PARTE II: PROCESSO DE SOFTWARE

CAPÍTULO 3 PROCESSO DE DESENVOLVIMENTO DE SOFTWARE

No desenvolvimento de qualquer projeto é desejável sempre seguir uma série de passos previsíveis – um roteiro que, por ocorrer várias vezes, ajuda a criar um resultado de alta qualidade. No que diz respeito ao desenvolvimento de sistemas, este roteiro é chamado "processo de software".

O pressuposto aqui é que cada instalação (departamento ou setor de desenvolvimento de sistemas) disponha de um processo padrão, de pleno conhecimento de todos os profissionais que atuem nela. A importância de se estabelecer um padrão é a possibilidade de aperfeiçoar cada vez mais o processo, visto que abrange todas as fases da produção do software, desde o planejamento, passando pelo desenvolvimento e implantação, indo até à fase de manutenção. Na fase de planejamento, a importância advém de se poder manter as mesmas tarefas na crucial etapa de estimativa de recursos e no estabelecimento dos cronogramas. À medida que mais projetos são desenvolvidos (desde que haja o registro histórico de seu planejamento e execução), cada vez mais precisas serão as estimativas dos recursos necessários (tempo, pessoal e outros) para o desenvolvimento de um projeto qualquer. A par disto, há uma padronização das tarefas desenvolvidas pelos profissionais (engenheiros de software, analistas de sistemas, projetistas e programadores).

A padronização do processo de software leva ainda forçosamente à definição de todos os documentos produzidos ao longo da fase de desenvolvimento. Estes documentos possibilitarão visualizar o estágio em que o projeto se encontra e serão úteis em dois aspectos: para avaliação da aderência a padrões estabelecidos por parte dos desenvolvedores, como também para avaliar se a solução até este ponto do desenvolvimento é satisfatória ou não. Este trabalho de avaliação pode ser feito, em alguns momentos, por meio de revisão técnica, ou por meio de revisão gerencial, ou ambas.

Definição de Engenharia de Software

O SWEBOK(2004) define Engenharia de Software como "a aplicação de uma abordagem sistemática, disciplinada, quantificável ao desenvolvimento, à operação e à manutenção de software; isto é, a aplicação de engenharia ao software". Acrescenta ainda que a Engenharia de Software inclui o estudo de abordagens que levem a disciplina de engenharia à área de software. Roger S. Pressman [Pressman, 2010] aponta que, além de disciplina, deve-se acrescentar adaptabilidade e agilidade, qualidades exigidas pela atual demanda de computação.

Os modelos de processos baseiam-se em vários princípios formulados ao longo do período de consolidação da área de desenvolvimento de software como uma disciplina de Engenharia. Na próxima seção estes princípios são apresentados resumidamente.

3.1 PRINCÍPIOS DE ENGENHARIA DE SOFTWARE

As disciplinas de Engenharia baseiam-se em princípios gerais, aplicáveis em todas as etapas do processo de desenvolvimento de produtos. Assim também ocorre na Engenharia de Software. Estes princípios gerais estão no cerne dos processos de desenvolvimento, das técnicas e das ferramentas utilizadas pelo engenheiro de software. Por isso, ele deve levá-los em conta durante a realização de seu trabalho. São eles:

- abstração
- simplicidade
- formalidade
- decomposição
- reutilização
- generalização
- robustez
- modularidade
- ergonomia (facilidade de uso)
- evolução
- classificação do conhecimento.

3.1.1 Abstração

Este princípio é dos mais significativos para a área de software. Como um software é nada mais que uma representação simplificada da realidade, a identificação dos aspectos relevantes de um dado componente é fundamental. Segundo Aurélio Ferreira [Aurélio, 1975], abstração é "o ato de separar mentalmente um ou mais elementos de uma totalidade complexa (coisa, representação, fato), os quais só mentalmente podem subsistir fora dessa totalidade".

Ao destacar alguns aspectos, outros são ignorados. O sucesso de um projeto de software decorre em grande parte do acerto na escolha do que é destacado e do que é ignorado. Erro comum é destacar o que deve ser ignorado e ignorar o que deve ser destacado. Requer-se em especial do modelador perspicácia neste momento para acerto nas escolhas.

Aliás, deve-se ressaltar que a computação assenta-se neste princípio: tudo o que é feito é extrair da realidade os aspectos relevantes que são levados para a implementação no computador; o que se julga desprezível, é deixado de lado.

Para ilustrar a aplicação da abstração, considere-se o desenvolvimento de um sistema de controle acadêmico de uma universidade. Podem-se citar como objetos de interesse óbvios desta aplicação alunos, disciplinas, cursos, turmas, salas. Com respeito ao objeto Aluno, os seguintes atributos podem ser relacionados como relevantes: número de matrícula, nome, endereço, sexo, número do RG, órgão de emissão do RG, data de emissão do RG, data de nascimento, CPF, filiação. Existem vários outros atributos que se poderia listar para caracterizar um aluno, como nomes dos avós, estado civil, número de dependentes do aluno.

Pela aplicação da abstração, o engenheiro de software relaciona somente atributos relevantes para a aplicação. Isto vale para as operações que serão captadas para a representação computacional: nem todas interessam à aplicação.

3.1.2 Simplicidade

A simplicidade deve ser um princípio a perseguir sempre. Roger S. Pressman [Pressman, 2006] afirma que "na verdade, os projetos mais elegantes costumam ser os mais simples". Lembrando Kepler (1619) *apud* [Giannetti, 2008]: "*Natura simplicitatem amat*. (A natureza ama a simplicidade)". Da mesma fonte, de Schopenhauer (1851): "Quem quer que tenha algo verdadeiro a dizer se expressa de modo simples. A simplicidade é o selo da verdade". Ainda desta fonte, de G. H. von Wright (1982): "O ensinamento dos grandes homens tem com frequência uma simplicidade e uma naturalidade que faz o que é difícil parecer fácil de aprender". Para finalizar as citações contidas em [Giannetti, 2008]: de Wittgenstein (1922): "Tudo o que pode ser dito pode ser dito claramente".

John Maeda [Maeda, 2007, p. 1], em "As Leis da Simplicidade", Novo Conceito Editora, afirma que "a maneira mais simples de alcançar a simplicidade é por meio de uma redução conscienciosa". Em caso de dúvida, ele sugere que se deve eliminar sempre. Maeda aponta que a verdadeira simplicidade é conseguida com a redução ou a eliminação de uma funcionalidade se não houver desvantagem significativa.

Maeda sugere outras estratégias adicionais quando os descartes já foram feitos e se busca maior simplificação: encolher, ocultar e agregar. Ao encolher um objeto, a simplicidade advirá do prazer inesperado de se ter algo que seria insignificante, mas que ultrapassa as expectativas pelo tamanho. A ocultação, por sua vez, é uma abordagem usual na computação: as barras de menu e de ferramentas ocultam funcionalidades, criando a ilusão de simplicidade. A operação de agregação tem a ver, na engenharia de produtos, com a utilização de material que garanta maior durabilidade a um produto.

Outra lei da simplicidade proposta por Maeda (e que faz parte de seu decálogo) é a organização. Deste decálogo, uma lei relacionada ao tempo: Maeda afirma que a economia de tempo transmite simplicidade, pois, para alguém obrigado a esperar, a vida lhe parece desnecessariamente complexa. Uma lei relacionada ao aprendizado: Maeda afirma que "o conhecimento torna tudo mais simples" [Maeda, 2007].

Um dos sete princípios centrais da Engenharia de Software propostos por David Hooker (*apud* Pressman, 2006) é o princípio KISS (*Keep It Simple, Stupid*!) – mantenha a simplicidade, idiota!

A experiência de utilização de qualquer software ratifica a aplicação de Pareto (a curiosa relação 80/20): 80% do tempo do usuário concentram-se na utilização de 20% das funcionalidades do software. Se é assim, deve-se dar destaque a este subconjunto de funcionalidades, priorizando-os no desenvolvimento e nos testes. Com frequência, este subconjunto é constituído das funcionalidades mais simples, mais básicas e mais importantes para o usuário.

3.1.3 Formalidade

Este princípio identifica o nível de rigor formal com que o software deve ser especificado. Uma especificação formal apresenta três propriedades básicas: não ambiguidade, consistência e integridade [Pressman, 2006]. Observadas estas três propriedades, pode-se esperar que a especificação tenha melhor qualidade.

3.1.4 Decomposição

Este é o princípio basilar que sustenta a Teoria de Sistemas. Dado um elemento complexo para analisar, este princípio sugere a identificação de seus elementos constitutivos. Desta forma, identificados estes elementos, eles são estudados, um a um, buscando-se compreender os inter-relacionamentos existentes. O princípio "Divida para conquistar" é uma forma adequada de tratar a complexidade dos objetos sob estudo.

De resto, como é característico no conceito de sistema, a decomposição pode ser feita em vários contextos. O sistema pode ser decomposto em seus subsistemas. Um produto pode ser decomposto em seus subprodutos. Aplicando a tecnologia de Orientação a Objetos, um produto vai ser decomposto em seus objetos constitutivos. Alguns destes objetos já podem ter sido implementados anteriormente e poderão ser reutilizados. Isto garante produtividade crescente, à medida que mais objetos são implementados.

3.1.5 Reutilização

Como mencionado no princípio anterior – decomposição -, ao identificar componentes de um sistema, alguns objetos podem atuar em vários deles. O princípio da reutilização busca racionalizar os esforços de desenvolvimento, de forma que os componentes necessários em mais de uma aplicação, sejam desenvolvidos uma única vez e reaproveitados.

Seguir este princípio significa: evitar redundância de tarefas, aumentar a produtividade de desenvolvimento, melhorar a qualidade do produto – componentes reutilizados já foram testados – evitando-se erros futuros que componentes novos poderiam potencialmente apresentar.

3.1.6 Generalização

Este princípio busca considerar uma solução geral para um dado problema a tratar. Por meio dele, por exemplo, um processo de pagamento num sistema de vendas consideraria todas as modalidades de pagamento existentes (dinheiro, cheque, transferência eletrônica de fundos, cartão de crédito, etc.). Esta solução genérica – apesar de exigir provavelmente mais tempo de desenvolvimento – apresenta mais alto grau de reutilização e de adaptação a mudanças nas regras de negócio.

3.1.7 Robustez

É característica que o software deve apresentar relacionada ao funcionamento mesmo em condições adversas, como ao receber dados reconhecidamente inválidos ou quando exigido a operar com recursos computacionais insuficientes. Para que o software apresente esta característica o trabalho desenvolvido

na fase de projeto e, em especial, na fase de programação deve ser cuidadoso. Os mecanismos de tratamento de exceção, disponíveis nas linguagens de programação, devem ser explorados adequadamente.

Diz-se que um software é robusto se, diante de condições desfavoráveis ou de um ambiente inadequado, ele não para de forma anormal (não "aborta"); nestas condições, o software percebe a situação de erro, a notifica ao operador ou usuário, recomendando, se possível, a ação corretiva e interrompe a execução.

3.1.8 Modularidade

É o princípio que estabelece que um software seja particionado em módulos (ou componentes) independentes, integrados para atender aos requisitos do problema em questão. Trabalhar com o particionamento de um problema em componentes ou módulos é uma forma de diminuir a complexidade inerente ao problema. Ao mesmo tempo, quando necessário corrigir ou alterar a funcionalidade de dado componente, o esforço exigido será menor. Há duas medidas para avaliar a efetividade do particionamento: coesão e acoplamento.

A coesão avalia em que medida um componente é constituído de partes comprometidas com a realização de uma dada função (a função do componente). Se todas as partes do componente concorrem para a realização da função, o nível de coesão é dito funcional (nível mais desejável); se as partes concorrem para a realização de mais de uma função, o componente não apresenta coesão funcional: poderá apresentar coesão sequencial, comunicacional ou outra. O nível de coesão mais indesejável é o coincidental, em que o componente executa mais de uma função, e estas funções não apresentam nenhum relacionamento lógico entre si. Além disto, há necessidade de passagem de uma informação de controle ao componente para ativar a função requerida dentre as funções existentes no componente. Analisando-se o componente, percebe-se que é por coincidência (daí o nome) que as partes estão juntas; não há outra justificativa para a formação do componente. Foge ao escopo deste livro detalhar todos os tipos de coesão que compõem a escala que vai da coesão desejável (funcional) até a menos desejável (coincidental).

O acoplamento mede a forma com que um componente se interconecta com outro. A forma de acoplamento desejável (mas utópico) é que a interconexão se dê sem acoplamento: um componente A chama um componente B sem lhe passar nenhum parâmetro, como também não recebe nada dele. Esta seria a forma de conexão ideal: um componente não recebe nenhuma influência da ação do outro componente. Em caso de existência de erro, consegue-se um isolamento completo deste, pois nenhum componente foi contaminado, já que não houve passagem de parâmetros. Dado que esta forma de acoplamento é irrealizável, a forma mais aceitável de acoplamento entre componentes é o de dados; ou seja, dados são passados de um componente para o outro. No outro extremo da escala da medida acoplamento, tem-se o acoplamento comum; nesta forma, há uma área comum acessível aos componentes. Neste caso, na ocorrência de erro, não se sabe *a priori* onde ele se originou: como a área é comum, qualquer um dos componentes pode ter gerado o erro. Este tipo de acoplamento ocorre quando o programador utiliza variável global no seu programa. Também foge ao escopo deste livro detalhar todos os tipos de acoplamento que compõem a escala que vai do acoplamento mais desejável (de dados) até o menos desejável (comum).

