra, junto com outros setores intensivos de informação), Bowen e Youngdahl (1998) argumentam que esta resistência de aceitação está associada à visão de um modelo de operação manufatureira ultrapassado. Sendo assim, as empresas de serviços precisam alterar o paradigma de produção em linha, adequando a sua realidade, e não abandoná-lo. Estes autores defendem esta visão, uma vez que a manufatura enxuta não traz apenas eficiência, mas também flexibilidade, característica cada vez mais importante num mercado cada vez mais competitivo.

7 Conclusões

Este capítulo apresenta as conclusões deste trabalho. Primeiramente são discutidas as principais contribuições do trabalho, em seguida são apresentadas algumas considerações com relação ao trabalho realizado e resultados obtidos. Por fim são apresentados os trabalhos futuros que poderão ser realizados, a partir dos resultados obtidos.

Do ponto de vista do objetivo apresentado na seção **1.2 Objetivo,** este trabalho atingiu o resultado esperado. Foi apresentado um modelo de gerenciamento de projetos, no capítulo **5**, tomando como base duas metodologias ágeis de desenvolvimento de software (*Scrum* e XP, apresentadas no capítulo **3**), e explorados os princípios da produção enxutas (capítulo **4**).

Após a proposição do modelo, ele foi aplicado e testado em um estudo de caso e em seguida, foi feita uma análise crítica dos resultados obtidos. As contribuições e as considerações em relação aos resultados obtidos são apresentadas nas seções a seguir.

7.1 Contribuições do Trabalho

O modelo proposto combinou duas metodologias que integram a corrente das Metodologias Ágeis de Desenvolvimento de Software (XP e *Scrum*). Esta combinação proporcionou um maior detalhamento e uma definição mais clara e objetiva das atividades relacionadas ao desenvolvimento de software e ao gerenciamento de projetos, facilitando sua implantação para realização do estudo de caso.

Foram incorporados ao modelo proposto princípios da produção enxuta. Esta transferência de teorias entre setores industriais distintos (manufatura e software), serviu para fortalecer o embasamento teórico do modelo proposto e aumentar a credibilidade da proposta, uma vez que estes princípios existem a mais de 50 anos e já tiveram sua eficiência comprovada na manufatura automobilística. Nos estágios iniciais da definição do projeto de estudo de caso, os princípios incorporados no modelo proposto facilitaram a aceitação por parte da diretoria para a realização do estudo de caso.

Foram apresentados diversos fatores que influenciam negativamente a produtividade e os resultados dos projetos de desenvolvimento de software (seção **2.2 Problemas Evidenciados na Indústria de Software**). A Tabela 14 apresenta novamente estes problemas, apontando como o modelo proposto evita a ocorrência de cada uma delas.

Tabela 14: Mapeamento dos problemas da indústria de software com o modelo cascata e o modelo proposto.

Problema	**Modelo Proposto**
Integrações tardias	A cada final do *sprint* é gerado um incremento funcional do produto que pode ser integrado gradativamente e colocado em operação para os usuários finais.
Muitas funcionalidades raramente ou nunca são utilizadas	A definição de funcionalidades é feita ao longo do projeto. Ao final de cada *sprint* os clientes podem incorporar, remover ou alterar o que será incorporado ao produto, de acordo com o aprendizado adquirido.
Pouca interação entre a equipe de projeto e os clientes	A participação dos clientes é ativa, eles participam frequentemente das reuniões de planejamento e avaliação do *sprint*
Intervalo longo entre o planejamento e o contato dos usuários finais com o produto	Ao final de cada *sprint* é apresentado um incremento funcional do produto aos clientes, que podem interferir no futuro desenvolvimento do produto.
Inabilidade de lidar com mudanças de requisitos	Durante as reuniões de planejamento do *sprint*, podem ser incorporados, alterados ou excluídos requisitos.
Pouco conhecimento na durante a especificação dos requisitos	Os requisitos são definidos incrementalmente, incentivando e facilitando a incorporação do conhecimento adquirido ao longo do projeto.

O modelo de gerenciamento proposto explora teorias alternativas de projeto e gerenciamento, apresentadas anteriormente na Tabela 1 (seção **1.3)**. Com base na classificação apresentada nesta tabela, foi elaborada a Tabela 15 que sumariza as teorias exploradas pelo modelo proposto.

