2
Introdução Teórica

2.1.
Processos de Desenvolvimento de Software: Conceitos Gerais

Segundo Humphrey [HUMPHREY89], um processo de desenvolvimento de software é um conjunto de atividades, métodos, práticas e tecnologias que as pessoas utilizam para desenvolver e manter software e produtos relacionados. Ele envolve uma série de etapas que produzem um produto. Um processo pode ser imaturo e utilizar de uma abordagem ad-hoc para organização do trabalho, ou um processo maduro, caracterizado por uma metodologia padronizada e documentada, que já tenha sido utilizada em projetos similares ou que seja habitualmente adotada dentro de uma organização.

Podemos ver um processo como um “programa” que é executado por pessoas ao invés de máquinas [OSTERWEIL87]. Um processo de desenvolvimento de software é tipicamente definido através de papéis, artefatos e técnicas. Artefatos são as entradas e saídas dos componentes do sistema. As técnicas e metodologias adotadas definem como estes artefatos são processados, e podem ser vistos como o fluxo do programa em si. Os papéis definem os responsáveis pela realização do trabalho, e podem ser comparados com hardware que executa o programa.

A qualidade do software está intimamente ligada ao processo de desenvolvimento utilizado [GUERRA02]. Quando um processo de desenvolvimento é imaturo tarefas não são rigorosamente cumpridas e controladas e, conseqüentemente a qualidade e as funcionalidades do produto final podem ficar comprometidas, prazos podem ser descumpridos e os custos de manutenção podem tornar-se excessivamente altos.

Mais adiante nesta seção descreveremos alguns dos modelos de processos de desenvolvimento, nas quais o conteúdo deste trabalho está fortemente baseado.

2.1.1.
Classificação de Processos quanto à Abordagem Utilizada

Muitas metodologias endereçam o problema de engenharia de software da mesma forma que em outras áreas da engenharia, e propõem um modelo em "cascata" para o processo de desenvolvimento. Basicamente, propõem um planejamento extremamente criterioso, com a identificação e representação a mais completa possível antes do início da implementação de cada sistema. Estes são os chamados modelos preditivos. Mais recentemente, alguns autores passaram a propor um modelo de desenvolvimento em ciclos: dividir o projeto de software considerando as funcionalidades ou os artefatos a serem desenvolvidos, de forma a realizar o mesmo planejamento criterioso, dividindo-o em alguns "ciclos" de planejamento, execução e testes.

Sobre este modelo foi proposto ainda uma abordagem iterativa, onde o escopo e a duração dos ciclos seriam o menor possível. Estas abordagens representam uma forma de lidar com a complexidade e volatilidade naturais em projetos de software. Com a complexidade, pois a conclusão dos ciclos anteriores garante um melhor entendimento das dificuldades e peculiaridades dos projeto, e com a volatilidade, pois torna-se possível a constante análise crítica e freqüentes alterações no modelo inicial.

Em um artigo, já em 1987 [BROOKS87], Brooks procura identificar características e dificuldades exclusivas da produção de software: sua complexidade, conformidade, volatilidade e invisibilidade. Estas propriedades únicas da disciplina de engenharia de software, segundo Brooks, representam barreiras diferentes daquelas encontradas nas disciplinas clássicas tratadas pela engenharia, como a construção civil, e justificariam abordagens metodológicas diferentes. Os métodos ágeis surgem como uma mudança de paradigma, uma abordagem alternativa àquelas propostas pelas outras áreas da engenharia, como forma de lidar com tais características essenciais do software.

Dentro deste contexto, alguns críticos das metodologias clássicas propostas pela engenharia de software alegam que o processo de desenvolvimento torna-se tão complexo que o objetivo final deste processo – a codificação do sistema em si – passa a representar uma parcela pequena do esforço total empregado. Como resultado, surgem os chamados "métodos leves" (*lightweight methods*), que

representam um meio termo entre a ausência de um processo e processos custosos demais, e propõem a realização somente daquilo que é essencial para garantir sucesso e qualidade no desenvolvimento de softwares. Estes métodos leves seriam mais tarde chamados de “métodos ágeis”, uma denominação dada por um conjunto de criadores e entusiastas deste tipo de abordagem, chamada Agile Alliance [W_AGILE]. Os métodos ágeis surgem como uma mudança de paradigma, uma abordagem alternativa àquelas propostas pelas outras áreas da engenharia, como forma de lidar com tais características essenciais do software.

2.1.2. Classificação de Processos quanto ao Nível de Modelagem

Pode ser útil classificar processos de desenvolvimento de software quanto ao nível de modelagem. Apresentaremos aqui a arquitetura de metamodelo definida pelo *Software Process Engineering Metamodel* (SPEM) [W_OMG02]. O metamodelo SPEM foi desenvolvido pela *Object Management Group, Inc* (OMG), e tem como objetivo descrever os elementos que compõem processos de desenvolvimento de software concretos através de um modelo orientado a objetos, utilizando o *Unified Modeling Language* (UML) [W_OMG03] como padrão de notação e definindo um *UML Profile* [W_OMG02]. Não é nosso objetivo aqui aprofundar-nos no modelo SPEM, mas tão somente fazer uso desta classificação com o objetivo de tornar mais claro o conceito de metamodelos, além de definir claramente o nível de abstração do processo descrito neste trabalho.