3.1.9 Ergonomia (Facilidade de Uso)

Este princípio associa-se à adequação da facilidade de uso do software pelos seus usuários. A observância deste princípio exige que toda a comunidade de usuários seja estudada, de modo a se conceber uma interface apropriada a cada perfil de usuário.

De nada adianta um software que implemente as funcionalidades necessárias, mas que apresente exigências ou habilidades que seus usuários não têm condições de atender.

3.1.10 Evolução

O desenvolvimento de um software exige um investimento considerável da empresa. Este investimento precisa ser preservado, deve ter um tempo de vida útil o mais longo possível. O princípio da evolução exige que acréscimos de funcionalidades sejam feitos sem pôr a perder o produto. Este é um princípio de difícil concretização: exige projeto e implementação criteriosos.

Na concepção do software, o engenheiro de software deve tê-lo na devida conta já que é inevitável que correções, aperfeiçoamentos, adaptações sejam feitos depois da implantação do produto.

M. Lehman *et als apud* [Pressman, 2006] relaciona oito leis que tratam de aspectos da evolução do software. Para citar duas destas leis: 1) a "Lei da Alteração Contínua" (1974) estabelece que "Sistemas Tipo-E (evolutivos) devem ser continuamente adaptados senão ficam cada vez menos satisfatórios"; 2) a "Lei da Complexidade Crescente" (1974) dispõe que "Quando um sistema Tipo-E evolui, sua complexidade aumenta a menos que seja realizado esforço para mantê-la ou reduzi-la". A fonte citada por Pressman (de onde se pode obter detalhamento das leis da evolução do software) é:
Lehman, M., et al, "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings of the 4th International Software Metrics Symposium (METRICS '97),* IEEE, 1997.

O *download* pode ser feito de**:**
http://www.ece.utexas.edu/~perry/work/papers/feast1.pdf.

3.1.11 Classificação do Conhecimento

Este princípio busca fazer com que o engenheiro de software perceba possíveis agrupamentos e relacionamentos do conhecimento da área de interesse. A observância deste princípio assegura simplificações e evita redundâncias e é a fonte vital para a formulação da arquitetura do software. Busca-se aqui perceber generalizações, especializações, estruturas todo-parte, associações de toda natureza presente no universo de discurso da área de conhecimento em questão, não só que se refiram a dados, mas também a processos ou operações realizados.

3.2 AFINAL, QUE É PROCESSO DE SOFTWARE?

Shari L. Pfleeger [Pfleeger, 2004] define processo de software como um conjunto de tarefas ordenadas que, se executadas, levam ao desenvolvimento de software. Tanto restrições como disponibilidade de recursos são levados em conta no desenvolvimento. A definição de um processo de software pressupõe três elementos:

- sequência de tarefas ordenadas;
- restrições a considerar no desenvolvimento (cronograma, etc.);
- recursos necessários para o desenvolvimento (pessoal, ferramentas, etc.).

Por sua vez, Ian Sommerville [Sommerville, 2003] define processo de software como o conjunto ordenado de atividades que levam a resultados intermediários e que têm como fim a produção de software de qualidade.

A produção de software se faz, em grande parte das vezes, a partir da adaptação de sistemas existentes. Daí que é desejável que o desenvolvedor tenha grande capacidade de avaliação crítica de sistemas para propor sua solução a partir da análise do sistema (automatizado ou não) que está em funcionamento.

Há Expressões Sinônimas para Processo de Software?

Sim. Uma expressão mais antiga é ciclo de vida de software. Outra expressão usual é paradigma de engenharia de software. No entanto, processo de software é a expressão mais usada hoje.

3.3 QUANTOS E QUAIS SÃO OS PROCESSOS DE SOFTWARE?

Há muitas variações de processos descritos na literatura de engenharia de software. Os modelos de processo mais referidos são [Pressman, 2006], [Sommerville, 2008], [Gustafson, 2003], [SWEBOK, 2004]:

- modelo cascata;
- modelo de protipagem;
- modelo incremental;
- modelos evolucionários;
- modelo baseado em componentes reutilizáveis;
- modelo espiral.

Há ainda outros modelos de processos que precisam ser citados: modelo de engenharia de software sala limpa, modelo de engenharia da *web*, métodos formais, modelo de desenvolvimento orientado a aspectos.

Todos estes modelos de processo são ditos prescritivos porque estabelecem estruturação e ordenação das tarefas. Mais recentemente foram propostos modelos de processo que subvertem todos os pressupostos existentes até então. São os chamados modelos não prescritivos que, como foi citado, se fundamentam exatamente no contrário do que recomenda a engenharia de software no seu desenvolvimento previsível ao longo de sua existência: qual seja, a formulação de princípios rígidos que procuram estruturar e disciplinar o processo de desenvolvimento de software. O chamado "movimento ágil" (por meio do manifesto ágil) colocou por terra muitos dos pressupostos vigentes, então. Resumidamente o que o manifesto expressa: entrega incremental do software desde o início, satisfação do cliente (há muito busca-se isto), pequenas equipes altamente motivadas, ênfase na simplicidade, poucos artefatos intermediários [Pressman, 2010].

Ressalte-se que este manifesto resultou de juntar tentativas isoladas que consagravam uma nova forma de desenvolver software. Assim, o primeiro trabalho nesta corrente é devido a Kent Beck, com a Programação Extrema (referida abreviadamente como XP). Adiante a razão do adjetivo "extrema" é explicado. O próprio nome já sugere a postura adotada pela abordagem. Adiante serão apresentados os pontos básicos em que se baseia esta abordagem. Muitas outras propostas de modelos de processo podem ser classificadas como modelos não prescritivos (e são citados no manifesto ágil): Scrum, Crystal, DSDM (www.dsdm.org), Desenvolvimento Adaptativo de Software, FDD – *Feature Driven Development* – Desenvolvimento Orientado por Características e Modelagem Ágil, de Scott Ambler (www.agilemodeling.com), Desenvolvimento de Software Simplificado (*Lean Software Development*) e Processo Unificado Ágil (*Agile Unified Process* - AUP). Alguns detalhes adicionais das abordagens XP, Modelagem Ágil e Processo Unificado Ágil são apresentados adiante.

3.4 COMO ESCOLHER UM MODELO DE PROCESSO PARA DADO SISTEMA?

Há alguns fatores que podem recomendar a utilização de determinado modelo de processo. Por exemplo, o emprego de certo método ou ferramenta pode levar à aplicação de dado modelo. A própria característica da aplicação (sua abrangência, sua complexidade, a maior ou menor experiência dos usuários envolvidos, etc.) pode recomendar a utilização de um modelo de processo.

O engenheiro de software (em acordo com a gerência de desenvolvimento de sistemas) pode aceitar até uma abordagem mista, dadas as peculiaridades da aplicação e da comunidade de usuários envolvidos e do controle de desenvolvimento exigido. Assim, por exemplo, pode-se iniciar usando prototipagem para formular um projeto preliminar de interação humano-computador, se este aspecto é determinante para o sucesso do sistema a ser desenvolvido. Concluído este trabalho, poder-se-ia prosseguir com o modelo em cascata ou com o modelo incremental.

Noutra situação, dado o nível de envolvimento dos usuários e sua experiência de participação em desenvolvimento de sistemas, poder-se-ia recomendar, de pronto, o modelo espiral. Ou se poderia escolher, mesmo, uma abordagem ágil.

Portanto, uma estratégia é ter a indicação do modelo de processo a ser utilizado por ocasião da etapa de planejamento. Ao definir-se o escopo do sistema, para fazer as estimativas de recursos, pode-se incluir no plano de desenvolvimento o modelo de processo a ser utilizado.

Há empresas que adotam um processo padrão, com base nos fatores mencionados acima (abrangência, complexidade, experiência prévia dos usuários, etc.), avaliam a conveniência de fazer uma simplificação nas fases (e suas tarefas) como também nos artefatos elaborados e documentos intermediários e finais produzidos e até na rigidez dos procedimentos de revisão que serão adotados.

Do exposto, depreende-se que o engenheiro de software deve conhecer bem todos os modelos de processo, para fazer a escolha mais apropriada para a aplicação que vai desenvolver, se lhe for concedida esta possibilidade de decisão.

Ian Sommerville [Sommerville, 2008] afirma que as tentativas de automatizar processos de software não apresentam resultados satisfatórios. A dependência de julgamento humano e a criatividade envolvida na sua aplicação impedem o sucesso destas iniciativas. Com isto, não se descartam a utilização de ferramentas CASE (*Computer Aided Software Engineering* – Engenharia de Software Assistida por Computador) para atividades específicas, principalmente quando se busca alterar algum requisito e consequentemente obter nova modelagem, como também a possibilidade de disponibilizar a documentação para outros participantes da equipe que vão prosseguir o trabalho.

3.5 CLASSIFICAÇÃO DE MODELOS DE PROCESSOS

Duas famílias de modelos de processo existem: Modelos Prescritivos e Modelos Não Prescritivos ou Ágeis.

3.5.1 Modelos Prescritivos

Os modelos ditos prescritivos são os mais tradicionais da Engenharia de Software. Eles preceituam que o desenvolvimento do software deve se dar com forte ênfase na organização e disciplina rigorosa de passos. A abordagem mais tradicional nesta família é o modelo em cascata. Os mais importantes modelos prescritivos propostos serão apresentados adiante.

3.5.2 Modelos Não Prescritivos ou Ágeis

Estes modelos de processo baseiam-se no trabalho de um grupo de desenvolvedores que propugna pela volta às práticas anteriores à formalização da disciplina Engenharia de Software. É sabido que o desenvolvimento de software chegou a ser considerado um trabalho artístico, pela natureza e características do produto elaborado. Buscou-se, então, a partir daí, formalizar métodos de trabalho para disciplinar a tarefa. Até chegar-se à proposição da tarefa de desenvolvimento de software como uma disciplina de engenharia: aí surgiu a Engenharia de Software. Todos os esforços realizados pelos que abraçaram esta área concentraram-se no sentido de formular métodos, metodologias, técnicas, elaborar ferramentas e estabelecer práticas voltadas para a organização e a disciplina da tarefa de desenvolvimento de software. Esta caminhada justificou-se plenamente em razão da demanda de software requerida pela sociedade, que os desenvolvedores da época não conseguiam dar conta. De outro lado, com o barateamento do preço dos recursos computacionais, decorrentes dos avanços da microeletrônica, mais e mais empresas puderam investir na compra de computadores, com o objetivo de melhorar seus processos de trabalho e sua administração. Isto intensificou uma ocorrência que virou lei: o aumento crescente da complexidade do software. Requer-se, cada vez mais, software mais e mais complexo. Na medida em que os usuários acostumam com as funcionalidades disponíveis em uma aplicação, passam a requisitar outras mais complexas, de modo a reduzir seu trabalho, melhorar sua gestão do negócio ou atingir algum benefício não previsto.

Ao longo dos anos, a tônica também tem sido o aumento de complexidade das organizações, requerendo a utilização de recursos computacionais mesmo para atividades antes inimagináveis.

Olhando de uma perspectiva histórica, avanços significativos foram registrados nas técnicas, nos métodos e nas ferramentas para desenvolvimento. E a perspectiva da área como uma engenharia, a engenharia de software, muito contribuiu para isto. Agora, é indiscutível que a exigência de disciplina e de organização irrestritas se contrapõe à forma como muitas aplicações se comportam: a exigência de mudanças, mais e mais frequentes, subverte esta lógica. O processo posto (organizado estritamente) não dá conta de superar as mudanças impostas por certas realidades empresariais.

Por perceber isto, alguns desenvolvedores e metodologistas se reuniram para lançar um manifesto (chamado Manifesto para Desenvolvimento Ágil de Software, disponível em www.agilemanifesto.org), expresso da seguinte forma [Beck, 2001]:

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan.
That is, while there is value in the items on the right, we value the items on the left more. “
Kent Back et als.

Tradução do autor: “Estamos descobrindo formas melhores de desenvolver software e ajudando outros a fazê-lo. Por meio deste trabalho, temos chegado a valorizar:

Indivíduos e interações ao invés de processos e ferramentas
Software operacional ao invés de documentação abrangente
Colaboração do cliente ao invés de negociação do contrato
Resposta às alterações ao invés de seguir um plano.
Isto é, enquanto há valor nos itens à direita, valorizamos muito mais os itens à esquerda.”

O manifesto é expresso, como se vê, em quatro pontos fundamentais:

1) ao invés de se fixar somente nos processos e nas ferramentas de desenvolvimento, as pessoas envolvidas e as interações entre elas têm que ser valorizadas; grande parte dos problemas com o desenvolvimento de software decorre de não se atentar adequadamente para as dificuldades decorrentes destas interações;

2) ao invés da fixação em produzir uma documentação abrangente (e volumosa), deve-se buscar produzir o mais cedo possível um software que funcione. Esta percepção favorece o entendimento entre as partes – desenvolvedores e interessados –, pois há algo – o software desenvolvido – que permite uma avaliação concreta. O paradoxo aqui é que o software é um bem abstrato, que representa algo concreto (a área da empresa ou a própria empresa representada por ele). A volumosa documentação é muito abstrata para ensejar avaliações mais pertinentes;

3) ao invés de a participação dos usuários restringirem-se à negociação do contrato de desenvolvimento e da definição do escopo do sistema, deve-se contar com a participação dos usuários na equipe de desenvolvimento;

4) ao invés de obedecer rigorosamente ao estabelecido no plano do projeto, havendo a percepção da necessidade de mudança nos requisitos, deve-se fazê-la imediatamente. O plano de projeto deve ser ajustado para refletir as mudanças necessárias.