Tabela 15: Teorias apresentadas por Koskela e Howell (2002)
exploradas pelo modelo proposto.

Tópicos de teorias		**Teorias**
Projeto		Fluxo
		Geração de valor
Gerenciamento	Planejamento	Gerenciamento por planejamento
		Gerenciamento por organização
	Execução	Perspectiva linguagem e ação
	Controle	Modelo termostato
		Modelo de experimentação cientifica

As seções a seguir descrevem como o modelo proposto explora cada uma das teorias apresentadas na Tabela 15.

7.1.1 Teoria de Projeto

A teoria de transformação é pouco utilizada no modelo proposto. As tarefas não são definidas de forma escrita, elas apenas são definidas oralmente momentos antes do inicio de sua realização. Através das reuniões diárias, as tarefas tornam-se visíveis para todos os envolvidos no projeto.

No lugar do modelo de transformação, o modelo baseado no fluxo é utilizado de muitas formas. No modelo proposto são utilizados ciclos diários e mensais de realimentação, para lidar com as incertezas e variações associadas ao projeto. A estrutura organizacional proposta, baseada em equipes auto-organizadas, permite um fluxo continuo de informações entre os envolvidos no projeto.

Os princípios do modelo de geração de valor também são aplicados pelo modelo proposto. Primeiramente, o valor desejado é expresso através da visão e do *backlog* do produto (contendo uma lista priorizada das funcionalidades desejadas). Segundo, a inclusão dos clientes ao longo da execução do projeto garante que os artefatos produzidos agregam valor ao produto final e incorpora o conhecimento adquirido iterativamente ao

longo do projeto. Por fim, através dos ciclos diários e mensais de realimentação, os clientes podem certificar-se de que os requisitos estão sendo compreendidos de forma satisfatória pela equipe do projeto.

7.1.2 Teoria de Planejamento

No modelo proposto, o planejamento é iniciado através do levantamento das funcionalidades desejadas pelos clientes (*backlog* do produto) e, devido às incertezas tecnológicas, não é possível construir com precisão uma estrutura analítica de trabalho do projeto.

Sem esta estrutura analítica, não é possível realizar as atividades de planejamento e elaboração de cronogramas conforme estipuladas pela doutrina de gerenciamento de projetos (PMBoK). Porém, no modelo proposto as condições de trabalho são padronizadas e é realizado antecipadamente um planejamento superficial (elaboração do *backlog* do produto e reunião de planejamento do *sprint*). Dois ciclos de trabalho são definidos, um com duração de um mês (*sprint*) e o outro com duração de um dia (trabalhos diários).

Do ponto de vista teórico, o modelo proposto segue as teorias relacionadas ao gerenciamento por organização, criando uma estrutura de trabalho com ciclos pré-determinados e comunicações padronizadas (reuniões diárias e de planejamento e revisão do *sprint*). Não existe uma representação central das atividades, elas fluem da situação momentânea criadas pelas ações anteriores. A coordenação é feita diretamente pelos membros da equipe ao invés de ser centralizada.

7.1.3 Teoria de Execução

No modelo clássico de delegação, um controlador central define as atribuições de cada membro da equipe e comunica ao mesmo. No modelo proposto, os papéis de controlador e membro da equipe foram agrupados. Cada membro da equipe define sua tarefa interagindo com o restante da equipe e torna pública a atribuição nas reuniões realizadas diariamente.

Não existe uma declaração formal da finalização de cada tarefa, mas é definido que as restrições ao progresso das tarefas são comunicadas nas reuniões realizadas

diariamente. Caso não seja comunicada nenhuma restrição, é presumido que a tarefa foi concluída. Desta forma, a comunicação de duas vias descrita pela perspectiva linguagem e ação é implementada no modelo proposto.

7.1.4 Teoria de Controle

Existem três níveis de controle no modelo proposto. No nível mais baixo, cada membro da equipe reporta as restrições e os progressos nas reuniões diárias. É responsabilidade do gerente, remover estas restrições para garantir o bom desempenho da equipe.