O SPEM define quatro camadas de processos, representadas na Figura 1. Um processo executável, isto é, um processo da forma como ele é aplicado em ambiente de produção, está no nível M0. Neste nível, temos o processo instanciado para um determinado projeto, com os papéis definidos (isto é, quem será responsável por cada papel), os artefatos gerados, o objetivo do projeto sendo desenvolvido, etc. Chamamos portanto este nível de nível da instância de processos.

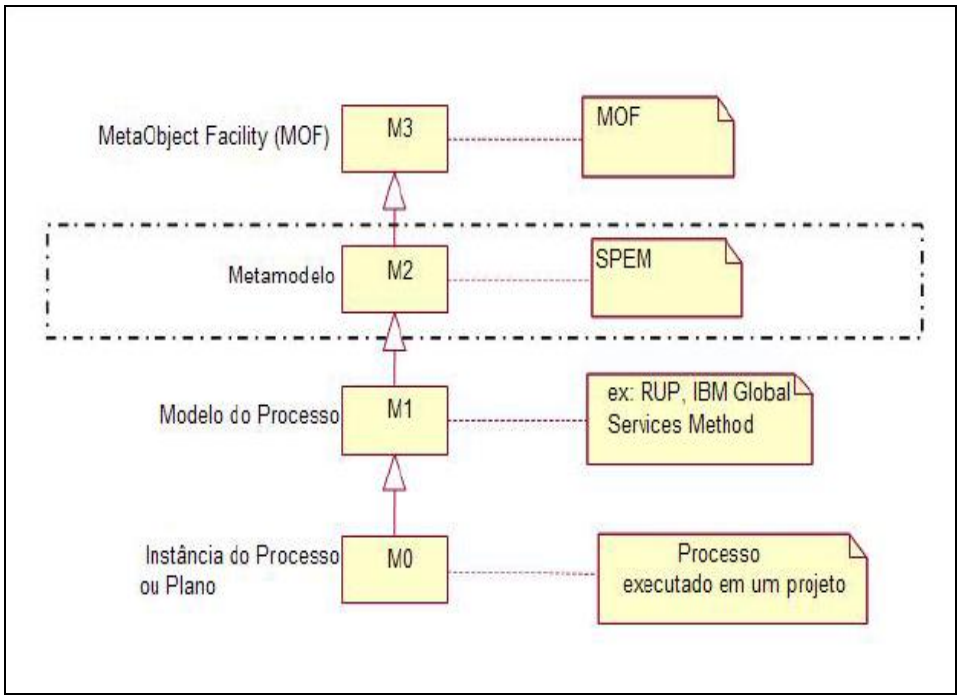


Figura 1 – Camadas de processos definidas pelo SPEM

A definição do processo correspondente está no nível M1. Este nível contém as definições genéricas de classes (entidades), modelos, relacionamentos, responsabilidades e comportamentos a serem instanciadas no nível M0. O processo desenvolvido e apresentado neste trabalho está neste nível.

O nível M2 é o nível de metamodelos. Os modelos neste nível define como as classes, relacionamentos e comportamentos mostrados no nível M1 se relacionam. É um metamodelo definido neste nível que usaremos para representar o processo proposto neste trabalho: o framework Process Engineering Perspective (PEP) [DAFLON03], que será apresentado de forma mais detalhada na seção 2.4. O próprio modelo SPEM também está definido neste nível.

O nível M3 é o repositório global dos vários domínios de modelagem, incluindo a própria UML, e provê a fonte para a construção de novos domínios de modelagem.

2.2. CMM e CMMI

O Modelo de Maturidade da Capacitação (ou Capability Maturity Model – CMM) é um framework que estabelece os elementos chave de um processo de desenvolvimento de software eficaz.

O CMM define um caminho evolutivo de etapas que vão desde um processo informal ou a ausência de processo, até um processo maduro e otimizado de melhoria contínua. Quando adotadas, as práticas ajudam as organizações a alcançar suas metas em termos de custo, prazo, funcionalidade e produtividade.

Para tal, o CMM classifica as organizações em cinco níveis distintos, cada um com suas características próprias. No nível 1, o das organizações mais imaturas, não há nenhuma metodologia implementada e tudo ocorre de forma desorganizada (ad-hoc). No nível 5, o das organizações mais maduras, cada detalhe do processo de desenvolvimento está definido, quantificado e acompanhado e a organização consegue absorver mudanças no processo sem prejudicar o desenvolvimento.