3.6 MODELOS PRESCRITIVOS

Os modelos prescritivos incluem o modelo cascata, modelo de prototipagem e o modelo evolucionário. Cada um destes modelos é descrito detalhadamente em seguida. O RUP – Processo Unificado também é um modelo prescritivo; será apresentado na seção 3.8 adiante, e com um pouco mais de detalhes no Apêndice B.

3.6.1 Modelo Cascata

Como o nome sugere, este modelo desdobra o desenvolvimento em várias etapas sequenciais; os resultados de uma etapa possibilitam a execução da próxima e, assim, sucessivamente, chega-se à conclusão do software. Este modelo teve origem em projetos de engenharia, anteriores à formulação da engenharia de software. Pressupõe que o sistema estará totalmente pronto quando a última etapa for concluída.

A Figura 3.1 apresenta o modelo cascata. A primeira fase do modelo (Decisão de Novo Sistema) começa com a percepção da necessidade de se desenvolver o novo sistema; segue-se ao Estudo de Viabilidade do desenvolvimento; considerando que seja viável (economicamente, tecnicamente,

administrativamente), faz-se o planejamento do desenvolvimento, com base nas informações já coletadas. A terceira fase (Requisitos) consiste em estudar minuciosamente os requisitos, buscando-se estabelecer prioridades, resolver possíveis conflitos. O resultado desta fase é o documento de especificação de requisitos, que deve ser submetido à comunidade de usuários para apreciação. Se aprovado, a equipe de desenvolvimento procederá ao detalhamento da Análise. A conclusão desta etapa é sinalizada com a elaboração do documento de especificação de análise. A fase seguinte (Projeto) consiste em produzir o modelo físico do sistema; os dados obtidos serão traduzidos em programas, componentes, interfaces, bancos de dados. O documento que sinaliza a conclusão desta etapa é o documento de especificação de projeto, que deve conter basicamente: projeto arquitetural, projeto de componentes, projeto de banco de dados, projeto de interação humano-computador. A fase seguinte (Programação) consiste em traduzir o modelo físico em modelo de implementação no computador; os programas e os componentes são programados numa linguagem de programação, os modelos de dados são traduzidos em esquemas do sistema de bancos de dados utilizado. À medida que os programas e componentes são programados, eles são testados. Concluída a fase de Testes, procede-se à Implantação do sistema. Concluída a implantação, o sistema entra em fase de Operação. Estando em operação, é possível haver necessidade de correções, adaptações, aperfeiçoamentos (fase de Manutenção). A avaliação da manutenção realizada pode determinar que um novo sistema seja desenvolvido; aí volta o ciclo para a fase inicial. Observa-se neste modelo que, numa fase, é sempre possível retornar à anterior. Neste livro, as três fases em que nos concentraremos são as fases de Requisitos, de Análise e de Projeto; parte significativa do trabalho do analista de sistemas localiza-se nestas fases.

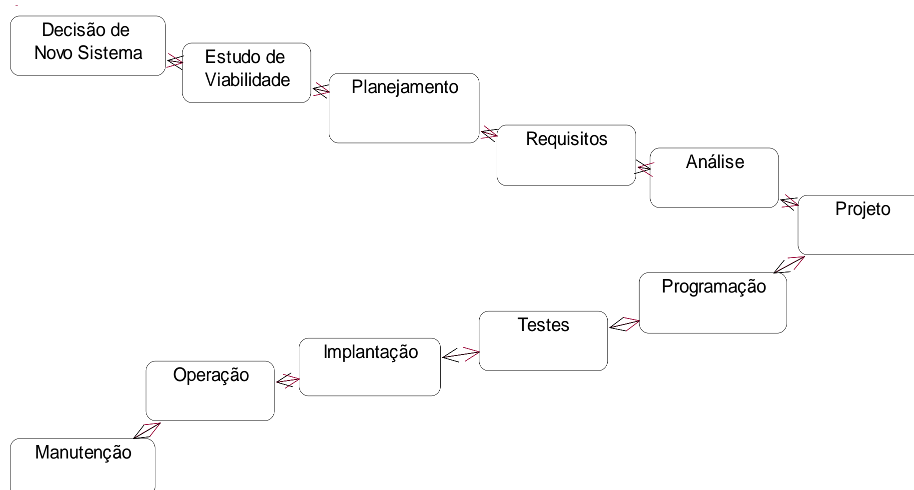

Figura 3.1 Modelo em cascata.

Esta abordagem considera que uma fase só iniciará quando a anterior tiver sido concluída. Exatamente aí reside a principal crítica a este modelo para sua completa aceitação. A prática de desenvolvimento de software mostra que há forte sobreposição entre as etapas do processo. Isto contraria frontalmente a metáfora do modelo. A água da cascata, depois que cai, não volta ao cimo.

Por mais que seja cuidadoso o trabalho numa fase, e por mais que se avalie este resultado, para que o fluxo siga sequencialmente, é recorrente a necessidade de fazer ajustes em etapas anteriores. Gane/Sarson [Gane & Sarson, 1983] dizem que, ao invés de o fluxo de desenvolvimento de software ser sequencial, ele tende mais ao espiralado, como uma mola distendida.

Toda vez que há alguma iteração, o trabalho das etapas anteriores deve ser revisto ou refeito. Isto inevitavelmente encarece o desenvolvimento.

Problemas do Modelo Cascata

O principal problema ocorre quando há modificações nos requisitos ou quando há incerteza ou dificuldade de explicitá-los. Ora, assim o projeto não pode avançar. Em especial, a questão da volatilidade de requisitos é problemática nesta abordagem. Certas mudanças de requisitos (por exemplo, aquelas

decorrentes de legislação e de tecnologia) ocorridas depois da etapa de elicitação, não podem ser adiadas, pois o que se vai produzir será totalmente inútil.

Também, o cliente do software precisa confiar na equipe de desenvolvimento, já que vai demorar bom tempo para ver algo palpável. Algum erro grosseiro havido na fase de coleta de requisitos (omissão de algum requisito relevante, erro de interpretação do que seja dado requisito, valorização pelo analista de requisito irrelevante para o usuário) é desastroso para o projeto.

Quando o Modelo Cascata é Aplicável?

Quando o analista de sistemas tem plena compreensão dos requisitos necessários e/ou quando o usuário consegue explicitá-los adequadamente ao analista e, por isso, há comunicação perfeita entre eles. Isto em geral ocorre quando o cliente é usuário frequente de sistemas, ou quando tem experiência da sua utilização e tem domínio das potencialidades e limitações dos computadores.

O modelo cascata é aplicável ainda quando o conjunto de requisitos é estável e de excelente qualidade; a duração do projeto é curta (menor que dois anos); é desejável (e aceitável) que o sistema esteja disponível completo. No entanto, estas condições não costumam ocorrer mais. As gerências desejam resultados cada vez mais rápidos, os requisitos mudam com frequência maior e precisam ser acomodados no sistema logo, pois, às vezes, sem sua consideração imediata, o sistema em desenvolvimento torna-se inútil.

Quais os Pontos Fortes do Modelo Cascata?

A sua aplicação exige que o desenvolvimento seja bastante disciplinado. A conclusão de cada fase é sinalizada por documentos que devem estar disponíveis para revisão, possibilitando avaliar se aquela fase foi ou não concluída. A existência de procedimentos formais de verificação[1] e validação[2] minimiza os problemas de erros cometidos numa fase repassarem para as posteriores.

_____________________ rodapé

[1]A palavra verificação refere-se a confrontar a especificação utilizada para produzir um artefato com este artefato, para avaliar sua correção.

[2]A palavra validação, quando aplicada em engenharia de software, refere-se a confrontar um artefato com as expectativas do usuário. Ou seja, o usuário será consultado neste processo, fazendo a avaliação do artefato quanto ao atendimento de suas expectativas.

3.6.2 Modelo de Prototipagem

Este modelo leva ao desenvolvimento de um protótipo, que é descartado depois. O protótipo é elaborado para ajudar na identificação de requisitos. Esta abordagem é adequada para reduzir riscos com o desenvolvimento, por conta de requisitos errados, incompletos, inconsistentes, omitidos.

O protótipo possibilita que o cliente avalie a implementação dos requisitos coletados até este ponto. Modificações podem ser feitas e, mesmo, novos requisitos podem ser acrescentados ao protótipo. A ferramenta utilizada na elaboração do protótipo deve garantir alta produtividade (a eficiência do código gerado neste caso não é relevante), de modo que as sessões de interação do analista com o cliente sejam frequentes e pouco espaçadas. Desta forma, o protótipo é aperfeiçoado continuamente, em sessões em que o ciclo "coleta requisito – implementa – avalia protótipo" se repete até que ele satisfaça ambas as partes. Neste ponto, o protótipo é descartado e o analista terá os requisitos desejados, que serão especificados para que o desenvolvimento siga então conforme o modelo clássico (ou em cascata) ou qualquer outro modelo de processo, provavelmente agora utilizando uma ferramenta de desenvolvimento cujo objetivo principal não é tanto a produtividade e, sim, a eficiência do código gerado e a economia de recursos computacionais que o sistema apresentará. Objetivos de qualidade do produto e garantia de manutenibilidade são perseguidos, o que, certamente, implicará em menor produtividade neste caso. Registre-se que durante a elaboração do protótipo o foco é obter uma solução rápida, de modo que, requisitos coletados num dia, sejam demonstrados um ou dois dias depois para o cliente. Portanto, é tolerável que o código que implementa o protótipo seja relativamente lento e até

excessivamente consumidor de recursos computacionais. Como foi dito, o destino do protótipo é o descarte.

Quando é Aplicável a Prototipagem?

Há situações em que esta abordagem é particularmente recomendada. Por exemplo, quando, por alguma razão, os requisitos não são identificados completamente. Isto pode ocorrer em razão de dificuldade de comunicação entre analista e cliente. Nem sempre o objeto do sistema (a área de conhecimento envolvida) é de domínio do analista; num caso deste, se acontecer de o cliente não estar familiarizado com potencialidades e limitações dos recursos computacionais, as dificuldades de comunicação podem ser insuperáveis. Uma solução aplicável: elaborar um protótipo para ajudar a comunicação entre as partes.

Há ainda outra situação em que esta abordagem é aplicável: na verificação da eficiência ou da correção de um algoritmo. Sem comprometer grandes recursos (tempo de pessoal, principalmente) pode-se obter uma resposta a questão relevante sobre o sistema.

Outro caso em que a prototipagem é aplicável é para estabelecer a interação dos usuários com o sistema. Isto se chama interação humano-computador (antes se dizia projeto de interfaces usuário-máquina). Cabe aqui, inclusive, a aplicação de um teste de usabilidade[3] do protótipo. Identifica-se uma amostra de cada parcela dos usuários e submete-se o protótipo a ela. Avalia-se então se o protótipo atende este perfil de usuários. Caso o resultado não seja satisfatório, alterações são efetuadas e nova submissão é feita. Este ciclo é repetido até que se tenha chegado à interface aceitável. Ou seja, definir a forma como os usuários interagirão com o sistema. Para certas aplicações, o estabelecimento de como a interação dos usuários se fará com o sistema é crucial para seu sucesso. Para citar um exemplo da importância da experimentação da interface, veja-se o caso do sistema de votação eletrônica, adotado pela Justiça Eleitoral no Brasil. A parte mais complexa desta aplicação foi exatamente o projeto da interface que garantisse que o eleitor, independente de seu nível de instrução (mesmo iletrados ou pouco habituados à utilização de máquinas) pudessem exercer seu direito de votar. De forma que experimentações com protótipos foram conduzidas, para orientar o projeto da urna eletrônica e do sistema a ser instalado para votação. Isto exigiu então que a Justiça Eleitoral antes de fazer investimentos na confecção da urna e nas considerações do software, providenciasse um protótipo para avaliar como os usuários interagiriam com o software de votação.

A falha no projeto de interfaces é grande causadora de fracassos de sistemas. Às vezes, sistemas conceitualmente muito bem desenvolvidos, com excelente projeto arquitetural e de componentes e com muito boa implementação, têm seus resultados comprometidos por conta de um projeto de interfaces pouco cuidadoso, que não levou em conta o perfil dos usuários para os quais foram desenvolvidos.

_______________rodapé

[3]Teste de usabilidade é o teste que se faz para avaliar como os usuários reais (ou uma amostra destes usuários), depois de treinados, utilizam um sistema. Busca-se avaliar potenciais problemas da interface (arrumação das informações nas telas, clareza das mensagens de erro e de advertência, etc. e de outros aspectos do sistema).

Portanto, o modelo de prototipagem tem sua aplicabilidade garantida sempre que houver riscos na forma como a interação dos usuários se dará com o sistema. Especialmente, quando houver usuários com perfis instrucionais muito variados. Um protótipo deve ser produzido e experimentações devem ser conduzidas com cada uma destas classes de usuários para se chegar a uma interface adequada.

Quais as Características de Sistemas Candidatos à Prototipagem?

