

**UNIVERSIDADE FEDERAL FLUMINENSE**  
**ALESSANDRA GALVÃO DA SILVA**

**A IMPORTÂNCIA DOS MÉTODOS ÁGEIS NA ENGENHARIA DE**  
***SOFTWARE***

**Niterói**  
**2016**

**ALESSANDRA GALVÃO DA SILVA**

**A IMPORTÂNCIA DOS MÉTODOS ÁGEIS NA ENGENHARIA DE  
*SOFTWARE***

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

**Orientador:**

**João Gabriel Felipe Machado Gazolla**

**NITERÓI**

**2016**

**ALESSANDRA GALVÃO DA SILVA**

**A IMPORTÂNCIA DOS MÉTODOS ÁGEIS NA ENGENHARIA DE  
SOFTWARE**

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, 01 de Junho de 2016.

Banca Examinadora:

---

Prof. João Gabriel Felipe Machado Gazolla - MSc. – IC/UFF  
UFF – Universidade Federal Fluminense

---

Prof. Daniel Luiz Alves Madeira DSc. – Avaliador  
UFSJ – Universidade Federal de São João del-Rei

Dedico este trabalho ao meu esposo, aos  
meus estimados filhos e aos meus pais.

## **AGRADECIMENTOS**

Agradeço a Deus, pois, sem ele eu não teria forças para essa longa jornada.

Agradeço ao meu esposo, Jaime que tanto amo, que de forma especial e carinhosa me deu força e coragem, me apoiando nos momentos de dificuldades.

Aos meus filhos, Rodrigo e Paloma, vidas da minha vida, que sempre me apoiaram e acreditaram em mim.

Aos meus maravilhosos pais, Nelson e Sandra, que Deus me concedeu, obrigada por acreditarem em mim.

A meu Orientador João Gabriel Gazolla pela paciência na orientação e atenção que me concedeu durante a conclusão deste TCC.

Porque dele e por ele, e para ele, são todas as coisas; glória, pois, a ele eternamente. Amém.

Romanos 11:36

## RESUMO

Este trabalho apresenta a importância dos métodos ágeis na engenharia de *software*. Relata a evolução dos métodos tradicionais até a chegada dos métodos ágeis e apresenta comparações entre os dois métodos para então ser apresentada a real importância do método ágil na engenharia de *software*.

**Palavras-chaves:** método ágil, método tradicional e comparação entre os métodos.

## **ABSTRACT**

This work presents the importance of agile methods in software engineering. Reports the evolution of the traditional methods until the arrival of agile methods and presents comparisons between the methods, showing how important are the agile methods in software engineering.

**Key words:** agile method, traditional method, software engineering.



## LISTA DE ILUSTRAÇÕES

Figura 1: A evolução do <i>software</i> ao longo dos anos (S.PRESSMAN, 1995).....	18
Figura 2: Camadas da Engenharia de <i>Software</i> [2, pg 39]. ....	19
Figura 3: O ciclo de vida clássico. (S.PRESSMAN, 1995) .....	22
Figura 4: O modelo incremental (S.PRESSMAN, 2011) .....	24
Figura 5: Processo de desenvolvimento de protótipo (SOMMERVILLE, 2011). ....	26
Figura 6: O exemplo das diversas formas de se entender um projeto (GOMES, 2013). ....	27
Figura 7: Modelo em espiral de processo de <i>software</i> (SOMMERVILLE, 2011). ....	29
Figura 8: Modelo Transformacional (SOMMERVILLE, 2003). ....	31
Figura 9: Processo da Extreme Programming (XP) (S.PRESSMAN, 2011).....	40
Figura 10: Fluxo de um Processo Scrum (BRAZ, 2011). ....	42
Figura 11: Estrutura FDD (FDD - <i>Feature Driven Development</i> , 2015). ....	45
Figura 12: A Distribuição do Método <i>Crystal</i> (RAFAEL DIAS RIBEIRO, 2015).....	46

## LISTA DE TABELAS

Tabela 1: Vantagens e Desvantagens do modelo Cascata. ....	32
Tabela 2: Vantagens e Desvantagens do modelo Incremental.....	33
Tabela 3: Vantagens e Desvantagens do modelo Prototipação .....	34
Tabela 4: Vantagens e Desvantagens do modelo Espiral.....	35
Tabela 5: Vantagens e Desvantagens do modelo Transformacional .....	36
Tabela 6: Características dos Métodos Ágeis.....	49
Tabela 7: Pontos Positivos e Negativos dos Métodos Ágeis. ....	50
Tabela 8: Comparativo Geral da Abordagem Clássico com a Ágil (Adaptado,GRANDO, 2010) .....	52
Tabela 9: Pontos Positivos e Negativos dos Métodos Tradicionais e Ágeis . ....	54

## LISTA DE ABREVIATURAS E SIGLAS

CLF – Construir a Lista de *Features*.

CPF – Construir por *Feature*.

CRC – Classes, Responsabilidades e Colaboração.

DMA – Desenvolver Modelo Abrangente.

DPF – Detalhar por *Feature*.

FDD – *Feature Driven Development*.

PPF – Planejar por *Feature*.

XP – *Extreme Programming*.

# SUMÁRIO

1	INTRODUÇÃO.....	15
1.1	OBJETIVO .....	16
1.2	ORGANIZAÇÃO DO TRABALHO .....	16
2	Histórico .....	17
2.1	O SURGIMENTO DA ENGENHARIA DE <i>SOFTWARE</i> .....	18
2.2	ORIGEM DOS MÉTODOS ÁGEIS.....	20
3	FUNDAMENTAÇÃO TEÓRICA .....	21
3.1	MÉTODOS TRADICIONAIS.....	21
3.1.1	MODELO CASCATA ( <i>WATERFALL</i> ).....	22
3.1.2	MODELO INCREMENTAL .....	24
3.1.3	MODELO PROTOTIPAÇÃO .....	25
3.1.4	MODELO ESPIRAL .....	28
3.1.5	MODELO TRANSFORMACIONAL .....	30
3.1.6	ANÁLISE DOS MÉTODOS TRADICIONAIS. ....	31
3.2	MÉTODOS ÁGEIS.....	37
3.2.1	<i>EXTREME PROGRAMMING (XP)</i> .....	39
3.2.2	SCRUM .....	41
3.2.3	<i>FEATURE DRIVEN DEVELOPMENT (FDD)</i> .....	44
3.2.4	<i>CRYSTAL</i> .....	46
3.2.5	ANÁLISE DAS METODOLOGIAS ÁGEIS .....	48
3.3	ANÁLISE DOS MÉTODOS CLÁSSICOS E MÉTODOS ÁGEIS .....	51
4	EXEMPLOS E SUGESTÕES DE ESCOLHA DE METODOLOGIA ....	55
4.1	EXEMPLO 01 – INFORMATIZAÇÃO DO ESTOQUE DE UMA PADARIA. ..	55
4.1.1	CENÁRIO E DEFINIÇÃO DO PROBLEMA .....	55
4.1.2	ANÁLISE DA METODOLOGIA SUGERIDA .....	55
4.2	EXEMPLO 02 – DESENVOLVIMENTO DE UM SISTEMA DE NAVEGAÇÃO PARA UMA AERONAVE NÃO TRIPULADA, <i>DRONE</i> . ....	56
4.2.1	CENÁRIO E DEFINIÇÃO DO PROBLEMA .....	56

4.2.2	ANÁLISE DA METODOLOGIA SUGERIDA .....	56
4.3	EXEMPLO 03 – DESENVOLVIMENTO DE UM SISTEMA <i>FOOD</i> PARA CELULAR.....	56
4.3.1	CENÁRIO E DEFINIÇÃO DO PROBLEMA .....	56
4.3.2	ANÁLISE DA METODOLOGIA SUGERIDA .....	56
4.4	EXEMPLO 04 – DESENVOLVIMENTO DE UM SISTEMA ALFABETIZAÇÃO INFANTIL .....	57
4.4.1	CENÁRIO E DEFINIÇÃO DO PROBLEMA .....	57
4.4.2	ANÁLISE DA METODOLOGIA SUGERIDA .....	57
5	CONCLUSÕES.....	58
5.1	TRABALHOS FUTUROS .....	59



# 1 INTRODUÇÃO

Atualmente a indústria de desenvolvimento de *software* tem se tornado uma das mais importantes indústrias, pois cria produtos essenciais para o estilo de vida atual, como os *softwares* que são utilizados desde para automação dos nossos negócios, até mesmo para a segurança dos veículos que dirigimos, controle de produção e em muitas outras áreas do nosso dia-a-dia. Para criação destes *softwares*, existem milhares de desenvolvedores e engenheiros em todo mundo.

Este mercado é extremamente competitivo no ambiente de desenvolvimento de *software*, para ser o melhor nesta indústria é necessário ter a habilidade de criar e entregar os produtos dentro do prazo, sem extrapolar o custo estipulado, obtendo uma qualidade superior aos demais concorrentes e que ainda responda de forma mais completa possível, as necessidades do cliente. Esta tarefa é difícil e com base nestas necessidades surgiram os métodos ágeis.

Os métodos ágeis chegaram com a incumbência de melhor organizar estes *softwares* que a cada vez se tornavam maiores e mais complexos, já que os métodos clássicos exigiam uma documentação extensa e se viu a necessidade de algo mais objetivo, além de um contato maior com cliente, uma relação mais estreita e próxima.

Os métodos ágeis possibilitam uma melhor comunicação com o cliente e a equipe que constrói o *software* tornando assim tudo mais claro nesta construção, com isso, os custos com manutenção devidos a erros de entendimento cliente-programador, cai de forma expressiva.

Estes métodos ágeis também são mais flexíveis às necessidades do cliente, já que agora (diferente dos modelos clássicos) mudanças são bem-vindas a cada interação com o cliente. Através do uso do método ágil, um trabalho possui mais chances de ser entregue no prazo pré-estabelecido com qualidade. Neste método o foco esta na equipe, na validação e na eliminação de erros.

## 1.1 OBJETIVO

Este trabalho descreve e compara as principais metodologias de desenvolvimento clássico, mas com foco no desenvolvimento ágil de *software*, que estão sendo utilizadas globalmente, na intenção de se desenvolver *software* de qualidade.

## 1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho foi organizado da seguinte forma: No capítulo 2 apresentaremos um histórico do surgimento da engenharia de *software* e a origem dos métodos ágeis. No capítulo 3 apresentaremos a Fundação Teórica, apresentando os modelos dos métodos clássicos, os métodos ágeis e uma comparação geral entre os métodos clássicos e ágeis. No capítulo 4 serão apresentados estudos de casos que estão divididos em cenário, definição do problema e análise e metodologia sugerida. Finalmente, no capítulo 5 veremos as conclusões do trabalho e sugestões de trabalhos futuros.



## 2 HISTÓRICO

No início da era do computador preocupavam-se muito com o *hardware*, precisavam construir máquinas menores e acessíveis a mais pessoas já que as máquinas eram muito grandes, *mainframes*, e de alto custo.

Devido à evolução dos *hardwares* como aquele processador que antes eram válvulas e agora são microprocessadores, junto a outras mudanças para que hoje tivéssemos um computador em cada escritório, em casa. O valor dessa máquina ficou mais acessível a todos.

No início os *softwares* eram desenvolvidos de maneira personalizada, fato que limitava sua distribuição e o alcance a todos os usuários. Devido a demanda criada pela popularização do computador houve um grande crescimento na criação de *software*, os programas antes personalizado, agora tinham que se adaptar a uma nova realidade, o *software* agora era algo muito importante e não havia até então programadores para atender a grande demanda e aqueles que haviam, usavam cada um o seu modo particular de programar, não se havia uma padronização ou um método específico causando assim alguns problemas como prazos não cumpridos, clientes pediam para construir um sistema que possuíam em mente e o programador desenvolvia outro sistema. Estes sistemas eram de difícil manutenção pois quando eram detectadas suas falhas somente que os programou eram capazes de corrigi-las.