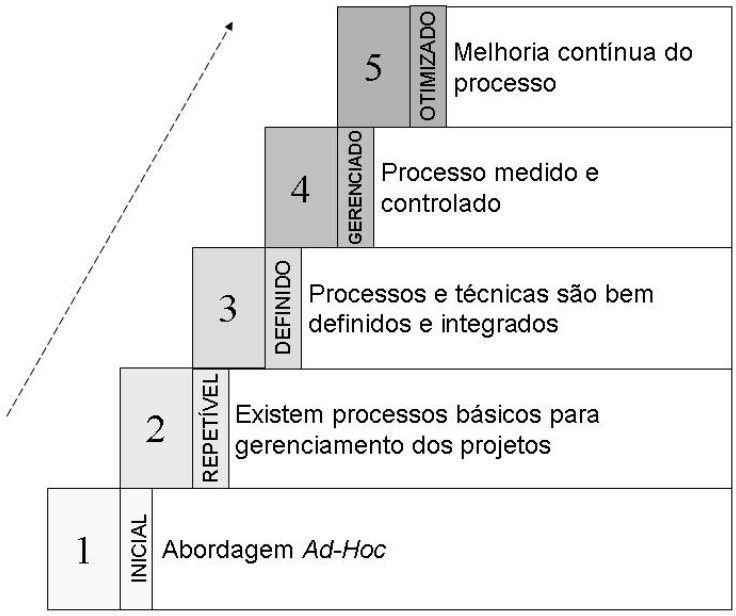


Figura 2 – Níveis de Maturidade definidos pelo CMM

A tabela a seguir apresenta uma descrição um pouco mais detalhada das evoluções do processo em cada nível de maturidade.

Nível	Evolução
(1) Inicial	O processo de desenvolvimento é desorganizado e caótico. Poucos processos são definidos e o sucesso depende de esforços individuais e heróicos.
(2) Repetitivo	Os processos básicos de gerenciamento de projeto estão estabelecidos e permitem acompanhar custo, cronograma e

	funcionalidade. É possível repetir o sucesso de um processo utilizado anteriormente em outros projetos similares.
(3) Definido	Tanto as atividades de gerenciamento quanto de engenharia do processo de desenvolvimento de software estão documentadas, padronizadas e integradas em um padrão de desenvolvimento da organização. Todos os projetos utilizam uma versão aprovada e adaptada do processo padrão de desenvolvimento de software da organização.
(4) Gerenciado	São coletadas medidas detalhadas da qualidade do produto e processo de desenvolvimento de software. Tanto o produto quanto o processo de desenvolvimento de software são entendidos e controlados quantitativamente.
(5) Otimizado	A melhoria contínua do processo é conseguida através da retroalimentação dos processos. Os gerentes são capazes de estimar e acompanhar quantitativamente o impacto das alterações realizadas no processo.

Tabela 1 – Evoluções do processo de desenvolvimento para cada nível de maturidade

O CMM estabelece um referencial que permite mensurar de forma pragmática a maturidade dos processos de uma organização. Maturidade neste contexto é definido pelo próprio CMM e representado através da estrutura utilizada por este modelo, que é apresentada a seguir.

2.2.1. Estrutura do CMM

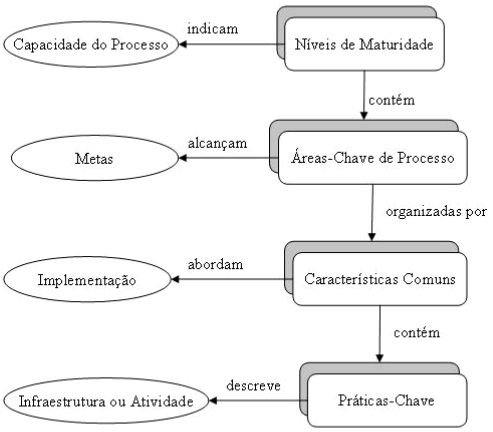


Figura 3 – Estrutura do CMM

Para alcançarmos um determinado nível de maturidade, precisamos cumprir determinadas Práticas-chave. Cada nível de maturidade se desdobra em diversas Áreas-chave de Processo, cada uma das quais relacionada ao atingimento de metas. Estas Áreas-chave agrupam Práticas-chave de acordo com características comuns. Cada área-chave tem no mínimo uma Prática-chave para cada Característica Comum. Tais Características Comuns podem referir-se à implementação (atividades) ou à institucionalização (compromissos, habilitações, medições, verificações). Já as Práticas-chave são descrições de atividades ou de elementos de infra-estrutura concretos [STAA98].

Podemos ver portanto que os elementos descritos na parte inferior da Figura 3 são descrições em um nível totalmente concreto, e o nível de abstração dos conceitos vai subindo conforme caminhamos para a parte superior do diagrama.

2.2.2. CMMI na Versão Contínua

Ao longo do tempo, o CMMI evoluiu, e a evolução do modelo levou à decomposição deste em duas abordagens distintas. Convencionou-se chamar estas abordagens por “representações” ou “versões” do modelo. Uma representação reflete a organização, aplicação e apresentação dos componentes do modelo, que permanece o mesmo.

A representação em estágios, apresentada anteriormente usa um conjunto determinado de áreas de processo para definir um “caminho” para alcance da maturidade da organização. Cada passo deste caminho é descrito pelo modelo como um componente denominado “nível de maturidade”, que foi conceituado anteriormente.

A segunda representação – contínua, possibilita que uma organização escolha áreas de processo específicas e obtenha o nível de maturidade para estas áreas individualmente. Esta representação é a que apresenta mais interesse para pequenas organizações, pois são mais adaptáveis e representam maior facilidade de adoção por pequenas equipes.