No segundo nível, o controle é realizado ao final de cada *sprint* (iteração). A equipe apresenta os resultados aos clientes e gerente, e em seguida, são comparados com relação aos resultados planejados. Ainda neste nível de controle, um incremento funcional do produto é apresentado, permitindo um aprendizado por partes dos clientes, gerente e equipe do projeto que influencia nas definições das funcionalidades futuras.

O nível mais alto de controle é relacionado ao projeto como um todo. Ao final de cada *sprint*, durante a reunião de revisão do *sprint*, o *backlog* do produto e o desempenho do projeto (relacionado à duração e custo) são revisados.

Teoricamente interpretado, o nível mais alto de controle é baseado no modelo termostato, enquanto que os dois níveis mais baixo são baseados no modelo de experimentação científica. Os dois níveis mais baixos são voltados para o aprendizado e a criação de conhecimento, o nível mais alto concentra-se nos aspectos de tempo e custo do projeto como um todo.

7.2 Considerações Finais

Apesar dos resultados satisfatórios obtidos na realização do estudo de caso, com relação a aplicação do modelo proposto, eles ainda podem ser questionados quanto ao seu domínio de aplicação em outros cenários.

A escolha favoreceu a obtenção destes resultados, uma vez que segundo a classificação apresentada na Figura 27 e na Tabela 13 (seção **6.3.3 Escopo de Aplicação do Modelo Proposto**), o projeto do estudo de caso se enquadrava no campo das metodologias ágeis (no estágio do produto de teste de conceito, a equipe e o produto eram

de pequeno porte, em caso de falhas não traria grandes perdas, os requisitos eram voláteis, a equipe era experiente e a cultura da empresa era de delegar responsabilidades aos empregados).

Apesar dos envolvidos na realização do estudo de caso considerarem-no como bem sucedido (uma vez que foi cumprido o prazo e atingida a expectativa do cliente), não foi realizada uma coleta de dados dos resultados com o objetivo de conduzir uma análise quantitativa, e assim, poder mensurar os ganhos de produtividade obtidos. Sem esta análise quantitativa não foi possível fazer uma comparação mais detalhada do modelo proposto em relação a outros modelos.

Outra consideração importante quanto a aplicação do modelo proposto na realização do estudo de caso foi que os papéis Treinador e Líder de Projeto, foram exercidos por uma mesma pessoa. A influência desta configuração do modelo proposto no resultado final do projeto também não foi mensurada.

7.3 Trabalhos Futuros

O modelo proposto identificou diversos artefatos que são produzidos ao longo da realização de um projeto e definiu os estágios em que são produzidos. Apesar destas definições, estes artefatos ainda são abstratos. A filosofia das metodologias ágeis sugere a não especificação formal destes artefatos pois eles variam conforme as características de cada projeto. Entretanto, poderia ser elaborado um conjunto de exemplos de artefatos para facilitar a adoção e implantação do modelo proposto.

A comprovação da eficiência do modelo proposto e do aumento da produtividade por ele proporcionado, ainda exige a realização de outros estudos de caso. As novas aplicações do modelo proposto devem explorar diferentes cenários de projetos para determinar as restrições do seu domínio de aplicação e diferentes configurações da equipe do projeto (como por exemplo, duas pessoas distintas exercendo os papéis de Treinador e Líder de Projeto).

A realização de novos estudos de casos, pode ser precedida pela definição de métricas que viabilizem a mensuração da produtividade e qualidade do produto final, e a sua posterior coleta e análise para depois iniciar uma discussão mais aprofundada e fundamentada sobre o modelo proposto.

Referências Bibliográficas

ABRAHAMSSON, P.; SALO, O.; JUSSI, R.; WARSTA J. **Agile Software Development Methods**: Review and Analysis. VTT Publications, v. 478. 2002. 107 p.

ANDERSON, D. **Managing Lean Software Development with Cumulative Flow Diagrams**. In: BORLAND CONFERENCE, 2004, San Jose. Disponível em: <http://www.agilemanagement.net/Articles/Papers/BorConManagingLeanDevWithCFDs.pdf>. Acesso em: 15 de março de 2006.

BECK, K. **Extreme Programming explained**: Embrace change. Reading, Mass., Addison-Wesley, 1999. 244 p.