A Figura 1 descreve a evolução do *software* dentro do contexto das áreas de aplicação de sistemas baseados em computador [17]. Nos primeiros anos estes *softwares* eram de distribuição limitada, já que eram feitos sob medida para cada cliente. Ainda não havia uma distribuição de *software* como vemos hoje, que se pode comprar um programa escolhido dentre muitos que atendam os objetivos do cliente. Era contruído um programa específico para um cliente e ali o colocava para funcionar, sem exigência de documentação ou formalidades.

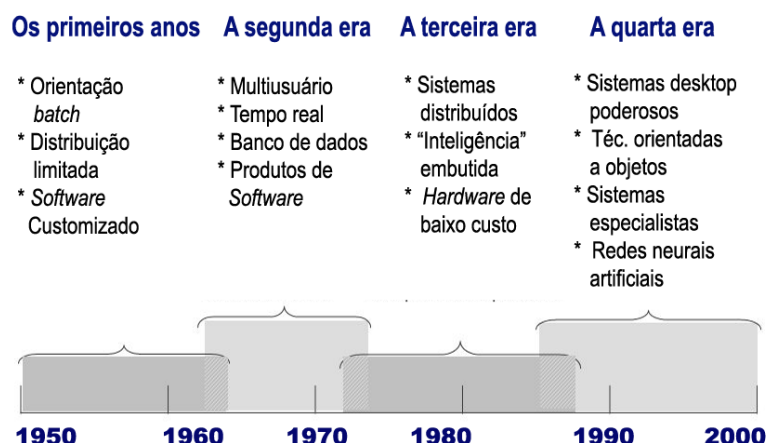


Figura 1: A evolução do *software* ao longo dos anos [17].

Na segunda era os *softwares* eram desenvolvidos para ampla distribuição sendo um único programa distribuído para muitos usuários. Esta interação proporcionou um avanço nos *softwares*. Sistemas em tempo real e banco de dados trouxeram agilidade e sistemas de gerenciamento de banco de dados. Os *softwares* começaram a chegar nas casas de seus usuários e cada vez mais utilizados para organização de empresas..

Na terceira era aumenta a comunicação entre computadores. O *hardware* já tem um preço mais acessível, e os *softwares* estão cada vez mais complexos e "inteligentes".

Na quarta era, é apresentado o que vemos nos dias atuais, uma infinidade de sistemas para infinitas necessidades. A inteligência artificial é realidade e deixou de ser uma ficção para ser colocada em prática. Nasce então a necessidade de criar metodologias e algumas padronizações, para criação de *softwares*.

## 2.1 O SURGIMENTO DA ENGENHARIA DE SOFTWARE.

Dado o crescimento no setor de *software* houve a necessidade de realizar algumas padronizações para que um programa pudesse ser manipulado por várias pessoas, numa empresa por exemplo, alguns passos padrões para melhor funcionamento ou para se obter um menor custo final do *software* a ser desenvolvido.

Existia então alguns problemas que precisavam ser resolvidos [17] :

1. A sofisticação do *software* ultrapassou nossa capacidade de construir um *software* que extraia o potencial do *hardware*.
2. Nossa capacidade de construir programas não pode acompanhar o ritmo da demanda de novos programas.
3. Nossa capacidade de manter os programas existentes é ameaçada por projetos ruins e recursos inadequados.

Surgia então a necessidade de trazer conceitos da engenharia para a construção de *software*. Foi então que em 1968, numa conferência que vários profissionais da área de construção de *software* utilizaram o termo Engenharia de *Software* [16]. Eles discutiram sobre metodologias, ferramentas, linguagens de programação e tudo mais a esse respeito. Fritz Bauer deu sua definição de Engenharia de *Software*:

“Engenharia de *software* é a criação e a utilização de sólidos princípios de engenharia a fim de obter *software* de maneira econômica, que seja confiável e que trabalhe eficientemente em máquinas reais”. [17]

Surgia então uma disciplina muito importante que abrange três etapas na construção de um *software*: métodos, ferramenta e procedimentos.



Figura 2: Camadas da Engenharia de *Software* [18].

A figura 2 demonstra que a engenharia de *software* é distribuída em camadas. Dentre todas as camadas a que se dá mais ênfase é o foco na qualidade. A qualidade final do produto, a satisfação do cliente. A camada de processo também é de grande importância já esta é a base da engenharia de *software*, é nela que é definida a metodologia a ser usada e se define prazos. Métodos é a camada onde se fornece as informações técnicas para desenvolver o programa, e por último, mas também relevante, a camada ferramentas que é a responsável por fornecer suporte para os métodos e processos.

## 2.2 ORIGEM DOS MÉTODOS ÁGEIS.

A construção de *software* utilizando métodos tradicionais não era capaz de atender de forma completa alguns programadores, pois seus métodos são muito documentacionais e possuíam pouca interação com o cliente. Muito das vezes só se descobria que o *software* não era o que o cliente planejou apenas no final, traduzindo-se em um alto custo para se adequar a verdadeira necessidade do cliente.

A construção de *software* precisa ser mais maleável, possuindo várias interações com os clientes. Pensando em tudo isso em 2001, Kent Beck e outros dezesseis renomados desenvolvedores, autores e consultores da área de *software* [18] assinaram o “Manifesto para o Desenvolvimento Ágil de *Software*” [10].

Esses desenvolvedores tinham seus conceitos e métodos na construção de *softwares* de qualidade que alcançavam a finalidade do sistema, atendiam bem ao cliente. Então eles começaram a listar o que era comum a todos eles na construção destes sistemas que geravam bons resultados.

Em base desta reunião [1] eles listaram as seguintes conclusões:

“Estamos descobrindo maneiras melhores de desenvolver *software* fazendo-o nós mesmos e ajudando outros a fazê-lo. Através deste trabalho, passamos a valorizar:

1. **Indivíduos e interação entre eles** mais que processos e ferramentas.
2. ***Software* em funcionamento** mais que documentação abrangente.
3. **Colaboração com o cliente** mais que negociação de contratos.
4. **Responder a mudanças** mais que seguir um plano.

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda [1]”.

O objetivo do método ágil não era descartar documentos ou métodos até então utilizados, mas sim, dar ênfase maior a indivíduos, ao *software* funcionando corretamente, ao contato maior com o cliente e saber que mudanças podem ser bem-vindas.

## 3 FUNDAMENTAÇÃO TEÓRICA

### 3.1 MÉTODOS TRADICIONAIS

A metodologia tradicional para a construção de *software* surgiu em uma época onde estas eram realizadas em terminais burros e *mainframes*. Michel [21] descreve como era este cenário:

Essas metodologias surgiram em um contexto de desenvolvimento de *software* muito diferente do atual, baseado apenas em um *mainframe* e terminais burros. Na época, o custo de fazer alterações e correções era muito alto, uma vez que o acesso aos computadores eram limitados e não existiam modernas ferramentas de apoio ao desenvolvimento do *software*, como depuradores e analisadores de código. Por isso o *software* era todo planejado e documentado antes de ser implementado [21].

Havia uma necessidade de se resolver o problemas na construção destes *software*, e a maneira encontrada foi planejar e documentar muito bem antes do início de seu desenvolvimento. Estes métodos tradicionais eram orientados a documentos bem detalhados e consequentemente muito extensos[17].

Os métodos tradicionais possuiu grande importancia no passado, mas hoje em dia é pouco utilizado devido o seu foco estar nas documentações, possuem a incumbência de colocar em ordem ao caos que se encontrava a área de *software*. Estes modelos trouxeram uma organização considerável às equipes que os construíam.

### 3.1.1 MODELO CASCATA (WATERFALL)

O modelo cascata, também conhecido como modelo clássico, sugere uma abordagem sequencial e sistemática para o desenvolvimento de *software*. [18]. Este modelo é muito bem sucedido nos casos onde os requisitos são bem compreendidos.

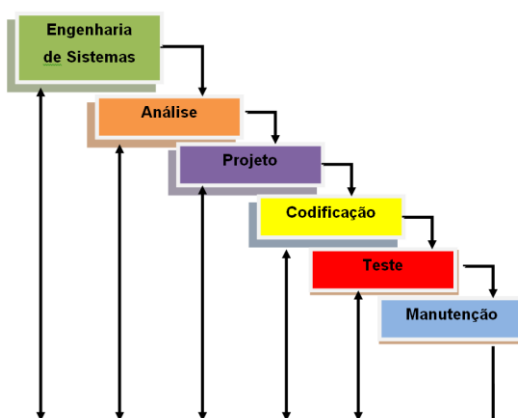


Figura 3: O ciclo de vida clássico. [17]

A figura 3 ilustra o ciclo de vida clássico da engenharia de *software* que descreve um método linear e sequencial [17]. Neste método a cada fase terminada, começa uma próxima etapa sem que haja retorno. Podemos descrever as etapas do ciclo clássico da seguinte forma:

- **Engenharia de Sistemas:** Esta etapa é de grande importância, pois é levantado requisitos para o sistema. É avaliada a interface com diversos elementos como *hardware*, pessoas e banco de dados.
- **Análise:** Nesta fase a engenharia de *software* coleta os requisitos do *software* em si, também se define as funções requeridas para este *software* comportamentos e interface exigidos. Nesta etapa os requisitos (do sistema e do *software*) são documentados e revisados pelo cliente.
- **Projeto:** Nesta etapa o projeto é na verdade um processo de vários passos que foca em quatro atributos distintos: estrutura de dados, arquitetura do *software*, caracterização de interface e detalhes de

procedimentos. O projeto, como os requisitos, é documentado e torna-se parte da configuração do *software*.

- **Codificação:** A codificação é a etapa onde o projeto deve ser traduzido em uma linguagem de programação.
- **Testes:** Finda a etapa de codificação é chegada à fase de testes, onde serão realizados testes para descobrir possíveis erros, avaliar se as entradas irão produzir os resultados esperados.
- **Manutenção:** Mudanças sempre ocorrem, e esta etapa se concentra nestas mudanças. Estas mudanças podem ocorrer por diversos motivos: erros foram encontrados, o *software se adaptará* a um novo sistema operacional, adaptações de funcionalidade existente ou o cliente faz novas exigências. A manutenção reaplica cada uma das etapas precedentes do ciclo de vida clássico.

O modelo cascata é o paradigma mais antigo da engenharia de *software* [18]. Ele tem a vantagem de nos permitir um controle gerencial. Suas fases são bem definidas e respeitadas. Mas há algumas desvantagens, como por exemplo, ao fim de uma etapa não existe possibilidade de retorna-la para uma correção. Pressman [18] relata alguns problemas encontrados quando se aplica ao modelo cascata:

- Os projetos reais raramente seguem o fluxo sequencial que o modelo propõe. Sempre existe a necessidade de fazer alguma iteração e estas podem trazer problemas no decorrer do projeto.
- É difícil para o cliente declarar todas as exigências explicitamente. O ciclo de vida clássico exige uma declaração completa das exigências do cliente, e isso se torna uma dificuldade para este método, já que é necessário se adequar a incerteza que sempre existe no início de um projeto.

### 3.1.2 MODELO INCREMENTAL

O modelo incremental surgiu na incumbência de tentar solucionar as desvantagens do modelo clássico. Foi observado que em várias situações não dava para utilizar um processo totalmente linear e poderia ser necessária, uma interação com os usuários para então prosseguir refinando e expandindo o projeto [18]. Neste caso, o método de desenvolvimento de sistema poderia então ser administrado numa série de incrementos.

Este modelo desenvolve o *software* incrementalmente permitindo assim, entender melhor e corrigir as versões anteriores de cada incremento.

O modelo incremental faz uma combinação entre os modelos lineares e paralelos. Na Figura 4, pode-se observar as sequencias lineares aplicadas de forma escalonada a medida que o tempo vai avançando [23]. Desse modo, pode-se a cada incremento, ser observado um ciclo de vida clássico.

