Metodologias e métodos

O termo metodologia é bastante controverso nas ciências em geral e na Engenharia de Software em particular. Muitos autores parecem tratar metodologia e método como sinônimos, porém seria mais adequado dizer que uma metodologia envolve princípios filosóficos que guiam uma gama de métodos que utilizam ferramentas e práticas diferenciadas para realizar algo.

Assim teríamos, por exemplo, a Metodologia Estruturada, na qual existem vários métodos, como Análise Estruturada e Projeto Estruturado (muitas vezes denominados SA/SD, e Análise Essencial). Tanto a Análise Estruturada quanto a Análise Essencial utilizam a ferramenta Diagrama de Fluxos de Dados para modelar o funcionamento do sistema.

1) Metodologia Estruturada
    1. Análise Estruturada
    2. Projeto Estruturado
    3. Programação Estruturada
    4. Análise Essencial
    5. SADT
    6. DFD - Diagrama de Fluxo de Dados
    7. MER - Modelo de Entidades e Relacionamentos

2) Metodologia Orientada a Objetos
    1. Orientação a Objetos
    2. Rational Unified Process (RUP)
    3. Desenvolvimento ágil de software
    4. Feature Driven Development (FDD)
    5. Enterprise Unified Process (EUP)
    6. Scrum (Scrum)
    7. Crystal (Crystal Clear, Crystal Orange, Crystal Orange Web)
    8. Programação extrema (XP)

3) Outras Metodologias
    1. Microsoft Solution Framework (MSF)

Começando pelas Metodologias Estruturadas...

Parte 01

1. Análise estruturada

A **análise estruturada** é uma atividade de construção de modelos. Utiliza uma notação que é própria ao método de análise estruturada para com a finalidade de retratar o fluxo e o conteúdo das informações utilizadas pelo sistema, dividir o sistema em partições funcionais e comportamentais e descrever a essência daquilo que será construído.

1.1 Modelo ambiental

O modelo ambiental descreve o ambiente no qual o sistema se insere, ou seja, descreve o contexto do sistema, que deve ter 3 componentes:

Componentes do modelo ambiental::

Definição de objetivos → Finalidade de sistema;

Lista de eventos → Os acontecimentos que ocorrem no exterior e que interagem com o sistema;

Diagrama de contexto → Representa o sistema como um único processo e as suas interações com o meio ambiente.

1.2 Modelo comportamental

O modelo comportamental descreve as ações que o sistema deve realizar para responder da melhor forma aos eventos definidos no modelo ambiental.

Técnicas utilizadas:

1.2.1 Diagrama de fluxos de dados (DFD)

O **diagrama de fluxos de dados** (DFD) é uma ferramenta para a modelagem de sistemas. Ela fornece apenas uma visão do sistema, a visão estruturada das funções, ou seja, o fluxo dos dados. Se estivermos desenvolvendo um sistema no qual os relacionamentos entre os dados sejam mais importantes que as funções, podemos dar menos importância ao DFD e dedicar-nos aos diagramas de entidade-relacionamento (DER)

Um DFD é uma ferramenta de modelagem que nos permite imaginar um sistema como uma rede de processos funcionais, interligados por "dutos" e "tanques de armazenamento de dados". (Edward Yourdon)

Outros nomes para este diagrama

Diagrama de bolhas

DFD (abreviatura)

Modelo de processo

Diagrama de fluxo de trabalho

Modelo funcional

Componentes de um DFD

DFD Entidades Externas

DFD Processos

Fluxo de dados

Depósito de dados

O DFD pode ter vários níveis de detalhamento de acordo com a necessidade do sistema. O diagrama de contexto é uma representação macro do sistema. Em seguida, temos os DFDs de níveis. O nível mais alto é conhecido como DFD de nível 0 e está logo abaixo do diagrama de contexto. Neste nível as principais funções do sistemas são mostradas. Caso o processo não esteja claro o suficiente o mesmo será aperfeiçoado a cada nível.

Quando se diz que o DFD fornece apenas uma visão do sistema,é pelo fato de que através de sua representação gráfica não nos comprometemos com a sua implementação física.

O DIAGRAMA DE FLUXO DE DADOS (DFD)

Todo modelo funcional de um sistema pode ser visto como sendo formado por uma representação gráfica (uma rede de funções ou processos interligados), acompanhada da descrição de cada função e suas interfaces.

A representação gráfica do modelo funcional costuma ser expressa por meio de um diagrama denominado diagrama de fluxo de dados−DFD.

O conceito de função → Caixa Preta

X o------ Y = F(X) ------o Y

por exemplo

Elevar o X o----- No X ao -----o Y

Quadrado

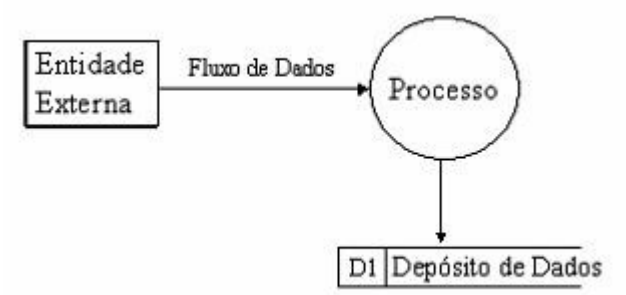
Há ligações de entrada e de saída da caixa.

Conhecem-se os elementos de entrada da caixa.

Conhecem-se os elementos de saída da caixa.

Sabe-se o que a caixa faz com as entradas para obter as saídas.

Não é preciso saber como a caixa realiza suas operações, e nem a ordem.



1.2.2 Dicionário de dados (DD);

Um **dicionário de dados** (do inglês *data dictionary*) é uma coleção de metadados que contêm definições e representações de elementos de dados.

Dentro do contexto de SGBD, um dicionário de dados é um grupo de tabelas, habilitadas apenas para leitura ou consulta, ou seja, é uma base de dados, propriamente dita, que entre outras coisas, mantém as seguintes informações:

Definição precisa sobre elementos de dados

Perfis de usuários, papéis e privilégios

Descrição de objetos

Restrições de integridade

Stored procedures (pequeno trecho de programa de computador, armazenado em um SGBD, que pode ser chamado freqüentemente por um programa principal) e gatilhos

Estrutura geral da base de dados

Informação de verificação

Alocações de espaço

Índices

Um dos benefícios de um dicionário de dados bem preparado é a consistência entre itens de dados através de diferentes tabelas. Por exemplo, diversas tabelas podem conter números de telefones; utilizando uma definição de um dicionário de dados bem feito, o formato do campo 'número de telefone' definido com "( )9999-9999" deverá ser obedecido em todas as tabelas que utilizarem esta informação.

Quando uma organização constrói um dicionário de dados de dimensão empresarial, o intuito

deve ser o de padronizar precisamente definições semânticas a serem adotadas na empresa toda; portanto, ele deve incluir tanto definições semânticas como de representação para elementos de dados, sendo que os componentes semânticos focam na criação precisa do significado dos elementos de dados, e de outro lado, as definições de representação indicam como os elementos de dados são armazenados em uma estrutura de computador de acordo com seu tipo, ou seja, se são dados do tipo inteiro, caracter ou formato de data (veja Tipos de Dados). Os dicionários de dados são mais precisos que glossários (termos e definições) porque costumam ter uma ou mais representações de como o dado é estruturado e podem envolver ontologias completas quando lógicas distintas sejam aplicadas a definições desses elementos de dados.

Os dicionários de dados são gerados, normalmente, separados do Modelo de Dados visto que estes últimos costumam incluir complexos relacionamentos entre elementos de dados.

Notação

Símbolo	Significado
=	é composto de
()	opcional (pode estar presente ou ausente)
{}	iteração
[]	escolha em uma das alternativas
**	comentário
@	identificador (chave) em um depósito
|	separa opções alternativas na construção [].

1.2.3 Diagrama de entidades e associações (ou relacionamentos) (Diagrama entidade relacionamento [DER] ou Modelo de entidades e relacionamentos [MER]);

Diagrama entidade relacionamento

Diagrama entidade relacionamento é um modelo diagramático que descreve o modelo de dados de um sistema com alto nível de abstração. Ele é a principal representação gráfica do Modelo de Entidades e Relacionamentos. É usado para representar o modelo conceitual do negócio.

MER: Conjunto de conceitos e elementos de modelagem que o projetista de banco de dados precisa conhecer. O Modelo é de Alto Nível.

DER: Resultado do processo de modelagem executado pelo projetista de dados que conhece o MER.

Tipos de relacionamentos

Os tipos de relacionamentos que são utilizadas neste diagrama:

Relacionamento 1..1 (lê-se relacionamento um para um) - indica que as tabelas têm relacionamento unívoca entre si. Você escolhe qual tabela vai receber a chave estrangeira;

Relacionamento 1..n (lê-se um para muitos) - a chave primária da tabela que tem o lado 1 vai para a tabela do lado N. No lado N ela é chamada de chave estrangeira;

Relacionamento n..n (lê-se muitos para muitos) - quando tabelas têm entre si relacionamento n..n, é necessário criar uma nova tabela com as chaves primárias das tabelas envolvidas, ficando assim uma chave composta, ou seja, formada por diversos campos-chave de outras tabelas. A relacionamento então se reduz para uma relacionamento 1..n, sendo que o lado n ficará com a nova tabela criada.

Modelo de entidades e relacionamentos

O **modelo de entidades e relacionamentos** é um modelo abstrato cuja finalidade é descrever, de maneira conceitual, os dados a serem utilizados em um sistema de informações ou que pertencem a um domínio. A principal ferramenta do modelo é sua representação gráfica, o diagrama entidade relacionamento. Normalmente o modelo e o diagrama são conhecidos por suas siglas: MER e DER.

Vários diagramas

O Modelo de Entidade e Relacionamento (MER) é um representação da realidade e pode ser representado por entidades, relacionamentos e atributos (Londeix, 1995). Existem muitas notações para diagrama de entidades e relacionamentos. A notação original foi proposta por Peter Chen e é composta de entidades (retângulos), relacionamentos (losangos), atributos (círculos) e linhas de conexão (linhas) que indicam a cardinalidade de uma entidade em um relacionamento. A cardinalidade pode ser 1:1, 1:N e N:M. Chen ainda propõe símbolos para entidades fracas e entidades associativas.

As notações modernas abandonaram o uso de símbolos especiais para atributos, incluindo a lista de atributo, de alguma forma, no símbolo da entidade. Consideramos as notações como as mais interessantes na atualidade:

- IDEF1X, utilizada pela ferramenta ERWIN, bastante difundida no mercado

- Engenharia de Informação, bastante difundida e também presente como notação alternativa no ERWIN.

- Notação de Setzer, difundida no Brasil por seu autor.

- Notação de Ceri, Bertini e Navathe, pouco difundida, mas com aspectos teóricos interessantes.

- Uso da UML para representar modelos de dados não-orientados a objetos.

1.2.4 Especificação de processos (EP) – (DESENHO);

Especificação de processos é a descrição do que ocorre dentro de cada bolha primitiva do nível mais baixo de um DFD (Diagrama de Fluxo de Dados), pode ser chamada de mini especificações.

O objetivo é definir o que deve ser feito para transformar entradas em saídas. É uma descrição detalhada, mas concisa da realização de processos pelos utilizadores.

1.2.5 Diagrama de transição de estados (DTE).

Em engenharia de software e eletrônica digital, um **diagrama de transição de estados** é uma representação do estado ou situação em que um objeto pode se encontrar no decorrer da execução de processos de um sistema. Com isso, o objeto pode passar de um estado inicial para um estado final através de uma transição.

Conceitos

Estado: Condição ou situação durante a vida de um objeto na qual ele satisfaz algumas condições, executa algumas atividades ou espera por eventos.

Transição: O relacionamento entre dois estados, indicando que o objeto que está no primeiro estado irá passar para o segundo estado mediante a ocorrência de um determinado evento e em certos casos uma condição.

Condição: causa necessária para que haja a transição de estado. Decorre da ocorrência de um evento ou circunstância que propicia a transição de estado.

Estado inicial: Estado por onde se começa a leitura de um diagrama de estado.

Estado final: Estado que representa o fim de uma máquina.

Barra de Sincronização: Semelhante a um Fork do Diagrama de atividade.

Estado composto: Estado composto por outras máquinas de estado organizadas em regiões que são executadas em paralelo.

Sincronização: permite que os relógios de dois ou mais processos paralelos estejam sincronizados em um determinado momento do processo.

Ação: atividade do sistema que efetua a transição de estado.

Exemplo

Um exemplo simples seria um semáforo (sinal de trânsito).

Cada estado corresponde a uma situação que ocorrerá. Quando verde, os carros podem prosseguir na via. Passado um tempo, é acionada a tarefa de **mudar para amarelo**. Então o semáforo passa de **verde** para **amarelo**. Aqui os carros ficam em estado de atenção e já aguardam a próxima transição.

O próximo passo é **passar para vermelho**. Nesse estado, os carros estão parados na via. De **vermelho**, o próximo estado somente será **verde**, assim, os carros podem voltar a trafegar na via.

Diagrama de transição de estados de um semáforo

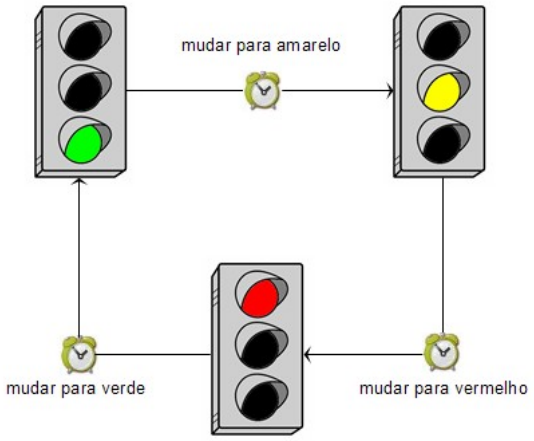
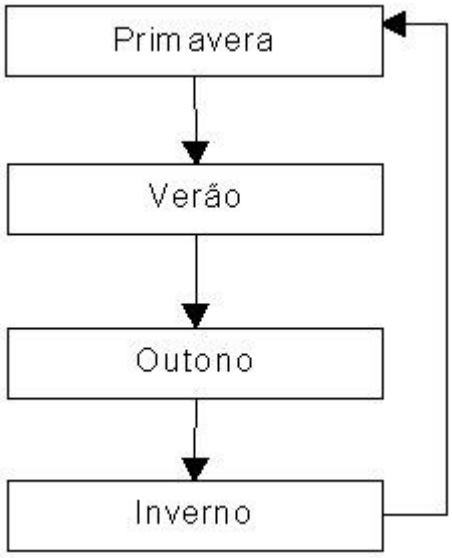


Diagrama de transição de estados das estações do ano



2. PROJETO ESTRUTURADO

Em engenharia de software, **projeto estruturado** é um método onde separa-se de forma hierárquica e determina-se quais projetos melhor solucionarão um problema. A atividade da especificação das atividades que compõem um modelo funcional de transformação das necessidades do usuário. São provenientes das fases de análise e diagramação e de plano de implementação.