2.3.
Processos Ágeis de Desenvolvimento

Nos anos recentes, observou-se um crescente interesse nos chamados "métodos ágeis" (MAs) de desenvolvimento de software. Estes métodos representam um paradigma alternativo no desenvolvimento de projetos de software, e pregam a simplicidade, objetividade e adaptabilidade no processo de desenvolvimento.

Estas metodologias são baseadas em princípios altamente pragmáticos, e recentemente vêm tendo grande aceitação pela indústria, principalmente por parte de grupos caracterizados por possuírem equipes pequenas trabalhando em um ambiente onde a comunicação é facilitada. MAs são especialmente apropriados quando o sistema sendo desenvolvido possui requisitos que evoluem rapidamente, operando em um ambiente de inovação, onde as especificações não podem ser completamente elicitadas em um primeiro instante.

Porém, apesar de argumentos contrários por parte dos entusiastas destas metodologias, certos aspectos básicos considerados importantes pela Engenharia de Software clássica são negligenciados por esta corrente.

É correto afirmar que as práticas propostas pelos MAs mais comuns não passam de uma coleção das melhores práticas já utilizadas há anos pela indústria, e que não devem ser usadas indiscriminadamente, mas sim adaptadas caso a caso de forma criteriosa.

2.3.1.
Características Gerais

É difícil elaborar uma definição precisa e genérica o suficiente que defina qualquer método ágil, apesar de haver alguns pontos comuns a todas as metodologias estudadas, como mostraremos a seguir. Isso acontece porque métodos ágeis são compostos por regras gerativas, e não regras inclusivas: a maior parte das metodologias consiste em um conjunto de regras inclusivas – tudo o que se deve fazer sob qualquer situação possível. Métodos Ágeis fornecem regras gerativas – um conjunto mínimo de coisas que devem ser feitas sob qualquer situação para gerar práticas apropriadas para esta situação especial. Talvez por esta característica, a definição mais genérica e adequada para métodos ágeis resida

no campo ideológico, e seja muito bem descrita pelo manifesto e pelos princípios propostos pela Agile Alliance [W_AGILE].

Jim Highsmith e A. Cockburn enumeraram algumas práticas essenciais para que uma metodologia possa ser considerada “ágil” [HIGHSMITH01]. Em um outro trabalho [W_MILLER], Randy Miller também enumera algumas características comuns em um conjunto de metodologias ágeis estudadas. Baseado estes autores, podemos afirmar que métodos ágeis devem ser modulares, iterativos, incrementais, temporalmente restritos, parcimoniosos, adaptáveis, convergentes, orientado a pessoas, colaborativos e retroalimentados. Tais características essenciais desta classe de processos são definidos na seções a seguir.

2.3.1.1. Modulares

Modularidade é a característica de um processo que permite ser dividido em atividades. Um bom processo deve definir claramente suas atividades de forma que as pessoas possam segui-lo. Modularidade permite que atividades sejam convenientemente retiradas ou inseridas no processo.

2.3.1.2. Iterativos

Um processo não pode ser classificado como ágil se o ciclo que inicia na especificação pelo cliente e vai até a validação dos artefatos desenvolvidos pelo próprio cliente for de mais do que seis meses. Metodologias ágeis indicam ciclos iterativos de curta duração, geralmente entre duas e seis semanas. O planejamento do trabalho a ser desenvolvido também deve ser realizado em ciclos curtos, no início ou fim de cada iteração.

2.3.1.3. Incrementais

Um processo ágil não objetiva a construção de todo o sistema de uma só vez. O projeto deve ser particionado em incrementos. Cada incremento pode levar várias iterações para ser completado, e podem ser desenvolvidos em paralelo.

2.3.1.4.
Temporalmente Restritos

Cada iteração deve ter duração limitada e curta. Cada iteração deve levar a um construto útil para o usuário. Usuário no sentido lato do termo pode envolver também a equipe de desenvolvimento. Neste caso o construto destina-se a ser avaliado.

2.3.1.5.
Parcimoniosos

Cronogramas são rígidos e devem ser seguidos, mas não pode ocorrer desgaste excessivo dos programadores. Programadores sobrecarregados rendem menos e têm a produtividade comprometida.

2.3.1.6.
Adaptáveis

Como a maioria das metodologias ágeis são aplicadas em ambientes "turbulentos", ou seja em constante mudança, devem haver práticas que garantam o *feedback* constante entre todos os atores envolvidos no projeto. Decisões técnicas, requisitos dos clientes, restrições gerenciais, e outros aspectos do projeto devem ter interação rápida garantida. Ao contrário de algumas metodologias, nas abordagens ágeis, alterações e mudanças são encorajadas a qualquer momento. Métodos ágeis são essencialmente não-preditivos.

Deve também haver priorização dinâmica. Isso significa que, ao fim de cada iteração, o cliente pode reordenar a prioridade das funcionalidades desejadas ou até mesmo excluir e adicionar novas funcionalidades.

2.3.1.7.
Convergentes

Após cada incremento, o sistema deve estar mais próximo do objetivo final.