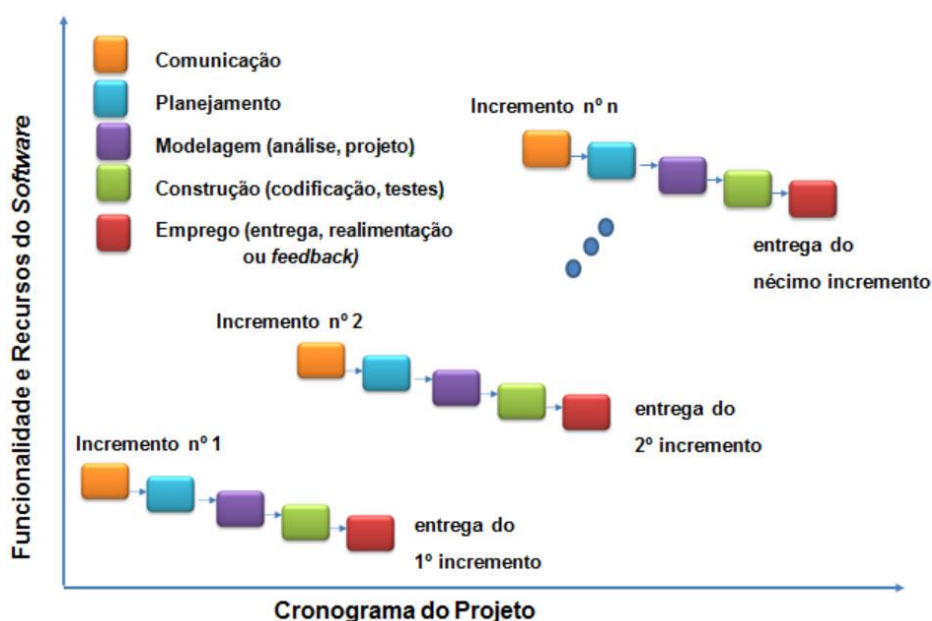


Figura 4: O modelo incremental [18].

Neste modelo o primeiro incremento é, em geral, denominado de núcleo do produto, isto se deve ao fato de serem definidos os requisitos básicos do projeto. A partir do incremento o cliente já pode avaliar se os requisitos por ele solicitados foram entendidos. Com base nesta avaliação é projetado um plano para o próximo



incremento já com as modificações, se existirem, solicitadas pelo cliente. Este processo é repetido a cada incremento até o fim do projeto.

Os primeiros incrementos servem como versões simplificadas do produto final, mas já podem ser utilizadas pelo usuário para avaliação[18].

O modelo incremental possibilita ao cliente identificar as funções a serem oferecidas pelo sistema de forma mais clara e simples. Desse modo o cliente identifica quais são as funções mais ou menos importantes para ele. Podendo então definir os estágios de entrega juntos com suas funcionalidades. Deste modo são entregues primeiramente as funções mais importantes para o cliente [23].

Medeiros [11] descreve que geralmente os primeiros incrementos podem ser implementados com um número pequenos de profissionais. Já nos incrementos posteriores estes profissionais poderão ser acrescidos de forma a implementar o incremento seguinte.

Existem algumas vantagens na utilização do método incremental, uma delas é que os requisitos básicos são atendidos no primeiro incremento com isso já na primeira avaliação poderá detectar problemas que culminaria em dimensões maiores se detectados na entrega final do *software*. Neste método existe a vantagem de o cliente esclarecer seus requisitos a cada incremento facilitando assim o trabalho do desenvolvedor do *software*.

### 3.1.3 MODELO PROTOTIPAÇÃO

O modelo de prototipação vem com o objetivo de esclarecer ao cliente e a equipe que constrói o *software* entenderem o projeto já que é comum o cliente definir diversos recursos que ele quer que faça parte de seu programa, mas, em muitos casos, a equipe que esta construindo o *software* não entende todos os requisitos solicitados pelo cliente, nesta situação a prototipação é uma ferramenta muito eficaz.

Bezerra [2] define a prototipagem como uma técnica que serve de complemento à análise de requisitos. O protótipo é um modelo de uma parte do sistema. Esta técnica tem o objetivo de assegurar que os requisitos dos sistemas foram completamente compreendidos.

Na prototipação o desenvolvedor constrói um modelo do *software* definido pelo cliente e através deste protótipo o cliente e o desenvolvedor podem eliminar suas dúvidas. O desenvolvimento rápido e iterativo é essencial para os custos sejam controlados e erros que custariam muito, se descoberto na fase final do projeto, sejam resolvidos o mais rápido possível.

A Figura 5 descreve as etapas do modelo de prototipação. No início do processo definido seus objetivos. A próxima etapa é definir o que terá neste projeto, o que será deixado de fora deste protótipo. Na etapa de construir um protótipo executável podem-se deixar de fora algumas funcionalidades par diminuição de custos. A etapa final é o processo de avaliar o protótipo e verificar se o projeto atingiu ou não o objetivo e quais alterações deverão ser feitas.

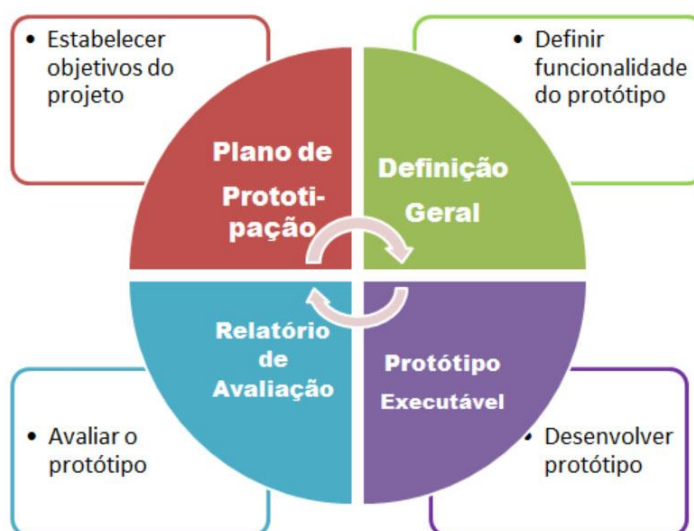


Figura 5: Processo de desenvolvimento de protótipo [23].

O modelo de prototipação é de grande importância quando se leva em consideração que muitas das vezes o cliente idealiza um projeto e o desenvolvedor entende outro projeto completamente diferente. Isso pode ser facilmente entendido na Figura 6, onde o cliente nem sempre sabe o que quer, e o protótipo vem esclarecendo as duas partes envolvidas [7].

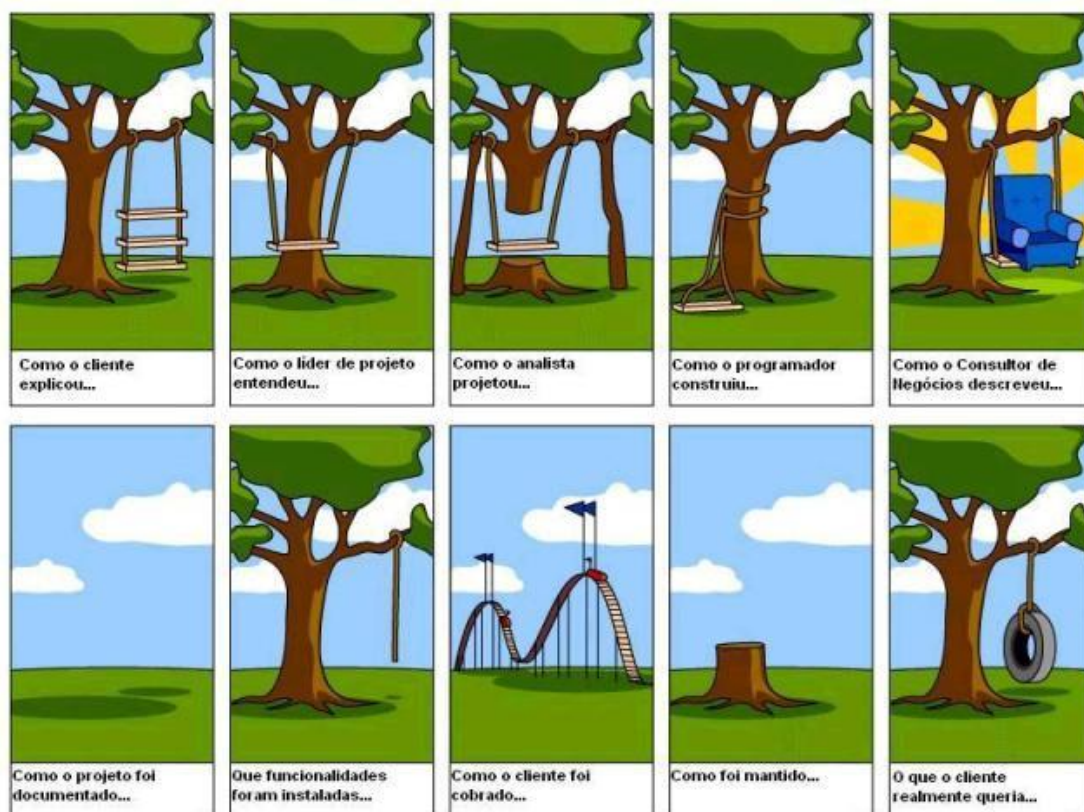


Figura 6: O exemplo das diversas formas de se entender um projeto [7].

Pode-se observar neste exemplo que a forma que o cliente explicou o projeto não era exatamente a forma que ele idealizou o mesmo. E isso trouxe vários outros problemas, pois uma vez não compreendido como o cliente deseja o projeto o líder entendeu de uma maneira, a analista de outra, ficando o projeto sem documentação, sem funcionalidade, sem manutenção e o cliente acaba sendo cobrado por um projeto diferente do idealizado por ele. E nesse contexto a prototipação vem acrescentar, pois, com ela há uma diminuição de erros de entendimento entre o cliente e a equipe que constrói seus projetos.

### 3.1.4 MODELO ESPIRAL

O modelo espiral é um processo de *software* evolucionário. O modelo espiral foi proposto por Boehm [23]. Este modelo acopla a natureza interativa da prototipação com o aspecto sistemático do modelo cascata [23].

Boehm [3] descreve este modelo da seguinte maneira:

“O modelo espiral de desenvolvimento é um gerador de modelos de processos dirigidos a riscos e é utilizado para guiar a engenharia de sistemas intensivos de *software*, que ocorre de forma concorrente e tem múltiplos envolvidos. Possui duas características principais o distinguem. A primeira consiste em uma abordagem cíclica voltada para ampliar, incrementalmente, o grau de definição e a implementação de um sistema, enquanto diminui o grau de risco do mesmo. A segunda característica consiste em uma série de pontos âncora de controle para assegurar o comprometimento de interessados quanto à busca de soluções de sistema que sejam mutualmente satisfatórias e praticáveis.”

A Figura 7 ilustra como este modelo se desenvolve. O processo de *software* é representado por uma espiral. O início da espiral começa com o Plano de Requisito/Plano do Ciclo de vida e a cada volta na espiral representa uma fase do processo de *software*. Neste modelo as atividades de manutenção são usadas para identificar problemas para que sejam resolvidos antes de evoluírem para a próxima volta, ou seja, cada *loop* na espiral representa uma fase do processo de *software*.