3. PROGRAMAÇÃO ESTRUTURADA

Programação estruturada é uma forma de programação de computadores que preconiza que todos os programas possíveis podem ser reduzidos a apenas três estruturas: sequência, decisão e iteração, desenvolvida por Michael A. Jackson no seu livro "Principles of Program Design" de 1975.

Tendo, na prática, sido transformada na Programação modular, a Programação estruturada orienta os programadores para a criação de estruturas simples em seus programas, usando as subrotinas e as funções. Foi a forma dominante na criação de software anterior à programação orientada por objetos.

Apesar de ter sido sucedida pela programação orientada por objetos, pode-se dizer que a programação estruturada ainda é muito influente, uma vez que grande parte das pessoas ainda aprendem programação através dela. Para a resolução de problemas relativamente mais simples e diretos a programação estruturada é muito eficiente. Além disso, por exigir formas de pensar relativamente complexas, a programação orientada a objetos até hoje ainda não é bem compreendida ou usada pela maioria.

Há de se acrescentar também que inúmeras linguagens ainda extremamente relevantes nos dias de hoje, como Delphi (Pascal), Cobol, PHP e Perl ainda utilizam o paradigma estruturado (muito embora possuam suporte para a orientação à objetos).

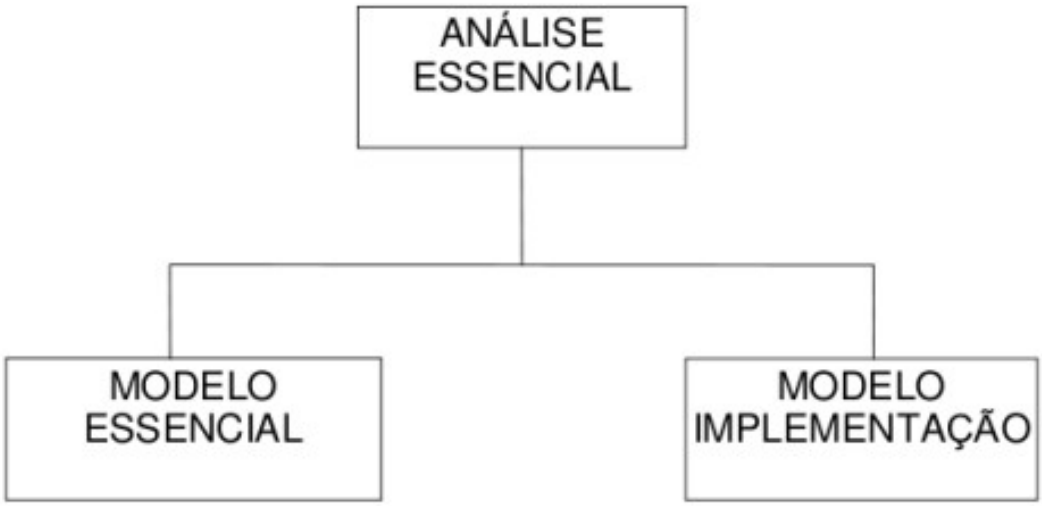
4. ANÁLISE ESSENCIAL

A **Análise Essencial** propõe o particionamento do sistema por eventos. A rigor, o valor de um sistema está na sua capacidade de responder com eficácia a todos os estímulos a que for submetido. Assim, um sistema é construído para responder a estímulos. A cada estímulo, o sistema deve reagir produzindo uma resposta predeterminada. A expressão Essential Analysis, traduzida por Análise Essencial, foi proposta em 1984 por McMenamim e Palmer para refletir a introdução dos novos conceitos que estavam sendo incorporados à Análise Estruturada clássica.

Conceito

A Análise Essencial é a técnica que orienta a análise de sistemas para a essência do negócio ao qual se destina, independente das soluções de informática que serão utilizadas em sua construção, partindo do princípio de que os sistemas existem independentemente dos computadores, e são feitos visando uma oportunidade de negócio.

Na Análise Essencial existem dois modelos, denominados de Modelo Essencial e Modelo de Implementação.



Modelo Essencial

Apresenta o sistema num nível de abstração completamente independente de restrições tecnológicas. Antes que um sistema seja implementado, é necessário conhecer-se a sua verdadeira essência, não importando saber se sua implementação vai ser manual ou automatizada, e nem mesmo que tipo de hardware ou software vai ser usado. O Modelo Essencial é formado por:

Modelo Ambiental: Define a fronteira entre o sistema e o resto do mundo

Modelo Comportamental: Define o comportamento das partes internas do sistema necessário para interagir com o ambiente;

Métodos Envolvidos:Modelagem de Dados e Modelagem Funcional.

Modelo Ambiental

O Modelo Ambiental é o modelo que define:

- A fronteira do sistema com o ambiente onde ele se situa, determinando o que é interno e o que é externo a ele.

- As interfaces entre o sistema e o ambiente externo, determinando que informações chegam ao sistema vindas do mundo exterior e vice-versa.

- Os eventos do ambiente externo ao sistema aos quais este deve responder.

- Ferramentas para definição do ambiente.

O Modelo Ambiental consiste de quatro componentes:

- Declaração de Objetivos

- Diagrama de Contexto

- Lista de Eventos

- Dicionário de Dados Preliminar (opcional)

Declaração dos Objetivos

Consiste de uma breve e concisa declaração dos objetivos do sistema.

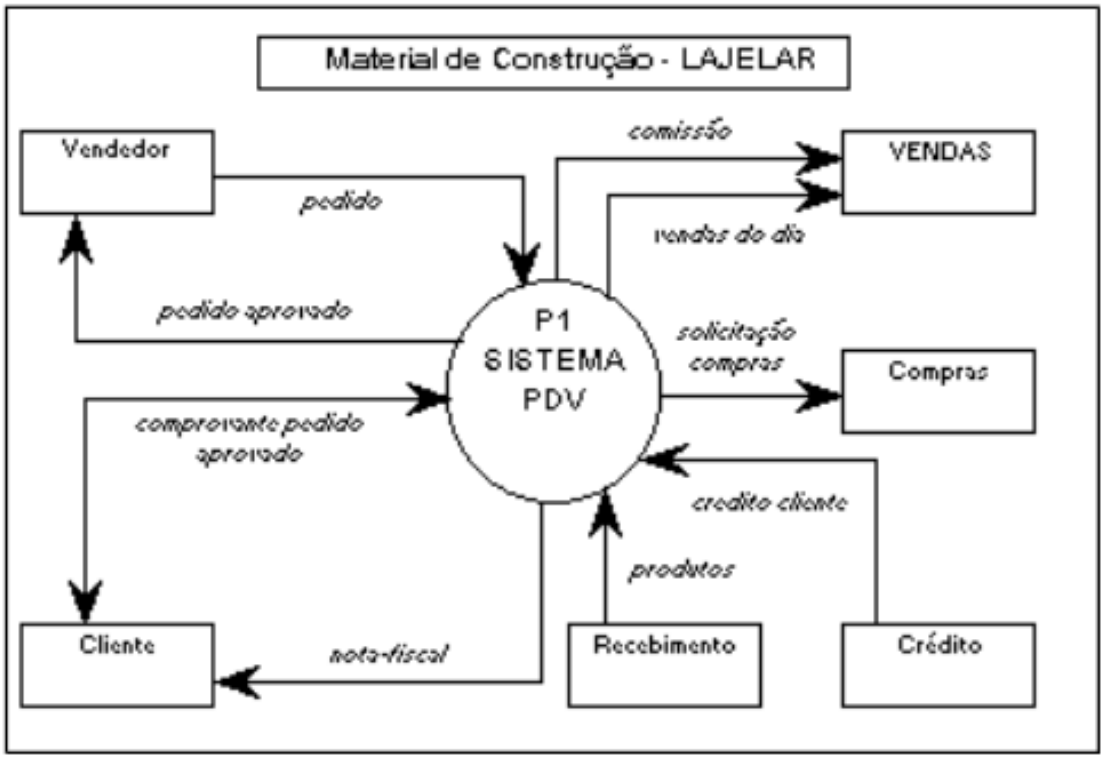
É dirigida para a alta gerência, gerência usuária ou outras pessoas não diretamente envolvidas no desenvolvimento do sistema.

Pode ter uma, duas ou várias sentenças mas não deve ultrapassar um parágrafo.

Não deve pretender dar uma descrição detalhada do sistema.

Diagrama de Contexto

Apresenta uma visão geral das características importantes do sistema, as pessoas, organizações ou sistemas com os quais o sistema se comunica (Entidades Externas), os dados que o sistema recebe do mundo exterior e que de alguma forma devem ser processados, os dados produzidos pelo sistema e enviados ao mundo exterior e a fronteira entre o sistema e o resto do mundo.



Lista de eventos

É uma relação de estímulos que ocorrendo no mundo exterior implicam que o sistema de algum tipo de resposta.

Também pode ser definido informalmente como um acontecimento do mundo exterior que requer do sistema alguma resposta. É um ativador de uma função. É a forma como o evento age sobre o sistema. É a consequência do fato de ter ocorrido um evento externo. É a chegada de um estímulo que indica que o evento ocorreu e isto faz com que o sistema então ative uma função pré-determinada para produzir a resposta esperada.

UM ESTÍMULO: É um ativador de uma função. É a forma como o evento age sobre o sistema. É a consequência do fato de ter ocorrido um evento externo. É a chegada de um estímulo que indica que o evento ocorreu e isto faz com que o sistema então ative uma função pré-determinada para produzir a resposta esperada.

UMA RESPOSTA: É o resultado gerado pelo sistema devido à ocorrência de um evento. Uma resposta é sempre o resultado da execução de alguma função interna no sistema como

consequência do reconhecimento pelo sistema de que um evento ocorreu.

Modelo Comportamental

Define o comportamento interno que o sistema deve ter para se relacionar adequadamente com o ambiente. Ou, o Modelo Comportamental é definido do ponto de vista interno, é o modelo interior do sistema. Descreve de que maneira o sistema, enquanto um conjunto de elementos inter-relacionados, reage, internamente, como um todo organizado, aos estímulos do exterior.

Consiste de quatro componentes:

- DFD Particionado
- Diagrama ER (DER)
- Diagrama de Transição de Estado (DTE)
- Dicionário de Dados Preliminar (opcional)
- Especificações de processos

Modelo de Implementação

Tem como objetivo definir a forma de implementação do sistema em um ambiente técnico específico. Apresenta o sistema num nível de abstração completamente dependente de restrições tecnológicas.

Simbologia

Conjunto de artefatos gráficos que permitem a montagem de diagramas na análise essencial

Processo: Conjunto de atividade que produzem, modificam ou atribuem qualidade às informações.

Depósito de Dados: Conjunto de informações armazenadas pelo processo para serem utilizadas por algum processo, a qualquer momento.

Entidade Externa: É algo situado fora do escopo do sistema, que é fonte ou destino das suas informações.

Fluxo de Dados: O nome deve expressar o significado do conjunto de informações que está fluindo.

Vantagens da Análise Essencial sobre a Estruturada

- A Análise Essencial começa pelo modelo essencial, o que equivale, na Análise Estruturada, começar diretamente pelo modelo lógico proposto.

- A Análise Estruturada aborda duas perspectivas do sistema - função e dados -, ao passo que a Análise Essencial aborda três perspectivas - função, dados e controle.

- Na Análise Estruturada o particionamento é feito através da abordagem top-down, enquanto na Análise Essencial, o particionamento é por eventos.

5. SADT

Structured Analysis and Design Technique

Structured Analysis and Design Technique — **SADT** é uma técnica para apresentação de idéias. Esta técnica foi proposta por Douglas T. Ross em 1977, no artigo Ross, D. (1977) “Structured Analysis (AS): A Language for Communicating Ideas”. IEEE Transactions on Software Engineering, vol. 3, no.1, pp. 16-34.

É uma técnica usada na construção de modelos. Segundo essa visão, cada modelo deve ter um objetivo e um ponto de vista.

Parte 2 – Orientação a Objetos

Metodologia Orientada a Objetos

2.1 Orientação a Objetos

A **orientação a objetos** é um paradigma de análise, projeto e programação de sistemas de *software* baseado na composição e interação entre diversas unidades de software chamadas de objetos.

Em alguns contextos, prefere-se usar modelagem orientada ao objeto, em vez de programação. De fato, o paradigma "orientação a objeto", tem bases conceituais e origem no campo de estudo da cognição, que influenciou a área de inteligência artificial e da linguística, no campo da abstração de conceitos do mundo real. Na qualidade de método de modelagem, é tida como a melhor estratégia para se eliminar o "gap semântico", dificuldade recorrente no processo de modelar o mundo real do domínio do problema em um conjunto de componentes de software que seja o mais fiel na sua representação deste domínio. Facilitaria a comunicação do profissional modelador e do usuário da área alvo, na medida em que a correlação da simbologia e conceitos abstratos do mundo real e da ferramenta de modelagem (conceitos, terminologia, símbolos, grafismo e estratégias) fosse a mais óbvia, natural e exata possível.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de *software*. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

C++, C♯, VB.NET, Java, Object Pascal, Objective-C, Python, SuperCollider, Ruby e Smalltalk são exemplos de linguagens de programação orientadas a objetos.

ActionScript, ColdFusion, Javascript, PHP (a partir da versão 4.0), Perl (a partir da versão 5) e Visual Basic (a partir da versão 4) são exemplos de linguagens de programação com suporte a orientação a objetos.