1) Quando os usuários não são capazes de examinar modelos abstratos; 2) quando os usuários não são capazes de articular seus requisitos, podendo expressá-lo somente por meio de processo de tentativa e erro; 3) todo sistema “on-line” é candidato natural à prototipagem; 4) quando o sistema não exige especificação de grande quantidade de detalhes algorítmicos; 5) quando o usuário está interessado na sua interação com o sistema, e não na computação básica oferecida por ele.

Quais são as Ferramentas dos Prototipadores?

Para produzir código rapidamente, o prototipador precisa de: geradores de tela, geradores de relatórios, linguagens de quarta geração, linguagem de consulta não procedural ou geradores de aplicação.

3.6.3 Modelo Evolucionário

Ao contrário do modelo de prototipagem, em que se elabora um protótipo que é descartado depois, no modelo evolucionário (Figura 3.2) o produto elaborado não é descartado. Ele evolui a cada iteração (do tipo “coleta requisitos – projeta – implementa – valida produto”). Isto já está estabelecido: a abordagem de desenvolvimento mais adequada é aquela que se dá prevendo vários incrementos. Isto possibilita participação dos interessados mais cedo na avaliação do software em construção, com maior chance de atendimento pleno das necessidades do usuário.

O modelo evolutivo pressupõe interação forte dos desenvolvedores com os interessados; cada novo incremento incorpora novas funcionalidades ao sistema.

A principal crítica a este modelo é a dificuldade de os usuários vislumbrarem o fim do desenvolvimento do sistema – a dificuldade de haver a convergência para o final do sistema. Esta crítica pode ser atenuada se, à maneira do modelo incremental, forem definidos os requisitos básicos para a primeira versão e as sucessivas versões com as extensões do sistema. Uma característica apreciável deste modelo é a evolução do sistema com o tempo, plenamente consonante com a realidade que vivemos.

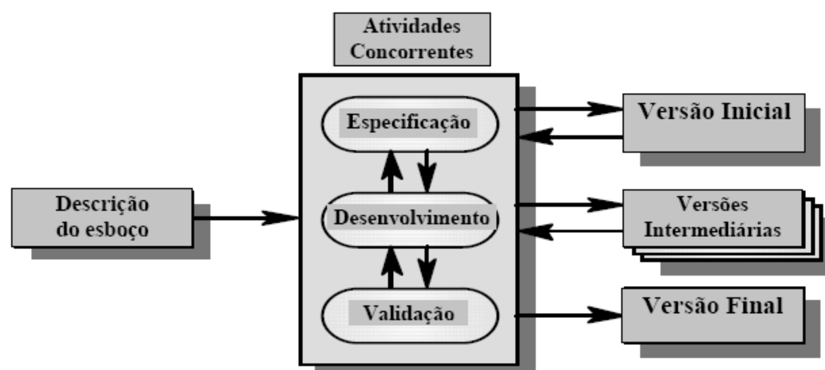

Figura 3.2 Modelo evolucionário [Sommerville, 2007].

Um exemplo de modelo evolucionário é o modelo espiral (proposto por Barry Boehm) (Figura 3.3). Este modelo apresenta quatro fases (cada quadrante da figura abaixo); o desenvolvimento se desenrola a partir do levantamento inicial de requisitos (objetivos, alternativas e restrições); em seguida, procede-se à análise de riscos para implementar os requisitos coletados. Segue-se ao desenvolvimento e teste do produto, que é entregue à comunidade de usuários para utilização e avaliação. Procede-se a novo ciclo de desenvolvimento (novos requisitos são identificados) e a espiral segue para os demais quadrantes. O software vai evoluindo até que a comunidade de usuários sinalize que nenhum outro requisito há para incorporar no produto.

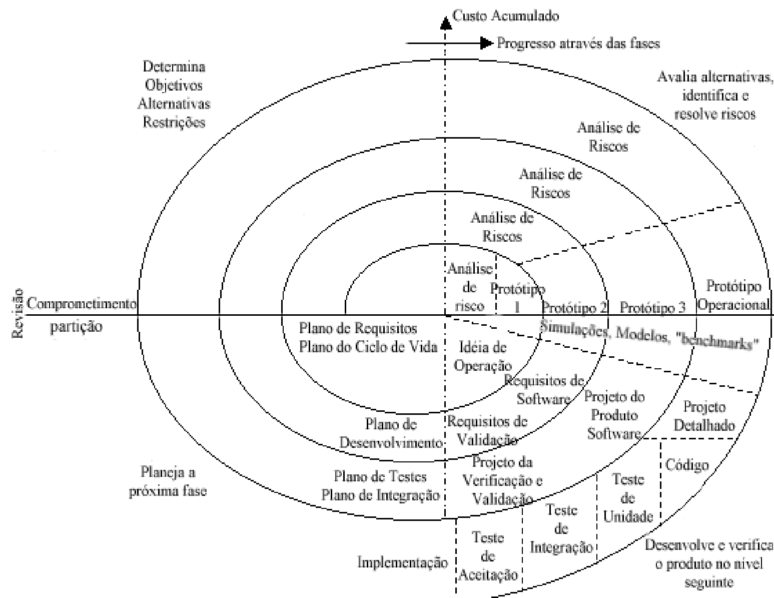

Figura 3.3 Modelo espiral [Sommerville, 2007].

Quais são os Modelos de Processos mais Usados?

O modelo em cascata é o mais tradicional e ainda é muito usado. O modelo evolucionário também é muito usado por propiciar às empresas que o adotam a possibilidade de associar a versões o *marketing* para comercialização inicial do produto. Como perspectiva de futuro, a chance maior de uso recai sobre os modelos baseados em componentes, cujo exemplo mais consagrado é o chamado Processo Unificado.

Que Critérios são Considerados para a Escolha de um Modelo de Processo?

Deve-se levar em conta a complexidade do produto a ser desenvolvido, como também sua abrangência, seu escopo. Ainda: os métodos e as ferramentas que serão utilizados no desenvolvimento; quão rigorosos serão os controles feitos sobre os produtos intermediários e finais elaborados. Quanto mais abrangente a aplicação a ser desenvolvida, certamente mais complexa será. Aplicações corporativas (aquelas que permeiam toda a organização, sendo utilizadas em todos os seus escalões) são mais complexas; exigem uma etapa de coleta de requisitos mais demorada e mais trabalhosa, e mais sujeita a erros decorrentes de conflitos de requisitos e discrepâncias em procedimentos e dados. Estas aplicações exigem um processo mais formal, com verificações e validações mais frequentes, com revisões técnicas e gerenciais a cada etapa.

3.7 MODELOS NÃO PRESCRITIVOS OU ÁGEIS

Os modelos não prescritivos ou ágeis incluem a programação extrema, a modelagem ágil, o processo unificado ágil, entre vários outros. Cada um destes modelos é descrito detalhadamente em seguida.

3.7.1 Programação Extrema

Kent Beck [Beck, 2000], criador da Programação Extrema (*eXtreme Programming*, referida como XP), acentua que ela é indicada especialmente para o desenvolvimento de software que apresenta requisitos vagos e altamente voláteis e envolvam equipes pequenas ou médias (equipes com até 10 programadores) e contem com o envolvimento dos interessados ("*stakeholders*"). Ele afirma que XP não é indicada para utilização por grandes equipes envolvidas num projeto, quando os interessados não

participam da equipe de desenvolvimento. E também quando se utiliza tecnologia inapropriada para concretização rápida de frequentes modificações.

O processo XP desdobra-se em quatro fases: planejamento, projeto, codificação e teste. Na fase de planejamento, os requisitos são coletados na forma de *user stories* (histórias do usuário); estas narrativas descrevem funcionalidades que o sistema deve apresentar e servem de base para composição das versões do sistema. A primeira versão (ou primeiro incremento) é implementada e a experiência adquirida com o desenvolvimento auxilia a elaboração das estimativas de recursos dos posteriores incrementos [Pressman, 2006].

A fase de projeto do XP prioriza a busca da simplicidade – as funcionalidades básicas são consideradas em detrimento de outras mais complexas, uma arquitetura de projeto simples é buscada. A técnica chamada refabricação (*refactoring*) é empregada; ela estabelece que, sempre que se perceber que a arquitetura projetada puder ser melhorada, isto deve ser feito; o mesmo vale para a codificação: sempre que a codificação de um componente puder ser aprimorada e, às vezes, percebe-se isto depois de escrita a primeira versão do código, a refatoração deve ser feita.

A fase de codificação começa com o projeto de casos de teste para cada componente do sistema. Depois da codificação do componente, o teste unitário é realizado. A atividade de programação é realizada por duplas de programadores: um tem participação ativa (domínio do teclado) e o outro observa, avalia, opina, critica o que é feito. Obtém-se desta forma de trabalho um resultado de melhor qualidade. Diz-se em XP que o trabalho de implementação é coletivo pelo forte envolvimento dos programadores e por não haver monopolização de um profissional no desenvolvimento de um componente.

A última fase – teste – começa com o teste unitário – teste do componente. Posteriormente, o componente é integrado ao sistema (teste de integração). Sugere-se a execução de integração contínua para eliminar mais cedo problemas de interfaces e outros. Os testes de aceitação finalizam esta fase com o objetivo de avaliar as narrativas do usuário coletadas na fase de planejamento.

Beck explica que o termo "extremo" do nome da metodologia advém de se levar a níveis extremos alguns princípios e práticas. Por exemplo, o senso comum recomenda que se revise o código produzido, então XP acentua que isto seja feito o tempo inteiro por meio da programação em duplas. Numa estação de trabalho, dois programadores trabalham: um revisa o que o outro produz, continuamente.

Da mesma forma, como é bom testar o código produzido, XP preceitua que isto seja feito sempre, por meio de testes de unidade (realizados pelos desenvolvedores) e testes funcionais (realizados pelos interessados). Os testes de unidade concentram-se no código escrito; buscam verificar se o código obedece à especificação e como ele faz isto. Os testes funcionais concentram-se nas funcionalidades executadas; buscam verificar se os requisitos funcionais são atendidos corretamente. Reforçando o que foi mencionado XP também recomenda que o código produzido seja reprojetado sempre que se perceber que é possível melhorá-lo, por meio da refatoração.

Assim que cada componente é concluído, é integrado logo ao sistema e os testes de integração são realizados; isto ocorre ao longo de todo o tempo de desenvolvimento.

Outro ponto fundamental é que os requisitos mais importantes sejam implementados primeiro. O princípio de Paretto estabelece que os usuários passam 80% do tempo executando 20% das funcionalidades existentes. Por isso, é importante descobrir as funcionalidades que compõem estes 20%.

Uma questão relevante para a XP é a execução de mudanças no sistema. Isto é indesejável, mas inevitável. Em geral, as abordagens prescritivas não convivem bem com mudanças de requisitos, mas isto é uma inevitabilidade do mundo real.

Kent Beck [Beck, 2004] fixou alguns princípios básicos para a XP:

- Buscar insistentemente a simplicidade: o princípio expresso no acrônimo KIS ("*Keep It Simple*" – mantenha a simplicidade) é ponto-chave.

- Mudanças incrementais: a partir do conjunto de funcionalidades que constituam o primeiro incremento do software, mudanças pequenas são realizadas. Esta prática é recomendada em todos os aspectos em que mudanças sejam requeridas no desenvolvimento de um sistema, seja a mudança de requisitos, seja o projeto do sistema, seja a codificação (com a refatoração), seja a abordagem de testes.

- Buscar *feedback* rápido: concluído um incremento, buscar *feedback* dos interessados o mais rapidamente possível. Garante-se assim convergência rápida entre desenvolvedor e interessado: se algo

estiver errado ou insatisfatório, os interessados notificarão os problemas, que poderão ser resolvidos tempestivamente.

XP também exercita algumas práticas importantes para o sucesso do método [Beck, 2004]:

- planejar sempre o próximo incremento;

- garantir entregas frequentes para os interessados.

- buscar implementar primeiramente as funcionalidades básicas. Lembrar que, em geral, os interessados vão gastar 80% do seu tempo com apenas 20% das funcionalidades do sistema.

- testar o mais cedo possível, e sempre.

- refatoração: o melhor código quase sempre não é obtido da primeira vez; sempre que se perceber que há algum trecho a melhorar, procurar fazê-lo.

- trabalhar em duplas: um programador codifica, o outro revisa. Garante-se assim melhor qualidade do código escrito.

- a propriedade do código é de todos, não é autoral. Os recursos e as técnicas utilizadas devem ser de domínio de todos; rejeitam-se construções e técnicas incompreensíveis. Os padrões da instalação são exigidos de todos.

- realizar integração continuamente. Logo que um componente é concluído, sua integração ao sistema é realizada.

- semana sem horas extras. XP prega que todo trabalho seja feito com carga horária normal. A utilização frequente de horas extras denuncia falhas gerenciais ou técnicas.

- participação de um interessado com conhecimento e poder decisório na equipe de desenvolvimento.

3.7.2 Modelagem Ágil

A Modelagem Ágil foi proposta por um dos signatários do Manifesto Ágil – Scott S. Ambler, com o objetivo de descrever como a modelagem de sistemas pode ser conduzida de maneira a abreviar a duração deste trabalho. Portanto, é dirigida às fases de análise e projeto de sistemas.

Ambler [Ambler, 2004] sugere a utilização da Modelagem Ágil (MA) seguida pela Programação Extrema (XP), para implementar a modelagem, assim como também sugere a utilização da MA ao longo do ciclo de vida do Processo Unificado.