BOEHM, B.; TURNER, R. **Balancing Agility and Discipline**: A Guide for the Perplexed. Primeira Edição. Addison-Wesley Professional, 2003. 304 p.

BOWEN, D.; YOUNGDAHL, W. "Lean" service: in defense of a production-line approach. **International Journal of Service Industry Managament**, v. 9, p. 207-225, 1998.

BROOKS, F. **The Mythical Man Month**: Essays on Software Engineering, 20th Anniversary Edition. Primeira Edição. Addison-Wesley Professional, 1995. 322 p.

COHEN, D.; LARSON, G.; WARE, B. Improving Software Investments through Requirements Validation. **Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop**. Washington, p. 106 – 114, 2001.

DELUCCA, J.; COAD, P.; LEFEBVRE, E. **Java Modeling in Color with UML**: Enterprise Components and Process. Prentice Hall, 1999. 221 p.

DEMARCO, T.; LISTER, T. **Peopleware**: Productivity Projects and Teams. Nova Iorque Dorset House Pub. Co, 1987. 188 p.

GOLDMAN, S. et. al **Agile Competitors and Virtual Organizations**: Strategies for Enriching the Customer (Industrial Engineering). Wiley, 1994. 414 p.

GOLDRATT, E. **Critical Chain**: A Business Model. Great Barrington: North River Press, 1997. 246 p.

HAIGH, T. Software in the 1960s as Concept, Service, and Product. **IEEE Annals of the History of Computing**, Piscataway, v. 14, p. 5-13, Janeiro 2002.

HARTMANN, D. Exemplo de implementação de uma planilha de gerenciamento do *Scrum*, 2004. Disponível em: <http://toronto.scrums.org/SprintBacklog_sample.zip>. Acesso em: 15 de março de 2006.

HIGHSMITH, J. **Agile Software Development Ecosystem**. Primeira Edição. Addison-Wesley Professional, 2002. 448 p.

_____. **Agile Project Management**: Creating Innovative Products (The Agile Software Development Series). Primeira Edição. Addison-Wesley, 2004. 312 p.

HOUAISS, A. **Dicionário Houaiss da Lingua Portuguesa**. Primeira Edição. Rio de Janeiro: Objetiva. 2004. 3008 p.

JOHNSON, J. et al. **CHAOS**: A Recipe for Success. Published Report, The Standish Group, 1999. 12 p.

JONES, C. **Patterns of Software Failures and Success**. Londres: International Thompson Press. 1995. 292 p.

KOSKELA, L.; HOWELL, G. The Underlying Theory of Project Management is Obsolete. In: PMI RESEARCH CONFERENCE, 2002, Seatle. **Proceedings of the PMI Research Conference**, p. 293-302, 2002.

KRAFCIK, JOHN. Triumph of the Lean Production System. **Sloan Management Review**, v. 30, n. 1, p. 41 – 52, 1988.

LARMAN, C. **Agile and Iterative Development**: A Manager's Guide. Primeira Edição. Boston: Addison-Wesley Professional, 2003. 368 p.

LEVY, L. **Lean Production in International Supply Chain**. Sloan Management Review, v. 38, p. 94-102, 1997.

MANIFESTO for Agile Software Development, 2001. Disponível em: <http://www.agilemanifesto.org>. Acesso em: 15 de março de 2006.

MOORE, GEOFFREY. **Crossing the Chasm**: Marketing and Selling High-Tech Products to Mainstream Customer. Nova Iorque. Harper Business, 1991. 227 p.

NATHAN, A. **Economy Impact Study**: Software and the U.S. Economy in 2002. Business Software Alliance, Janeiro 2003. Disponível em: <http://www.bsa.org/usa/research/upload/SoftwareandUS-Economy-2002.pdf>. Acesso em: 22 de maio de 2006.

NEILL, C.; LAPLANTE P. Requirement Engineering: the state of the practice. **IEEE Software**, Los Alamitos, v. 20, n. 6, p. 40 – 45, Novembro 2003.

OHNO, T. **O Sistema Toyota de Produção**: Além da Produção em Larga Escala. Primeira Edição. Porto Alegre, Bookman. 1997. 152 p.