Conceitos essenciais

Classe representa um conjunto de objetos com características afins. Uma classe define o comportamento dos objetos através de seus métodos, e quais estados ele é capaz de manter através de seus atributos. Exemplo de classe: Os seres humanos.

Subclasse é uma nova classe que herda características de sua(s) classe(s) ancestral(is)

Objeto / instância de uma classe. Um objeto é capaz de armazenar estados através de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos. Exemplo de objetos da classe Humanos: João, José, Maria

Atributo são características de um objeto. Basicamente a estrutura de dados que vai representar a classe. Exemplos: Funcionário: nome, endereço, telefone, CPF,...; Carro: nome, marca, ano, cor, …; Livro: autor, editora, ano. Por sua vez, os atributos possuem valores. Por exemplo, o atributo cor pode conter o valor azul. O conjunto de valores dos atributos de um determinado objeto é chamado de estado

Método definem as habilidades dos objetos. Bidu é uma instância da classe Cachorro, portanto tem habilidade para latir, implementada através do método deUmLatido. Um método em uma classe é apenas uma definição. A ação só ocorre quando o método é invocado através do objeto, no caso Bidu. Dentro do programa, a utilização de um método deve afetar apenas um objeto em particular; Todos os cachorros podem latir, mas você quer que apenas Bidu dê o latido. Normalmente, uma classe possui diversos métodos, que no caso da classe Cachorro poderiam ser sente, coma e morda

Mensagem é uma chamada a um objeto para invocar um de seus métodos, ativando um comportamento descrito por sua classe. Também pode ser direcionada diretamente a uma classe (através de uma invocação a um método estático)

Herança (ou generalização) é o mecanismo pelo qual uma classe (sub-classe) pode estender outra classe (super-classe), aproveitando seus comportamentos (métodos) e variáveis possíveis (atributos). Um exemplo de herança: Mamífero é super-classe de Humano. Ou seja, um Humano **é um** mamífero. Há herança múltipla quando uma sub-classe possui mais de uma super-classe. Essa relação é normalmente chamada de relação "é um"

Associação é o mecanismo pelo qual um objeto utiliza os recursos de outro. Pode tratar-se de uma associação simples "usa um" ou de um acoplamento "parte de". Por exemplo: Um humano usa um telefone. A tecla "1" é parte de um telefone

Encapsulamento consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados. Exemplo: você não precisa conhecer os detalhes dos circuitos de um telefone para utilizá-lo. A carcaça do telefone encapsula esses detalhes, provendo a você uma interface mais amigável (os botões, o monofone e os sinais de tom)

Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstração de entidades existentes no domínio do sistema de software

Polimorfismo consiste em quatro propriedades que a linguagem pode ter (atente para o fato de que nem toda linguagem orientada a objeto tem implementado todos os tipos de polimorfismo):

Universal:

Inclusão: um ponteiro para classe mãe pode apontar para uma instância de uma classe filha (exemplo em Java: "List lista = new LinkedList();" (tipo de polimorfismo mais básico que existe)

Paramétrico: se restringe ao uso de *templates* (C++, por exemplo) e *generics* (Java/C♯)

Ad-Hoc:

Sobrecarga: duas funções/métodos com o mesmo nome mas assinaturas diferentes

Coerção: a linguagem que faz as conversões implicitamente (como por exemplo atribuir um int a um float em C++, isto é aceito mesmo sendo tipos diferentes pois a conversão é feita implicitamente)

Interface é um contrato entre a classe e o mundo externo. Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface

Pacotes (ou Namespaces) são referências para organização lógica de classes e interfaces

2.2 Rational Unified Process (RUP)

O **RUP**, abreviação de **Rational Unified Process** (ou Processo Unificado Racional), é um processo proprietário de Engenharia de software criado pela Rational Software Corporation, adquirida pela IBM, ganhando um novo nome **IRUP** que agora é uma abreviação de **IBM Rational Unified Process** e tornando-se uma *brand* na área de Software, fornecendo técnicas a serem seguidas pelos membros da equipe de desenvolvimento de software com o objetivo de aumentar a sua produtividade no processo de desenvolvimento.

O RUP usa a abordagem da orientação a objetos em sua concepção e é projetado e documentado utilizando a notação UML (*Unified Modeling Language*) para ilustrar os processos em ação. Utiliza técnicas e práticas aprovadas comercialmente.

É um processo considerado pesado e preferencialmente aplicável a grandes equipes de desenvolvimento e a grandes projetos, porém o fato de ser amplamente customizável torna possível que seja adaptado para projetos de qualquer escala. Para a gerência do projeto, o RUP provê uma solução disciplinada de como assinalar tarefas e responsabilidades dentro de uma organização de desenvolvimento de software.

O RUP é, por si só, um produto de software. É modular e automatizado, e toda a sua metodologia é apoiada por diversas ferramentas de desenvolvimento integradas e vendidas pela IBM através de seus "Rational Suites".

Métodos concorrentes no campo da engenharia de software incluem o "Cleanroom" (considerado pesado) e os Métodos Ágeis (leves) como a Programação Extrema (XP-Extreme Programming), Scrum, FDD e outros.

1 Linhas mestras

1.1 Gestão de requisitos

1.2 Uso de arquitetura baseada em componentes
1.3 Uso de software de modelos visuais
1.4 Verificação da qualidade do software
1.5 Gestão e Controle de Mudanças do Software
2 Fases
2.1 Fase de Concepção
2.2 Fase de Elaboração
2.3 Fase de Construção
2.4 Fase de Transição
3 Disciplinas
3.1 Seis Disciplinas de Engenharia
3.1.1 Disciplina de Modelagem de Negócios
3.1.2 Disciplina de Requisitos
3.1.3 Disciplina de Análise e Projeto("Design")
3.1.4 Disciplina de Implementação
3.1.5 Disciplina de Teste
3.1.6 Disciplina de Implantação
3.2 Três Disciplinas de Apoio/Suporte
3.2.1 Disciplina de Ambiente
3.2.2 Disciplina de Configuração e Gerência de Mudança
3.2.3 Disciplina de Gerência de Projeto
4 Princípios e melhores prática
4.1 Desenvolvimento iterativo
4.2 Gerenciamento de requisitos
4.3 Uso de arquitetura baseada em componentes
4.4 Modelagem visual de software
4.5 Verificar qualidade de software
4.6 Controle de alterações no software

Linhas mestras

O RUP define as seguintes linhas-mestras e esqueletos (*templates*) para os membros da equipe de um ciclo de produção: parte do cliente, e uma avaliação do progresso do projeto pela sua gerência. Além disso, ajuda os programadores a manterem-se concentrados no projeto.

Gestão de requisitos

Uma documentação apropriada é essencial para qualquer grande projeto; note que o RUP descreve como documentar a funcionalidade, restrições de sistema, restrições de projeto e requisitos de negócio.

Os casos de uso (em inglês *Use Cases*) e os cenários são exemplos de artefatos dependentes do processo, que têm sido considerados muito mais eficazes na captura de requisitos funcionais.

Uso de arquitetura baseada em componentes

A arquitetura baseada em componentes cria um sistema que pode ser facilmente extensível, promovendo a reutilização de software e um entendimento intuitivo. Um componente normalmente se relaciona com um objeto na programação orientada a objetos.

O RUP oferece uma forma sistemática para construir este tipo de sistema, focando-se em produzir uma arquitetura executável nas fases iniciais do projeto, ou seja, antes de comprometer recursos em larga escala.

Estes componentes são normalmente incluídos em infraestruturas existentes como o CORBA e o COM (Modelo de Componentes de Objetos).

Uso de software de modelos visuais

Ao abstrair a programação do seu código e representá-la utilizando blocos de construção gráfica, o RUP consegue uma maneira efetiva de se ter uma visão geral de uma solução.

O uso de modelos visuais também pode permitir que indivíduos de perfil menos técnico (como clientes) tenham um melhor entendimento de um dado problema, e assim se envolvam mais no projeto como um todo.

A linguagem de modelagem UML tornou-se um padrão industrial para representar projetos, e é amplamente utilizada pelo RUP!

Verificação da qualidade do software

Não assegurar a qualidade do software é a falha mais comum em todos os projetos de sistemas computacionais. Normalmente pensa-se em qualidade de software após o término dos projetos, ou a qualidade é responsabilidade de uma equipe diferente da equipe de desenvolvimento.

O RUP visa auxiliar no controle do planejamento da qualidade, verificando-a na construção de todo o processo e envolvendo todos os membros da equipe de desenvolvimento.

Gestão e Controle de Mudanças do Software

Em todos os projetos de software a existência de mudanças é inevitável. O RUP define métodos para controlar e monitorar mudanças. Como uma pequena mudança pode afetar aplicações de formas inteiramente imprevisíveis, o controle de mudanças é essencial para o sucesso de um projeto.

O RUP também define *áreas de trabalho seguras*, garantindo a um programador que as mudanças efetuadas noutro sistema não afetarão o seu sistema.

Fases

Até agora estas linhas de guia são gerais, a serem aderidas ao percorrer do ciclo de vida de um projeto. As fases indicam a ênfase que é dada no projeto em um dado instante. Para capturar a dimensão do tempo de um projeto, o RUP divide o projeto em quatro fases diferentes:

Concepção: ênfase no escopo do sistema;

Elaboração: ênfase na arquitetura;

Construção: ênfase no desenvolvimento;

Transição: ênfase na implantação.

O RUP também se baseia nos 4 Ps:

Pessoas

Projeto

Produto

Processo

As fases são compostas de iterações. As iterações são janelas de tempo; as iterações possuem prazo definido enquanto as fases são objetivas.

Todas as fases geram artefatos. Estes serão utilizados nas próximas fases e documentam o projeto, além de permitir melhor acompanhamento.

Fase de Concepção

A fase de concepção contém os *workflows* necessários para que as partes interessadas (*stakeholders*) concordem com os objetivos, arquitetura e o planejamento do projeto. Se as partes interessadas tiverem bons conhecimentos, então, pouca análise será requerida. Caso contrário, uma análise maior será requerida. Nesta fase os requisitos essenciais do sistema são transformados em casos de uso. O objetivo não é fechar todos os requisitos, mas apenas aqueles necessários para se formar uma opinião. Esta fase é geralmente curta e serve para se definir se vale a pena continuar com o projeto e definir os riscos e o custo deste. Um protótipo pode ser feito para que o cliente possa aprovar.

Como cita o RUP, o ideal é que sejam feitas iterações, mas estas devem ser bem definidas quanto a sua quantidade e objetivos.

Fase de Elaboração

A fase de elaboração será apenas para o projeto do sistema, buscando complementar o levantamento / documentação dos casos de uso, voltado para a arquitetura do sistema, revisa a modelagem do negócio para os projetos e inicia a versão do manual do usuário. Deve-se aceitar: Visão geral do produto (incremento + integração) está estável?; O plano do projeto é confiável?; Custos são admissíveis?

Fase de Construção

Na fase de construção, começa o desenvolvimento físico do software, produção de códigos, testes alfa e beta.

Deve-se aceitar testes, e processos de testes estáveis, e se os códigos do sistema constituem "baseline".

Fase de Transição

Nesta fase ocorre a entrega ("deployment") do software, é realizado o plano de implantação e entrega, acompanhamento e qualidade do software. Produtos (releases, versões) devem ser entregues, e ocorrer a satisfação do cliente. Nesta fase também é realizada a capacitação dos usuários.

Disciplinas

Seis Disciplinas de Engenharia

Disciplina de Modelagem de Negócios

As organizações estão cada vez mais dependentes de sistemas de TI, tornando-se imperativo que os engenheiros de sistemas de informação saibam como as aplicações em desenvolvimento se inserem na organização. As empresas investem em TI, quando entendem a vantagem competitiva do valor acrescentado pela tecnologia. O objetivo de modelagem de negócios é, primeiramente, estabelecer uma melhor compreensão e canal de comunicação entre engenharia de negócios e engenharia de software. Compreender o negócio significa que os engenheiros de software devem compreender a estrutura e a dinâmica da empresa alvo (o cliente), os atuais problemas na organização e possíveis melhorias. Eles também devem garantir um entendimento comum da organização-alvo entre os clientes, usuários finais e desenvolvedores.

Modelagem de negócios, explica como descrever uma visão da organização na qual o sistema será implantado e como usar esta visão como uma base para descrever o processo, papéis e responsabilidades.

Disciplina de Requisitos

Esta disciplina explica como levantar pedidos das partes interessadas ("stakeholders") e transformá-los em um conjunto de requisitos que os produtos funcionam no âmbito do sistema a ser construído e fornecem requisitos detalhados para o que deve fazer o sistema.

Disciplina de Análise e Projeto("Design")

O objetivo da análise e projeto é mostrar como o sistema vai ser realizado. O objetivo é construir um sistema que:

- Execute, em um ambiente de execução específica, as tarefas e funções especificadas nas descrições de casos de uso

- Cumpra todas as suas necessidades

- Seja fácil de manter quando ocorrerem mudanças de requisitos funcionais

Resultados de projeto em um modelo de análise e projeto tem, opcionalmente, um modelo de análise. O modelo de design serve como uma abstração do código-fonte, isto é, o projeto atua como um modelo de "gabarito" de como o código-fonte é estruturado e escrito. O modelo de projeto consiste em classes de design estruturado em pacotes e subsistemas com interfaces bem definidas, representando o que irá se tornar componentes da aplicação. Ele também contém descrições de como os objetos dessas classes colaboram para desempenhar casos de uso do projeto.

Disciplina de Implementação

Os efeitos da implementação são:

- Para definir a organização do código, em termos de subsistemas de implementação organizadas em camadas

- Para implementar classes e objetos em termos de componentes (arquivos-fonte, binários, executáveis e outros)

- Para testar os componentes desenvolvidos como unidades

- Integrar os resultados produzidos por implementadores individuais (ou equipes), em um sistema executável

Sistemas são realizados através da aplicação de componentes. O processo descreve como reutilizar componentes existentes ou implementar novos componentes com responsabilidades bem definidas, tornando o sistema mais fácil de manter e aumentar as possibilidades de reutilização.