2.3.1.8.
Orientados a Pessoas

Processos ágeis são altamente dependentes das pessoas envolvidas. Os desenvolvedores são providos de alto poder de decisão, já que quase todas as decisões são tomadas localmente.

2.3.1.9.
Colaborativos

A coesão entre equipe de desenvolvimento e os *stakeholders* é essencial em um projeto ágil. Programação em duplas é uma característica marcante no XP, e outros métodos incentivam formas diferentes de garantir uma forte colaboração, e uma comunicação eficiente, como por exemplo, um ambiente de desenvolvimento sem paredes ou baias. O relacionamento com o cliente também deve ser próximo, e o sucesso de projetos que utilizam metodologias ágeis depende fortemente disso.

2.3.1.10.
Retroalimentados

As atividades devem ser controladas. Ao fim de cada atividade, alguma forma de análise deve ser feita e podem ser necessárias alterações ou adaptações.

2.3.2.
eXtreme Programming

Extreme Programming (XP) [BECK00] é um processo “ágil” de desenvolvimento de software, conforme caracterizado na seção anterior. Como qualquer metodologia ágil, ela pode ser vista com reservas em relação a metodologias e técnicas da engenharia de software clássica. Porém, uma análise mais cuidadosa desta alternativa revela um processo altamente pragmático, que oferece alternativas razoáveis para o desenvolvimento de sistemas de software por pequenas equipes, em contrapartida a processos custosos (com grande *overhead* gerencial) definidos por metoldologias tradicionais. XP difere de tais metodologias tradicionais por mudar o foco de atenção para a geração de código, que é visto como o produto final e objetivo de qualquer projeto. Ele advoga a

construção de sistemas simples, limpos e testados em passos pequenos através do esfoço coordenado de uma equipe de desenvolvimento.

XP segue uma estrutura simples, baseada em dois compontentes apenas: valores e atividades. Os quatro valores, comunicação, simplicidade, feedback e coragem podem ser vistos como a base ética que guiará a força de trabalho. Já as quatro atividades, projetar, codificar, testar e planejar são o embasamento técnico por trás da metodologia [HISLOP02]. Os valores são seguidos através da aplicação de 15 princípios, enquanto as atividades são implementadas através da adoção de 28 práticas. Os princípios e práticas, enumerados a seguir são descritos de forma de forma bastante resumida, com o propóstio de caracterizar de forma objetiva este processo ágil de grande importância.

2.3.2.1. Valores

2.3.2.1.1. Comunicação

Espera-se que a comunicação entre os membros da equipe de desenvolvimento e entre esta e os clientes seja qualitativamente alta e freqüente. Isto ocorre porque no XP, a documentação no sentido tradicional é limitada. Clientes comunicam requisitos funcionais através de estorietas escritas cartões que são usados como linha-guia para o desenvolvimento. Os desenvolvedores tiram proveito do pequeno tamanho da equipe e de um ambiente único e livre de baias para comunicarem-se sempre que necessário. Reuniões rápidas e diárias alinham a equipe em torno do andamento global do projeto.

2.3.2.1.2. Simplicidade

Os desenvolvedores devem objetivar a construção do sistema mais simples possível que realize de forma eficaz o trabalho a que se propõe. Conforme novos requisitos e funcionalidades são incorporadas no sistema, este é modificado o quanto for necessário, mas o mínimo possível para não complicar o *design*. Modificar o sistema pode envolver a necessidade de refatoramento, uma atividade que visa melhorar a estrutura do código sem alterar a funcionalidade do sistema.

2.3.2.1.3. Retroalimentação

A retroalimentação no XP funciona na escala de minutos e dias e ocorre em diversos aspectos na organização do trabalho. Integração constante do sistema, por exemplo, fornece aos desenvolvedores informação imediata sobre o estado do sistema. Programação em pares é uma forma de implementar a inspeção de código através de um modelo que é retroalimentado continuamente. Cada execução de testes oferece aos desenvolvedores feedback imediato quanto a possíveis inconsistências no código.

2.3.2.1.4. Coragem

Para o XP, coragem é um valor essencial. O conceito por trás deste valor passa pela mudança de paradigma necessária para adoção de uma metodologia como esta, mas atinge também o processo em todos os níveis e tarefas. Por exemplo, desenvolvedores são responsáveis por garantir que a evolução do design do sistema é feita da melhor e mais simples forma possível. Mas o uso de um design simples pode depender de decisões difíceis de serem tomadas: grande quantidade de código pode ter que ser jogado fora, ou uma grande parte do sistema pode ter que ser totalmente re-projetado.

2.3.2.2. Práticas

2.3.2.2.1. Cliente Presencial

Ao longo do projeto, um representante do cliente deve estar sempre presente no ambiente de trabalho da equipe de desenvolvimento. O cliente é responsável por criar estorietas, que são tipos de cenários resumidos, cada uma das quais representando uma funcionalidade do sistema. A presença do cliente no ambiente de desenvolvimento é essencial para criar uma comunicação rápida com a equipe de desenvolvimento, uma vez que estorietas são uma representação bastante informal das funcionalidades, e os desenvolvedores podem deparar-se com

dúvidas e dificuldades de interpretação constantemente. Além de criar as estorietas, o cliente deve validar cada release, através de testes de aceitação.