Ambler afirma que a MA tem três objetivos:

1) “Definir e mostrar como colocar em prática um conjunto de valores, princípios e práticas relativas a uma modelagem eficaz e leve”;
2) “Lidar com a questão de como aplicar técnicas de modelagem em projetos de software adotando uma perspectiva ágil”;
3) “Discutir como se pode melhorar as atividades de modelagem adotando uma perspectiva ‘quase ágil’ no caso de se ter adotado um processo prescritivo qualquer (RUP ou outro)”.

Ambler diz que o foco da MA é a modelagem e a documentação eficazes, e baseia-se em princípios provenientes do Manifesto Ágil (mostrado no item 3.6.2), como a busca da simplicidade ao modelar e a encampação da mudança (é inevitável que ocorra: os requisitos mudam, por mais que não se queira que aconteça).

Alguns dos princípios citados por Ambler não constituem nenhuma novidade em relação ao que foi dito no Manifesto e em outras abordagens:

- o software é o principal objetivo; a modelagem é apenas um passo na direção de se produzir o software;

- a simplicidade é a busca constante;

- a mudança deve ser incorporada o mais cedo possível, e feita de forma incremental;

- se necessário, utilizar mais de um modelo, desde que cada um acrescente um aspecto relevante do problema;

- como o usuário está presente, obter avaliação dele do que é produzido o mais cedo, evitando retrabalho.

Ambler afirma que MA baseia-se na Programação Extrema: aproveita seus princípios gerais, incluindo poucos outros.

Ambler é defensor de um posicionamento bastante discutível, questionável: "o desenvolvimento de software é mais uma arte do que uma ciência, uma arte que requer artesãos habilidosos". Isto significa remontar aos primórdios da área de software, quando havia o convencimento de que programar computadores constituía uma arte.

3.7.3 Processo Unificado Ágil (AUP – *Agile Unified Process*)

Trata-se de uma versão simplificada do *Rational Unified Process* – RUP, ajustada para desenvolver aplicações comerciais, com o uso de técnicas ágeis, para melhorar a produtividade.

Na proposta formulada por Scott W. Ambler para o Processo Unificado Ágil são mantidas as fases do RUP, a saber: Concepção, Elaboração, Construção e Transição. Há, no entanto, mudança nos fluxos de trabalho, sendo propostos: Modelo, Implementação, Teste, Implantação, Gestão de Configuração, Gestão de Projeto e Ambiente. O assim chamado "gráfico de baleias" do RUP toma a forma mostrada na Figura 3.4.

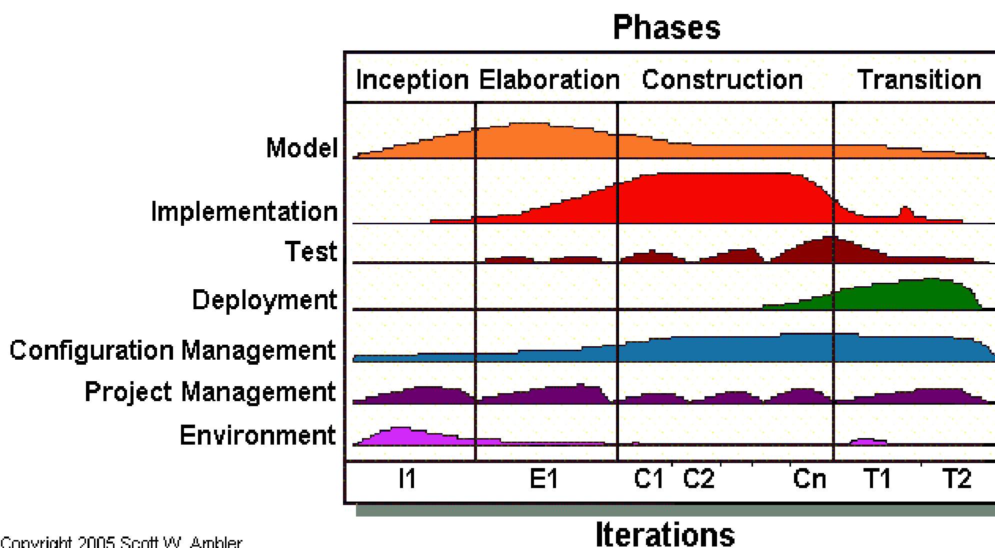

Figura 3.4 "Gráfico de Baleias" do AUP.

Observando-se a Figura 3.4, percebe-se que os fluxos de trabalho Modelo (*Model*), Gestão de Configuração (*Configuration Management*) e Gestão de Projeto (*Project Management*) abrangem as quatro fases (concepção, elaboração, construção e transição). A disciplina Modelo concentra-se mais na fase de elaboração; a disciplina Implementação concentra-se principalmente na fase de construção; a disciplina Teste inicia na fase de elaboração e vai até a transição; a disciplina Implantação concentra-se na fase de transição.

As Grandes Fases do AUP

Como visto na Figura 3.4, as quatro fases sequenciais do AUP são: 1) **Concepção**: o objetivo é identificar o escopo inicial do projeto, uma arquitetura potencial para o sistema e obter os recursos iniciais do projeto e a aceitação dos interessados; 2) **Elaboração**: o objetivo é elaborar a arquitetura do sistema; 3) **Construção**: seu objetivo é desenvolver o software de maneira incremental para atender os requisitos de maior prioridade dos interessados do projeto; 4) **Transição**: o objetivo é validar e implantar o sistema no seu ambiente para produção.

Iteração nos Fluxos de Trabalho

As disciplinas (*workflows*) são executadas de maneira iterativa, definindo as atividades que membros da equipe de desenvolvimento executam para construir, validar e entregar o software que implemente os requisitos dos interessados [Ambler, 2005].

As disciplinas são [Ambler, 2005]:

Modelo: o objetivo desta disciplina é buscar compreender o negócio da organização e ter domínio do problema com vista a identificar uma solução viável para o domínio em questão.

Implementação: o objetivo desta disciplina é transformar o(s) modelo(s) em código executável e testá-lo, em particular executar teste unitário de cada componente.

Teste: o objetivo desta disciplina é executar uma avaliação objetiva do incremento para garantir qualidade. Isto inclui procurar erros, para validar que o sistema funciona como projetado e confirmar que os requisitos são atendidos adequadamente.

Implantação: o objetivo desta disciplina é planejar a entrega do sistema e executar o plano com vista a entregá-lo para os usuários finais.

Gestão de Configuração: o objetivo desta disciplina é gerenciar e controlar o acesso aos artefatos do projeto, em especial quando modificações são realizadas, de modo a manter registro das versões dos artefatos ao longo do tempo.

Gestão de Projeto: o objetivo desta disciplina é gerenciar as atividades que compõem o projeto. Isto inclui: gerenciar riscos, administrar pessoal (alocar pessoal, acompanhar progresso, etc.) para assegurar que o projeto será entregue no prazo e dentro do orçamento.

Ambiente: o objetivo desta disciplina é garantir que o processo apropriado, os padrões e as ferramentas necessárias à execução do projeto (hardware, software, etc.) estejam disponíveis quando necessário.

Filosofias do AUP

Scott Ambler [Ambler, 2005] aponta algumas filosofias do AUP: a simplicidade (cada artefato é descrito concisamente); a agilidade (o AUP obedece aos princípios constantes do Manifesto Ágil); o foco recai sobre as atividades de mais alto valor, naquelas que efetivamente contam; a independência de ferramenta (neste quesito, devem-se buscar as ferramentas mais simples, por exemplo, aquelas de código aberto); o pessoal sabe o que está fazendo, sem precisar ler documentação extensa sobre o processo – têm isto por meio de manual reduzido e/ou por meio de treinamento periódico.

3.8 PROCESSO UNIFICADO RATIONAL - RUP

O que consta nesta seção foi extraído parcialmente do artigo "RUP – Processo Unificado *Rational*", de [Costa Jr & Furtado, 2007] – com o objetivo de fazer uma apresentação inicial sobre o assunto; mais informações sobre o RUP podem ser encontradas no Apêndice B deste livro.

Este modelo de processo é destacado aqui pela sua importância: é resultado de um esforço de unificação dos trabalhos de três grandes metodologistas; é um processo prescritivo com características incorporadas a partir dos trabalhos de vários outros autores.

O Processo Unificado Rational (*Rational Unified Process - RUP)* surgiu, já na década de 1990, como uma proposta de unificação das metodologias de desenvolvimento de software orientadas a objetos. O RUP não apenas engloba os trabalhos de seus três autores (Ivar Jacobson, Grady Booch e James Rumbaugh), como também incorpora inúmeras contribuições de outros autores. Dessa forma, ele consegue capturar muitas das melhores práticas de desenvolvimento de software em um único processo de desenvolvimento.

O RUP é baseado em componentes, o que significa que o sistema é construído a partir de componentes de software interconectados por meio de interfaces muito bem definidas. Além disso, o RUP utiliza como notação a Linguagem de Modelagem Unificada (UML) na representação de todos os artefatos do sistema. Na verdade, a UML e o RUP são elementos complementares no esforço conjunto de Ivar Jacobson, Grady Booch e James Rumbaugh no sentido de unificar os aspectos referentes à

modelagem e criação de software orientado a objetos. O primeiro fruto desses esforços foi a UML, que unificou a modelagem dos artefatos de software. Posteriormente, o RUP foi desenvolvido para que servisse de base a um processo orientado a objetos.

Ainda sobre o RUP, é importante enfatizar que seu objetivo primordial é servir de *framework* de processos orientados a objetos. Em outras palavras, o RUP apresenta as melhores técnicas de desenvolvimento orientado a objetos. Estas técnicas devem ser utilizadas de acordo com as características de cada equipe de desenvolvimento, levando-se em consideração questões cruciais, como o tempo disponível para desenvolvimento, assim como a quantidade de pessoas na equipe, e outras características que permearão a criação do software. Entende-se que cada projeto, dependendo de suas características intrínsecas, deve ser desenvolvido por meio de um processo customizado (adaptado ao desenvolvedor) e que levem em consideração essas características. O RUP é o *framework* (arcabouço) que permite aos engenheiros de software esse tipo de customização.

Os aspectos que distinguem o RUP são definidos em três conceitos-chave: o RUP é baseado em casos de uso, centrado na arquitetura, e iterativo e incremental. O RUP possui um ciclo de vida próprio, que é constituído pelas fases de Concepção, Elaboração, Construção e Transição. Durante este ciclo, ocorrem execuções repetidas dos fluxos de atividades do RUP. Estes fluxos são de dois tipos: Fluxos de Processo (Modelagem de Negócios, Requisitos, Análise e Projeto, Implementação, Testes e Implantação) e Fluxos de Suporte (Gerenciamento de Configuração e Mudanças, Gerenciamento de Projeto e Ambiente).

Esta seção se propõe a dar uma visão geral do RUP, proporcionando um entendimento de seus conceitos, características, organização, ciclo de vida e atividades relacionadas. Ele foi baseado no RUP 2000 [RUP, 2000].

3.8.1 Características Fundamentais do RUP

O RUP é Baseado em Casos de Uso

Um sistema de software é feito para servir seus usuários. Portanto, para construir um sistema adequadamente deve-se saber quem são seus usuários potenciais e o que eles querem e precisam. O termo *usuário* representa alguém ou alguma coisa (como um outro sistema) que interage com o sistema que está sendo desenvolvido.

Um caso de uso é um "pedaço" da funcionalidade do sistema, que interage com um usuário e produz algum resultado. Cada caso de uso captura determinados requisitos funcionais, e todos eles juntos resultam no modelo de casos de uso, o qual descreve a funcionalidade completa do sistema. Este modelo substitui a especificação funcional tradicional, cujo papel é responder à questão: "o que o sistema faz?" A estratégia de casos de uso pode ser caracterizada pela adição de três palavras no final dessa pergunta: "para cada usuário?" Estas palavras têm uma implicação muito importante. Forçam-nos a pensar em termos da importância dos usuários, não apenas em funções que poderiam ser interessantes no sistema.

Os casos de uso direcionam o processo de desenvolvimento, já que, baseados no modelo de casos de uso (que é o resultado da atividade de requisitos), os desenvolvedores criam uma série de modelos (ver Figura 3.5). Cada caso de uso é, então, especificado (Análise), efetivamente realizado (Projeto) e construído (Implementação). Além disso, os responsáveis pelos testes realizam seu trabalho com o propósito de garantir que os componentes do modelo de implementação cumpram corretamente os objetivos estabelecidos nos casos de uso. Desta forma, os casos de uso não apenas iniciam o processo de desenvolvimento, mas também o mantêm coeso.