Disciplina de Teste

As finalidades da disciplina de teste são:

- Para verificar a interação entre objetos
- Para verificar a integração adequada de todos os componentes do software
- Para verificar se todos os requisitos foram corretamente implementados
- Identificar e garantir que os defeitos são abordados antes da implantação do software
- Garantir que todos os defeitos são corrigidos, reanalisados e fechados

O Rational Unified Process propõe uma abordagem iterativa, o que significa que deve-se testar todo o projeto. Isto permite encontrar defeitos tão cedo quanto possível, o que reduz radicalmente o custo de reparar o defeito. Os testes são realizados ao longo de quatro dimensões da qualidade:*confiabilidade, funcionalidade, desempenho da aplicação, e o desempenho do sistema*. Para cada uma destas dimensões da qualidade, o processo descreve como você passar pelo teste do ciclo de planejamento, projeto, implementação, execução e avaliação.

Disciplina de Implantação

O objetivo da implantação é o de produzir com sucesso lançamentos de produtos e entregar o software para seus usuários finais. Ele cobre uma vasta gama de atividades, incluindo a produção de releases externos do software, a embalagem do software e aplicativos de negócios, distribuição do software, instalação do software e prestação de ajuda e assistência aos usuários. Embora as atividades de implantação estejam principalmente centradas em torno da fase de transição, muitas das atividades devem ser incluídas nas fases anteriores para se preparar para a implantação, no final da fase de construção. Os processos ("workflows") de "Implantação e Ambiente" do RUP contêm menos detalhes do que outros workflows.

Três Disciplinas de Apoio/Suporte

Disciplina de Ambiente

O ambiente enfoca as atividades necessárias para configurar o processo para um projeto. Ele descreve as atividades necessárias para desenvolver as diretrizes de apoio a um projeto. A proposta das atividades de ambiente é prover à organização de desenvolvimento de software os processos e as ferramentas que darão suporte à equipe de desenvolvimento. Se os usuários do RUP não entendem que o RUP é um framework de processo, eles podem percebê-lo como um processo pesado e caro. No entanto, um conceito-chave dentro do RUP foi que o processo RUP pode e, muitas vezes, deve ser refinado. Este foi inicialmente feito manualmente, ou seja, por escrito, um documento de "caso de desenvolvimento" que especificou o processo refinado para ser utilizado. Posteriormente, o produto IBM Rational Method Composer foi criado para ajudar a tornar esta etapa mais simples, engenheiros de processos e gerentes de projeto poderiam mais facilmente personalizar o RUP para suas necessidades de projeto. Muitas das variações

posteriores do RUP, incluindo OpenUP/Basic, a versão open-source e leve do RUP, são agora apresentados como processos distintos, por direito próprio, e atendem a diferentes tipos e tamanhos de projetos, tendências e tecnologias de desenvolvimento de software. Historicamente, como o RUP, muitas vezes é personalizado para cada projeto por um perito do processo RUP, o sucesso total do projeto pode ser um pouco dependente da capacidade desta pessoa.

Disciplina de Configuração e Gerência de Mudança

A disciplina de Gestão de Mudança em negócios com RUP abrange três gerenciamentos específicos: de configuração, de solicitações de mudança, e de status e medição.

Gerenciamento de configuração: A gestão de configuração é responsável pela estruturação sistemática dos produtos. Artefatos, como documentos e modelos, precisam estar sob controle de versão e essas alterações devem ser visíveis. Ele também mantém o controle de dependências entre artefatos para que todos os artigos relacionados sejam atualizados quando são feitas alterações

Gerenciamento de solicitações de mudança: Durante o processo de desenvolvimento de sistemas com muitos artefatos existem diversas versões. O CRM mantém o controle das propostas de mudança

Gerenciamento de status e medição: Os pedidos de mudança têm os estados: *novo*, *conectado*, *aprovado*, *cedido* e *completo*. A solicitação de mudança também tem atributos como a causa raiz, ou a natureza (como o defeito e valorização), prioridade, etc. Esses estados e atributos são armazenados no banco de dados para produzir relatórios úteis sobre o andamento do projeto. A Rational também tem um produto para manter a solicitações de mudança chamado ClearQuest. Esta atividade têm procedimentos a serem seguidos

Disciplina de Gerência de Projeto

O planejamento de projeto no RUP ocorre em dois níveis. Há uma baixa granularidade ou planos de **Fase** que descreve todo o projeto, e uma série de alta granularidade ou planos de **Iteração** que descrevem os passos iterativos. Esta disciplina concentra-se principalmente sobre os aspectos importantes de um processo de desenvolvimento iterativo: Gestão de riscos; Planejamento um projeto iterativo através do ciclo de vida e para uma iteração particular; E o processo de acompanhamento de um projeto iterativo, métricas. No entanto, esta disciplina do RUP não tenta cobrir todos os aspectos do gerenciamento de projetos.

Por exemplo, não abrange questões como:

Gestão de Pessoas: contratação, treinamento, etc

Orçamento Geral: definição, alocação, etc

Gestão de Contratos: com fornecedores, clientes, etc

Princípios e melhores prática

RUP é baseado em um conjunto de princípios de desenvolvimento de software e melhores práticas, por exemplo:

- Desenvolvimento de software iterativo

- Gerenciamento de requisitos

- Uso de arquitetura baseada em componente

- Modelagem visual de software

- Verificação da qualidade do software

- Controle de alteração no software

Desenvolvimento iterativo

Dado o tempo gasto para desenvolver um software grande e sofisticado, não é possível definir o problema e construir o software em um único passo. Os requerimentos irão freqüentemente mudar no decorrer do desenvolvimento do projeto, devido a restrições de arquitetura, necessidades do usuário ou a uma maior compreensão do problema original. Alterações permitem ao projeto ser constantemente refinado, além de assinalarem itens de alto risco do projeto como as tarefas de maior prioridade. De forma ideal, ao término de cada iteração haverá uma versão executável, o que ajuda a reduzir o risco de configuração do projeto, permitindo maior retorno do usuário e ajudando ao desenvolvedor manter-se focado.

O RUP usa desenvolvimento iterativo e incremental pelas seguintes razões:

- A integração é feita passo a passo durante o processo de desenvolvimento, limitando-se cada passo a poucos elementos

- A integração é menos complexa, reduzindo seu custo e aumentado sua eficiência

- Partes separadas de projeto e/ou implementação podem ser facilmente identificadas para posterior reuso

- Mudanças de requisitos são registradas e podem ser acomodadas

- Os riscos são abordados no inicio do desenvolvimento e cada iteração permite a verificação de riscos já percebidos bem como a identificação de novos

- Arquitetura de software é melhorada através de um escrutino repetitivo

- Usando iterações, um projeto terá um plano geral, como também múltiplos planos de iteração. O envolvimento dos stakeholders é freqüentemente encorajado a cada entrega. Desta maneira, as entregas servem como uma forma de se obter o comprometimento dos envolvidos, promovendo também uma constante comparação entre os requisitos e o desenvolvimento da organização para as pendências que surgem.

Gerenciamento de requisitos

O Gerenciamento de requisitos no RUP está concentrado em encontrar as necessidades do usuário final pela identificação e especificação do que ele necessita e identificando aquilo que deve ser mudado. Isto traz como benefícios:

- A correção dos requisitos gera a correção do produto, desta forma as necessidadades dos usuários são encontradas.

- As características necessárias serão incluídas, reduzindo o custo de desenvolvimentos posteriores.

RUP sugere que o gerenciamento de requisitos tem que seguir as atividades:

Análise do problema é concordar com o problema e criar medições que irão provar seu valor para a organização

Entender as necessidades de seus *stakeholders* é compartilhar o problema e valores com os stakeholders-chave e levantar quais as necessidades que estão envolvidas na elaboração da ideia

Definir o problema é a definição das características das necessidades e esquematização dos casos de uso, atividades que irão facilmente mostrar os requisitos de alto-nível e esboçar o modelo de uso do sistema

Gerenciar o escopo do sistema trata das modificações de escopo que serão comunicadas baseadas nos resultados do andamento e selecionadas na ordem na qual os fluxogramas de casos de uso são atacados

Refinar as definições do sistema trata do detalhamento dos fluxogramas de caso de uso com os *stakeholders* de forma a criar uma especificação de requerimentos de software que pode servir como um contrato entre o seu grupo e o do cliente e poderá guiar as atividades de teste e projeto

Gerenciamento das mudanças de requisitos trata de como identificar as chegadas das mudanças de requerimento num projeto que já começou

Uso de arquitetura baseada em componentes

Arquitetura baseada em componentes cria um sistema que é facilmente extensível, intuitivo e de fácil compreensão e promove a reusabilidade de software. Um componente freqüentemente se relaciona com um conjunto de objetos na programação orientada ao objeto.

Arquitetura de software aumenta de importância quando um sistema se torna maior e mais complexo. RUP foca na produção da arquitetura básica nas primeiras iterações. Esta arquitetura então se torna um protótipo nos ciclos iniciais de desenvolvimento. A arquitetura desenvolve-se em cada iteração para se tornar a arquitetura final do sistema. RUP também propõe regras de projeto e restrições para capturar regras de arquitetura. Pelo desenvolvimento iterativo é possível identificar gradualmente componentes os quais podem então ser desenvolvidos, comprados ou reusados. Estes componentes são freqüentemente construídos em infra-estruturas existentes tais como CORBA e COM, ou JavaEE, ou PHP

Modelagem visual de software

Abstraindo sua programação do seu código e representando-a usando blocos de construção gráfica constitui-se de uma forma efetiva de obter uma visão geral de uma solução. Usando esta representação, recursos técnicos podem determinar a melhor forma para implementar a dado conjunto de interdependências lógicas. Isto também constrói uma camada intermediária entre o processo de negócio e o código necessário através da tecnologia da informação. Um modelo neste contexto é uma visualização e ao mesmo tempo uma simplificação de um projeto complexo. RUP especifica quais modelos são necessários e porque.

A Linguagem modelagem unificada (UML) pode ser usada para modelagem de Casos de Uso, diagrama de classes e outros objetos. RUP também discute outras formas para construir estes modelos.

Verificar qualidade de software

Garantia da qualidade de software é o ponto mais comum de falha nos projetos de software, desde que isto é freqüentemente algo que não se pensa previamente e é algumas vezes tratado por equipes diferentes. O RUP ajuda no planejamento do controle da qualidade e cuida da sua construção em todo processo, envolvendo todos os membros da equipe. Nenhuma tarefa é especificamente direcionada para a qualidade; o RUP assume que cada membro da equipe é responsável pela qualidade durante todo o processo. O processo foca na descoberta do nível de qualidade esperado e provê testes nos processos para medir este nível.

Controle de alterações no software

Em todos os projetos de software, mudanças são inevitáveis. RUP define métodos para controlar, rastrear e monitorar estas mudanças. RUP também define *espaços de trabalho seguros* (do inglês secure workspaces), garantindo que um sistema de engenharia de software não será afetado por mudanças em outros sistemas. Este conceito é bem aderente com arquiteturas de software baseados em componentização.

Observação: E a UML??

Diagramas da UML 2.0

A **Unified Modeling Language** (**UML**) é uma linguagem de modelagem não proprietária de terceira geração. A UML não é uma metodologia de desenvolvimento, o que significa que ela não diz para você o que fazer primeiro e em seguida ou como projetar seu sistema, mas ela lhe auxilia a visualizar seu desenho e a comunicação entre objetos.

Basicamente, a UML permite que desenvolvedores visualizem os produtos de seus trabalhos em diagramas padronizados. Junto com uma notação gráfica, a UML também especifica significados, isto é, semântica. É uma notação independente de processos, embora o RUP (Rational Unified Process) tenha sido especificamente

desenvolvido utilizando a UML.

É importante distinguir entre um modelo UML e um diagrama (ou conjunto de diagramas) de UML. O último é uma representação gráfica da informação do primeiro, mas o primeiro pode existir independentemente. O XMI (XML Metadata Interchange) na sua versão corrente disponibiliza troca de modelos mas não de diagramas.

Objetivos da UML

Os objetivos da UML são: especificação, documentação, estruturação para sub-visualização e maior visualização lógica do desenvolvimento completo de um sistema de informação. A UML é um modo de padronizar as formas de modelagem.

O futuro da UML

Embora a UML defina uma linguagem precisa, ela não é uma barreira para futuros aperfeiçoamentos nos conceitos de modelagem. O desenvolvimento da UML foi baseado em técnicas antigas e marcantes da orientação a objetos, mas muitas outras influenciarão a linguagem em suas próximas versões. Muitas técnicas avançadas de modelagem podem ser definidas usando UML como base, podendo ser estendida sem se fazer necessário redefinir a sua estrutura interna.

A UML será a base para muitas ferramentas de desenvolvimento, incluindo modelagem visual, simulações e ambientes de desenvolvimento. Em breve, ferramentas de integração e padrões de implementação baseados em UML estarão disponíveis para qualquer um.

A UML integrou muitas ideias adversas, e esta integração acelera o uso do desenvolvimento de softwares orientados a objetos.

História

A UML tem origem na compilação das "melhores práticas de engenharia" que provaram ter sucesso na modelagem de sistemas grandes e complexos. Sucedeu aos conceitos de Booch, OMT (Rumbaugh) e OOSE (Jacobson) fundindo-os numa única linguagem de modelagem comum e largamente utilizada. A UML pretende ser a linguagem de modelagem padrão para modelar sistemas concorrentes e distribuídos.

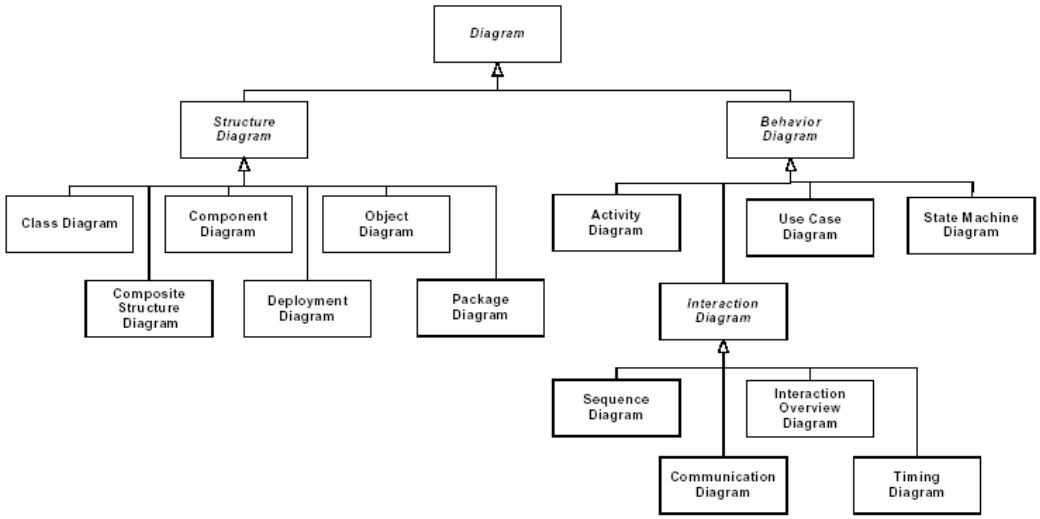
A UML ainda não é um padrão da indústria, mas esse objetivo está a tomar forma sob os auspícios do Object Management Group (OMG). O OMG pediu informação acerca de metodologias orientadas a objetos que pudessem criar uma linguagem rigorosa de modelagem de software. Muitos líderes da indústria responderam na esperança de ajudar a criar o padrão.