POPPENDIECK, M. **Make More Money**: improve our standard of living, 2004. Disponível em: <http://www.poppendieck.com/pdfs/Productivity.pdf>. Acesso em: 15 de março de 2006.

POPPENDIECK, M.; POPPENDIECK, T. **Lean Software Development**: An Agile Toolkit for Software Development Managers. Primeira Edição. Boston: Addison-Wesley Professional, 2003. 240 p.

RACCOON, L. Fifty Years of Progress in Software Engineering. **ACM SIGSOFT. Software Engineering Notes**, Nova Iorque, v. 22, n. 1. p. 88 – 104, Janeiro 1997.

ROYCE, W. Managing the Development of Large Software Systems: Concepts and Techniques. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING,

9., 1970, Califórnia. **Proceedings of the 9th International Conference on Software Engineering**, Califórnia, p. 328 – 338, 1970.

SCHWABER, K. **Scrum Development Process**. OOPLSA'95 Workshop on Business Object Design and Implementation. Austin, 1995.

SCHWABER, K.; BEEDLE, M. **Agile Software Development With Scrum**. Primeira Edição. Upper Saddle River: Prentice-Hall, 2001. 150 p.

SHINGO, S. **Study of "Toyota" Production System from Industrial Engineering Viewpoint**: Produce What Is Needed, When It's Needed. Cambridge: Productivity Press, 1981. 291 p.

TAKEUCHI, H.; NONAKA, I. The New New Product Development Game. **Harvard Business Review**, p. 137 – 146, Janeiro - Fevereiro 1986. Disponível em: <http://apln-richmond.pbwiki.com/f/New New Prod Devel Game.pdf>. Acesso em: 22 de maio 2006.

_____. **The Knowledge Creating Company**: How Japanese Companies Create the Dynamics of Innovation. Nova Iorque: Oxford University Press, 1995. 304 p.

TAYLOR, A. IT Projects Sink or Swin. **British Computer Society**, 2001. Disponível em: <http://archive.bcs.org/BCS/review01/articles/itservices/itprojectssinkorswim.htm>. Acesso em: 12 de setembro de 2005.

THOMKE, S. **Experimentation Matters**: Unlocking the Potential of New Technologies for Innovation. Boston: Harvard Business School Press. 2003. 320 p.

WOMACK, J.; JONES, D.; ROOS, D. **The Machine That Changed the World**: The History of Lean Production. Nova Iorque York: Harper Perennial, 1991. 336 p.

Bibliografia Complementar

ALLWAY, M.; CORBETT, S. Shifting to Lean Service: Stealing a Page from Manufacturers' Playbooks. **Journal of Organizational Excellence**, v. 21, n. 2, p. 45-54, Spring, 2002.

ANDERSON, D.; O'BYRNE, B. Lean Interaction Design and Implementation: Using Statecharts with Feature Drive Development. In: SECOND ITERNATIONAL CONFERENCE ON USAGE-CENTERED DESIGN, 2003, Portsmouth. **Proceedings of ForUse**, 2003.

APTE, U.; GOH, C. Applying Lean Manufacturing Principles to Information Intensive Services. **International Journal of Services Technology and Management**, v. 5, n. 5/6, p. 488 – 506, 2004.

BECK, K. Embracing Change With Extreme Programming. **IEEE Computer Society**, Los Alamitos, v. 32, n. 10, p. 70 - 77. 1999.

BOEHM, B. Get Ready for Agile Methods, With Care. **IEEE Computer Society**, Los Alamitos, v. 35, n. 1, p. 64-69, 2002.

BOEHM, B.; HUANG, L. Value-Based Software Engineering: Reinventing "Earned Value" Monitoring and Control. **ACM SIGSOFT Software Engineering Notes**, Nova Iorque, v. 28, n. 2, 2003. 4 p.

BUHRER, H. Software Development: What it is, What it should be, and How to get There. **ACM SIGSOFT Software Engineering Notes**, Nova Iorque, v. 28 n. 2, 2003. 4 p.

CHAU, T.; MAURER, F.; MELNIK G. **Knowledge Sharing: Agile Methods vs. Tayloristic Methods**, Proceedings of 12th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise, IEEE 2003.

COCKBURNM A.; HIGHSMITH, J. Agile Software Development: The People Factor. **IEEE Computer Society**, v. 34, n. 11, p. 131 – 133, Novembro 2001.