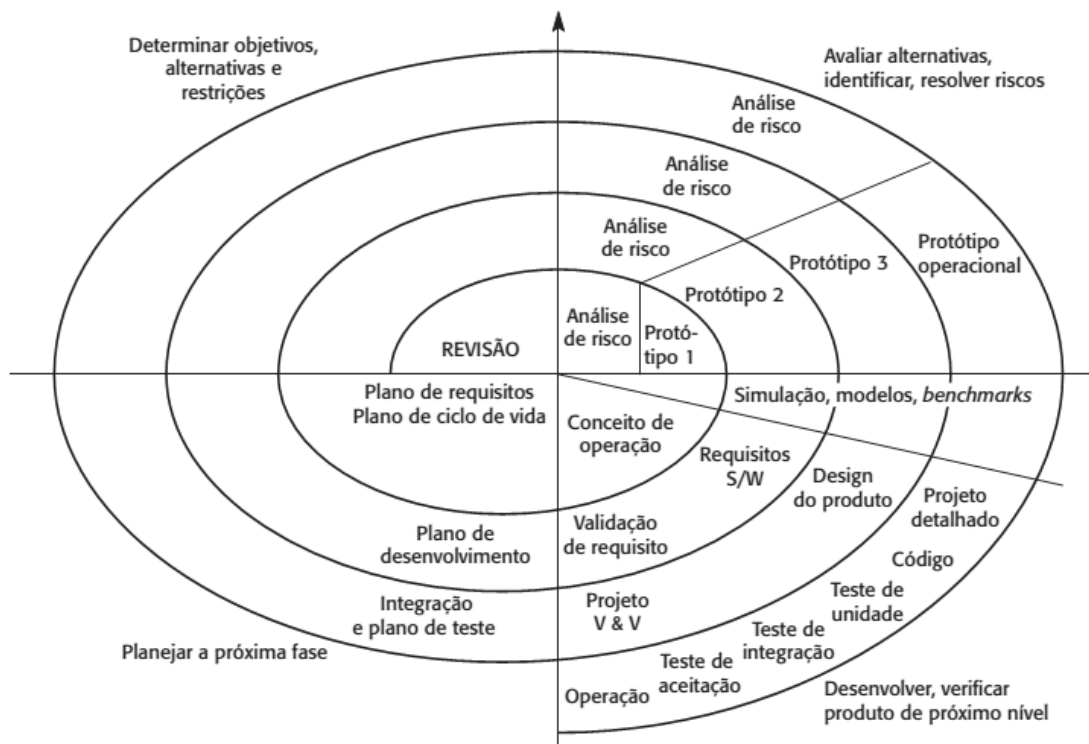


Figura 7: Modelo em espiral de processo de *software* [23].

Analisando a Figura 7 pode-se observar que a cada volta da espiral é dividida em quatro setores:

- **Definição de *software*:** Neste setor são definidos os objetivos para esta fase do projeto, também são definidas soluções alternativas específicas e restrições.
- **Avaliação e redução de riscos:** No segundo setor devem-se analisar os riscos das decisões do estágio anterior. Toda vez que passar por este setor pode ser construído protótipos.
- **Desenvolvimento e validação:** Agora já com as avaliações de risco, pode-se escolher um modelo para desenvolver o sistema. Neste contém as atividades da fase de desenvolvimento como o *design*, a especificação, a codificação e a verificação.
- **Planejamento:** No quarto e último, setor o projeto é feito a revisão das etapas anteriores e é tomada uma decisão de ir ou não para o próximo laço. Se a decisão tomada for positiva, será traçado o pla-

no para este novo laço. Caso a decisão for negativa serão feitas as devidas alterações e assim reavaliar seus novos riscos.

O modelo espiral tem uma técnica que facilita ao desenvolvedor e ao cliente entender e resolver problemas a cada etapa dos laços. Isso porque o modelo espiral consegue usar o que o modelo cascata e a prototipação tem de melhor, juntando a tudo isso a análise de risco.

### 3.1.5 MODELO TRANSFORMACIONAL

O modelo transformacional abrange um conjunto de atividades que levam a uma especificação matemática formal do *software* [18]. Neste modelo a especificação dos requisitos é formalmente detalhada matematicamente.

Durante o uso deste modelo são utilizados diversos mecanismos para eliminações de erros. Aplicações matemáticas precisas (não ambíguas) são utilizadas para resolver problemas de ambiguidade, inconclusão e inconsistência.

Os métodos formais são utilizados para sistemas que precisam dar prioridade a coesão. O modelo transformacional é desenvolvido a partir de princípios matemáticos garantindo exatidão em seus resultados. Este modelo auxilia todas as etapas de desenvolvimentos de *software* como especificação formal de requisitos, síntese para implementação, prototipagem e provas [13].

A Figura 8 ilustra o funcionamento do modelo transformacional. Neste modelo a especificação formal é refinada por meio de uma série de transformações, (T1, T2, T3 e T4). Estas transformações produzirão versões que serão analisadas até a versão final. Todas estas transformações embasadas em suas regras de transformações (R1, R2 e R3) e a cada etapa é gerado um programa equivalente (P1, P2, P3 e P4).

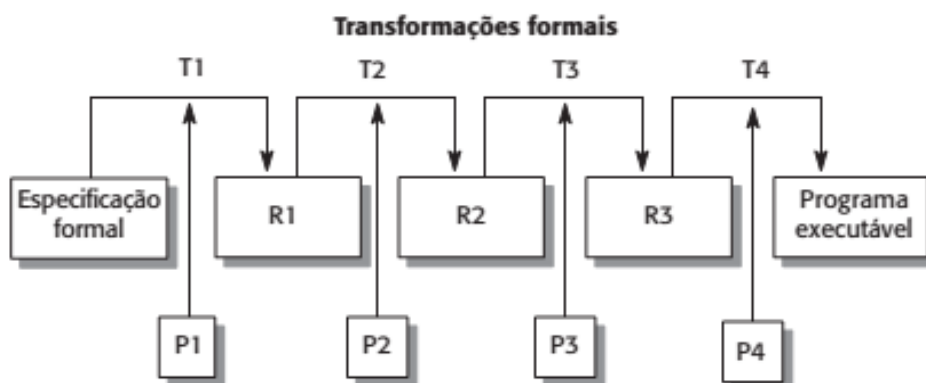


Figura 8: Modelo Transformacional [22].

O modelo transformacional tem um delineamento preciso do processo de desenvolvimento. Neste modelo o processo de desenvolvimento é incremental e é baseado em regras de transformações.

Estes métodos não ajudam a tomar as decisões referentes ao projeto, mas estabelecem métodos de escrever as decisões tomadas, métodos para colocar em prática as decisões tomadas.

### 3.1.6 ANÁLISE DOS MÉTODOS TRADICIONAIS.

Neste capítulo foram observados diversos modelos do método clássico. E a escolha do modelo certo para cada tipo de projeto pode garantir o sucesso de um projeto. Na escolha de um projeto devem-se considerar as características do mesmo. É necessário analisar quais as vantagens e desvantagens de cada modelo no projeto a desenvolver já que diferentes projetos possuem necessidades diferentes.

Para auxiliar a escolha do modelo mais adequado de um método para o desenvolvimento de um *software*, pode-se fazer uso de tabelas como estas ilustradas nas Tabelas 1, 2, 3, 4 e 5 que mostram as vantagens, desvantagens e aplicabilidade de cada modelo.

Tabela 1: Vantagens e Desvantagens do modelo Cascata.

CASCATA		
VANTAGENS	DESVANTAGENS	APLICABILIDADE
<ul style="list-style-type: none"> <li>• Ótimo controle gerencial;</li> <li>• Produz sistema de alta confiança;</li> <li>• Processo de desenvolvimento estruturado;</li> <li>• Cada fase cai em cascata sobre a próxima somente se terminada completamente;</li> <li>• Fases são bem definidas e respeitadas;</li> <li>• Ordem sequencial das fases.</li> </ul>	<ul style="list-style-type: none"> <li>• Não permite retornar a uma fase encerrada para uma possível alteração;</li> <li>• Não permite manutenção;</li> <li>• Exige uma declaração completa das exigências do cliente no início do projeto;</li> <li>• Caso haja um atraso, todo processo será afetado;</li> <li>• Entrega demorada do <i>software</i> para o cliente;</li> <li>• Não permite reutilização;</li> <li>• É excessivamente sincronizado;</li> <li>• Uma versão executável só estará disponível no final do projeto.</li> </ul>	<ul style="list-style-type: none"> <li>• Utilizados em projetos que seus requisitos sejam bem conhecidos;</li> <li>• Quando a probabilidade de os requisitos mudarem for pequena;</li> <li>• Utilizado na administração de etapas de projetos a serem documentalizados.</li> </ul>



Tabela 2: Vantagens e Desvantagens do modelo Incremental

INCREMENTAL		
VANTAGENS	DESVANTAGENS	APLICABILIDADE
<ul style="list-style-type: none"> <li>• Facilidade na manutenção;</li> <li>• Facilidade na identificação e correção de erros entre componentes;</li> <li>• Produz sistema de alta confiança;</li> <li>• Requisitos básicos são atendidos no primeiro incremento;</li> <li>• Clientes podem usar os primeiros protótipos para avaliação e obtenção de requisitos para os próximos estágios;</li> <li>• Menor possibilidade de falhas, devido ao grande número de testes;</li> <li>• Os incrementos podem ser desenvolvidos por um número menor de profissionais;</li> <li>• Os prazos para entrega dos incrementos são cumpridos.</li> </ul>	<ul style="list-style-type: none"> <li>• O número de interações não são definidos no início do projeto;</li> <li>• Manutenção complexa;</li> <li>• Dificilmente há possibilidade, de estipular data de entrega no início do projeto;</li> <li>• O produto final pode ficar mal estruturado já que a mudança contínua pode corromper a modularidade do sistema;</li> <li>• Muitas alterações na documentação do usuário; Difícil gerenciamento do custo.</li> </ul>	<ul style="list-style-type: none"> <li>• Utilizados em projetos de qualquer tamanho ou natureza;</li> <li>• Utilizar-se quando não há mão-de-obra disponível para uma implementação completa.</li> </ul>

Tabela 3: Vantagens e Desvantagens do modelo Prototipação

PROTOTIPAÇÃO		
VANTAGENS	DESVANTAGENS	APLICABILIDADE
<ul style="list-style-type: none"> <li>• Fáceis modificações em versões futuras;</li> <li>• Fácil definição de requisitos;</li> <li>• As incertezas e riscos do desenvolvimento são reduzidos;</li> <li>• Todas as funcionalidades são lembradas;</li> <li>• Podem ser utilizados posteriormente como documentação;</li> <li>• Agiliza validação do processo.</li> </ul>	<ul style="list-style-type: none"> <li>• Conscientizar o cliente que foi construído apenas um protótipo, e o produto será refeito.</li> </ul>	<ul style="list-style-type: none"> <li>• Utilizado para construções de interfaces gráficas;</li> <li>• Utilizado quando não se tem conhecimento pleno do sistema;</li> <li>• Quando existe dificuldade de se entender os requisitos do sistema.</li> </ul>

Tabela 4: Vantagens e Desvantagens do modelo Espiral

ESPIRAL		
VANTAGENS	DESVANTAGENS	APLICABILIDADE
<ul style="list-style-type: none"> <li>• A cada nova versão pode ser adicionada novas funcionalidades;</li> <li>• Produz sistema de alta confiança;</li> <li>• Ótimo gerenciamento de riscos;</li> <li>• Problemas importantes são descobertos no início, tornando o tempo estimável mais preciso;</li> <li>• Alto controle de riscos do projeto;</li> <li>• Possibilita ao desenvolvedor o uso de prototipação em qualquer etapa da evolução do produto;</li> <li>• Não faz distinção entre desenvolvimento e manutenção.</li> </ul>	<ul style="list-style-type: none"> <li>• Fazem-se necessários gerentes e técnicos experientes devido às avaliações e planejamentos baseados em risco;</li> <li>• Uso exclusivo para desenvolvimento de <i>software</i> interno;</li> <li>• Em geral, é difícil convencer aos clientes que a abordagem evolutiva pode ser controlável;</li> </ul>	<ul style="list-style-type: none"> <li>• Usado em grandes projetos;</li> <li>• É adequado para sistemas complexos com alto nível de interação com usuários.</li> </ul>

Tabela 5: Vantagens e Desvantagens do modelo Transformacional

TRANSFORMACIONAL		
VANTAGENS	DESVANTAGENS	APLICABILIDADE
<ul style="list-style-type: none"> <li>• <i>Software</i> livre de defeitos;</li> <li>• Garantia de qualidade, devido sua inspeção formal;</li> <li>• Ênfase em testes <i>do software</i>.</li> </ul> <p>Elimina problemas como ambiguidade, incompletude e inconsistência.</p>	<ul style="list-style-type: none"> <li>• Lento e dispendioso;</li> <li>• O desenvolvedor precisará de um treinamento extensivo;</li> </ul>	<ul style="list-style-type: none"> <li>• Na construção de <i>software</i> críticos em termos de segurança;</li> </ul>

O que ficou evidenciado neste capítulo é que cada modelo do método clássico possui sua importância. Este modelo só deverá ser usado quando tiver pleno conhecimento dos requisitos do cliente e do projeto em si. Ele é muito confiável, eficaz e ainda grandemente utilizado.