Os esforços para a criação da UML tiveram início em outubro de 1994, quando Rumbaugh se juntou a Booch na Rational. Com o objetivo de unificar os métodos Booch e OMT, decorrido um ano de trabalho, foi lançado, em outubro de 1995, o esboço da versão 0.8 do *Unified Process* - Processo Unificado (como era conhecido). Nesta mesma época, Jacobson se associou à Rational e o escopo do projeto da UML foi expandido para incorporar o método OOSE. Nasceu então, em junho de 1996, a

versão 0.9 da UML.

Finalmente em 1997, a UML foi aprovada como padrão pelo OMG (Object Management Group), um consórcio internacional de empresas que define e ratifica padrões na área de Orientação a Objetos.

Visão Geral da UML



Diagramas Estruturais

Diagrama de classes

Diagrama de objetos

Diagrama de componentes

Diagrama de instalação

Diagrama de pacotes

Diagrama de estrutura

Diagramas Comportamentais

Diagrama de Caso de Uso

Diagrama de transição de estados
Diagrama de atividade

Diagramas de Interação
Diagrama de sequência
Diagrama de Interatividade
Diagrama de colaboração ou comunicação
Diagrama de tempo

Diagramas
Diagrama de contexto
Diagrama de fluxos de dados
Diagrama entidade relacionamento
Lista de eventos
Tabela de decisão
Árvore de decisão
Diagrama de transição de estados

Elementos

De estrutura:
Classe
Objetos
Interface
Componente
Colaboração
Nó

De comportamento:
Casos de uso
Interação
Máquina de estados

De agrupamento:
Pacote
Modelo

Subsistema
Framework

De anotação:
Notas

Relacionamentos

Agregação
Associação (bidirecional ou unidirecional)
Composição
Generalização

2.3 Desenvolvimento ágil de software

Desenvolvimento ágil de software (do inglês *Agile software development*) ou **Método ágil** é um conjunto de metodologias de desenvolvimento de *software*. O desenvolvimento ágil, tal como qualquer metodologia de *software*, providencia uma estrutura conceitual para reger projetos de engenharia de software.

Introdução

Existem inúmeros frameworks de processos para desenvolvimento de software. A maioria dos métodos ágeis tenta minimizar o risco pelo desenvolvimento do software em curtos períodos, chamados de iteração, os quais gastam tipicamente menos de uma semana a até quatro. Cada iteração é como um projecto de software em miniatura de seu próprio, e inclui todas as tarefas necessárias para implantar o mini-incremento da nova funcionalidade: planejamento, análise de requisitos, projeto, codificação, teste e documentação. Enquanto em um processo convencional, cada iteração não está necessariamente focada em adicionar um novo conjunto significativo de funcionalidades, um projecto de software ágil busca a capacidade de implantar uma nova versão do software ao fim de cada iteração, etapa a qual a equipe responsável reavalia as prioridades do projecto.

Métodos ágeis enfatizam comunicações em tempo real, preferencialmente face a face, a documentos escritos. A maioria dos componentes de um grupo ágil deve estar agrupada em uma sala. Isso inclui todas as pessoas necessárias para terminar o software: no mínimo, os programadores e seus *clientes* (clientes são as pessoas que definem o produto, eles podem ser os gerentes, analistas de negócio, ou realmente os clientes). Nesta sala devem também se encontrar os testadores, projectistas de iteração, redactores técnicos e gerentes.

Métodos ágeis também enfatizam trabalho no software como uma medida primária de progresso. Combinado com a comunicação face-a-face, métodos ágeis produzem pouca documentação em relação a outros métodos, sendo este um dos pontos que podem ser considerados negativos. É recomendada a produção de documentação que realmente será útil.

Princípios

Os princípios do desenvolvimento ágil valorizam:

- Garantir a satisfação do consumidor entregando rapidamente e continuamente softwares funcionais;
- Softwares funcionais são entregues frequentemente (semanas, ao invés de meses);
- Softwares funcionais são a principal medida de progresso do projecto;
- Até mesmo mudanças tardias de escopo no projecto são bem-vindas.
- Cooperação constante entre pessoas que entendem do 'negócio' e desenvolvedores;
- Projetos surgem através de indivíduos motivados, entre os quais existe relação de confiança.
- Design do software deve prezar pela excelência técnica;
- Simplicidade;
- Rápida adaptação às mudanças;
- Indivíduos e interações mais do que processos e ferramentas;
- *Software* funcional mais do que documentação extensa;
- Colaboração com clientes mais do que negociação de contratos;
- Responder a mudanças mais do que seguir um plano.

História

As definições modernas de desenvolvimento de software ágil evoluíram a partir da metade de 1990 como parte de uma reação contra métodos "pesados", caracterizados por uma pesada regulamentação, regimentação e micro gerenciamento usado o modelo em cascata para desenvolvimento. O processo originou-se da visão de que o modelo em cascata era burocrático, lento e contraditório a forma usual com que os engenheiros de software sempre realizaram trabalho com eficiência.

Uma visão que levou ao desenvolvimento de métodos ágeis e iterativos era retorno a prática de desenvolvimento vistas nos primórdios da história do desenvolvimento de software.

Inicialmente, métodos ágeis eram conhecidos como *métodos leves*. Em 2001, membros proeminentes da comunidade se reuniram em Snowbird e adotaram o nome *métodos ágeis*, tendo publicado o Manifesto ágil, documento que reúne os princípios e práticas desta metodologia de desenvolvimento. Mais tarde, algumas pessoas formaram a *Agile Alliance*, uma organização não lucrativa que promove o desenvolvimento ágil.

Os métodos ágeis iniciais—criado a priore em 2000— incluíam Scrum (1986), Crystal Clear, Programação extrema (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (1995).

Comparações com outros métodos

Métodos Ágeis são algumas vezes caracterizados como o oposto de metodologias *guiadas pelo planejamento* ou *disciplinadas*. Uma distinção mais acurada é dizer que os métodos existem em um contínuo do *adaptativo* até o *preditivo*.[1] Métodos ágeis existem do lado adaptativo deste contínuo. Métodos adaptativos buscam a adaptação rápida a mudanças da realidade. Quando uma necessidade de um projeto muda, uma equipe adaptativa mudará também. Um time adaptativo terá dificuldade em descrever o que irá acontecer no futuro. O que acontecerá em uma data futura é um item de difícil predição para um método adaptativo. Uma equipe adaptativa pode relatar quais tarefas se iniciarão na próxima semana. Quando perguntado acerca de uma implantação que ocorrerá daqui a seis meses, uma equipe adaptativa deve ser capaz somente de relatar a instrução de missão para a implantação, ou uma expectativa de valor versus custo.

Métodos preditivos, em contraste, colocam o planejamento do futuro em detalhe. Uma equipe preditiva pode reportar exatamente quais aspectos e tarefas estão planejados para toda a linha do processo de desenvolvimento. Elas porém tem dificuldades de mudar de direção. O plano é tipicamente otimizado para o objetivo original e mudanças de direção podem causar a perda de todo o trabalho e determinar que seja feito tudo novamente. Equipes preditivas freqüentemente instituem um comitê de controle de mudança para assegurar que somente as mudanças mais importantes sejam consideradas.

Métodos ágeis têm muito em comum com técnicas de *Desenvolvimento rápido de aplicação* de 1980 como exposto por James Martin e outros.

Comparação com o desenvolvimento iterativo

A maioria dos métodos ágeis compartilha a ênfase no Desenvolvimento iterativo e incremental para a construção de versões implantadas do software em curtos períodos de tempo. Métodos ágeis diferem dos métodos iterativos porque seus períodos de tempo são medidos em semanas, ao invés de meses, e a realização é efetuada de uma maneira altamente colaborativa. estendendo-se a tudo.

Comparação com o modelo em cascata

O desenvolvimento ágil tem pouco em comum com o modelo em cascata. Na visão de alguns este modelo é desacreditado, apesar de ser um modelo de uso comum. O modelo em cascata é uma das metodologias com maior ênfase no planejamento, seguindo seus passos através da captura dos requisitos, análise, projeto, codificação e testes em uma seqüência pré-planejada e restrita. O progresso é geralmente medido em termos de entrega de artefatos—especificação de requisitos, documentos de projeto, planos de teste, revisão do código, e outros. O modelo em cascata resulta em uma substancial integração e esforço de teste para alcançar o fim do ciclo de vida, um período que tipicamente se estende por vários meses ou anos. O tamanho e dificuldade deste esforço de integração e teste é uma das causas das falhas do projeto em cascata. Métodos ágeis, pelo contrário, produzem um desenvolvimento completo e teste de aspectos (mas um pequeno subconjunto do todo) num período de poucas semanas ou meses. Enfatiza a obtenção de pequenos pedaços de funcionalidades executáveis para agregar valor ao negócio cedo, e continuamente agregar novas funcionalidades através do ciclo de vida do projeto.

Algumas equipes ágeis usam o modelo em cascata em pequena escala, repetindo o ciclo de cascata inteiro em cada iteração. Outras equipes, mais especificamente as equipes de Programação extrema, trabalham com atividades simultaneamente.

Comparação com a "codificação cowboy"

A codificação cowboy, também chamada de Modelo Balbúrdia, é a ausência de metodologias de desenvolvimento de Software: os membros da equipe fazem o que eles sentem que é correto. Como os desenvolvedores que utilizam métodos ágeis freqüentemente reavaliam os planos, enfatizam a comunicação face a face e fazem o uso relativamente esparso de documentos, ocasionalmente levam as pessoas a confundirem isto com codificação *cowboy*. Equipes ágeis, contudo, seguem o processo definido (e freqüentemente de forma disciplinada e rigorosa).

Como em todas as metodologias, o conhecimento e a experiência dos usuários definem o grau de sucesso e/ou fracasso de cada atividade. Os controles mais rígidos e sistematizados aplicados em um processo implicam altos níveis de responsabilidade para os usuários. A degradação de procedimentos bem-intencionados e organizados pode levar as atividades a serem caracterizadas como codificação *cowboy*.

Aplicabilidade dos métodos ágeis

Embora os métodos ágeis apresentem diferenças entre suas práticas, eles compartilham inúmeras características em comum, incluindo o desenvolvimento iterativo, e um foco na comunicação interativa e na redução do esforço empregado em artefatos intermediários. (Cohen et al., 2004) A aplicabilidade dos métodos ágeis em geral pode ser examinada de múltiplas perspectivas. Da perspectiva do produto, métodos ágeis são mais adequados quando os requisitos estão emergindo e mudando rapidamente, embora não exista um consenso completo neste ponto (Cohen et al., 2004). De uma perspectiva organizacional, a aplicabilidade pode ser expressa examinando três dimensões chaves da organização: cultura, pessoal e comunicação. Em relação a estas áreas inúmeros fatores chave do sucesso podem ser identificados (Cohen et al., 2004):

- A cultura da organização deve apoiar a negociação.
- As pessoas devem ser confiantes.
- Poucas pessoas, mas competentes.
- A organização deve promover as decisões que os desenvolvedores tomam.
- A Organização necessita ter um ambiente que facilite a rápida comunicação entre os membros.
- O fator mais importante é provavelmente o tamanho do projeto (Cohen et al., 2004).. Com o aumento do tamanho, a comunicação face a face se torna mais difícil. Portanto, métodos ágeis são mais adequados para projetos com pequenos times, com no máximo de 20 a 40 pessoas.

De forma a determinar a aplicabilidade de métodos ágeis específicos, uma análise mais sofisticada é necessária. O método dinâmico para o desenvolvimento de sistemas, por exemplo,

provê o denominado 'filtro de aplicabilidade' para este propósito. Também, a família de métodos Crystal provê uma caracterização de quando selecionar o método para um projeto. A seleção é baseada no tamanho do projeto, criticidade e prioridade. Contudo, outros métodos ágeis não fornecem um instrumento explícito para definir sua aplicabilidade a um projeto.

Alguns métodos ágeis, como DSDM e Feature Driven Development, afirmam se aplicar a qualquer projeto de desenvolvimento ágil, sem importar suas características (Abrahamsonn et al., 2003).

A comparação dos métodos ágeis irá revelar que eles suportam diferentes fases de um ciclo de vida do software em diferentes níveis. Estas características individuais dos métodos ágeis podem ser usadas como um critério de seleção de sua aplicabilidade.

Desenvolvimentos ágeis vêm sendo amplamente documentados (ver Experiências relatadas, abaixo, como também em Beck, e Boehm & Turner) como funcionando bem para equipes pequenas (< 10 desenvolvedores). O desenvolvimento ágil é particularmente adequado para equipes que têm que lidar com mudanças rápidas ou imprevisíveis nos requisitos.

A aplicabilidade do desenvolvimento ágil para os seguintes cenários é ainda uma questão aberta:

- esforços de desenvolvimento em larga escala (> 20 desenvolvedores), embora estratégias para maiores escalas tenham sido descritas.

- esforços de desenvolvimento distribuído (equipes não co-alocadas). Estas estratégias tem sido descritas em *Bridging the Distance* e *Using an Agile Software Process with Offshore Development*.

- esforços críticos de missão e vida.

- Companhias com uma cultura de comando e controle.