DEMARCO, T.; BOEHM, B. The Agile Methods Fray. **IEEE Computer Society**, Los Alamitos, v. 35, n. 6, p. 90 – 92, 2002.

GIL, A. C. **Como elaborar projetos de pesquisa**. Quarta Edição. São Paulo: Atlas, 1991. 176 p.

GILB, T. **Principles of Software Engineering Management**. Massachusetts: Addison-Wesley Professional, 1988. 464 p.

HIGHSMITH, J. What is Agile Software Development? **CrossTalk, The Journal of Defense Software Engineering**, v. 15, n. 10, p. 4 – 9, Outubro 2002.

HIGHSMITH, J.; COCKBURN, A. Agile Software Development: The Business of Innovation. **IEEE Computer Society**, Los Alamitos, v. 43, n. 9, p. 120 – 122, 2001.

HIRANO, H. **JIT Factory Revolution**: A Pictorial Guide to Factory Design of the Future. Productivity Press, 1989. 218 p.

JARZOMBEK, J. **The 5th Annual Joint Aerospace Weapons System Support, Sensor, and Simulation Symposium (JAWS S3) Proceedings**, Government Printing Office Press, 1999.

JOHNSTON, B.; BRENNAN, M. Planning or Organizing: the Implications of Theories of Activity for Management Operations. **Omega, International Journal of Management. Science**, v. 24, n. 4, p. 367-384, 1996.

KOSKELA, L. **An exploration towards a production theory and its application to construction**. VTT Publications, v. 408, 2000. 296 p.

KOSKELA, L.; HOWELL, G. Reforming Project Management: The Role Of Planning, Execution and Controlling. **Proceedings of the 9th International Group for Lean Construction Conference**, p. 185 – 198, 2001.

LAPLANTE, P. & NEILL, C. The Demise of the Waterfall Model is Eminent and Other Urban Myths. **ACM Queue**, v. 1, n. 10, Fevereiro 2004.

LAREAU, W. **Office Kaizen**: Transforming Office Operations into a Strategic Competitive Advantages. American Society for Quality, Quality Press, 2002. p. 174.

MACCOMACK, A. Product-Development Practices That Work: How Internet Companies Build Software. **MIT Sloan Management Review**, v. 42, n. 2, pp. 75 – 84, 2001.

MCBREEN, P. **Questioning Extreme Programming**. Primeira Edição. Boston, MA: Pearson Education, 2003. 224 p.

MEYER, A.; LOCH, C.; PICH, M. Managing Project Uncertainty: From Variation to Chaos. **MIT Sloan Management Review**, v. 43, n. 2, p. 60 – 67, 2002.

NEILL, C. J.; LAPLANTE, P. A. Requirements Engineering: the state of the practice. **IEEE Software**, v. 20, n. 6, p. 40 – 45, 2003.

OGILVY, J. **What Strategies Can Learn from Sartre**. Strategy & Competition, v. 33, Winter 2003. 10 p.

ORRECHIA, F.; HOWELL, G. Reflection on Money and Lean Construction. **Proceedings of International Group for Lean Construction 7 (IGLC)**, Berkley, p. 253 – 262, 1999.

PATRICK, F. Promises and Perceptions: How the Theory of Constraints Can Help Cure Common Projects Ailments. **Better Software Magazine**, v. 6, n. 1, p. 35 – 47, 2004.

POPPENDIECK, M. **Principles of Lean Thinking**. In: ANNUAL ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA), 17., 2002, Seattle. November, 2002.

REIFER, D. How Good Are Agile Methods? **IEEE Software**, Los Alamitos, v. 19, n. 4, p. 16 – 18, Julho 2002.

SHENHAR, J. From Theory to Practice: Toward a Typology of Project Management Styles. **IEEE Transactions on Engineering Management**, v. 45, n. 1, p. 33 – 48, Fevereiro 1998.

SKAGGS, B.; YOUNDT, M. Strategic Positioning, Human Capital, and Performance in Service Organizations: A Customer Interaction Approach. **Strategic Management Journal**, v. 25, p. 85 – 99, 2004.