O modelo incremental por sua vez, tem as vantagens do método cascata e ainda pode ter várias sequências para avaliação da equipe que cria o projeto e para o cliente, enquanto o método clássico tem uma única sequência. Isso quando levado em conta um projeto grande, que, geralmente necessita de mudanças, o modelo incremental torna-se mais adequado.

Algumas vezes os desenvolvedores do projeto não conseguem entender bem os requisitos que o cliente deseja então o modelo de prototipação atende bem a necessidade de explorar estes requisitos e funcionalidades do sistema. A prototipagem vem para esclarecer ao desenvolvedor e o cliente para poder prosseguir no projeto.

Existem ainda projetos altamente complexos, onde a interação com o cliente, usuário é muito grande, para este tipo de projeto é válido a utilização do modelo espiral, pois neste modelo é utilizado as vantagens do método clássico mais as

vantagens do uso da prototipação mais com o diferencial de ter nele a análise de risco, o tornando ideal para projetos complexos.

Em fim, o modelo transformacional é bem recomendado em projetos críticos em segurança, em projetos onde uma falha do *software* acarreta um prejuízo alto para o cliente. Este modelo permite uma evolução do projeto, algo que o modelo cascata não permite e ainda tem a garantia de regras e notações matemáticas que o torna mais preciso.

### 3.2 MÉTODOS ÁGEIS

Os métodos ágeis foram construídos para se buscar maneiras melhores de desenvolver software [1]. Estes métodos são construídos tendo como base os 12 princípios do Manifesto Ágil [1] que são:

- Satisfazer o cliente, através da entrega adiantada e contínua de software de valor.
- Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas.
- Entregar *software* funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.
- Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto.
- Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.
- O método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.
- *Software* funcional é a medida primária de progresso.
- Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes.
- Contínua atenção à excelência técnica e bom design, aumenta a agilidade.

- Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.
- As melhores arquiteturas, requisitos e designs emergem de times organizáveis.
- Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo.

Vimos nestes princípios que sua prioridade está em satisfazer o cliente, entregar o *software* funcionando com qualidade e na data especificada. Agora mudanças são bem-vindas em todo o processo da construção de um *software*, pois mudanças tardias os custos são muito altos. Existe agora uma necessidade de entregar o *software* funcionando num menor tempo possível, o processo passa por diversas iterações de melhoria contínua.

Este processo necessita de uma constante comunicação entre os clientes e os desenvolvedores do *software* todos juntos em prol do projeto, motivando todos os envolvidos, fornecendo um ambiente e suporte necessário para o êxito do projeto. Os desenvolvedores devem se sentir confiantes com uma boa comunicação e *feedbacks* constantes, sempre tendo em mente a prioridade é satisfazer o cliente com prazos cumpridos e *software* funcionando, e isso é possível como uma boa comunicação entre as partes envolvidas no projeto.

Este modelo sugere maior sustentabilidade para todos os envolvidos no processo da construção do *software*. Um código quando bem construído elimina documentações exaustivas e trás uma facilidade para tomadas de decisões, assim a entrega de versões do projeto para avaliação do cliente se tornam mais construtivas. Sempre tendo em vista a simplicidade, focando sempre no que é importante, peneirando aquilo que realmente importa no projeto.

Para a construção de um excelente projeto é necessário um time de desenvolvedores auto-organizáveis que estejam preparados para se ajustarem conforme as mudanças ocorram já que estas sempre são bem-vindas ao projeto, por isso é muito importante intervalos regulares para que a equipe reflita o que fazer para se tornar cada vez mais eficaz em seus projetos.

### 3.2.1 EXTREME PROGRAMMING (XP)

A *Extreme Programming* (Programação Extrema) assume que a volatilidade dos requisitos existe, em vez de tentar eliminá-la, trata o desenvolvimento do *software* a partir de uma abordagem flexível e colaborativa, na qual desenvolvedores e clientes fazem parte de uma única equipe que tem o propósito de produzir *software* de alto valor agregado [14].

O primeiro projeto *Extreme Programming* foi iniciado em 6 de março de 1996 [5]. Este método é muito eficiente para qualquer tamanho de projeto. Ele consegue deixar o cliente extremamente satisfeito. Na Programação Extrema é incentivado o trabalho em equipe, todos em prol do sucesso do projeto. A Programação Extrema combina uma abordagem colaborativa com um conjunto de boas práticas de engenharia de *software*. A XP é definida através de um conjunto de cinco (5) princípios:

**Comunicação:** A XP enfatiza a comunicação informal, uma comunicação mais pessoal com os clientes e a equipe de desenvolvedores para um *feedback* contínuo evitando assim uma documentação extensa.

**Simplicidade:** É incentivado praticas que reduzam a complexidade do sistema. Os desenvolvedores são incentivados a projetar para necessidades imediatas e então depois serem implementados em código. *Design*, processos e códigos poderão ser modificados a qualquer momento.

**Feedback:** Através do cliente, dos desenvolvedores e do próprio *software* se obtém a realimentação para prosseguir com o projeto. Esses *feedbacks* permite uma maior agilidade, pois os requisitos são reavaliados constantemente e erros são detectados e corrigidos imediatamente.

**Coragem:** Devido aos diversos testes, uma ótima integração entres todos que estão ligados ao projeto e a uma programação em pares os programadores se sentem mais encorajados investir no projeto. O programador tem que ter em mente que precisa programar para hoje, tendo que ter assim disciplina, investindo em tes-

tes, melhoramento de códigos funcionais para torna-los mais simples, mexer no *design* procurando melhorá-lo.

**Respeito:** A equipe de desenvolvedores deve estar comprometida com o respeito entre todos que estejam ligados ao projeto e até mesmo com o próprio *software*.

A XP utiliza, geralmente, uma abordagem orientada a objeto como paradigma de desenvolvimento.



Figura 9: Processo da Extreme Programming (XP) [18].

Analisando a Figura 9 pode-se observar que um processo XP é dividido em 4 etapas:

**Planejamento:** Um projeto XP tem como primeiro passo ouvir o cliente para entender o que o mesmo deseja em seu projeto. Ouvindo faz-se o levantamento de requisitos e os desenvolvedores tem um amplo conhecimento do que o cliente já que a XP tem práticas que dirigem e guiam para uma melhor comunicação. Cada história ouvida é colocada em uma ficha onde o cliente atribui uma prioridade, assim os desenvolvedores atribuem a esta história custos onde são medidos o tempo gasto na mesma em semanas. Uma história muito extensa, maior que três semanas, pode



ser dividida. A XP é baseada em comunicação e prioriza a comunicação verbal de qualidade e uma menor importância a documentação formal.

**Projeto:** O projeto XP atende o princípio da simplicidade. A XP aceita a evolução natural do sistema, e isso implica em mudanças constante. Ele também utiliza o uso de Cartões CRC (Classes, Responsabilidades e Colaboração) que identificam e organizam as classes orientadas a objeto. Às vezes quando um problema é de difícil solução é utilizado a Re-fabricar (*refactor*) onde é projetado um protótipo para melhor avaliação, e com os dados obtidos se aprimora a estrutura interna do *software*.

**Codificação:** Nesta etapa escreve-se o código que é refinado através de algumas práticas. Primeiro se desenvolve testes de unidade, a programação é feita em pares, e essa dupla trabalham juntos numa história para criar seu código, juntos eles podem desenvolver aquela história ouvida do cliente, agora transformada em um código. Este código desenvolvido por esta dupla é integrado ao trabalho de outros da equipe, pois o código é propriedade coletiva.

**Testes:** A XP entende que quanto mais cedo pequenos problemas forem resolvidos menos custoso será o projeto. O teste é um passo integrado no processo de desenvolvimento. Estes testes podem ser feitos diariamente. Os desenvolvedores escrevem os testes antes de desenvolverem os códigos. Cada unidade deve ser testada antes de liberada.

### 3.2.2 SCRUM

*Scrum* é uma metodologia ágil para gerenciamento de projetos de *software*. Foi criada por Jeff Sutherland, Ken Schwaber e John Scumniotales na década de 1990, baseada no Pensamento Lean (*Lean Thinking*), desenvolvimento iterativo e incremental, e novas estratégias de criação de produtos [19].

A metodologia *Scrum* segue os princípios do manifesto ágil para a construção de um projeto. Um processo, no *Scrum*, tem as seguintes atividades estruturais: requisitos, análise, projeto, evolução e entrega.

Analisando a Figura 10 podemos observar que a cada atividade, num processo *Scrum*, ocorrem algumas tarefas a serem cumpridas e são eficazes até mesmo para quando se tem um prazo de entrega apertado e quando seus requisitos precisam ser mudados durante o processo. A seguir podemos analisar cada etapa deste método.

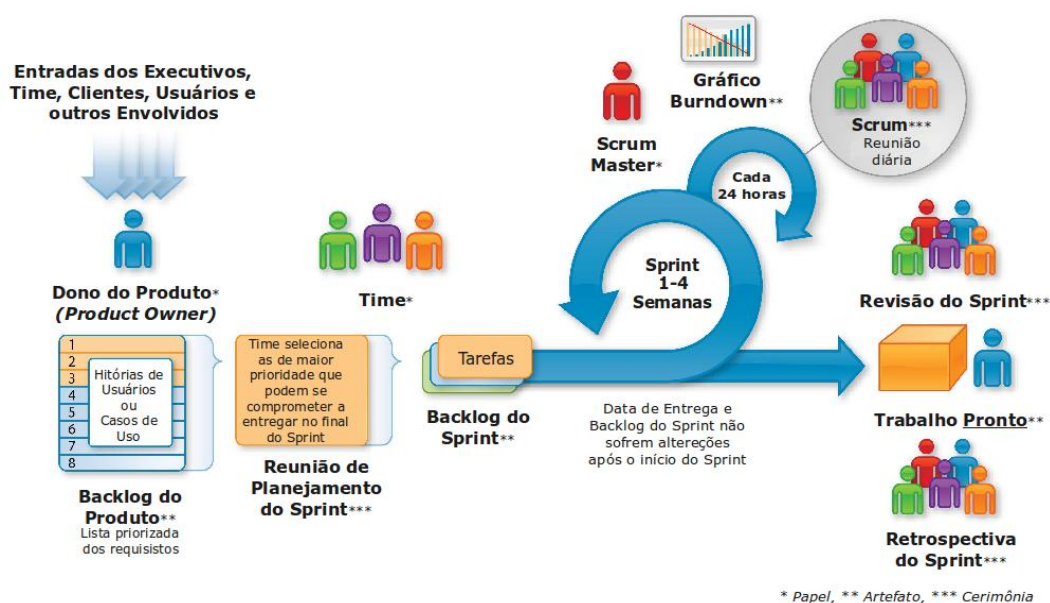


Figura 10: Fluxo de um Processo Scrum [4].

**Product Backlog:** Inicialmente define-se uma lista onde se formaliza tudo que se pretende construir no projeto inclusive e requisitos funcionais e não funcionais. Nesta lista contém as funcionalidades, prioridades desejadas para o projeto. A ela pode ser adicionado novos itens a qualquer momento ou até mesmo eliminados, os itens serão avaliados pelo gerente do projeto.

**Product Owner:** É o dono do projeto. Ele prioriza os *Backlog* da lista de funcionalidades. Ele é responsável por definir a ordem dos requisitos que serão produzidos pela equipe.

**Sprint Backlog:** É uma tarefa, um conjunto de funcionalidade separadas pelo *scrum master* julgadas ser possível de ser entregue num ciclo de 30 dias.