Barry Boehm e Richard Turner sugeriram que análise de risco pode ser usada para escolher entre métodos adaptativos ("ágeis") e preditivos ("dirigidos pelo planejamento"). Os autores sugerem que cada lado deste contínuo possui seu *ambiente ideal"*

Ambiente ideal para o desenvolvimento ágil:

- Baixa criticidade
- Desenvolvedores senior
- Mudanças freqüente de requisitos
- Pequeno número de desenvolvedores
- Cultura que tem sucesso no caos.
- Ambiente ideal para o desenvolvimento direcionado ao planejamento:
- Alta criticidade
- Desenvolvedores Junior
- Baixa mudança nos requisitos
- Grande número de desenvolvedores

- Cultura que procura a ordem.

Adaptabilidade dos métodos ágeis

Um método deve ser bastante flexível para permitir ajustes durante a execução do projeto. Há três problemas chaves relacionados ao tópico de adaptação dos métodos ágeis: **a aplicabilidade dos métodos ágeis** (no geral e no particular), e finalmente, o **suporte ao gerenciamento de projeto**.

Métodos ágeis e o gerenciamento de projeto

Os métodos ágeis diferem largamente no que diz respeito a forma de serem gerenciados. Alguns métodos são suplementados com guias para direcionar o gerenciamento do projeto, mas nem todos são aplicáveis..

PRINCE2™ tem sido considerado como um sistema de gerenciamento de projeto complementar e adequado.

Uma característica comum dos processos ágeis é a capacidade de funcionar em ambientes muito exigentes que tem um grande número de incertezas e flutuações (mudanças) que podem vir de várias fontes como: equipe em processo de formação que ainda não trabalhou junto em outros projetos, requisitos voláteis, baixo conhecimento do domínio de negócio pela equipe, adoção de novas tecnologias, novas ferramentas, mudanças muito bruscas e rápidas no ambiente de negócios das empresas: novos concorrentes, novos produtos, novos modelos de negócio.

Sistemas de gerenciamento de projetos lineares e prescritivos, neste tipo de ambiente, falham em oferecer as características necessárias para responder de forma ágil as mudanças requeridas. Sua adoção pode incrementar desnecessariamente os riscos, o custo, o prazo e baixar a qualidade do produto gerado, desgastando a equipe e todos os envolvidos no processo.

A abordagem Scrum, para gestão de projetos ágeis, leva em consideração planejamento não linear, porém de maneira mais exaustiva e está focada em agregar valor para o cliente e em gerenciar os riscos, fornecendo um ambiente seguro. Pode ser utilizada na gestão do projeto aliada a uma metodologia de desenvolvimento como Programação Extrema, FDD, OpenUP, DSDM, Crystal ou outras.

Metodologias

- Programação extrema
- Scrum
- Feature Driven Development
- DSDM
- Adaptive Software Development
- Crystal
- Pragmatic Programming
- Test Driven Development

Críticas

O método de desenvolvimento ágil é algumas vezes criticado como codificação cowboy. O início da Programação extrema soava como controverso e dogmático, tal como a programação por pares e o projeto contínuo, tem sido alvo particular de críticos, tais como McBreen e Boehm e Turner. Contudo, muitas destas críticas têm sido vistas pelos defensores dos métodos ágeis como mal entendidos a respeito do desenvolvimento ágil.

Em particular, a Programação extrema é revista e criticada por Matt Stephens' Extreme Programming Refactored.

As críticas incluem

- falta de estrutura e documentação necessárias
- somente trabalhar com desenvolvedores de nível sênior
- incorpora de forma insuficiente o projeto de software
- requer a adoção de muita mudança cultural
- poder levar a maiores dificuldades nas negociações contratuais

2.4 Feature Driven Development (FDD)

O **Desenvolvimento Guiado por Funcionalidades** (do inglês, *Feature Driven Development*; FDD) é uma das seis metodologias ágeis originais do desenvolvimento de software. Seus representantes redigiram o Manifesto Ágil para Desenvolvimento de Software, em 2001. Nessa ocasião, o representante da FDD foi Jon Kern, que trabalhava na TogetherSoft, substituindo Peter Coad.

Origens

A FDD nasceu num projeto em Cingapura, entre 1997 e 1999, a partir do Método Coad (uma metodologia completa para Análise, Desenho e Programação Orientados por Objetos, desenvolvida por Peter Coad nas décadas de 1980 e 1990) e das técnicas de gerenciamento iterativo, incremental e enxuto de projetos utilizadas por Jeff De Luca, um gerente de projetos australiano.

Seu lema é **"Resultados freqüentes, tangíveis e funcionais"**.

A primeira descrição oficial dos processos foi publicada no livro "Java Modeling in Color with UML", por Peter Coad, Eric Lefebvre e Jeff De Luca, em 1999.

O livro de referência é "A Practical Guide to Feature-Driven Development", por Stephen Palmer e John Mac Felsing, publicado em 2002 pela Prentice-Hall, compondo uma série editada pelo próprio Peter Coad.

FDD e a Família Ágil

Com relação às outras metodologias de desenvolvimento de software, situa-se numa posição intermediária entre as abordagens mais prescritivas (Processo Unificado, Cascata tradicional - Waterfall) e as abordagens Ágeis (XP - Programação Extrema, Scrum, Crystal, etc.).

Oferece um conjunto coeso de princípios e práticas tanto para a Gestão de Projetos quanto para a Engenharia de Software, mas convive bem com abordagens mais especialistas, como Scrum.

Apesar de algumas divergências pontuais com a XP, várias práticas propostas por esta última também são utilizadas por equipes usando FDD, como os testes unitários, refatoração, programação em pares, integração contínua, entre outras. Apenas a ênfase na FDD é que não é tão grande quanto na XP. A FDD também propõe práticas como inspeção formal (de desenho e de código) e posse individual/situacional de código/classe, que podem contrastar com algumas das práticas fundamentais da XP. A experiência da equipe e dos gerentes é que deve julgar quais práticas são mais apropriadas.

Os Processos

Os cinco processos da FDD

A FDD é, classicamente, descrita por cinco processos:

Desenvolver um Modelo Abrangente: pode envolver desenvolvimento de requisitos, análise orientada por objetos, modelagem lógica de dados e outras técnicas para entendimento do domínio de negócio em questão. O resultado é um modelo de objetos (e/ou de dados) de alto nível, que guiará a equipe durante os ciclos de construção.

Construir uma Lista de Funcionalidades: decomposição funcional do modelo do domínio, em três camadas típicas: áreas de negócio, atividades de negócio e passos automatizados da atividade (funcionalidades). O resultado é uma hierarquia de funcionalidades que representa o produto a ser construído (também chamado de *product backlog*, ou lista de espera do produto).

Planejar por Funcionalidade: abrange a estimativa de complexidade e dependência das funcionalidades, também levando em consideração a prioridade e valor para o negócio/cliente. O resultado é um plano de desenvolvimento, com os pacotes de trabalho na seqüência apropriada para a construção.

Detalhar por Funcionalidade: já dentro de uma iteração de construção, a equipe detalha os requisitos e outros artefatos para a codificação de cada funcionalidade, incluindo os testes. O projeto para as funcionalidades é inspecionado. O resultado é o modelo de domínio mais detalhado e os esqueletos de código prontos para serem preenchidos.

Construir por Funcionalidade: cada esqueleto de código é preenchido, testado e inspecionado. O resultado é um incremento do produto integrado ao repositório principal de código, com qualidade e potencial para ser usado pelo cliente/usuário.

2. 2.5 Enterprise Unified Process (EUP) – É uma extensão do RUP.

3. 2.6 Scrum (Scrum)

O **Scrum** é um processo de desenvolvimento iterativo e incremental para gerenciamento de projetos e desenvolvimento ágil de software. Apesar de a palavra não ser um acrônimo, algumas empresas que implementam o processo a soletram com letras maiúsculas como SCRUM. Isto pode ser devido aos primeiros artigos de Ken Schwaber, que capitalizava SCRUM no título.

Scrum não é um processo prescribente, ou seja, ele não descreve o que fazer em cada situação. Ele é usado para trabalhos complexos nos quais é impossível predizer tudo o que irá ocorrer.

Apesar de Scrum ter sido destinado para gerenciamento de projetos de software, ele pode ser utilizado em equipes de manutenção de software ou como uma abordagem geral de gerenciamento de projetos/programas.

História

Inicialmente, o SCRUM foi concebido como um estilo de gerenciamento de projetos em empresas de fabricação de automóveis e produtos de consumo, por Takeuchi e Nonaka no artigo "The New Product Development Game" (Harvard Business Review, Janeiro-Fevereiro 1986). Eles notaram que projetos usando equipes pequenas e multidisciplinares (*cross-functional*) produziram os melhores resultados, e associaram estas equipes altamente eficazes à formação Scrum do Rugby (utilizada para reinício do jogo em certos casos). Jeff Sutherland, John Scumniotales e Jeff McKenna conceberam, documentaram e implementaram o Scrum, conforme descrito abaixo, na empresa Easel Corporation em 1993, incorporando os estilos de gerenciamento observados por Takeuchi e Nonaka. Em 1995, Ken Schwaber formalizou a definição de *Scrum* e ajudou a implantá-lo no desenvolvimento de softwares em todo o mundo.

Scrum junta conceitos de Lean, desenvolvimento iterativo e do estudo de Hirotaka Takeuchi e Ikujiro Nonaka.

A função primária do Scrum é ser utilizado para o gerenciamento de projetos de desenvolvimento de software. Ele tem sido usado com sucesso para isso, assim como Extreme Programming e outras metodologias de desenvolvimento. Porém, teoricamente pode ser aplicado em qualquer contexto no qual um grupo de pessoas necessitem trabalhar juntas para atingir um objetivo comum, como iniciar uma escola pequena, projetos de pesquisa científica, ou até mesmo o planejamento de um casamento.

Mesmo que idealizado para ser utilizado em gestão de projetos de desenvolvimento de software ele também pode ser usado para a gerência de equipes de manutenção, ou como uma abordagem para gestão de programas: *Scrum de Scrums*.

Características

Scrum é um esqueleto de processo que contém grupos de práticas e papéis pré-definidos. Os principais papéis são:

- o **ScrumMaster**, que mantém os processos (normalmente no lugar de um gerente de projeto)

- o **Proprietário do Produto**, ou **Product Owner**, que representa os *stakeholders* e o negócio

- a **Equipe**, ou **Team**, um grupo multifuncional com cerca de 7 pessoas e que fazem a análise, projeto, implementação, teste etc.

Sprint (corrida)

Um sprint é a unidade básica de desenvolvimento em Scrum. Sprints tendem a durar entre uma semana e um mês, e são um esforço dentro de uma "caixa de tempo" (ou seja, restrito a uma duração específica) de um comprimento constante.

Cada sprint é precedido por uma reunião de planejamento, onde as tarefas para o sprint são identificadas e um compromisso estimado para o objetivo do sprint é definido e seguido por uma reunião de revisão ou de retrospectiva, onde o progresso é revisto e lições para os próximos sprints são identificadas.

Durante cada sprint, a equipe cria um incremento de produto potencialmente entregável (por exemplo, software funcional e testado). O conjunto de funcionalidades que entram em um sprint vêm do "backlog" do produto, que é um conjunto de prioridades de requisitos de alto nível do trabalho a ser feito. Quais itens do backlog entram para o sprint são determinados durante a reunião de planejamento do sprint. Durante esta reunião, o **Product Owner** informa a equipe dos itens no backlog do produto que ele ou ela quer concluídos. A equipe então determina quantos eles podem se comprometer a concluir durante o próximo sprint, e registram isso no backlog do sprint. Durante um sprint, ninguém está autorizado a alterar o backlog do sprint, o que significa que os requisitos são congelados para esse sprint. O desenvolvimento está dentro de uma caixa de tempo, o que significa que o sprint deve terminar a tempo. Se os requisitos não são completados por qualquer motivo, eles são deixados de fora e voltam para o backlog do produto. Depois que um sprint é completado, a equipe demonstra como usar o software.

O Scrum permite a criação de equipes auto-organizadas, encorajando a co-localização de todos os membros da equipe e a comunicação verbal entre todos os membros e disciplinas da equipe no projeto.

Um princípio chave do Scrum é o reconhecimento de que, durante um projeto, os clientes podem mudar de idéia sobre o que eles querem e precisam (muitas vezes chamados requisitos churn), e que os desafios imprevisíveis não podem ser facilmente tratados de uma maneira preditiva ou planejada tradicional. Como tal, o Scrum adota uma abordagem empírica, aceitando que o problema não pode ser totalmente entendido ou definido, focando na maximização da habilidade da equipe para entregar rapidamente e responder às necessidades emergentes.

Como outras metodologias de desenvolvimento ágil, o Scrum pode ser implementado através de uma ampla gama de ferramentas. Muitas empresas utilizam ferramentas de software universal, como planilhas para construir e manter artefatos como o backlog do sprint. Há também pacotes de software open-source e proprietários dedicados à gestão de produtos no âmbito do processo Scrum. Outras organizações implementam o Scrum sem o uso de quaisquer ferramentas de software, e mantêm seus artefatos na forma de cópias impressas, como papel, quadros e notas.

Cada *sprint* é uma iteração que segue um ciclo (PDCA) e entrega incremento de software pronto.

Um *backlog* é conjunto de requisitos, priorizado pelo Product Owner (responsável pelo ROI e por conhecer as necessidades do cliente);

Há entrega de conjunto fixo de itens do *backlog* em série de interações curtas ou *sprints*;

Breve reunião diária, ou daily scrum, em que cada participante fala sobre o progresso conseguido, o trabalho a ser realizado e/ou o que o impede de seguir avançando (também chamado de Standup Meeting ou Daily Meeting, já que os membros da equipe geralmente ficam em pé para não prolongar a reunião).

Breve sessão de planejamento, na qual os itens do *backlog* para uma *sprint* (iteração) são definidos;

Retrospectiva, na qual todos os membros da equipe refletem sobre a *sprint* passada.

O Scrum é facilitado por um *Scrum Master*, que tem como função primária remover qualquer impedimento à habilidade de uma equipe de entregar o objetivo do *sprint*. O Scrum Master não é o líder da equipe (já que as equipes são auto-organizadas), mas atua como um mediador entre a equipe e qualquer influência desestabilizadora. Outra função extremamente importante de um Scrum Master é o de assegurar que a equipe esteja utilizando corretamente as práticas de Scrum, motivando-os e mantendo o foco na meta da Sprint.