2.3.2.2.2. Uso de Metáfora

Metáforas são usadas como uma representação sistemática e abrangente dos objetivos do sistema. Elas podem ser criadas pela equipe de desenvolvimento, pelo cliente, ou por ambos de forma colaborativa. O objetivo da metáfora é criar uma visão que servirá para alinhar toda a equipe em torno de um mesmo objetivo.

2.3.2.2.3. Jogo do Planejamento

Ocorre no intervalo das iterações e no início do projeto. Para o XP, o planejamento é visto como um jogo, onde todos participam – desenvolvedores, gerentes e o cliente. O Cliente é responsável por especificar o projeto através de estorietas. Os cartões com as estorietas são então avaliados pela equipe de desenvolvimento e classificadas quanto a sua complexidade, utilizando a medida de *homem-hora ideal*[1].

2.3.2.2.4. Pequenos *Releases*

O cliente, num trabalho conjunto com a equipe de desenvolvimento, durante o *Jogo do Planejamento*, classifica e agrupa as estorietas com base na prioridade de cada uma. Finalmente, com base nas estimativas de custo dadas pelos desenvolvedores, as estorietas são agrupadas em iterações de no máximo duas semanas. A saída de diversas iterações é um *release*. Os *releases* representam partes do sistema com uma macro-funcionalidade completa e que pode ser avaliado pelo cliente através dos testes de aceitação.

[1] Uma homem-hora ideal equivale a uma hora de trabalho de um desenvolvedor, considerando que não haja desperdício de tempo ou contratempos na realização do trabalho

2.3.2.2.5.
Uso de Testes de Aceitação

Ao final de cada *release*, os testes de aceitação são feitos pelo cliente. Ter o sistema entregue aos poucos e validado a cada passo ajuda a mitigar riscos quanto a mudanças de requisitos e inadequação da especificação, e apresenta-se como uma forma eficiente de garantir que o que está sendo desenvolvido está sempre de acordo com as expectativas do cliente.

2.3.2.2.6.
Desenvolvimento Orientado a Testes (*TDD*)

Desenvolvimento orientado a testes é uma prática que consiste em escrever os testes antes e só posteriormente escrever o código que irá passar nos testes. Embora pouco ortodoxa esta prática é extremamente proveitosa, pois elimina a necessidade de criar-se um papel exclusivamente dedicado à implementação dos *scripts* de teste (o 'testador'). Sabemos que quando os testes automatizados são escritos pela mesma pessoa ou grupo que codificou o sistema, os scripts implementados tendem a ser beligerantes e conter viés, pois as falhas de especificação ou os pontos negligenciados tendem a se repetir.

Na prática, implementar os scripts de teste antes do código também é uma forma de especificar o sistema. Ao implementar um teste, os desenvolvedores passam a ter que considerar aspectos de mais baixo nível, abordados somente de forma superficial ou totalmente ausente nas estorietas, com os quais teriam que lidar após o início da implementação.

Esta prática também é uma forma eficiente de garantir a abrangência da aplicação de testes automatizados: nenhum código será escrito antes de seus testes, logo, todo código será coberto por pelo menos uma parca quantidade de *scripts* de teste.

2.3.2.2.7.
Integração Contínua

O sistema deve ser integrado com a maior freqüência possível, de preferência diariamente. Após cada integração, todos os testes deverão ser re-executados para garantir a consistência do sistema.

2.3.2.2.8. Design Simples

Utilizar sempre o *design* mais simples que passe em todos os testes. Nunca tentar adicionar funcionalidades antes do planejado (ou seja, antes de chegar à estorieta que descreve tal funcionalidade). Da mesma forma, se encontrarmos algo que poderia ser feito de forma mais simples, indica-se simplificá-lo imediatamente. Deve-se manter a estrutura do sistema o mais simples possível pelo maior tempo possível. Norman Kerth [W_KERTH01] apresenta uma metáfora interessante ao defender que não se deve investir em um *design* completo a princípio: segundo esta metáfora, uma lagarta é projetada para comer a maior quantidade de folhas possível, mas é incapaz de reproduzir-se. Mas ela precisa cumprir seu papel inicial antes de ser transformada para a forma de uma borboleta, capaz de alimentar-se e reproduzir-se de forma eficiente. Esta metáfora é eficiente em demonstrar como esta prática é intimamente dependente da prática 2.3.3.2.9: “Refatoramento Constante”.

2.3.2.2.9. Refatoramento Constante

XP parte do pressuposto de que o uso e reuso de código que não é mais manutenível leva invariavelmente ao desperdício de recursos [BECK00]. Desta forma, eliminar funcionalidades que não são mais utilizadas e rever *design* obsoleto constantemente ao longo do ciclo de vida do projeto ajuda a economizar tempo e melhorar a qualidade do sistema.

2.3.2.2.10. Programação em Pares

Todo o código do sistema deve ser criado por duas pessoas trabalhando cooperativamente em uma única estação de trabalho. Pode ser intuitivo pensar nesta prática como um desperdício de recursos, mas o contra-argumento utilizado pelos defensores do XP está na qualidade do código escrito. Programação em pares é uma forma contínua de revisão e os artefatos escritos por uma dupla de programadores estão menos propensos a conter falhas e são teoricamente mais consistentes e bem projetados.