**Scrum Master:** É o líder da equipe. Ele é responsável por conduzir reuniões, como *standup meeting*, avaliar respostas dos demais membros da equipe e dividir tarefas. Ele garante que o processo do *Scrum* estejam sendo conforme as regras e mantém a equipe sempre no foco do projeto.

**Sprint:** É um ciclo que pode durar de 1 a 4 semanas (até 30 dias).

**Standup Meeting:** Reunião em pé. São realizadas diariamente pelo *Scrum Master*, geralmente pela manhã com duração de 15 minutos, onde ajuda a equipe a revelar problemas potenciais o mais cedo possível e também ajuda a equipe unificar pensamentos mantendo a equipe auto-organizada. Nesta reunião cada membro da equipe precisa responder três perguntas:

1. O que você realizou desde a última reunião?
2. Você está tendo alguma dificuldade?
3. O que fará para o projeto até a próxima reunião?

**Burndown Chart:** É um painel de progresso onde é demonstrado as funcionalidades que foram obtidas até um certo momento da *Sprint*. Este gráfico de consumo ajuda a acompanhar o progresso e a velocidade da equipe de desenvolvimento.

**Revisão do Sprint:** É uma reunião onde será apresentado ao *Product Owner* o produto funcionando para que ele avalie se o produto está de acordo com o que foi definido no início do projeto.

**Retrospectiva do Sprint:** No final de cada *Sprint* é realizada uma reunião onde analisado tudo o que foi feito na *Sprint* atual, pontuando aquilo que não foi bom, para que se possa melhorar, e o que foi bom para então manter na próxima *Sprint*.

### 3.2.3 *FEATURE DRIVEN DEVELOPMENT* (FDD)

O desenvolvimento dirigido à funcionalidade foi criado originalmente por Peter Coad (1980/1990) para servir de modelo prático em Engenharia de *Software* orientada a objeto [18]. Este modelo foi aperfeiçoado por Stephen Palmer e John Felsing entre os anos de 1997 e 1999.

O FDD tem como um de seus princípios a colaboração entre todos os integrantes da equipe, busca o desenvolvimento por um requisito funcional do sistema, ou seja, baseado por funcionalidade. Este método conquista os clientes, gerentes e desenvolvedores com sua abordagem prática e eficaz. Como em outros métodos ágil uma comunicação de detalhes técnicos usando meio verbais, gráfico e textos. É uma metodologia muito prática para projetos iniciais ou mesmo com códigos já existentes.

O lema da FDD é: “Resultados frequentes, tangíveis e funcionais” [6]. No FDD a funcionalidade “é uma função valorizada pelo cliente passível de ser implementada em duas semanas ou menos” [18]. Analisando a Figura 11, podemos observar que a metodologia FDD é muito objetiva. Possui duas fases [5]:

- **Concepção e Planejamento:** Pensar um pouco antes de fazer (tipicamente de 1 a 2 semanas)
- **Construção:** Fazer de forma interativa (tipicamente em interações de 2 semanas)

O FDD possui cinco processos básicos e bem definidos[8]:

- **Desenvolver Modelo Abrangente (DMA):** Análise orientada por objetos – É a atividade inicial realizada por membros do domínio do negócio e por desenvolvedores, todos sob a orientação de um modelado de objeto. Tem o objetivo de atribuir mais uma forma do que um conteúdo.
- **Construir a Lista de *Features* (CLF):** Decomposição funcional - O objetivo dessa etapa é identificar as funcionalidades agrupadas em conjunto que satisfação aos requisitos.

- **Planejar por *Feature* (PPF):** Planejamento incremental – O objetivo desta etapa é construir um plano de desenvolvimento proprietários de conjuntos de funcionalidades. Essas atividades são consideradas em conjunto após serem feitos diversos refinamentos.
- **Detalhar por *Feature* (DPF):** Desenho orientado a objetos – Nesta etapa é produzido o pacote de projeto (*design*). É desenvolvido um plano de proprietário de classes e conjuntos de funcionalidades.
- **Construir por *Feature* (CPF):** Programação e teste orientado a objetos– Produz uma função com valor para o cliente. Agora que DPF foi completado e aprovado pode-se então, implementar os itens necessário para satisfazer os requisitos para cada funcionalidade.

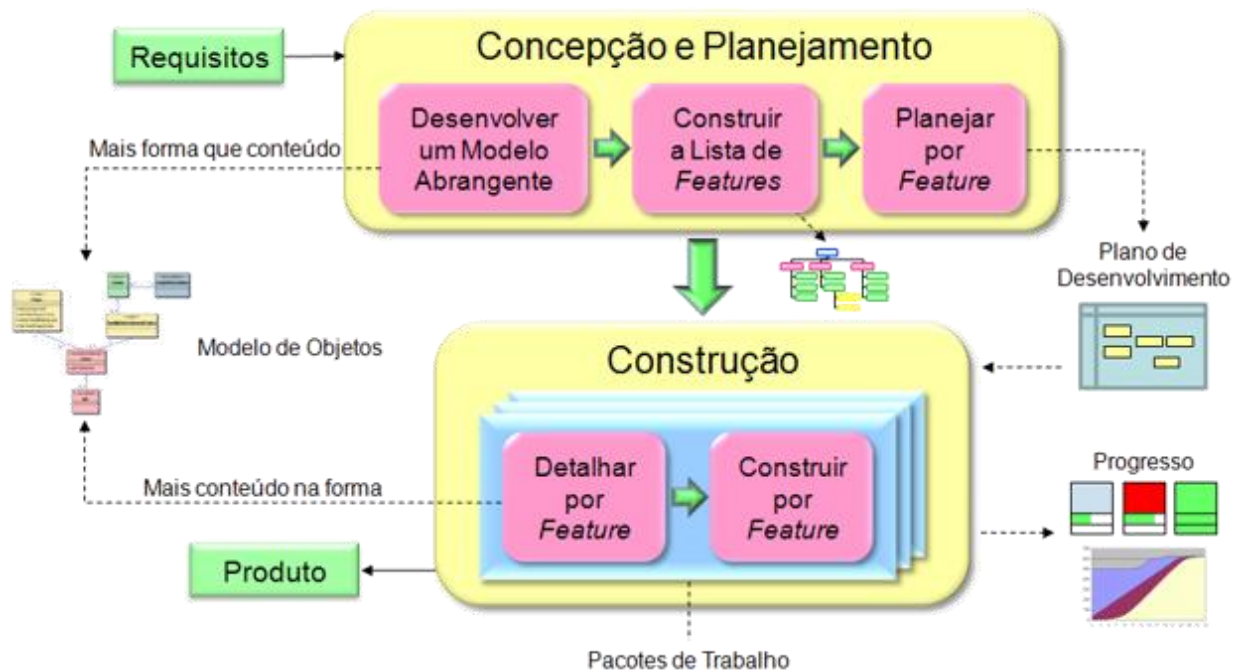


Figura 11: Estrutura FDD [5].

### 3.2.4 CRYSTAL

O método *Crystal* é uma família de metodologia criada por Alistair Cockburn, em meados da década 1990. Esta metodologia é útil em projetos para uma equipe pequena com baixa criticidade ou até mesmo em projetos com grandes equipes com alta criticidade [15]. Esta metodologia se adequa o tamanho da equipe e sua criticidade. Em cada novo projeto é realizado uma análise onde se verifica o número de pessoas na equipe de desenvolvimento, a criticidade do sistema e a prioridade do projeto.

Esta metodologia utiliza cores diferentes para indicar o “peso” do projeto e a melhor metodologia a ser utilizada. Analisando a Figura 12, cada método é moldado para ter a quantidade suficiente para o processo. No *Crystal*, cada projeto é realizado seguindo três fatores: Número de Pessoas Envolvidas, Criticidade e Prioridade do Projeto. Os métodos *Crystal* tem ênfase nos talentos e habilidade das pessoas.

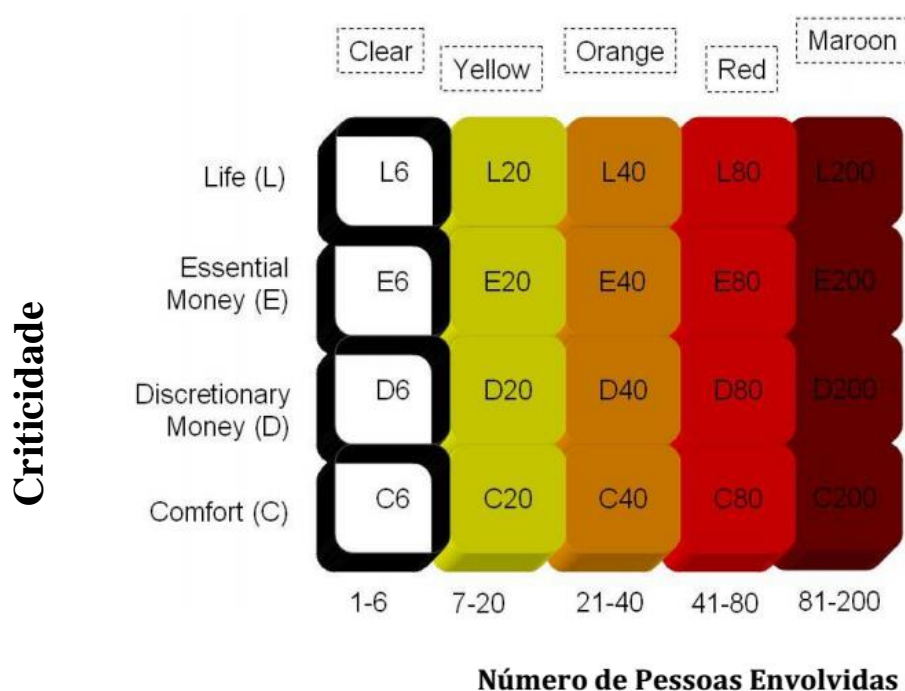


Figura 12: A Distribuição do Método *Crystal* [15]

A família de metodologia *Crystal* é caracterizado por uma cor conforme o Número de Pessoas Envolvidas:

**Crystal Clear:** Metodologia leve, equipes de 1 a 6 pessoas;

**Crystal Yellow:** Para equipes de 7 a 20 membros;

**Crystal Orange:** Para equipes de 21 a 40 pessoas;

**Crystal Red:** Para equipes de 41 a 80 pessoas;

**Crystal Maroon:** Para equipes de 81 a 200 pessoas.

Cada um dos métodos com graus de gerenciamento e de comunicação ajustados de acordo com o tamanho da equipe [12].

Neste modelo os métodos são concentrados nas pessoas e na comunicação, mudanças são sempre bem vindas e tem duas regras principais: o desenvolvimento deve ser incremental, que durem até quatro meses mas prefere-se que dure de um a três meses, e deve possuir “*workshops* de reflexão” antes de depois de cada incremento. É muito importante que os participantes de uma equipe *Crystal* sejam concentrados em sua tarefa individualmente dentro do projeto e também sejam concentrados no projeto como todo. É muito importante a integração total, pois qualquer alteração que seja necessário fazer o projeto todos da equipe saberá.

O método *Crystal* possui princípios, são os seguintes [20]:

- Comunicação iterativa, face-a-face é o canal mais barato e rápido para o intercâmbio de informações.
- O excesso de peso do método é custoso.
- Times maiores precisam de métodos mais pesados.
- A maior cerimônia é apropriada para projetos com maior criticidade.
- Aumentando a retro-alimentação (feedback) e comunicação é reduzida a necessidade de itens intermediários entregues.
- Disciplina, habilidade e entendimento contra processo, formalidade e documentação.
- Pode-se perder eficiência em atividades que não são o gargalo do processo.

### 3.2.5 ANÁLISE DAS METODOLOGIAS ÁGEIS

Neste capítulo foram observados diversos modelos do método ágil. Observa-se que como nos métodos clássico, há diversos modelos e a escolha do modelo certo para cada tipo de projeto pode garantir o sucesso de um projeto. Apesar de todos os modelos utilizarem o manifesto ágil [10] cada modelo tem sua particularidade, prática, vantagens e desvantagens que serão analisadas pelo responsável do projeto então adotado a que melhor se adeque ao projeto, considerando as habilidades da equipe de desenvolvimento.