Uma das grandes vantagens do Scrum, porém, é que não tem abordagem "receita de bolo" do gerenciamento de projetos exemplificado no Project Management Body of Knowledge ou PRINCE2, que tem como objetivos atingir qualidade através da aplicação de uma série de processos prescritos.

Papéis

Equipes Scrum consistem de três papéis principais e uma série de papéis auxiliares - papéis principais são frequentemente referidos como *porcos* e papéis auxiliares como *galinhas* (após a história A Galinha e o Porco).

Papéis principais

Os papéis principais em equipes Scrum são aqueles comprometidos com o projeto no processo do Scrum - são os que produzem o produto (objetivo do projeto).

Product Owner (dono do produto)

O Product Owner representa a voz do cliente e é responsável por garantir que a equipe agregue valor ao negócio. O Product Owner escreve centrado nos itens do cliente (histórias tipicamente do usuário), os prioriza e os adiciona para o product backlog. Equipes de Scrum devem ter um Product Owner, e, embora esse possa também ser um membro da equipe de desenvolvimento, recomenda-se que este papel não seja combinado com o de ScrumMaster.

Equipe (*Development Team*)

A equipe é responsável pela entrega do produto. A equipe é tipicamente composta de 5-9 pessoas com habilidades multifuncionais que fazem o trabalho real (analisar, projetar, desenvolver, testar técnicas de comunicação, documentos, etc.) Recomenda-se que a equipe seja auto-organizada e auto-conduzida, mas que muitas vezes trabalhem com alguma forma de projeto ou gestão de equipe.

Scrum Master

Scrum é facilitado por um Scrum Master, também escrito como Scrum Master, que é responsável pela remoção de impedimentos à capacidade da equipe para entregar o objetivo do sprint / entregas. O Scrum Master não é o líder da equipe, mas age como um tampão entre a equipe e qualquer influência ou distração. O Scrum Master garante que o processo Scrum seja usado como pretendido. O Scrum Master é o responsável pela aplicação das regras. Uma parte fundamental do papel do Scrum Master é proteger a equipe e mantê-la focada nas tarefas em mãos. O papel também tem sido referido como um líder-servo para reforçar essa dupla perspectiva.

Papéis auxiliares

Os papéis auxiliares em equipes Scrum são aqueles com nenhum papel formal e envolvimento frequente no processo de Scrum, mas, ainda assim, devem ser levados em conta.

Partes interessadas (clientes, fornecedores)

Estas são as pessoas que permitem o projeto e para quem o projeto vai produzir o acordado benefício, que justifica a sua produção. Eles só estão diretamente envolvidos no processo durante as revisões sprint.

Gerentes (incluindo gerentes de projeto)

Pessoas que irão configurar o ambiente para desenvolvimento de produtos.

Gerenciamento Ágil de Projetos com Scrum

Scrum não só reforçou o interesse em gerenciamento de projetos de software, mas também desafiou as idéias convencionais sobre essa gestão. Scrum é voltado para instituições de gerenciamento de projetos, onde é difícil planejar o futuro com mecanismos de controle de processos empíricos, como loops de feedback, onde constituem o elemento central do desenvolvimento do produto em comparação com a gestão de comando e controle tradicionais orientado. Ela representa uma abordagem radicalmente nova para o planejamento e gerenciamento de projetos de software, trazendo poder de decisão ao nível das propriedades operação e certezas. Scrum reduz defeitos e torna o processo de desenvolvimento mais eficiente, bem como reduzindo os custos de manutenção a longo prazo.

Artefatos

Product Backlog

Um *backlog* é uma lista de itens priorizados a serem desenvolvidos para um software. O *Product Backlog* é mantido pelo *Product Owner* e é uma lista de requisitos que tipicamente vêm do cliente. O *Product Backlog* pode ser alterado a qualquer momento pelo *Product Owner* ou por decisão deste.

Sprint backlog

O *Sprint backlog* é uma lista de itens selecionados do *Product backlog* e contém tarefas concretas

que serão realizadas durante o próximo *sprint* para implementar tais itens selecionados. O *Sprint Backlog* é uma representação em tempo real do trabalho que o *Development Team* planeja concluir na *sprint* corrente, e ele pertence unicamente ao *Development Team*.

Planejamento de *sprint*

Antes de todo *sprint*, o Product Owner, o Scrum Master e a Equipe decidem no que a equipe irá trabalhar durante o próximo *sprint*. O Product Owner mantém uma lista priorizada de itens de *backlog*, o *backlog* do produto, o que pode ser repriorizado durante o planejamento do *sprint*. A Equipe seleciona itens do topo do *backlog* do produto. Eles selecionam somente o quanto de trabalho eles podem executar para terminar. A Equipe então planeja a arquitetura e o design de como o *backlog* do produto pode ser implementado. Os itens do *backlog* do produto são então destrinchados em tarefas que se tornam o *backlog* do *sprint*.

Reuniões

Daily Scrum

Cada dia durante o sprint, uma reunião de status do projeto ocorre. Isso é chamado de "scrum diário", ou "de pé o dia". Esta reunião tem diretrizes específicas:

- A reunião começa precisamente no horário marcado.
- Todos são bem-vindos, mas apenas "poucos" podem falar.
- O encontro tem duração determinada (Time-Box) e dura 15 minutos.
- A reunião deve acontecer no mesmo local e mesma hora todos os dias

Durante a reunião, cada membro da equipe responde a três perguntas:

- O que você tem feito desde ontem?
- O que você está planejando fazer hoje?
- Você tem algum problema impedindo você de realizar seu objetivo?

É papel do Scrum Master para facilitar a resolução desses impedimentos. Normalmente, isso deve ocorrer fora do contexto do Daily Scrum para que a reunião possa durar menos de 15 minutos.

Reunião de Planejamento da Sprint (*Sprint Planning Meeting*)

No início do ciclo de sprint (a cada 7-30 dias), um *Sprint Planning Meeting* é realizado.

Selecione o trabalho que está a ser feito.

Prepare o Sprint Backlog que detalha o tempo que levará para fazer esse trabalho, com toda a equipe.

Identificar e comunicar o quanto o trabalho é susceptível de ser feito durante o sprint atual.

Dividida em duas partes:

Parte 1 (Primeiras quatro horas): Team Product Owner: diálogo para priorizar o Product Backlog.

Parte 2 (Próximas quatro horas): Team apenas: hash de um plano para a Sprint, resultando na Sprint Backlog.

No final de um ciclo de sprint, são realizadas duas reuniões: a "Sprint Review" e do "Sprint Retrospective".

Reunião de Revisão da Sprint (*Sprint Review*)

Rever o trabalho que foi concluído e não concluído.

Apresentar o trabalho realizado para os interessados (ou "a demo"). Um trabalho incompleto não pode ser demonstrado.

Retrospectiva da Sprint (*Sprint Retrospective*)

Todos os membros da equipe refletem sobre a sprint passada.

Faça melhorias contínuas de processos.

Duas questões principais são feitas na retrospectiva do sprint: O que correu bem durante a corrida? O que poderia ser melhorado na próxima sprint?

Scrum simplificado

Muitas organizações têm sido resistentes às metodologias introduzidas em baixos níveis da organização. Porém, a adaptabilidade do Scrum permite que ele seja introduzido de forma invisível (*"stealth"*), usando os três passos:

Agende uma demonstração do software com seu cliente em um mês a partir de agora;

Como equipe, tome um mês para deixar o software pronto para uma demo, com funcionalidades prontas, não simplesmente telas;

Na demonstração, obtenha *feedback* e use-o para guiar o seu próximo mês de trabalho de desenvolvimento.

Algumas características de Scrum

- Clientes se tornam parte da equipe de desenvolvimento (os clientes devem estar genuinamente interessados na saída);
- Entregas frequentes e intermediárias de funcionalidades 100% desenvolvidas;
- Planos frequentes de mitigação de riscos desenvolvidos pela equipe;
- Discussões diárias de status com a equipe;
- A discussão diária na qual cada membro da equipe responde às seguintes perguntas:
    - O que fiz desde ontem?

- O que estou planejando fazer até amanhã?
- Existe algo me impedindo de atingir minha meta?

Transparência no planejamento e desenvolvimento;

Reuniões frequentes com os *stakeholders* (todos os envolvidos no processo) para monitorar o progresso;

Problemas não são ignorados e ninguém é penalizado por reconhecer ou descrever qualquer problema não visto;

Locais e horas de trabalho devem ser energizadas, no sentido de que "trabalhar horas extras" não necessariamente significa "produzir mais".

Agendando discussões diárias

Um momento bom para as discussões diárias é depois do almoço. Durante a manhã pode ser complicado. Estas discussões de status não demoram e uma forma eficiente de fazer estas reuniões seria ficar em pé e em frente a um quadro negro. Como as pessoas tendem a ficar cansadas depois do almoço, ter uma viva reunião em pé nessa hora permite que a equipe mantenha a sua energia alta. Como todos estiveram trabalhando durante a manhã, suas mentes estão focadas no trabalho e não em questões pessoais. Grandes usuários do processo são enfáticos na necessidade de os membros da equipe estarem em pé durante a reunião, para permitir maior agilidade e evitar perdas no foco. Recomenda-se inclusive evitar lugares onde as pessoas possam se apoiar.

Scrum Solo

Scrum é baseado em pequenas equipes. Ele permite a comunicação entre os membros da equipe. Entretanto, há uma grande quantidade de softwares desenvolvidos por programadores solos. Um software sendo desenvolvido por um só programador pode ainda se beneficiar de alguns princípios do Scrum, como: um *backlog* de produto, um *backlog* de *sprint*, um *sprint* e uma retrospectiva de *sprint*. Scrum Solo é uma versão adaptada para uso de programadores solo.

2.7 Programação extrema (XP)

Programação extrema (do inglês *eXtreme Programming*), ou simplesmente **XP**, é uma metodologia ágil para equipes pequenas e médias e que irão desenvolver software com requisitos vagos e em constante mudança. Para isso, adota a estratégia de constante acompanhamento e realização de vários pequenos ajustes durante o desenvolvimento de software.

Os cinco valores fundamentais da metodologia **XP** são: comunicação, simplicidade, *feedback*, coragem e respeito. A partir desses valores, possui como princípios básicos: *feedback* rápido, presumir simplicidade, mudanças incrementais, abraçar mudanças e trabalho de qualidade.

Dentre as variáveis de controle em projetos (custo, tempo, qualidade e escopo), há um foco explícito em escopo. Para isso, recomenda-se a priorização de funcionalidades que representem maior valor possível para o negócio. Desta forma, caso seja necessário a diminuição de escopo, as funcionalidades menos valiosas serão adiadas ou canceladas.

A **XP** incentiva o controle da qualidade como variável do projeto, pois o pequeno ganho de curto prazo na produtividade, ao diminuir qualidade, não é compensado por perdas (ou até impedimentos) a médio e longo prazo.

Valores

Comunicação

Simplicidade

Feedback

Coragem

Respeito

Princípios básicos

Feedback rápido

Presumir simplicidade

Mudanças incrementais

Abraçar mudanças

Trabalho de alta qualidade.

Práticas

Para aplicar os valores e princípios durante o desenvolvimento de *software*, **XP** propõe uma série de práticas. Há uma confiança muito grande na sinergia entre elas, os pontos fracos de cada uma são superados pelos pontos fortes de outras.

Jogo de Planejamento (*Planning Game*): O desenvolvimento é feito em iterações semanais. No início da semana, desenvolvedores e cliente reúnem-se para priorizar as funcionalidades. Essa reunião recebe o nome de Jogo do Planejamento. Nela, o cliente identifica prioridades e os desenvolvedores as estimam. O cliente é essencial neste processo e assim ele fica sabendo o que está acontecendo e o que vai acontecer no projeto. Como o escopo é reavaliado semanalmente, o projeto é regido por um contrato de escopo negociável, que difere significativamente das formas tradicionais de contratação de projetos de software. Ao final de cada semana, o cliente recebe novas funcionalidades, completamente testadas e prontas para serem postas em produção.

Pequenas Versões (*Small Releases*): A liberação de pequenas versões funcionais do projeto auxilia muito no processo de aceitação por parte do cliente, que já pode testar uma parte do sistema que está comprando. As versões chegam a ser ainda menores que as produzidas por outras metodologias incrementais, como o RUP.

Metáfora (*Metaphor*): Procura facilitar a comunicação com o cliente, entendendo a realidade dele. O conceito de rápido para um cliente de um sistema jurídico é diferente para um programador experiente em controlar comunicação em sistemas em tempo real, como controle de tráfego aéreo. É preciso traduzir as palavras do cliente para o significado que ele espera dentro do projeto.

Projeto Simples (*Simple Design*): Simplicidade é um princípio da XP. Projeto simples significa dizer que caso o cliente tenha pedido que na primeira versão apenas o usuário "teste" possa entrar no sistema com a senha "123" e assim ter acesso a todo o sistema, você vai fazer o código exato para que esta funcionalidade seja implementada, sem se preocupar com sistemas de autenticação e restrições de acesso. Um erro comum ao adotar essa prática é a confusão por parte dos programadores de código *simples* e código *fácil*. Nem sempre o código mais fácil de ser desenvolvido levará a solução mais simples por parte de projeto. Esse entendimento é fundamental para o bom andamento do XP. Código fácil deve ser identificado e substituído por código simples.

Time Coeso (*Whole Team*): A equipe de desenvolvimento é formada por pessoas engajadas e de forma multidisclipinar (no sentido de incluir pessoas com cada uma das habilidades necessárias para o projeto).

Testes de Aceitação (*Customer Tests*): São testes construídos pelo cliente e conjunto de analistas e testadores, para aceitar um determinado requisito do sistema.

Ritmo Sustentável (*Sustainable Pace*): Trabalhar com qualidade, buscando ter ritmo de trabalho saudável (40 horas/semana, 8 horas/dia), sem horas extras. Horas extras são permitidas quando trouxerem produtividade para a execução do projeto. Outra prática que se verifica neste processo é a prática de trabalho energizado, onde se busca trabalho motivado sempre. Para isto o ambiente de trabalho e a motivação da equipe devem estar sempre em harmonia.