SLIGER, M. Fooling Around with XP: Why I Lost Interest in PMI and Took Up With Something More Extreme. **Better Software Magazine**, v. 6, n. 5, p. 16 – 18, Maio 2004.

SOLON, R; STAZ, J. Benchmarking the ROI for Software Process Improvement (SPI): Some New Thoughts on an Old Problem. **The DoD SoftwareTech News**, Nova Iorque, v. 5, p. 6-11. Novembro 2002.

SUZAKI, K. **New Manufacturing Challenge**: Techniques for Continuous Improvement, Free Press, 1987. 255 p.

TURNER, R. Project Management: A Profession Based on Knowledge or Faith? **International Journal of Project Management**, v. 17, n. 6, p.329-330, 1999.

YOFFIE, D.; KWAK, M. **Judo Strategy**: Turning Your Competitor's Strength to Your Advantage. Harvard Business Press, 2003. 256 p.

WOMACK, J.; JONES, D. **Lean Thinking**: Banish Waste and Create Wealth in Your Corporation, Revised and Updated. Segunda Edição. Free Press, 1990. p. 384.

Anexo A - Produto Final do Estudo de Caso

Apesar de o produto ser voltado para instituições financeiras, os usuários finais seriam os clientes de instituições que utilizam os canais eletrônicos para realizar transações financeiras.

A utilização do produto inicia-se com o usuário transferindo e instalando o módulo cliente de um servidor seguro, através do protocolo WAP (*Wireless Application Protocol*), para um dispositivo móvel como celulares ou PDAs. Este processo é representado na Figura 28.

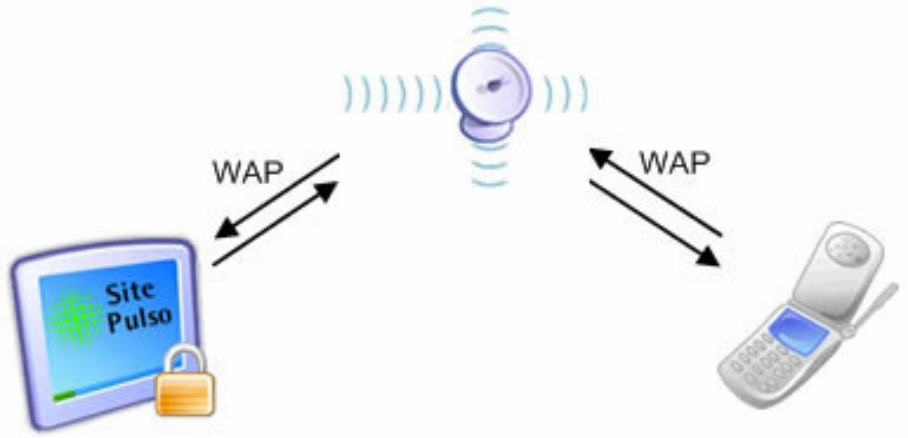


Figura 28: Instalação do aplicativo no celular.

O usuário utiliza o módulo cliente, instalado no seu celular ou PDA, primeiramente para autenticação em sistemas de internet oferecidos pelas instituições financeiras. Um exemplo de uso é ilustrado na Figura 29, onde um usuário acessa, através da internet, um sistema de "*E-Banking*".

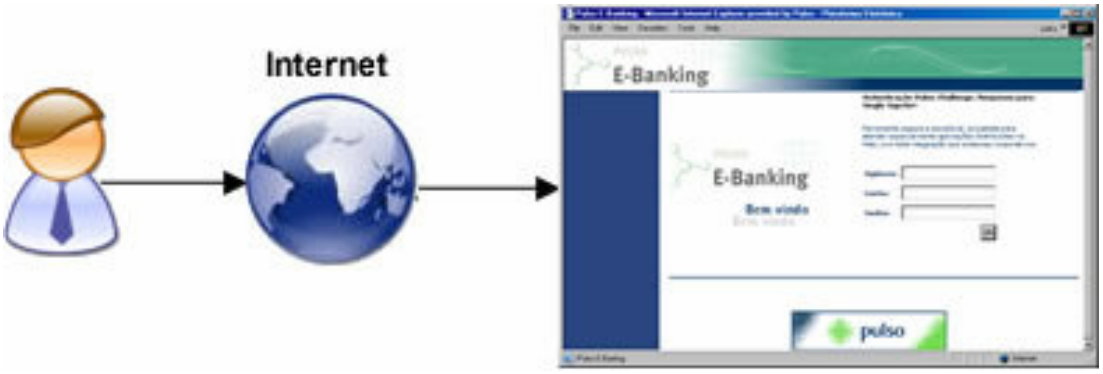


Figura 29: Tela de autenticação do internet *banking*.