O XP tem uma abordagem deliberada e disciplinada, ele integra um conjunto de boas práticas de programação, melhorias do projeto e participação do cliente com a equipe de desenvolvimento no projeto. O FDD já possui uma metodologia centrada na modelagem e utilizam as funções de valor do cliente que permite agradar tanto ao cliente como a equipe de desenvolvimento. O método *Scrum* utiliza o desenvolvimento iterativo e incremental, ele fornece um *framework* de gerenciamento de projeto, enquanto o modelo *Crystal* utiliza uma família de metodologias que varia de acordo com a necessidade do projeto, este método não se destina a empresas utilizam padronização já que o *Crystal* exige mudanças e adaptações constantes. Fábio Levy [20] relata que este modelo pode ser facilmente utilizado por uma equipe que utilize o XP mas o resultado final do XP é melhor, por isso algumas equipes inicializam o projeto com o *Crystal* e posteriormente, finalizam com o XP ou até mesmo equipes que não tenham se adaptado ao XP podem passar usar o *Crystal*.



Para auxiliar a escolha do modelo ágil mais adequado a um projeto de desenvolvimento de um *software*, pode-se fazer uso de tabelas como estas ilustradas, na Tabela 6, que mostram as características cada modelo e a Tabela 7 que mostra os Pontos Positivos e Negativos de cada modelo:



Tabela 6: Características dos Métodos Ágeis

Características	<b>XP</b>	<b>SCRUM</b>	<b>FDD</b>	<b>CRYSTAL</b>
<b>Equipes</b>	Pequenas	Qualquer Tamanho	Médias/ Grandes	Qualquer Tamanho
<b>Ênfase</b>	Codificação/manutenção do código	Interações entre os indivíduos	Arquitetura / modelagem	Comunicação e cooperação entre membros da equipe
<b>Abordagem</b>	Incrementos Iterativos	Incrementos Iterativos	Interativo	Incremental
<b>Documentação</b>	Básica	Básica	Grande Importância	Básica
<b>Requisitos</b>	Equipe técnica e cliente definem a <i>user stories</i> .	Definições do <i>Sprint Backlog</i>	Características agrupadas, priorizadas e distribuídas.	
<b>Testes</b>	Teste Incremental a partir de cenários	Revisão de <i>Sprints</i>	Construir por <i>Feature</i> (CPF)	
<b>Interação (tempo)</b>	1 a 6 semanas	1 a 4 semanas	2 dias a 2 semanas	Máximo de 4 meses
<b>Especialidade</b>	TDD, <i>refactoring</i> , <i>user stories</i>	<i>Sprint</i> , <i>product backlog</i> , <i>Planning poker</i> , <i>scrum master</i>	UML Diagramas	Métodos adaptativos
<b>Entrega Final</b>	Cliente satisfeito com projeto	Todos os itens no <i>Product Backlog</i> desenvolvidos	Após todos os conjuntos de características implementados	

Tabela 7: Pontos Positivos e Negativos dos Métodos Ágeis.

Métodos Ágeis	 <b>Pontos Positivos</b>	<b>Pontos Negativos</b> 
<i>XP</i>	<ul style="list-style-type: none"> <li>• Método Ágil possui maior popularidade;</li> <li>• Processo ágil e flexível;</li> <li>• <i>Feedback</i> constantes;</li> <li>• Ganho de eficiência e eficácia no desenvolvimento de sistemas;</li> <li>• Clientes determinam as prioridades;</li> </ul>	<ul style="list-style-type: none"> <li>• Documentação baseada apenas no código e na comunicação verbal;</li> <li>• Não tem uma avaliação de riscos;</li> <li>• Análise de risco informal;</li> </ul>
<i>SCRUM</i>	<ul style="list-style-type: none"> <li>• Times auto-organizados e com <i>feedback</i>;</li> <li>• Participação direta do cliente;</li> <li>• Prioridades definidas pelo cliente;</li> <li>• Valorização do indivíduo;</li> <li>• Equipe comprometida;</li> <li>• Melhor visualização do projeto</li> </ul>	<ul style="list-style-type: none"> <li>• Não especifica técnicas prática;</li> <li>• Suporte apenas para o gerenciamento do projeto.</li> </ul>
<i>FDD</i>	<ul style="list-style-type: none"> <li>• Aplicado em vários tipos de projeto;</li> <li>• Foco em “características de valor para o cliente”;</li> <li>• Vários Times podem trabalhar em paralelo;</li> <li>• Escalável para equipes e grandes projetos;</li> <li>• Fácil modelagem e desenvolvimento por funcionalidade.</li> </ul>	<ul style="list-style-type: none"> <li>• O desenvolvedor de um código torna-se “dono” dele;</li> <li>• Difícil manutenção;</li> <li>• Indefinição sobre o tamanho mínimo de uma equipe FDD.</li> </ul>
<i>Crystal</i>	<ul style="list-style-type: none"> <li>• Possibilita adaptações no projeto;</li> <li>• Metodologia escalável quanto ao tamanho e criticidade;</li> <li>• Menos especulações e mais visibilidade de tarefas.</li> </ul>	<ul style="list-style-type: none"> <li>• Não funciona bem em projetos longos;</li> <li>• Muita documentação, em projetos pequenos não funciona bem.</li> </ul>

### 3.3 ANÁLISE DOS MÉTODOS CLÁSSICOS E MÉTODOS ÁGEIS

Na construção de um *software* algumas decisões precisam ser tomadas, uma delas é a escolha de um método para o desenvolvimento deste projeto. E então surge o dilema: “Qual o melhor método a ser utilizado o Método Clássicos ou o Método Ágil? ” Para responder esta pergunta precisamos avaliar alguns pontos que diferenciam estas metodologias.

Os métodos clássicos são muito utilizados ainda hoje quando adequados à necessidade do cliente. Neste método o cliente só irá avaliar o projeto quando ele se encontra 100% cumprido. Enquanto que nos métodos ágeis este projeto pode ser avaliado progressivamente, basta ter um conjunto de funcionalidades pronto, para então o projeto ser avaliado pelo cliente e a equipe de desenvolvedores.

Será que isso torna o método ágil melhor? A resposta para essa pergunta seria “nem sempre”. Pois o importante não são somente as avaliações intermediárias ou finais, mas se a qualidade total do projeto foi excelente. Isso porque o método clássico também tem suas qualidades, vimos isso no que tange ao custo do produto, no método clássico é possível saber o preço total do projeto no início do projeto, isso porque o valor é combinado inicialmente, enquanto que no método ágil o preço não fica bem definido no início, já que geralmente, são feitas diversas alterações das funcionalidades do projeto, podendo levar a um acréscimo no valor final do projeto. Então esse ponto tornaria o método clássico melhor? “Nem sempre”, o que vai responder esta pergunta é a necessidade do cliente e do projeto.

Na Tabela 8 tem-se um comparativo geral da abordagem clássica e ágil onde podemos observar algumas características destas metodologias para ajudar na decisão do método certo para determinado projeto.

Tabela 8: Comparativo Geral da Abordagem Clássico com a Ágil [Adaptado, 9].

Abordagem Clássico	Abordagem Ágil
<b>Preditivo:</b> Detalhar o que ainda não é bem conhecido	<b>Adaptativo:</b> Conhecer problema e resolver o crítico primeiro
<b>Rígido:</b> Seguir especificação predefinida, a qualquer custo	<b>Flexível:</b> Adaptar-se a requisitos atuais, que podem mudar
<b>Burocrático:</b> Controlar sempre, para alcançar objetivo planejado	<b>Simplista:</b> Fazer algo simples de imediato e alterar se necessário
<b>Orientado a processos:</b> Segui-los possibilita garantir a qualidade	<b>Orientado a pessoas:</b> Motivadas, comprometidas e produtivas
<b>Documentação</b> gera confiança	<b>Comunicação</b> gera confiança
Sucesso = <b>entregar o planejado</b>	Sucesso = <b>entregar o desejado</b>
Gerência = “ <b>comando-e-controle</b> ” voltado para trabalho em massa, ênfase: papel do gerente, com planejamento e disciplina fortes	Gerência = <b>liderança/orientação</b> trabalhadores do conhecimento, ênfase: criatividade, flexibilidade e atenção às pessoas
<b>Desenvolvedor</b> hábil (variedade)	<b>Desenvolvedor</b> ágil (colaborador)
<b>Cliente</b> pouco envolvido	<b>Cliente</b> comprometido (autonomia)
<b>Requisitos</b> conhecidos, estáveis	<b>Requisitos</b> emergentes, mutáveis
<b>Retrabalho/reestruturação</b> caro	<b>Retrabalho/reestruturação</b> barata
<b>Planejamento</b> direciona os resultados (incentiva controlar)	<b>Resultados</b> direcionam o planejamento (incentiva mudar)
<b>Conjunto de processos</b> , com metodologia definida	<b>Conjunto de valores</b> , com atitudes e princípios definidos
<b>Premia</b> a garantia da qualidade	<b>Premia</b> o valor rápido obtido
<b>Foco:</b> Grandes projetos ou os com restrições de confiabilidade, planej. estratégico / priorização (exigem mais formalismo)	<b>Foco:</b> Projetos de natureza exploratória e inovadores, com equipes pequenas/médias (exigem maior adaptação)
<b>Objetivo:</b> Controlar, em busca de alcançar o objetivo planejado (tempo, orçamento, escopo)	<b>Objetivo:</b> Simplificar o processo de desenvolvimento, minimizando e dinamizando tarefas e artefatos

Para se escolher então o método certo a utilizar temos que analisar a necessidade do cliente. Se a exigência deste cliente é definir totalmente o projeto desde o início, vale apenas pensar nos métodos clássicos já que neste método é necessário que os requisitos sejam totalmente esclarecidos, que sua documentação esteja bem detalhada, e que tudo que se precisa saber para realizar o projeto esteja bem definido, tendo então um preço fechado do projeto para o cliente.



No entanto, quando a necessidade do cliente seja um projeto que ele possa avaliar mais de perto, informando suas principais funcionalidades no início, para então ir avaliando e acrescentando de forma incremental, onde o preço fechado no início não tenha tanta importância para o cliente, pode-se então utilizar os métodos ágeis. Precisamos ter em vista também que é necessário avaliar a cultura da empresa que está realizando o projeto. Algumas realmente preferem um método ao outro.

Os métodos ágeis têm seu caráter adaptativo e orientado a pessoas, isso é muito útil em uma empresa cujo número da equipe é pequeno, onde se fazem necessárias a flexibilidade e adaptações.

Vale apenas deixar claro que algumas empresas costumam utilizar os dois métodos, trabalhando lado a lado, buscando os pontos fortes de cada método ambos num só projeto, de acordo com a necessidade do cliente e do projeto como um todo.

Analisando a Tabela 9 podemos analisar os Pontos Positivos e Negativos dos Métodos Ágeis e Tradicionais, para que possa facilitar na escolha do método escolhido para o desenvolvimento de um possível projeto.

Tabela 9: Pontos Positivos e Negativos dos Métodos Tradicionais e Ágeis.

Métodos	 <b>Pontos Positivos</b>	<b>Pontos Negativos</b> 
<i>Tradicional</i>	<ul style="list-style-type: none"> <li>• Entrega do produto em sua totalidade;</li> <li>• Planejamento intenso, para eliminar riscos;</li> <li>• Preço total geralmente mais claro para o cliente;</li> <li>• Monitoramento de custos;</li> <li>• Análise de risco constante;</li> </ul>	<ul style="list-style-type: none"> <li>• Planejamento com pouca flexibilidade;</li> <li>• Documentação extensa;</li> <li>• Foco nos processos;</li> <li>• Uma nova funcionalidade no final do projeto tende a custar caro;</li> <li>• Processo centralizado no gerente do projeto;</li> </ul>
<i>Ágil</i>	<ul style="list-style-type: none"> <li>• Planejamento com alta flexibilidade;</li> <li>• Foco nas pessoas</li> <li>• Melhor visualização do projeto, menos especulações;</li> <li>• Integração com o cliente;</li> <li>• Reduz o tempo de entrega da primeira versão;</li> <li>• Equipe de desenvolvedores unidos em prol do projeto;</li> <li>• <i>Feedbacks</i>;</li> </ul>	<ul style="list-style-type: none"> <li>• Menos controle de custos e lucro;</li> <li>• Menos escalares;</li> <li>• Difícil em equipes grandes;</li> <li>• Insegurança por parte dos clientes;</li> </ul>

## 4 EXEMPLOS E SUGESTÕES DE ESCOLHA DE METODOLOGIA

Neste capítulo serão descritos casos de situações hipotéticas em que se necessita desenvolver um *software*, e para cada cenário é eleita uma metodologia sugerida que melhor se acomoda ao caso.