Reuniões em pé (*Stand-up Meeting*): Reuniões em pé para não se perder o foco nos assuntos, produzindo reuniões rápidas, apenas abordando tarefas realizadas e tarefas a realizar pela equipe.

Posse Coletiva (*Collective Ownership*): O código fonte não tem dono e ninguém precisa solicitar permissão para poder modificar o mesmo. O objetivo com isto é fazer a equipe conhecer todas as partes do sistema.

Programação em Pares (*Pair Programming*): é a programação em par/dupla num único computador. Geralmente a dupla é formada por um iniciante na linguagem e outra pessoa funcionando como um instrutor. Como é apenas um computador, o novato é que fica à frente fazendo a codificação, e o instrutor acompanha ajudando a desenvolver suas habilidades. Desta forma o programa sempre é revisto por duas pessoas, evitando e diminuindo assim a possibilidade de defeitos. Com isto busca-se sempre a evolução da equipe, melhorando a qualidade do código fonte gerado.

Padrões de Codificação (*Coding Standards*): A equipe de desenvolvimento precisa estabelecer regras para programar e todos devem seguir estas regras. Desta forma parecerá que todo o código fonte foi editado pela mesma pessoa, mesmo quando a equipe possui 10 ou 100 membros.

Desenvolvimento Orientado a Testes (*Test Driven Development*): Primeiro crie os testes unitários (*unit tests*) e depois crie o código para que os testes funcionem. Esta abordagem é complexa no início, pois vai contra o processo de desenvolvimento de muitos anos. Só que os testes unitários são essenciais para que a qualidade do projeto seja mantida.

Refatoração (*Refactoring*): É um processo que permite a melhoria continua da programação, com o mínimo de introdução de erros e mantendo a compatibilidade com o código já existente. Refabricar melhora a clareza (leitura) do código, divide-o em módulos mais coesos e de maior reaproveitamento, evitando a duplicação de código-fonte;

Integração Contínua (*Continuous Integration*): Sempre que produzir uma nova funcionalidade, nunca esperar uma semana para integrar à versão atual do sistema. Isto só aumenta a possibilidade de conflitos e a possibilidade de erros no código fonte. Integrar de forma contínua permite saber o status real da programação.

Parte 3 Outras Metodologias

Microsoft Solution Framework (MSF)

A MSF foi criado em 1994, e originou-se da análise de times de projetos e grupo de produtos, estas análises eram constatadas com a indústria de práticas e métodos. Estes resultados combinados eram consolidados em melhores praticas entre pessoas e processos.

O MSF (Microsoft Solutions Framework) tem sido usado pela Microsoft como o seu "método" para desenvolvimento de soluções de software dentro da Microsoft e também para os milhares de clientes e parceiros da Microsoft em todo o mundo.

A disseminação deste método, agora na versão 4.0 no Visual Studio 2005, normalmente induz as pessoas a compará-lo com outros "métodos" da indústria, como o RUP ou XP, entre outros. É importante entender, entretanto, o que são estes elementos antes de compará-los.

O MSF para Desenvolvimento Ágil de Software é um guia de procedimentos, uma coleção de boas práticas para projetos de desenvolvimento de softwares.

Um projeto MSF é regido por ciclos ou iterações. A cada ciclo, cada componente da equipe executa suas funções e atualiza o resultado do seu trabalho conforme a necessidade. Os ciclos se repetem até que o projeto seja concluído ou cada versão seja lançada.

Cada componente da equipe será responsável por um ou mais papéis, dependendo do tamanho ou da complexidade do projeto.

A Microsoft não classifica o MSF como uma metodologia, mas sim como uma disciplina. O que isso quer dizer? Basicamente que o MSF serve como um grande guia e uma coleção de boas práticas. Porém, o MSF não se aprofunda em detalhes.

Por exemplo, em um dado momento do projeto, o MSF diz que você terá que fazer uma especificação funcional.

Entretanto, ele não define se você deve usar UML, análise essencial ou outras técnicas. Isso fica a seu critério.

A falta de detalhes do MSF pode parecer uma deficiência a princípio, mas essa característica permitiu uma abordagem simples e direta das técnicas apresentadas. Ou seja, o MSF permite uma fácil compreensão tanto por parte da equipe como do cliente, além de ser bastante flexível em sua aplicação.

PRINCÍPIOS DA MSF

Foco no negócio: Entender porque o projeto existe da perspectiva do negócio e como este valor é medido.

Comunicação: MSF aconselha a comunicação aberta em toda a equipe, clientes e outros componentes do time.

Visão de projeto compartilhado: O processo de compartilhamento de visão de projeto é especificado no início do projeto. Na criação desta visão o time se comunica no intuito de identificar e resolver conflitos e resolver visões enganosas. Isto permite definir a direção do projeto.

Esclarecer as responsabilidades compartilhadas: Todo o time compartilha várias responsabilidades para ensinar ao time e seu relacionamento aos respectivos stakeholders.

Mais poderes aos membros do time: Baseado em time de pares MSF dá poderes aos membros do time por ter que atingir as metas e entregas, aceitando o fato de terem as responsabilidades compartilhadas por tomar decisões, direções quando necessário.

Agilidade: As iterações do ciclo de vida do modelo de processo habilitam ajustes de cursos para a entrega do projeto em cada milestone.

Investimento em qualidade: MSF tem por premissa que todo o time é responsável por balancear os custos, e funcionalidades para preservar a solução em qualidade e assegurar a qualidade. Membros do time precisam construir qualidade em todas as fases até o sucesso da solução, e por sua vez a organização deve investir em seu time em educação, treinamento, e experiência.

Aprender com todas as experiências: Nos últimos 20 anos houve um crescimento colossal no que diz respeito à taxa de sucesso de projetos. Dados que a maior causa de falha são praticamente os mesmos, as organizações de TI não aprendem com as suas falhas de projeto. O MSF engloba o conceito de contínuo crescimento baseado em aprendizado individual e de time.

GERENCIAMENTO DE PREPARAÇÃO No início de um projeto da solução, antes da fase de visão/escopo, a organização precisa ter um claro entendimento destes aspectos:

Seu cenário e requisitos de segurança específicos: para atender às necessidades das organizações que iniciam implementações de soluções de segurança, o Microsoft Solutions for Security (MSS) criou o SRMG. O SRMG é um processo detalhado usado para determinar quais ameaças e vulnerabilidades têm o maior impacto potencial em uma determinada organização. Como cada empresa tem requisitos comerciais diferentes, é impossível criar uma lista de vulnerabilidades que tenham o mesmo impacto em todos os ambientes. Por isso, o SRMG permite que uma organização aumente de forma incremental sua segurança e identifique áreas potenciais que precisam de correção no futuro.

Suas competências internas: o objetivo desta solução é ser facilmente entendida e imediatamente implementada por um Microsoft Certified Systems Engineer (MCSE) com boa experiência e pelo menos uma familiaridade básica com os materiais do Microsoft Official Curriculum (MOC).

MODELOS MSF: TIME E PROCESSOS Modelo de Time (Team Model) habilita a escalabilidade do projeto, identifica quem vai trabalhar durante o projeto e linca cada time com um responsável Modelo de Processo (Process Model) provê a alta qualidade através do ciclo de vida do projeto (Disciplina de Gerência de Riscos; Disciplina de Gerência de Projetos). O process model trabalha em conjunto com o Team Model organizando o processo em fases distintas criação, teste, publicação.

MSF VERSÂO 4.0

A Microsoft Solution Framework versão 4.0 é uma combinação de um metamodelo, que pode ser usado como base para processos de engenharia de software prescritivo, e dois personalizável e escalável processos de engenharia de software. Os MSF metamodelo consiste de princípios fundamentais, uma equipa modelo e ciclos e iterações. MSF 4.0 fornece uma ferramenta de alto nível quadro de orientações e princípios que podem ser mapeados para uma variedade de modelos prescritivos processo.

O programa está estruturado em duas metodologias descritivas e prescritivas. A componente descritiva é chamado a MSF 4,0 metamodelo, que é uma descrição teórica do SDLC melhores práticas para a criação de metodologias SDLC. Microsoft é da opinião de que as organizações têm diferentes dinâmicas e contrariando as suas prioridades durante o desenvolvimento de software; algumas organizações necessitam de uma sensível e softwares adaptáveis ambiente de desenvolvimento, enquanto outros precisam de uma padronizado, repetitivo e mais ambiente controlado.

Para atender estas necessidades, a Microsoft representa o metamodelo de 4,0 MSF em dois modelos que prevêem prescritivo metodologia específica processo de orientação, chamado Microsoft Solutions Framework para Agile Software Development (MSF4ASD) e Microsoft Solutions Framework para Capability Maturity Model Integration Melhoria de Processos (MSF4CMMI).

Note-se que estes processos de engenharia de software podem ser modificados e personalizados de acordo com as preferências da organização, clientes e equipe do projeto.

A filosofia MSF afirma que não existe uma única estrutura ou processo que se aplica optimamente os requisitos e ambientes para todos os tipos de projetos. MSF, portanto, suporta múltiplas abordagens processo, para que possa ser adaptado para apoiar qualquer projecto, independentemente da sua dimensão ou complexidade. Esta flexibilidade significa que ele pode oferecer suporte a uma ampla variação no grau de execução dos processos de engenharia de software, mantendo porém a um conjunto de princípios fundamentais e mentalidades.

A Microsoft Solutions Framework Process Model consiste em séries curtas de ciclos de desenvolvimento e iterações. Este modelo abraça iterativo rápido desenvolvimento com a aprendizagem contínua e refinamento, devido ao progressivo conhecimento da empresa e do projeto de todos os agentes envolvidos. Identificar necessidades, desenvolvimento de produtos, e os ensaios ocorrem nas sobreposições iterações, resultando em acréscimo conclusão de assegurar um fluxo de valor do projecto. Cada iteração tem um foco diferente eo resultado é uma porção estável do sistema global.

Seguem-se os oito princípios fundamentais, que constituem a espinha dorsal para os outros modelos e disciplinas do MSF:

1. comunicação aberta
2. Trabalhar em prol de uma visão compartilhada
3. Capacitar os membros da equipa

4. Estabelecer uma responsabilidade clara e partilhada

5. Concentre-se em negócio prestação valor

6. Fique ágil, esperam mudanças

7. Invista na qualidade

8. Aprenda com todas as experiências

MSF Models

MSF consiste de dois modelos:

1. MSF equipe modelo. Este artigo descreve o papel dos vários membros da equipe de projeto de desenvolvimento de software. Os membros dessa equipe seria:

Gerente de Produto: Principalmente lida com clientes e definir requisitos projecto, garante ainda a cliente as expectativas são cumpridas.

Gestão de Programas: Mantém projeto desenvolvimento e entrega ao cliente

Arquitetura: Responsável pela concepção solução, garantindo a solução óptima concepção satisfaz todas as necessidades e expectativas

Desenvolvimento: Desenvolve de acordo com as especificações.

Teste: Ensaios e garante a qualidade dos produtos

Lançamento / Operações: Garante a implantação eo bom funcionamento do software

Experiência do Usuário: Apoia questões dos usuários.

Uma pessoa pode ser atribuído a desempenhar múltiplos papéis. MSF também tem sugestão sobre como combinar a responsabilidades, como o dono da obra não deve ser atribuída a qualquer outro papel.

2. MSF modelo de governação. Esta descreve as diferentes fases da transformação de um projeto. O Modelo de Governança MSF tem cinco faixas sobreposição de atividade (ver abaixo), cada um com uma meta de qualidade definidos. Essas faixas de atividade definir o que precisa ser cumprida e deixar como eles se realizem para a equipa seleccionada metodologia. Por exemplo, as faixas podem ser pequenos e realizada no âmbito rapidamente para ser coerente com uma metodologia Agile, ou pode ser serializada e alongada para ser coerente com uma metodologia Cachoeira.

Envision - pense sobre o que deve ser realizado e identificar constrangimentos

Plano - planejar e projetar uma solução para atender as necessidades e expectativas dentro dessas limitações

Build - construir a solução

Estabilizar - validar a solução que vá ao encontro das necessidades e expectativas

Implantar - implantar a solução

MSF Project Management Process

Integrar planejamento e condução mudança de controlo

Definir e gerenciar o escopo do projeto

Preparar um orçamento e gerenciar os custos

Preparar e acompanhar os horários

Garantir que os recursos são atribuídos à direita do projecto

Gerir contratos e vendedores projeto e adquirir recursos

Facilitar a equipe e comunicações externas

Facilitar o processo de gestão do risco

Documento e monitorar a qualidade da equipe do processo de gestão

MSF para Agile Software Development metodologia

O MSF para Agile Software Development (MSF4ASD) se destina a ser um peso leve, processo iterativo e adaptável.

O MSF4ASD usa os princípios do desenvolvimento ágil abordagem formulada pela Agile Alliance.

O MSF4ASD prevê um processo de orientação que incide sobre as pessoas e as mudanças. Inclui aprendizagem de oportunidades e de avaliações usando iterações em cada iteração.

MSF para Capability Maturity Model Integration metodologia Melhoria de Processos

O MSF para Capability Maturity Model Integration Melhoria de Processos (MSF4CMMI) dispõe de mais artefatos, mais processos, mais signoffs, mais planejamento e destina-se a projectos que requerem um maior grau de formalidade e cerimônia.

O MSF4 CMMI é uma metodologia formal para a engenharia de software. Capability Maturity Model foi criada a partir do Software Engineering Institute da Universidade Carnegie Mellon, e é um processo de melhoria abordagem que fornece às organizações com os elementos essenciais do processo de melhoria contínua, resultando em uma redução SDLC, a melhoria da capacidade de cumprir as metas custo e cronograma, materiais de construção de alta qualidade.

O MSF4CMMI que ampliou o MSF4ASD orientação adicional com formalidade, comentários, verificação e de auditoria. Isso resulta em um processo que depende do SEP e ao processo e não conformidades confiando unicamente na confiança e na capacidade de cada um dos membros da equipe. O MSF4CMMI tem mais documentos obrigatórios e relatórios do que a versão ágil, mais formal e desenvolvimento deste processo reduz risco em grandes projetos de software e fornece um estatuto mensuráveis. Uma das vantagens de utilizar o CMMI é o processo pelo qual uma avaliação padrão poderão comparar a capacidade de desenvolver software em outras organizações.