Este exemplo de caso de uso também foi desenvolvido durante o projeto de estudo de caso, para ser apresentado na feira onde seria exposto o novo produto. Antes de iniciar a autenticação no sistema, é necessário ativar o módulo cliente instalado no dispositivo móvel (celular ou PDA).

Este processo é feito através do preenchimento de um formulário (Figura 30), via internet, onde o usuário insere algumas informações pessoais para confirmar a posse do dispositivo onde está instalado o módulo cliente do produto.

Figura 30: Formulário de ativação do aplicativo.

Depois de preenchido o formulário, são geradas algumas informações (sementes) que deverão ser inseridas no módulo cliente. Estas sementes são pessoais e intransferíveis e garantem a autenticação e autorização entre servidor e cliente, através de um segredo de posse única e exclusivamente do cliente (ao contrário da autenticação pelo par usuário e senha onde o segredo é compartilhado). Este processo está representado graficamente na Figura 31.

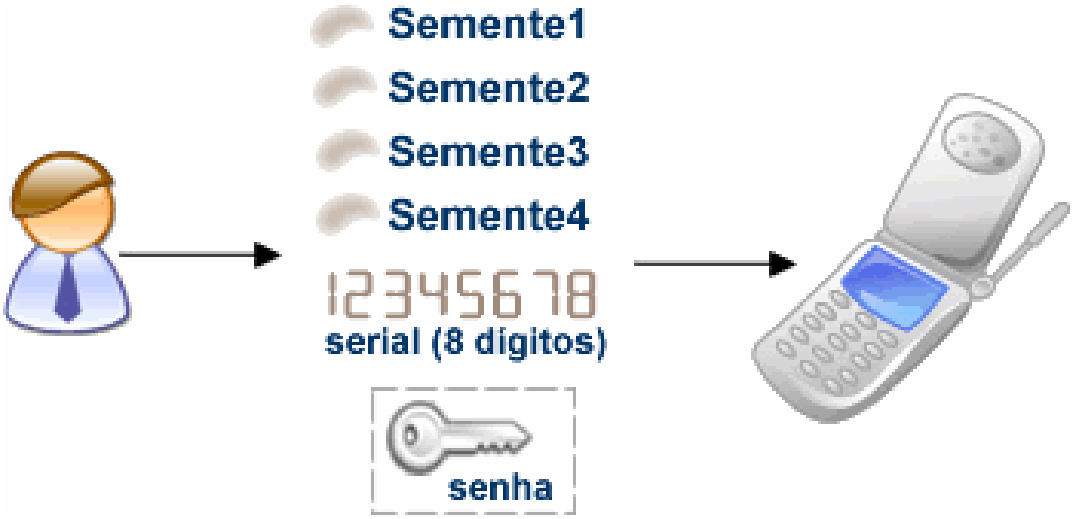


Figura 31: Usuário insere as sementes no aplicativo.

Após a ativação do módulo cliente, realizada através da inserção das sementes, ele está pronto para ser utilizado. Um dos exemplos de caso de uso é a utilização do dispositivo no processo de autenticação de usuários em um sistema de banco pela internet ("*E-Banking*").

A Figura 32 ilustra uma tela de autenticação, desenvolvida no estudo de caso, onde além dos campos de entrada de dados "Agência" e "Conta", é apresentado um "Desafio" e esperado uma "Resposta".

Figura 32: Site de demonstração do produto.

O "Desafio", apresentado na Figura 32, deve ser inserido no módulo cliente (instalado no celular ou PDA) que gerará uma resposta. Esta seqüência de caracteres deve ser inserida no campo "Resposta" para efetuar a autenticação. A Figura 33 apresenta graficamente este processo.

Figura 33: Ilustração da utilização do aplicativo do celular.