Na programação em pares, um dos programadores é o “piloto”, e está no controle do mouse e do teclado. Este programador tem uma visão local do código. O co-piloto deve observar o trabalho do primeiro, e opinar sobre tudo que é escrito. Por não estar ocupado com a codificação, o co-piloto tende a ser mais atento a pequenos erros que passariam desapercebidos, ao não uso das melhores práticas, e tende a ter uma visão mais abrangente e sistemática do artefato sendo codificado. Piloto e co-piloto devem revezar papéis com freqüência, e podem fazê-lo à vontade.

2.3.2.2.11.Revezamento

Os desenvolvedores devem revezar sempre que possível o artefato em que estão trabalhando, tal como o posto de “piloto” e “co-piloto” na programação em duplas. O objetivo desta prática é manter toda a equipe a par do sistema como um todo, e evitar o desenvolvimento de vícios e paradigmas.

2.3.2.2.12.Propriedade Coletiva do Código

Qualquer desenvolvedor pode adicionar funcionalidade, consertar falhas ou refatorar em qualquer artefato. Esta prática é suportada pelas práticas “Integração Freqüente” e “Desenvolvimento Orientado a Testes”, pois somente através delas podemos garantir que qualquer alteração indevida em um artefato que gere uma inconsistência no sistema será imediatamente identificada e sanada.

2.3.2.2.13.Padrões de Codificação

Todo o código deve ser formatado de acordo com padrões de codificação comuns para toda a equipe. Esta é uma forma de garantir que o código permaneça consistente e legível para que toda a equipe possa analisar e refatorar.

2.3.2.2.14.Semana de 40h

XP defende a idéia de que desenvolvedores que trabalham mais de oito horas diárias são menos produtivos e tendem a escrever código de pior qualidade, e defende o limite superior de quarenta horas semanais como jornada de trabalho.

Uma maior necessidade de produtividade deve ser suprida com uma equipe maior, e não com a dedicação dos desenvolvedores além deste limite.

2.4. O *Framework* PEP

O *framework* PEP é um metamodelo de processo [DAFLON03], que dá suporte à definição de processos de desenvolvimentos. Apresentamos a seguir uma visão geral da estrutura deste *framework*. Como o PEP estende o metamodelo SPEM, algumas características relevantes deste metamodelo também serão abordadas de forma conjunta. No capítulo 3, utilizaremos este metamodelo para descrever o processo que figura como tema principal desta dissertação.

O PEP dá suporte à definição de processos utilizando o seguinte conjunto de diagramas, definidos pelo SPEM:

a. Diagrama de Unidade de Processo
b. Diagrama de Tarefas
c. Diagrama de Pacote
d. Diagrama de Casos de Uso
e. Diagrama de Seqüência

2.4.1. A Estrutura do *Framework* PEP

O *framework* está estruturado em seis pacotes. Cinco dele são descritos de forma resumida a seguir. O pacote “fachada” não será descrito por não ser relevante para o trabalho aqui desenvolvido.

2.4.1.1. O Pacote Repositório

O Repositório é composto por quatro *containers* de elementos básicos e dois *containers* de elementos compostos. Os *containers* são responsáveis por armazenar e gerenciar o ciclo de vida dos elementos de modelo de um processo de desenvolvimento de software.

Os *containers* de elementos básicos armazenam atividades, papéis, artefatos e projetos. Já os *containers* de unidades compostas armazenam as unidades de

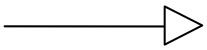
processo e os processos definidos. Os elementos compostos são definidos a partir dos elementos básicos.

2.4.1.2. O Pacote Unidade de Processos

As unidades de processo representam blocos de construção utilizados na definição de novos modelos de processo e abstraem os conceitos de padrões e componentes de uma instância do processo. O PEP utiliza dois diagramas para representar a solução de uma unidade de processo: (1) o diagrama de unidade de processo e (2) o diagrama de tarefas. A notação usada nestes diagramas é descrita de forma mais detalhada na seção 2.4.2. Para cada diagrama de unidade de processo existe um diagrama de tarefas detalhando as atividades que representam a solução da unidade de processo.

2.4.1.3. O Pacote Conectores

O pacote conectores define os diversos conectores usados para a construção dos diagramas. Um conector liga dois elementos dos modelos. A tabela 2 mostra uma descrição resumida dos diferentes tipos de conectores e suas respectivas representações gráficas.

Conector	Notação	Descrição
Sucessão		Descreve que uma unidade de processo A (denominado predecessor) e uma unidade de processo B (o sucessor) estão relacionados por sucessão caso A produza todos os artefatos que a unidade de processo B consome. Isto é, o contexto inicial de B é um subconjunto do contexto resultado de A.
Especialização		Uma unidade de processo A (subunidade) é especialização de uma unidade de processo B (superunidade) se o contexto inicial e o contexto final das unidades A e B se casam e o processo A é descrito em mais detalhe do que o processo B.
Exclusão Mútua		Uma unidade de processo A e uma unidade de processo B estão relacionadas por exclusão mútua se ambas resolvem o mesmo problema com o mesmo contexto.