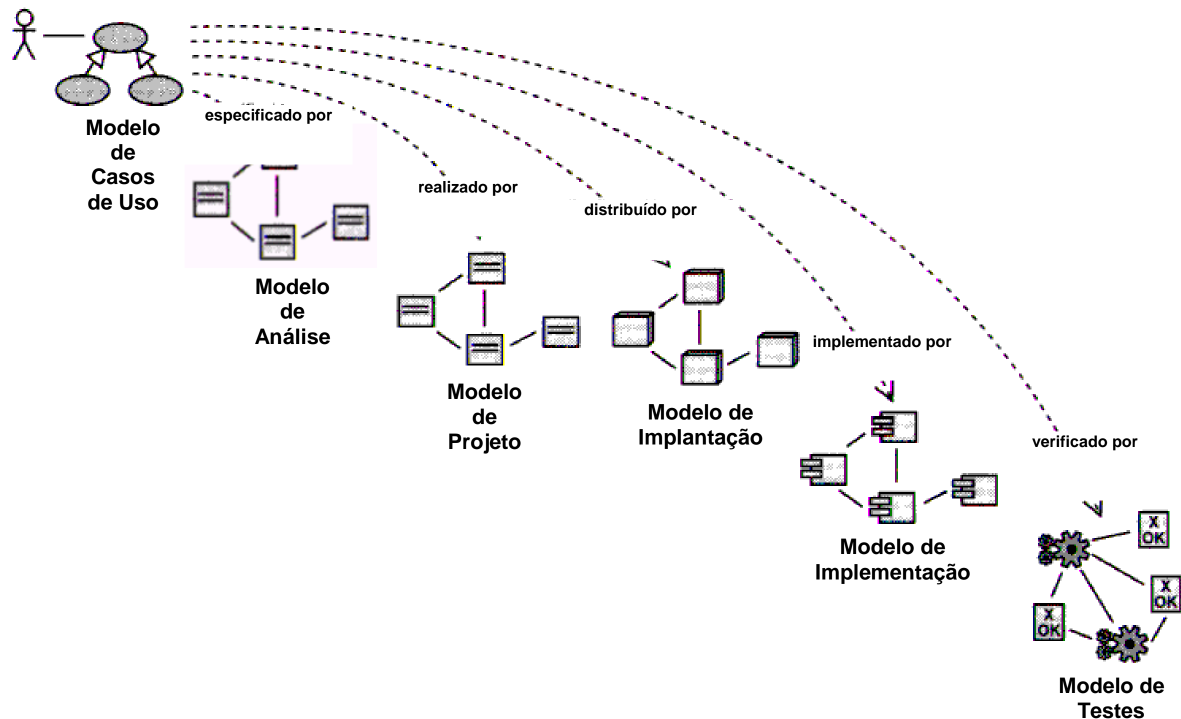

Figura 3.5 A influência do Modelo de Casos de Uso no processo de desenvolvimento [RUP, 2000].

Baseado em casos de uso significa que o processo de desenvolvimento executa uma sequência de tarefas derivadas dos casos de uso. Os Casos de Uso são especificados, projetados e servem de base para a definição dos casos de teste.

Os casos de uso ajudam na identificação das classes. Desenvolvedores podem, a partir da descrição de um caso de uso, "procurar" pelas classes necessárias para realizá-lo. Casos de uso também ajudam na definição das interfaces do sistema. Fica mais fácil projetar as interfaces se cada caso de uso é analisado separadamente.

Os casos de uso são pontos de partida para a elaboração de manual do usuário. Cada caso de uso descreve uma maneira de utilizar o sistema, por isso constitui um ponto de partida ideal para explicar como o usuário interage com o sistema.

Os casos de usos também ajudam os gerentes de projeto a planejar e definir as tarefas. Especificar um caso de uso, projetar um caso de uso, testar um caso de uso são exemplos de tarefas. Cada membro da equipe pode ficar responsável por determinadas tarefas: especificar cinco casos de uso, projetar três casos de uso, especificar casos de teste para dois casos de uso, etc. Definir tarefas a partir de casos de uso pode ajudar os gerentes até mesmo a estimar esforços e tempo para realizá-las. Há métodos para estimativas de projetos baseados em casos de uso [Pressman, 2006].

Embora seja verdade que os casos de uso dirigem o processo, eles não são selecionados isoladamente. São desenvolvidos juntamente com a arquitetura do sistema, ou seja, os casos de uso direcionam a arquitetura do sistema, que por sua vez influencia a seleção dos casos de uso. Portanto, ambos amadurecem no decorrer do ciclo de vida do sistema.

O RUP é Centrado na Arquitetura

O papel da arquitetura no software é de natureza similar ao papel da arquitetura na construção civil. As construções são observadas sob vários pontos de vista: estrutura, serviços, condução de calor, encanamento, eletricidade, etc. Da mesma forma, a arquitetura em um sistema de software é descrita como diferentes visões deste sistema.

O conceito de arquitetura de software incorpora os aspectos estáticos e dinâmicos mais importantes do sistema. A arquitetura é influenciada por muitos fatores, tais como a plataforma de software sobre a qual o sistema vai rodar (sistema operacional, sistema gerenciador de banco de dados, protocolos para comunicação em rede, etc.), blocos de construção reutilizáveis disponíveis (por exemplo, um *framework* para construção de interface gráfica com o usuário), possibilidade de extensibilidade e requisitos não funcionais (desempenho, confiabilidade, etc.). A arquitetura representa uma visão do projeto como um todo, na qual as características mais importantes são colocadas em destaque, deixando os detalhes de lado.

Como os casos de uso estão relacionados à arquitetura? Todo produto tem função e forma e nenhum desses elementos sozinho é suficiente. Essas duas forças devem ser balanceadas para se obter um bom produto. Neste caso, a função corresponde aos casos de uso e a forma corresponde à arquitetura. Por um lado, os casos de uso devem, quando construídos, encaixar-se na arquitetura. Por outro lado, a arquitetura deve fornecer espaço para a construção de todos os casos de uso necessários, agora e no futuro. Na realidade, ambos devem ser desenvolvidos em paralelo. Os casos de uso dirigem o desenvolvimento da arquitetura, e a arquitetura guia quais casos de uso podem ser realizados.

Os arquitetos definem o sistema em uma forma. Para encontrar essa forma, eles devem trabalhar a partir de uma compreensão geral das funções-chave do sistema, isto é, dos casos de uso chave. Estes devem ficar em torno de 5% a 10% de todos os casos de uso do sistema, mas são os mais significativos, aqueles que constituem o núcleo das funções do sistema.

Em termos simplificados, os arquitetos criam um esboço da arquitetura iniciando com a parte que não é específica dos casos de uso (por exemplo, a plataforma). Embora seja uma parte independente dos casos de uso, o arquiteto deve ter uma compreensão geral destes antes da criação do esboço. Depois o arquiteto trabalha com um subconjunto dos casos de uso identificados, aqueles que representam as funções chave do sistema em desenvolvimento. Cada caso de uso selecionado é especificado em detalhes e construído em termos de subsistemas, classes e componentes.

À medida que os casos de uso são especificados e atingem maturidade, mais detalhes da arquitetura são descobertos. Isto, por sua vez, leva ao surgimento de mais casos de uso. Este processo continua até que a arquitetura seja considerada estável.

A descrição da arquitetura é a extração de um conjunto de visões dos modelos do sistema, exceto o modelo de testes. Essa descrição deve ser mantida atualizada durante todo o ciclo de vida do sistema, refletindo as mudanças e adições que são relevantes para a arquitetura como, por exemplo, a adição de novas funcionalidades a subsistemas existentes.

O RUP é Iterativo e Incremental

O desenvolvimento de um produto comercial de software é uma grande tarefa que pode ser estendida por vários meses, possivelmente um ano ou até mais. Dessa forma, torna-se mais prático dividir o trabalho em pedaços menores ou miniprojetos. Cada miniprojeto é uma iteração que resulta em um incremento. Iterações são passos em um fluxo de trabalho e incrementos são crescimentos do produto. Ver a figura 3.6.

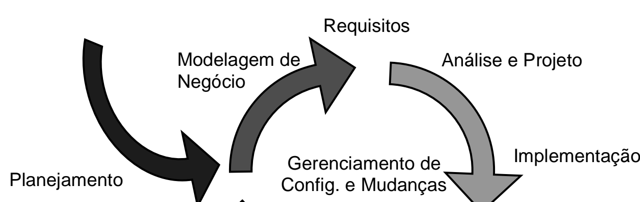

Figura 3.6 O desenvolvimento iterativo e incremental [RUP, 2000].

Os desenvolvedores selecionam o que deve ser feito em cada iteração baseados em dois fatores. Primeiro, a iteração deve trabalhar com um grupo de casos de uso que, juntos, estendam a usabilidade do produto em desenvolvimento. Segundo, a iteração deve tratar os riscos mais importantes.

Um incremento não é necessariamente a adição do código executável correspondente aos casos de uso que pertencem à iteração em andamento. Especialmente nas primeiras fases do ciclo de desenvolvimento, os desenvolvedores podem substituir um projeto superficial por um mais detalhado ou sofisticado. Em fases avançadas, os incrementos são tipicamente aditivos.

Em cada iteração, os desenvolvedores identificam e especificam os casos de uso relevantes, criam um projeto utilizando a arquitetura escolhida como guia, implementam o projeto em componentes e verificam se esses componentes satisfazem aos casos de uso. Se uma iteração atinge seus objetivos, e isso normalmente ocorre, o desenvolvimento prossegue com a próxima iteração. Caso contrário, os desenvolvedores devem rever suas decisões e tentar uma nova abordagem. Portanto, a cada iteração são desenvolvidas todas as atividades no ciclo de vida do sistema, gerando uma versão do produto. Veja a Figura 3.7.

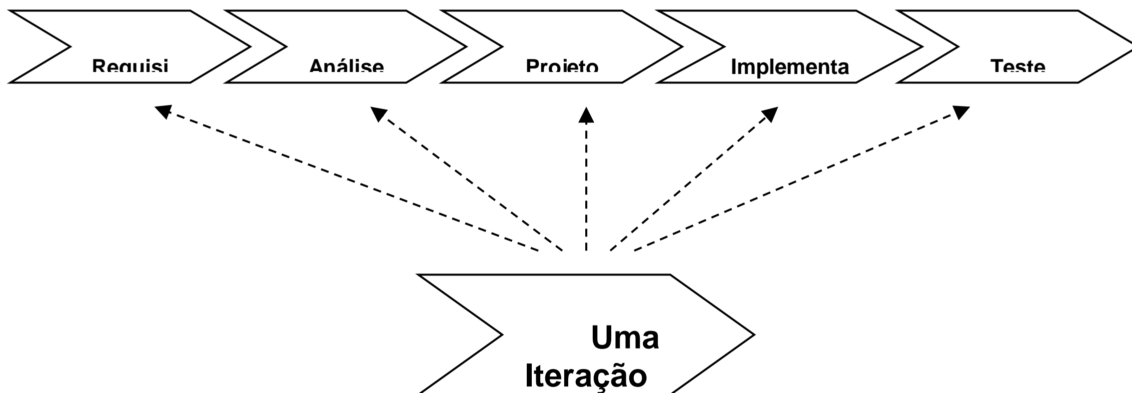

Figura 3.7 Todas as atividades do ciclo de vida são realizadas a cada iteração [RUP, 2000].

Há vários benefícios em se adotar um processo iterativo controlado, entre os quais se destacam:

- Redução dos riscos envolvendo custos a um único incremento (veja a Figura 3.8 que compara o desenvolvimento iterativo/incremental com o desenvolvimento tradicional). Se os desenvolvedores precisarem repetir a iteração, a organização perde somente o esforço mal direcionado de uma iteração, não o valor de um produto inteiro.

- Redução do risco de lançar o projeto no mercado fora da data planejada. Identificando os riscos numa fase inicial, o esforço despendido para gerenciá-los ocorre cedo, quando as pessoas estão sob menos pressão do que numa fase final de projeto.
- Aceleração do tempo de desenvolvimento do projeto como um todo, porque os desenvolvedores trabalham de maneira mais eficiente quando buscam resultados de escopo pequeno e claro.
- Reconhecimento de uma realidade frequentemente ignorada: as necessidades dos usuários e os requisitos correspondentes não podem ser totalmente definidos no início do processo. Eles são tipicamente refinados em sucessivas iterações. Este modelo de operação facilita a adaptação a mudanças de requisitos.

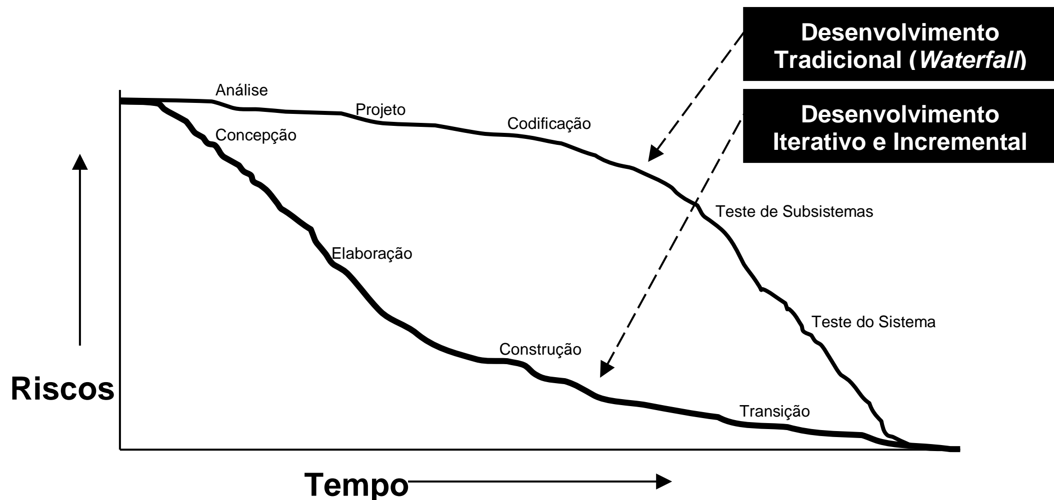

Figura 3.8 Relação Risco X Tempo nos modelos Iterativo e Tradicional (cascata) [RUP, 2000].