### 4.1 EXEMPLO 01 – INFORMATIZAÇÃO DO ESTOQUE DE UMA PADARIA.

#### 4.1.1 CENÁRIO E DEFINIÇÃO DO PROBLEMA

Uma padaria de porte pequeno, de um bairro classe média, necessita informatizar o seu estoque de insumos para a padaria, bem como o estoque de produtos a ser vendidos na mesma. A padaria necessita que este estoque seja possível cadastrar o produto, foto, preço, quantidade, validade e descrição do produto. Este sistema não será responsável pela venda dos produtos, somente controle de estoque.

#### 4.1.2 ANÁLISE DA METODOLOGIA SUGERIDA

A metodologia mais indicada, seria o modelo incremental pois, se trata de um *software* simples onde os requisitos estão bem definidos. Este modelo irá possibilitar ao cliente identificar as funções a serem oferecidas pelo sistema de forma mais clara e simples. Desse modo o cliente identifica quais são as funções mais ou menos importantes para ele. Podendo então definir os estágios de entrega juntos com suas funcionalidades. Com este modelo com poucas interações o *software* poderá ser entregue ao cliente com eficiência garantida.

## 4.2 EXEMPLO 02 – DESENVOLVIMENTO DE UM SISTEMA DE NAVEGAÇÃO PARA UMA AERONAVE NÃO TRIPULADA, *DRONE*.

### 4.2.1 CENÁRIO E DEFINIÇÃO DO PROBLEMA

Um agricultor, responsável por um grande campo precisa que um programador desenvolva sistema de navegação para um *drone*, para monitorar sua agricultura com câmeras de alta resolução e pulverização contra pragas. Esse *drone* deverá realizar sobrevoos semanais durante todo período de produção.

### 4.2.2 ANÁLISE DA METODOLOGIA SUGERIDA

A metodologia mais indicada é o modelo transformacional. Os métodos formais são utilizados para sistemas que precisam dar prioridade a coesão. O *drone* necessita de uma matemática que garanta exatidão e o modelo transformacional é desenvolvido a partir de princípios matemáticos garantindo exatidão em seus resultados.

## 4.3 EXEMPLO 03 – DESENVOLVIMENTO DE UM SISTEMA *FOOD* PARA CELULAR

### 4.3.1 CENÁRIO E DEFINIÇÃO DO PROBLEMA

Uma lanchonete deseja aumentar suas vendas e deseja construir um aplicativo para celular que ofereça seus produtos. Este *software* necessita ter todos os produtos oferecidos pela loja, como as formas de pagamento, envio do pedido diretamente para a produção e envio a casa do cliente.

### 4.3.2 ANÁLISE DA METODOLOGIA SUGERIDA

A metodologia mais indicada seria o modelo FDD, pois oferece uma metodologia ágil com ênfase em arquitetura e modelagem e possui interação



com o cliente. Esta metodologia permitirá intervalos pequenos entre as iterações e a entrega do projeto em um curto prazo. Com esta metodologia a equipe desenvolvedora conseguirá eficácia neste projeto

#### **4.4 EXEMPLO 04 – DESENVOLVIMENTO DE UM SISTEMA ALFABETIZAÇÃO INFANTIL**

##### **4.4.1 CENÁRIO E DEFINIÇÃO DO PROBLEMA**

Uma rede de ensino deseja adquirir um *software* para auxiliar a alfabetização de seus alunos. O projeto deve ter diversos níveis, onde as crianças serão alfabetizar de forma lúdica e eficiente. Neste projeto o cliente exige uma participação ativa de uma pedagoga para auxiliar a construção do mesmo.

##### **4.4.2 ANÁLISE DA METODOLOGIA SUGERIDA**

A metodologia mais indicada, seria o modelo SCRUM, pois oferece uma metodologia ágil com grande interação com o cliente. O cliente poderá está sempre presente definindo requisitos que considera importante para o sistema. Logo no início o sistema já poderá ser avaliado a cada *Sprint*. Com esta metodologia a equipe desenvolvedora conseguirá eficácia neste projeto.

## 5 CONCLUSÕES

Neste trabalho foi possível entender um pouco mais a respeito da importância dos métodos clássicos e ágeis na engenharia de *software*. Neste trabalho foi possível entender que a metodologia ágil surgiu não para substituir os métodos clássicos, mas sim para complementar. Os métodos clássicos possuem seus valores e vantagens na engenharia de *software* e que os métodos ágeis os complementam.

Foi apresentado o início da engenharia de *software*, o porquê de se construir modelos para a construção de *software*. Foi observado algumas reflexões relativas a importância deste processo, a importância do levantamento de requisito e da interação com o cliente. Foi observado também, como o cliente e os desenvolvedores podem escolher o método que mais se enquadre em seu projeto.

Foi apresentada também a importância dos métodos tradicionais para a engenharia de *software*. Muitas empresas ainda utilizam estes métodos, eles são bem aceitos quando cenários necessitam ser mais formais, onde a documentação é uma exigência do negócio. Foi analisado que cada método tradicional tem sua funcionalidade.

Em contrapartida, foi possível observar a chegada do método ágil com a esperança de que todos os problemas encontrados nos métodos tradicionais seriam resolvidos, e em alguns requisitos realmente é superior, os métodos ágeis enfatizam a colaboração humana e a auto-organização das equipes. O método ágil é um ótimo método quando se trabalham com pequenas equipes e que a flexibilidade é necessária. Cada método ágil tem sua funcionalidade para cada tipo de projeto.

Não é necessário abrir mão dos métodos clássicos, mas em alguns projetos ambos os métodos podem trabalhar lado a lado, em complementação. Foi observado ainda, que existe uma dificuldade para se buscar um modelo ideal que possa contribuir produtivamente na construção de um *software*. A escolha de um modelo está ligada aos recursos e os custos que este modelo irá requerer. Fica então evi-

denciado que a engenharia de *software* a cada dia tenta aperfeiçoar suas técnicas para que se encontre o modelo perfeito para cada projeto. Mas que, sem dúvida os métodos ágeis vieram para agregar, consolidar, flexibilizar e humanizar a construção de projetos de *software*. Finalmente, vimos que esta evolução dos métodos ágeis não para por aqui e que a cada dia se buscarão métodos que elevarão o conhecimento de todos os envolvidos em projeto, para a construção de processos de *software* ainda mais eficazes.

## 5.1 TRABALHOS FUTUROS

Visando trabalhos futuros, é interessante um estudo sobre métrica para métodos ágeis de desenvolvimento, no intuito de possibilitar um acompanhamento mais preciso das atividades do processo.

## REFERÊNCIAS BIBLIOGRÁFICAS

- [1] AGIL, M. **Manifesto Ágil**. <http://www.manifestoagil.com.br/>. Acesso em: 24 fev. 2016.
- [2] BEZERRA, E. **Princípios de Análise e Projeto de Sistemas com UML**. [S.l.]: Campus, 2007.
- [3] BOEHM, B. W. **A Spiral Model of Software Development and Enhancement**. [S.l.]: Prentice Hall Press, 1988.
- [4] BRAZ, A. **Precisa-se de Projetos Scrum pra Estudo de Caso**. *Agile, travel and more*, 2011. Disponível em: <<https://alanbraz.wordpress.com/2011/05/17/precisa-se-de-projetos-scrum/>>. Acesso em: 25 mar. 2016.
- [5] **EXTREME Programming: Uma introdução suave**. Extreme Programming, 2013. Disponível em: <<http://www.extremeprogramming.org/>>. Acesso em: 22 mar. 2016.
- [6] **FDD - Feature Driven Development**. Heptagon Tecnologia da Info, 2015. Disponível em: <<http://www.heptagon.com.br/fdd>>. Acesso em: 26 mar. 2016.
- [7] GOMES, C. T. N. **Introdução a prototipação e apresentação do Axure RP 6.5**. DevMedia, 2013. Acesso em: 10 mar. 2016.
- [8] GOMES, F. **Introdução ao FDD - Feature Driven Development**. DevMedia, 2013. Disponível em: <<http://www.devmedia.com.br/introducao-ao-fdd-feature-driven-development/27971>>. Acesso em: 25 mar. 2016.
- [9] GRANDO, N. Metodologias Ágeis no Desenvolvimento de Projetos de Software. **Blog do Nei**, 2010. Disponível em: <<https://neigrandi.wordpress.com/2010/09/06/metodologias-ageis-no-desenvolvimento-de-projetos-de-software/>>. Acesso em: 31 mar. 2016.
- [10] MANIFESTO, A. <http://www.agilemanifesto.org/>. Acesso em: 24 fev. 2016.
- [11] MEDEIROS, H. **Introdução aos Processos de Software e o Modelo Incremental e Evolucionário**. DevMedia, 2013. Disponível em:

<<http://www.devmedia.com.br/introducao-aos-processos-de-software-e-o-modelo-incremental-e-evolucionario/29839>>. Acesso em: 10 mar. 2016.

[12] MÉTODOS Ágeis. Neurobox, 2009. Disponível em: <<http://www.neurobox.com.br/metodos-ageis/>>. Acesso em: 28 mar. 2016.

[13] NASCIMENTO, R. J. O. D. **Como usar os Métodos Formais no desenvolvimento de Software**. DevMedia, 2014. Disponível em: <<http://www.devmedia.com.br/como-usar-os-metodos-formais-no-desenvolvimento-de-software/31339>>. Acesso em: 10 mar. 2016.

[14] PRIKLADNICKI, R.; WILLI, R.; MILANI, F. **Métodos Ágeis para Desenvolvimento de Software**. Porto Alegre: Bookman, 2014.

[15] RAFAEL DIAS RIBEIRO, H. D. C. E. S. R. **Métodos Ágeis em Gerenciamento de Projetos**. Rio de Janeiro: SPIN, 2015.

[16] RANDELL, N. P. E. B. **Software Engineering: A Report on Conference Spnsored by the SATO Science Commitee**. [S.l.]: [s.n.], 1969.

[17] S.PRESSMAN, R. **Engenharia de Software**. São Paulo: Makron Books, 1995.

[18] S.PRESSMAN, R. **Engenharia de Software: Uma abordagem profissional**. [S.l.]: Dados eletrônicos, 2011.

[19] SCRUM. Heptagon Tecnologia da Informação, 2015. Disponível em: <<http://www.heptagon.com.br/scrum>>. Acesso em: 25 mar. 2016.

[20] SIQUEIRA, F. L. **Métodos ágeis**, 2003. Disponível em: <[www.levysiqueira.com.br/artigos/metodos\\_ageis.pdf](http://www.levysiqueira.com.br/artigos/metodos_ageis.pdf)>. Acesso em: 25 mar. 2016.

[21] SOARES, M. D. S. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**, Conselheiro Lafaiete, 2004. Disponível em: <<http://jeltex.googlecode.com/svn/trunk/Jeltex/Material%20de%20Leitura/UML/Comp ara%C3%A7%C3%A3o%20entre%20metodologias.PDF>>. Acesso em: 30 mar. 2016.

[22] SOMMERVILLE, I. **Engenharia de Software**. 6ª. ed. São Paulo: Pearson Education do Brasil, 2003.

[23] SOMMERVILLE, I. **Engenharia de Software**. 9ª. ed. São Paulo: Pearson Education, 2011.