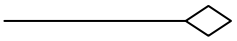
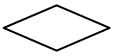
Conector	Notação	Descrição
Composição		Uma unidade de processo A (unidade composta) e uma unidade de processo B (unidade componente) estão relacionadas através de composição se a unidade B representa um subprocesso da unidade A. Neste caso, a unidade de processo B descreve parte da solução da unidade de processo A e a notação é orientada da forma que a ponta com o losango esteja próxima à unidade A.
Sincronismo		Utilizado para iniciar atividades em paralelo ou juntar o fluxo de atividades iniciadas em paralelo.
Ponto de Decisão		Simboliza alterações no fluxo de execução dependendo da avaliação de uma determinada condição.

Tabela 2 – Principais conectores definidos pelo SPEM

2.4.1.4. O Pacote Avaliação

O pacote avaliação tem como objetivo dar suporte à avaliação da qualidade de um processo, através de critérios de conformidade a uma norma ou modelo de maturidade, como o CMM. Este pacote do metamodelo contém uma estrutura genérica, de forma que o modelo de qualidade ou de maturidade a ser utilizado é um ponto de flexibilidade (*hotspot*) do *framework* PEP.

No modelo adotado pelo *framework*, a interface do pacote avaliação é definida pela classe GerenteDeAvaliação. O gerente de avaliação está relacionado a um modelo de maturidade ou padrão de qualidade e a um conjunto de diretivas. As diretivas são os elementos básicos de avaliação que estão associados aos elementos executáveis da estrutura da norma. Cada diretiva contém um conjunto de itens de conformidade, na qual cada um destes itens apresenta um conjunto de unidades de processo que tornam aquela diretiva conforme. Além disso, um item de conformidade é composto por um conjunto de artefatos que representam as evidências de realização da diretiva.

O pacote também contém uma estrutura que representa o diagrama de avaliação. Este diagrama é a representação da avaliação do processo definido. Por depender da estrutura da norma ou modelo de maturidade, este diagrama é também um *hot-spot* do framework.

2.4.1.5. O Pacote Processo

O pacote processo estende o pacote de componentes do SPEM, embutindo o conceito de unidade de processo. A unidade de processo especializa o conceito de pacote do metamodelo SPEM e é especializada pelos conceitos de *padrão de processo* e *componente de processo*. A unidade de processo, assim como as atividades, implementam a interface de elementos conectáveis.

Um componente de processo é um conjunto estruturado de atividades, papéis e artefatos que são utilizados como blocos de construção de um processo [UNHELKAR02]. Componentes de processo transformam um contexto inicial em um contexto resultado após sua aplicação. Para reutilizar um componente de processo é necessário identificar o contexto no qual ele está inserido, certificando que suas pré-condições sejam atendidas e que sua saída seja compatível com a entrada do próximo componente.

O conceito de Unidade de Processo definido por Daflon [DAFLON03] refere-se a blocos de construção utilizados na definição de novos modelos de processo e extendem os conceitos de Componentes de Processo.

2.4.2. Notação

Para descrever as Unidades de Processo foi utilizada a notação adotada pelo framework PEP, que leva em conta a especificação do Metamodelo de Processos de Engenharia de Software proposto pelo *Object Management Group* (OMG) [W_OMG02], juntamente com as contribuições de Unhelkar [UNHELKAR02] e Dittman et al. [DITTMAN02] para descrição das Unidades de Processo.

O PEP utiliza os diagramas básicos da UML para representar as diferentes perspectivas de modelos de processo. Neste contexto, são utilizados os seguintes diagramas: (1) Diagrama de Unidades de Processo, (2) Diagrama de Pacotes, (3) Diagrama de Casos de Uso, (4) Diagrama de Seqüência e (5) Diagrama de Classe. O processo desenvolvido neste trabalho será descrito utilizando este profile, através dos seus diagramas de Unidade de Processo. As outras perspectivas não serão apresentadas graficamente, mas somente o conteúdo dos pacotes definidos na seção 2.4.1.

O SPEM UML Profile define os seguintes estereótipos e suas notações gráficas:

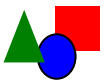
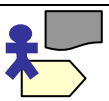
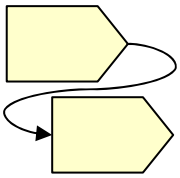
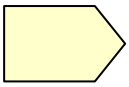
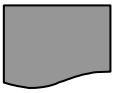
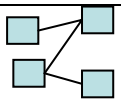
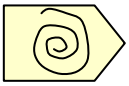
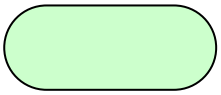
Estereótipo	Notação
Papel	
Artefato	
Pacote de Processo	
Definição de Trabalho	
Atividade (estende Definição de Trabalho)	
Documento (estende Artefato)	
Modelo UML (estende Artefato)	
Processo (estende Pacote Processo)	
Fase (estende Definição Trabalho)	
Tarefa	
Unidades de Processo (estende Pacote de Processo)	

Tabela 3 – Principais esteriótipos definidos pelo *SPEM UML Profile*