Os três conceitos apresentados (dirigido a casos de uso, centrado em arquitetura, iterativo e incremental) são igualmente importantes. A arquitetura provê a estrutura que conduz o trabalho ao longo das várias iterações, enquanto que os casos de uso conduzem o trabalho dentro de cada iteração. Remover um deles poderia reduzir drasticamente o valor do RUP.

O Apêndice B complementa a descrição abrangente sobre o Processo Unificado, apresentando detalhadamente cada fluxo proposto. Esta descrição é particularmente útil para a customização do processo para uma dada instalação, simplificando os passos para o desenvolvimento de software neste ambiente – dadas a natureza e a complexidade dos problemas que aí são tratados –, mas beneficiando-se da abrangência do Processo Unificado.

3.9 ABORDAGEM PRESCRITIVA X ABORDAGEM ÁGIL: QUAL ESCOLHER?

O gerente de desenvolvimento de sistemas deveria, no momento oportuno, com a participação da equipe encarregada do desenvolvimento de sistemas, fazer a escolha da abordagem mais apropriada para desenvolver dada aplicação.

Alguns fatores podem orientar esta escolha:

3.9.1 Escopo Abrangente da Aplicação

Se a aplicação é corporativa, pois abrange aos escalões operacionais, táticos e estratégicos da empresa, uma abordagem prescritiva seria mais indicada. Para as aplicações não corporativas, pode-se adotar uma abordagem ágil.

3.9.2 Número de Participantes da Equipe Desenvolvedora

Se o número de desenvolvedores (analistas, projetistas, programadores, testadores, usuários) é grande (acima de dez), recomenda-se a adoção de uma abordagem prescritiva, pela exigência de forte comunicação entre os membros da equipe; para equipes reduzidas, uma abordagem ágil pode ser adotada.

3.9.3 Volatilidade de Requisitos

Aplicações que apresentem alta volatilidade de requisitos, ou seja, os requisitos mudam com muita frequência, uma abordagem ágil pode ser adotada. Certas aplicações para Internet têm esta característica: alta volatilidade. Por exemplo, os portais corporativos.

3.10 AVALIAÇÃO DE PROCESSOS DE SOFTWARE

A percepção de que, quanto mais bem formalizado, tornado padrão e efetivamente utilizado pelas empresas for o processo de software, maior é a probabilidade de que o software desenvolvido atinja os requisitos de qualidade desejáveis. Em razão disto, modelos de avaliação da maturidade de processos foram propostos. O primeiro dos quais, o CMM – *Capability Maturity Model* – Modelo de Maturidade de Capacidade, proposto pelo Software Engineering Institute – SEI [SEI], pertencente à Universidade Carnegie-Mellon (www.sei.cmu.edu), em 1994. Em 2002, o SEI publicou o CMMI – *Capability Maturity Model Integration,* agora com seis níveis de capacitação:

Nível 0 – Incompleto
Nível 1 – Realizado
Nível 2 – Gerenciado
Nível 3 – Definido
Nível 4 – Quantitativamente Gerenciado
Nível 5 – Otimizado.

Outros modelos de avaliação propostos: SPICE (ISO/IEC 15504) e ISO 9001:2000 para Software.

Modelo Brasileiro de Avaliação de Processo de Software

Este modelo é chamado MPS.Br – Melhoria de Processo de Software Brasileiro e baseia-se no CMMI, nas normas ISO/IEC 12207 e ISO/IEC 15504. Leva em conta a realidade do mercado de software nacional [SOFTEX, 2008]. A vantagem de se ter um modelo brasileiro, que considera aspectos da realidade nacional, é a redução dos custos de certificação; caso se usasse um modelo estrangeiro, os custos de certificação seriam maiores.

O MPS.Br foi desenvolvido pelo Softex, com apoio de órgãos do governo (MCT, FINEP, BID) e por universidades.

O MPS.Br desdobra-se em três modelos: MR-MPS (modelo de referência), MA-MPS (método de avaliação) e MN-MPS (modelo de negócio).

O modelo de referência do MPS apresenta os seguintes níveis:

A – Em otimização
B – Gerenciado Quantitativamente
C – Definido
D – Largamente Definido
E – Parcialmente Definido
F – Gerenciado
G – Parcialmente Gerenciado.

Os esforços das empresas visam caminhar, nível a nível, desde o nível inicial (G) até atingir o nível A, ponto em que a empresa já apresenta um modelo de processo de software maduro e há compromisso de todos os agentes envolvidos (gerência de desenvolvimento, desenvolvedores, testadores, usuários, alta administração) com a melhoria contínua deste modelo.

O método de avaliação do MPS.Br estabelece como as avaliações são realizadas nas empresas que implementam o MR-MPS. Define composição da equipe, duração, validade e estruturação da avaliação.

O modelo de negócio do MPS.Br define o documento com as informações que a empresa interessada em implementar o MPS.Br fornece para credenciado.

Como se obtém a Melhoria de Processos?

Para conseguir melhoria do processo de software adotado numa empresa, o primeiro passo é definir um modelo padrão. Definido este padrão – para cuja formulação os engenheiros de software contribuiriam – deve-se procurar adotá-lo rigorosamente. Deve-se providenciar treinamento para todos os novos engenheiros de software que sejam admitidos. Estabelecer que, na conclusão de cada sistema, a equipe desenvolvedora prepare um relatório com apreciação de possíveis problemas encontrados. Qualquer ajuste que possa ser feito para corrigir problema identificado deve ser avaliado e, se for o caso, incorporado ao processo. Desta maneira, aproxima-se a área de desenvolvimento de software do objetivo desejável que é a busca da melhoria contínua.

Exercícios de Fixação

1) O Processo Unificado (RUP – *Rational Unified Process*) é um moderno processo de desenvolvimento de software constituído de quatro fases. Assinale a opção que apresenta as quatro fases do RUP, na ordem em que elas devem ser executadas.
A) concepção, elaboração, construção, teste.
B) elaboração, transição, concepção, construção.
C) concepção, elaboração, construção, transição.
D) concepção, elaboração, transição, construção.
E) elaboração, concepção, teste, transição.
(MEC/ENADE/2005)

2) Todo modelo de processo compreende várias fases; em cada fase, há tarefas específicas a desenvolver, artefatos devem ser elaborados.
O Processo Unificado compreende as seguintes fases: Concepção, Elaboração, Construção e Transição. Em qual (quais) destas fases é produzido o Modelo de Projeto Preliminar?
A) Concepção.
B) Concepção e Elaboração.
C) Elaboração.
D) Elaboração e Construção.
E) Construção

3) Quais são as quatro atividades do *framework* do modelo de processo Programação Extrema (XP)?
A) Planejamento, Análise, Projeto, Codificação.
B) Análise, Projeto, Codificação, Teste.
C) Comunicação, Planejamento, Modelagem, Codificação.
D) Planejamento, Análise, Codificação, Teste.
E) Planejamento, Projeto, Codificação, Teste.

4) No processo unificado, cinco *workflows* acompanham o conjunto das fases de desenvolvimento de software. Cada *workflow* é um conjunto de atividades executadas por vários membros do projeto. Considerando o desenvolvimento de um sistema integrado de gestão (ERP), o empacotamento em componentes de software dos elementos do modelo de projeto – tais como arquivo de código-fonte, biblioteca de ligação dinâmica e componentes executáveis – é descrito pelo *workflow* de
A) Teste; B) Análise; C) Projeto; D) Implementação; E) Requisito.

(MEC/ENADE/2005)

5) No processo de desenvolvimento de um sistema de controle de materiais (matérias-primas) para uma metalúrgica, a equipe de projeto, responsável pelo mapeamento dos requisitos, desenvolveu seus trabalhos seguindo os quatro subprocessos da engenharia de requisitos. Inicialmente, foram feitas a análise e a avaliação para se verificar se o sistema seria útil ao negócio. Em um segundo momento, os requisitos foram identificados e analisados e, logo em seguida, foram documentados. Finalmente, foi verificado se os requisitos identificados atendiam às demandas dos usuários. Tendo sido executado esse procedimento, uma empresa independente de auditoria, após análise, identificou dois problemas no processo: a documentação dos requisitos (formulários e padrões utilizados) estava inadequada e não possibilitava o entendimento correto dos requisitos; o processo de checagem entre as demandas dos usuários e as especificações relatadas não foi bem conduzido e seus resultados eram insatisfatórios.
Considerando o relatório da auditoria independente, quais foram as duas fases do processo de engenharia de requisitos que apresentaram problemas?
A) Entendimento do domínio e especificação; B) Elicitação e validação; C) Validação e entendimento do domínio; D) Especificação e validação; E) Validação e elicitação.
(MEC/ENADE/2005)

6) O modelo de processo prescritivo que se caracteriza por forte preocupação com análise de riscos é o modelo:
A) em cascata; B) incremental; C) de prototipagem; D) espiral; E) de desenvolvimento concorrente.

7) A vida de um sistema de *software* pode ser representada como uma série de ciclos. Cada ciclo termina com a liberação de uma versão do sistema para os clientes.
No Processo Unificado, cada ciclo contém quatro fases, a saber: Concepção, Elaboração, Construção e Transição. Cinco *workflows* atravessam o conjunto das quatro fases. Cada *workflow* é um conjunto de atividades que vários membros do projeto executam.
O *workflow* em que são desenvolvidos o modelo de projeto e o modelo de instalação do sistema é o *workflow* de:
A) Requisitos; B) Análise; C) Projeto; D) Implementação; E) Teste.

8) O *workflow* em que são construídos protótipos da interface com o usuário é o *workflow* de:
A) Requisitos; B) Análise; C) Projeto; D) Implementação; E) Teste.

9) Um sistema tem um ciclo de vida, que passa pela concepção, desenvolvimento e vida útil. Este ciclo inclui fases e subfases, algumas das quais listadas a seguir:
1) Codificação
2) Estudo de viabilidade
3) Manutenção
4) Teste de programa
5) Conversão do sistema.
6) Projeto
7) Operação do sistema
8) Teste do sistema
9) Análise.
A ordem cronológica correta das fases e subfases acima é:
A) 9-2-6-1-4-8-5-7-3.
B) 9-2-6-1-4-5-8-3-7.
C) 2-9-6-1-4-5-8-7-3.
D) 2-9-6-1-4-8-5-7-3.
E) 2-9-6-1-4-5-8-3-7.

10) O nível do CMMI (*Capability Maturity Model Integration*) cujo foco é o aperfeiçoamento contínuo do processo é o nível:
A) Realizado; B) Definido; C) Otimizado; D) Gerido; E) Quantitativamente gerido.

11) Um exemplo de modelo de processo ágil é:
A) O modelo RAD; B) A Programação Extrema; C) O modelo incremental; D) O Processo Unificado; E) O modelo espiral.

12) O princípio de Engenharia de Software que permite que problemas complexos sejam vistos e analisados em diferentes níveis de profundidade, com destaque de aspectos relevantes de um determinado fenômeno, ignorando-se os detalhes é a:

A) Modularidade; B) Generalidade; C) Formalidade; D) Abstração; E) Decomposição.

13) O *workflow* (fluxo de trabalho) de Requisitos do RUP (*Rational Unified Process*) é realizado em que fase(s):
A) Somente na fase de Concepção.
B) Somente na fase de Elaboração.
C) Nas fases de Concepção, Elaboração, Construção e Transição.
D) Somente nas fases de Concepção e Elaboração.
E) Somente nas fases de Concepção, Elaboração e Construção.

14) Formule um modelo geral de processo de software para sua empresa, que leve em conta a possibilidade de escolha entre abordagens diferentes (prescritiva ou ágil), considerando as características de uma dada aplicação a desenvolver.

15) Com relação à forma como o RUP trata a análise de requisitos, assinale a opção correta:
A) A análise de requisitos ocorre na fase de construção, quando são descritos todos os casos de uso, e em seguida modelados por meio de diagramas de casos de uso UML.
B) A análise de requisitos ocorre na fase de elaboração, em que são feitas entrevistas com usuários e definição do escopo do projeto.
C) A maior parte da análise de requisitos ocorre durante a fase de elaboração.
D) Por se tratar de um processo iterativo e evolutivo, a análise de requisitos ocorre na fase de construção juntamente com a programação, o que permite que os requisitos sejam revistos.
E) A análise de requisitos deve acontecer antes da programação e testes do sistema, não podendo sofrer alterações a partir do momento que estejam definidos.
(MEC/ENADE/2008)

16) Considere que você trabalhe em uma empresa de desenvolvimento de software e que a empresa tenha decidido desenvolver um novo editor de texto para colocar no mercado. Esse editor deve ser um software que forneça recursos adicionais de apoio à autoria, embasado no estilo de escrita do usuário, o que o torna um software de funcionalidade mais complexa. Considere que a empresa deseje disponibilizar o produto no mercado em versões que agreguem esse suporte de forma gradativa, fazendo análise de risco para avaliar a viabilidade de desenvolvimento de uma nova versão. Tendo de escolher um modelo de processo para desenvolver esse editor, e conhecendo as características dos modelos existentes, entre os modelos abaixo, qual é o modelo apropriado para esse caso?
A) cascata; B) espiral; C) RAD (*rapid application development*); D) prototipação; E) *cleanroom*.
(MEC/ENADE/2008)

17) Após realizar uma análise de Mercado em busca de soluções para aprimorar o seu negócio, uma empresa adquiriu um sistema de ERP (*Enterprise Resource Planning*) contendo um conjunto de módulos que integra todos os departamentos existentes. Após um ano de utilização, houve uma mudança na legislação e, para atender as novas exigências, foi necessária uma manutenção no sistema ERP.
Considerando essa situação hipotética, é correto afirmar que a empresa irá realizar uma manutenção:
A) corretiva; B) adaptativa; C) aperfeiçoadora; D) preventiva; E) perfectiva.
(MEC/ENADE/2008)

18) Considere as seguintes afirmativas sobre os modelos prescritivos de processos de desenvolvimento de software:
I. Uma das vantagens do modelo de prototipação é servir como base para entendimento dos requisitos do sistema.
II. Um dos problemas do modelo RAD (*Rapid Application Development*) é a necessidade de conseguir recursos suficientes para a montagem de vários grupos operando em paralelo.
III. O caso negócio (*Business Case*) é um dos produtos da fase de Concepção do Processo Unificado (*Unified Process*).
Assinale a alternativa ***CORRETA:***
A) Apenas a afirmativa I é verdadeira.
B) Apenas a afirmativa II é verdadeira.
C) Apenas a afirmativa III é verdadeira.
D) Apenas as afirmativas I e II são verdadeiras.
E) Todas as afirmativas são verdadeiras.
(SBC/POSCOMP/2009)

19) Formule uma organização da área de desenvolvimento de sistemas (com o estabelecimento dos padrões de documentação necessários) para tratar adequadamente o problema de mudança de requisitos em aplicações em

desenvolvimento ou implantadas (gerência de configuração). Não deixe de considerar a possibilidade de utilização de abordagens ágeis para o desenvolvimento de sistemas, quando for aplicável. **Sugestão**: Siga o Apêndice B (Processo Unificado) para conceber seu arcabouço de desenvolvimento de sistemas, com as simplificações necessárias. O que se pede aqui é a elaboração de uma cartilha para nortear o trabalho de analistas, projetistas e programadores, e assim alcançar o objetivo de padronizar o processo de desenvolvimento. Não esquecer de identificar as revisões técnicas formais e as revisões gerenciais necessárias.

20) É possível a adoção de abordagens prescritivas e abordagens ágeis na mesma instalação de desenvolvimento de software? É possível conciliar seus usos? Se a resposta é sim, como se pode fazer para adotá-las? Se a resposta é não, que razão(ões) justifica(m) a negativa?

21) Dada a engenharia de fabricação como um exemplo da engenharia convencional, a abordagem comum nestas áreas de engenharia vai do abstrato para o concreto, e o produto final é a realização física de um projeto abstrato. Contudo, na engenharia de software, a abordagem é inversa. Ela vai do concreto para o abstrato. O produto de software final é a virtualização (codificação) e representação invisível de um projeto original que expressa um problema do mundo real. A única parte tangível de um produto de software é o seu meio de armazenamento ou seus comportamentos de "*run-time*".
Como ilustrado na figura 3.10, isto é provavelmente a característica mais singular e interessante da Engenharia de Software.

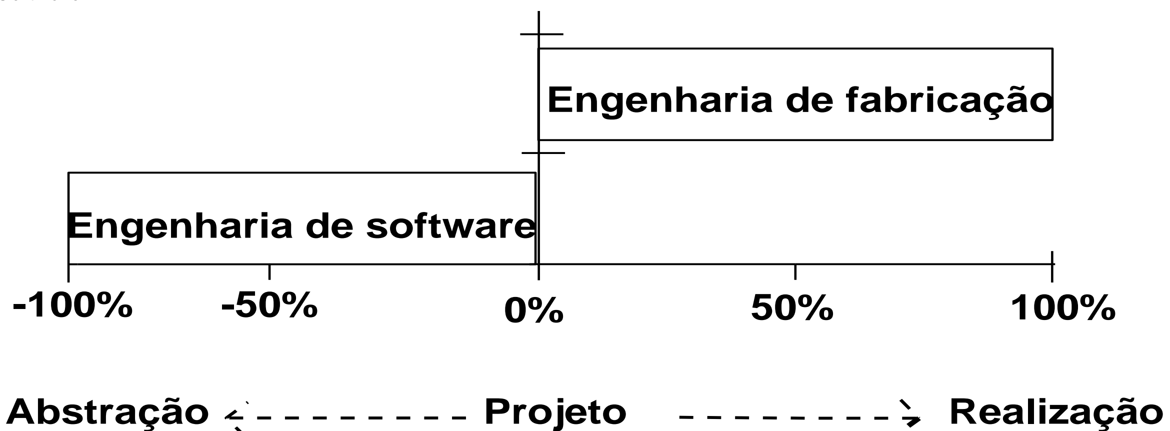

Figura 3.10 Engenharia de Software X Engenharia de Fabricação.

Cite e comente duas outras características que distinguem a Engenharia de Software da Engenharia Convencional.

22) Faça uma definição abrangente da Engenharia de Software.

---

23) Os requisitos de software mudam. Isto é um fato. O impacto da mudança varia com o momento em que é introduzida. Pesquisas atestam que se a mudança é feita ainda na fase de definição, seu custo é de 1x; se ocorre na fase de desenvolvimento é de 1,5x a 6x; se após a entrega do software, é de 60 a 100x [Pressman, 2006]. Na sua avaliação, por que isto ocorre?

24) A Engenharia de Software é uma disciplina para o desenvolvimento de software de alta qualidade para sistemas baseados em computador. Várias representações de software são derivadas à medida que se evolui dos passos conceituais para a concretização. Tipicamente, três grandes fases podem ser identificadas para compor todo o processo de desenvolvimento de software: 1) Fase de definição, 2) Fase de desenvolvimento e 3) Fase de verificação, liberação e manutenção.
Para cada uma destas fases, identifique subprodutos típicos e pontos de revisão para garantia de qualidade.

25) Qual é a opção listada abaixo que contém características utilizadas na fase de Projeto da Programação Extrema ("*Extreme Programming*" – XP)?
A) Criação de narrativas do usuário ("*stories*"); utilização de cartões CRC.
B) Programação em dupla; teste de unidade.
C) Teste de unidade; teste de aceitação.
D) Aplicação do Princípio KIS ("*Keep It Simple*"); utilização de cartões CRC.
E) Definição de critérios de aceitação; elaboração do plano de iteração.

26) Qual é o nível do CMMI ("*Capability Maturity Model Integration*") em que o foco é a padronização do processo?
A) Nível Realizado; B) Nível Gerido; C) Nível Definido; D) Nível Quantitativamente Gerido; E) Nível Otimizado.

Exercícios Propostos

1) O nível do modelo CMM (*Capability Maturity Model*) em que o objetivo principal é a padronização do processo é o:
A) definido; B) repetível; C) inicial; D) gerenciado.

2) O modelo de ciclo de vida, no qual os principais processos são executados em estrita sequência, o que permite demarcá-los com pontos de controle bem-definidos, é o:
A) Cascata; B) Dirigido por ferramenta; C) Codifica-remenda; D) Prototipagem evolutiva.
(Concurso Público – FADESP/2009)

3) No modelo CMM, o nível em que a organização estabelece metas quantitativas para os seus produtos e processos é o:
A) repetível; B) definido; C) otimizado; D) gerenciado. (Concurso Público – FADESP/2008)

4) A fase do Processo Unificado em que o Modelo de Análise é elaborado é a fase de:
A) construção; B) concepção; C) transição; D) elaboração. (Concurso Público – FADESP/2008)

5) A abordagem proposta por Barry Boehm com foco nos objetivos do projeto, seus marcos e cronogramas, responsabilidades, abordagens gerenciais e técnicas, e recursos necessários é o:
A) Modelo COCOMO; B) Princípio W^5HH; C) Princípio de Pareto; D) Modelo espiral.
(Concurso Público – FADESP/2008)

6) O fluxo de processo de Implementação do RUP – *Rational Unified Process* abrange a(s) fase(s) de:
A) Construção, somente; B) Elaboração e Construção, somente; C) Elaboração, Construção e Transição, somente; D) Concepção, Elaboração, Construção e Transição; E) Concepção, Elaboração e Construção, somente.
(Concurso Público – UFRA/2008)

7) Inovação e Implantação Organizacional, Análise e Resolução Causal são áreas de processo necessárias para alcançar qual nível de maturidade do CMMI – *Capability Maturity Model Integration*?
A) Realizado; B) Gerido; C) Definido; D) Quantitativamente Gerido; E) Otimizado.
(Concurso Público – UFRA/2008)

8) As seguintes afirmações dizem respeito ao modelo de desenvolvimento em espiral – proposto por Barry Boehm na década de 1970:
I – suas atividades de desenvolvimento são conduzidas por riscos;
II – cada ciclo da espiral inclui 4 passos: passo 1 – identificação dos objetivos; passo 2 – avaliação das alternativas tendo em vista os objetivos e os riscos (incertezas, restrições) do desenvolvimento; passo 3 – desenvolvimento de estratégias (simulação, prototipagem) para resolver riscos; e passo 4 – planejamento do próximo passo e continuidade do processo determinada pelos riscos restantes;
III – é um modelo evolutivo em que cada passo pode ser representado por um quadrante num diagrama cartesiano: assim, na dimensão radical da espiral, tem-se o custo acumulado dos vários passos do desenvolvimento enquanto na dimensão angular tem-se o progresso do projeto.
Levando-se em conta as três afirmações I, II e III acima, identifique a única alternativa válida:
A) Apenas a I e a II estão corretas; B) Apenas a II e a III estão corretas; C) Apenas a I e a III estão corretas; D) As afirmações I, II e III estão corretas; E) Apenas a III está correta. (SBC/POSCOMP/2003)

9) O conjunto básico de atividades e a ordem em que são realizadas no processo de construção de um software definem o que é habitualmente denominado de ciclo de vida do software. O ciclo de vida tradicional (também denominado *waterfall*) ainda é hoje em dia um dos mais difundidos e tem por característica principal:
A) O uso de formalização rigorosa em todas as etapas de desenvolvimento; B) A abordagem sistemática para realização das atividades do desenvolvimento de software de modo que elas seguem um fluxo sequencial; C) A codificação de uma versão executável do sistema desde as fases iniciais do desenvolvimento, de modo que o sistema final é incrementalmente construído, daí a alusão à ideia de "cascata" (*waterfall*); D) A priorização da análise dos riscos do desenvolvimento; E) A avaliação constante dos resultados intermediários feita pelo cliente.
(SBC/POSCOMP/2003)

10) Qualidade é uma das premissas básicas para se desenvolver software hoje em dia. Contudo, gerenciar a qualidade dentro do processo de software não é uma etapa trivial. Requer preparação, conhecimento técnico

adequado e, sobretudo, comprometimento de todos os *stakeholders* envolvidos. A esse respeito, considere as seguintes afirmativas:
I. O MPS.br é uma iniciativa para Melhoria de Processo do Software Brasileiro. O MPS.br adequa-se à realidade das empresas brasileiras e está em conformidade com as normas ISO/IEC 12207. No entanto, não apresenta uma estratégia de compatibilidade com o CMMI – *Capability Maturity Model Integration*.
II. A rastreabilidade de requisitos de software proporciona uma melhor visibilidade para a gerência de qualidade do projeto.
III. Uma empresa de tecnologia certificada por meio de modelos como CMMI ou MPS.br oferece produtos de software também certificados.
IV. A padronização é um dos fundamentos básicos da gerência da qualidade. A padronização pode acontecer em diversos níveis: na documentação, no código e, principalmente, no processo.
Considerando a gerência da qualidade, assinale a alternativa CORRETA:
A) Todas as afirmativas são verdadeiras; B) Nenhuma das afirmativas é verdadeira; C) Somente as afirmativas II e III são verdadeiras; D) Somente as afirmativas II e IV são verdadeiras; E) Somente as afirmativas I, II e III são verdadeiras.

(SBC/POSCOMP/2007)

11) Qual é a ênfase da fase de Concepção do *Rational Unified Process*?

12) Qual é a ênfase da fase de Elaboração do *Rational Unified Process*?

13) Qual é a ênfase da fase de Construção do *Rational Unified Process*?

14) Qual é a ênfase da fase de Transição do *Rational Unified Process*?

Resumo do Capítulo 3: Processo de Desenvolvimento de Software

Este capítulo trata do processo de desenvolvimento de software. Processo é definido como um conjunto ordenado de passos que, quando executado, leva ao desenvolvimento de software. Uma definição de Engenharia de Software foi apresentada, como também os princípios que norteiam esta área. O domínio de qualquer tarefa complexa se dá pela identificação das fases que a compõem, pelo particionamento das etapas de cada fase em subtarefas e atividades. Isto vale para o desenvolvimento de software, que exige a execução ordenada de várias fases; estas fases, por sua vez, desdobram-se em subtarefas e atividades. A este conjunto chama-se processo de desenvolvimento de software. Quanto mais detalhado, mais conhecido, mais aplicado for um modelo de processo adotado por uma organização, provavelmente de melhor qualidade será o software desenvolvido com este processo. As duas grandes famílias de processos de software foram apresentadas: modelos prescritivos (com destaque para os modelos Cascata, Incremental, Prototipação, Espiral e Processo Unificado) e modelos ágeis (com destaque para a Programação Extrema, Modelagem Ágil e Processo Unificado Ágil). O capítulo finaliza com os modelos de avaliação de processo de software (CMMI e MPS.br) e com exercícios de concursos diversos.