Universidade Federal do Piauí
Centro de Educação Aberta e a Distância

INTRODUÇÃO À ENGENHARIA DE SOFTWARE

Pedro de Alcântara dos Santos Neto

Ministério da Educação - MEC
Universidade Aberta do Brasil - UAB
Universidade Federal do Piauí - UFPI
Universidade Aberta do Piauí - UAPI
Centro de Educação Aberta e a Distância - CEAD

Introdução à Engenharia de Software

Pedro de Alcântara dos Santos Neto

UFPI
2013

S237i Santos Neto, Pedro de Alcântara dos.
Introdução à engenharia de software / Pedro de Alcântara dos Santos Neto. - Teresina: EDUFPI/CEAD, 2013.
92 p.

ISBN

1. Engenharia de Sftware. 2. Software - Desenvolvimento. 3. Software. 4. Educação a Distância. I. Título.

CDD 004.21

Apresentação

Os Sistemas de Informação (SI) são produtos de softwares cada vez mais importantes no dia a dia das pessoas. Tais sistemas são usualmente compostos por muitos elementos que operam juntos para recuperar, processar, armazenar e distribuir informações, que normalmente são utilizadas para auxiliar a análise e tomada de decisão em uma organização. Por conta disso, é necessário o uso de métodos, técnicas e ferramentas para sua construção, para que se possa obter altos níveis de qualidade a baixos custos.

Neste livro apresentamos uma introdução à Engenharia de Software, que é a área da computação que aborda a sua construção, incluindo os Sistemas de Informação, como um produto de engenharia.

Na Unidade I apresentamos as origens da Engenharia de Software, bem como os conceitos básicos existentes, explorando os conceitos de processo e dos modelos de ciclo de vida, os quais são elementos básicos para entender a área.

Na Unidade II faremos uma breve apresentação dos principais fluxos da Engenharia de Software que são chaves para o desenvolvimento de um projeto e serão abordados com mais profundidade em outras disciplinas do curso.

Por fim, na Unidade III apresentamos um Processo de Software baseado nos princípios ágeis, exemplificando seu uso para controle de um projeto envolvendo tarefas simples.

Sumário

67

UNIDADE 3

EXEMPLO DE UM PROCESSO DE *SOFTWARE*

UNIDADE 01

Conceitos Básicos

Resumindo

Nesta unidade apresentamos os conceitos básicos da Engenharia de *Software,* definindo os pontos básicos para o seu entendimento. Apresentamos o evento que deu origem à área, conhecido como a "Crise do *software*", discutindo se ela realmente foi superada. Fazemos ainda uma definição precisa sobre o conceito de processo, que é fundamental para a Engenharia de *software.*

Por fim, apresentamos os mitos do *software* que, diferentemente dos mitos tradicionais, trazem desinformação e geram problemas, em vez de histórias interessantes e cativantes.

1 CONCEITOS BÁSICOS

INTRODUÇÃO

A partir de 1961, o mundo presenciou o surgimento de novos computadores mais modernos e com mais poder computacional. A partir dessa data o *software* ganhou notoriedade e, por conta disso, uma série de problemas relacionados ao "amadorismo" utilizado na sua construção ficou evidente. Esses fatores originaram a "crise do *software*", em meados de 1968. O termo expressava as dificuldades do desenvolvimento de *software* frente ao rápido crescimento da demanda existente, da complexidade dos problemas a serem resolvidos e da inexistência de técnicas estabelecidas para o desenvolvimento de sistemas que funcionassem adequadamente ou pudessem ser validados.

A crise do software surgiu devido ao amadorismo existente na condução do processo de desenvolver software. Em 1968, surgiu a Engenharia de Software.

Figura 1: Fotos da NATO Software Engineering Conference.

A NATO *Software* Engineiring Conference marcou o início de uma nova área na computação. A Figura 1 mostra registros dessa conferência, que teve como um dos participantes o ilustre professor de Matemática, da Universidade Eindhoven de Tecnologia, Edsger Dijkstra.

Em 1972, Dijkstra recebeu o Prêmio Turing (Turing Award), que é

dado anualmente pela Association for Computing Machinery (ACM), para uma pessoa selecionada pelas contribuições de natureza técnica feitas para a comunidade da computação. Seu discurso no recebimento do prêmio, intitulado “O Pobre Programador” (The Humble Programmer), tornou-se um clássico da Engenharia de *Software*. Um trecho desse discurso, traduzido para o português e que resume a crise, é exibido a seguir:

“A maior causa da crise do *software* é que as máquinas tornaram-se várias ordens de magnitude mais poderosas! Para esclarecer melhor: quando não havia máquinas, a programação não era um problema; quando tínhamos poucos computadores, a programação tornou-se um problema razoável, e agora que nós temos computadores gigantes, a programação tornou-se um problema igualmente grande”.

Dijkstra é um personagem ilustre da computação, com diversas contribuições em diversas áreas.

Mas, o que realmente seria a crise do *software*? Podemos resumir a crise à imaturidade no desenvolvimento de *software*, causando diversos problemas, como por exemplo:

1. Projetos estourando o orçamento;
2. Projetos estourando o prazo;
3. *Software* de baixa qualidade;
4. *Software* muitas vezes não atendendo aos requisitos;
5. Projetos não gerenciáveis e código de difícil manutenção.

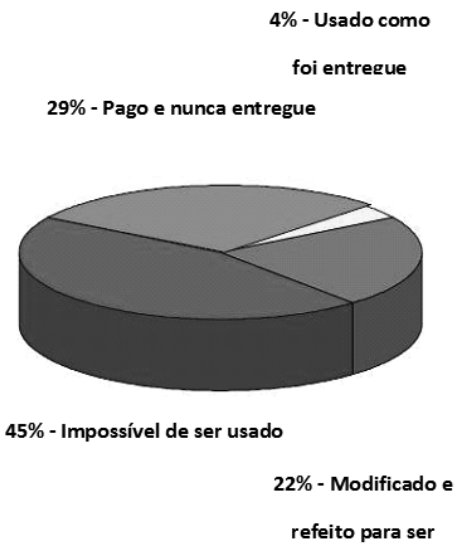

Figura 2: Exemplo de um quadro da situação na época da “crise do software.”

A Figura 2 exibe um quadro da situação relacionada ao desenvolvimento de *software* na época da crise. Nesse período, apenas 4% dos projetos eram finalizados e o produto gerado era utilizado tal qual foi entregue; 29% dos produtos não eram entregues, embora tenham sido pagos; 45% eram entregues, porém, o produto resultante era impossível de ser utilizado, normalmente, por conter uma quantidade de incorreções que impediam sua utilização e 22% eram modificados e em muitos casos extremamente modificados para que pudessem ser utilizados.

Mesmo com a evolução dos computadores, das técnicas e ferramentas nos últimos anos, a produção de *software* confiável, correto e entregue dentro dos prazos e custos estipulados, ainda é muito difícil. Dados do STANDISH GROUP, em 2004, usando como base mais de 8000 projetos mostrou que apenas 29% dos projetos foram entregues respeitando os prazos e os custos e com todas as funcionalidades especificadas. Aproximadamente 18% dos projetos foram cancelados antes de estarem completos e 53% foram entregues, porém com prazos maiores, custos maiores ou com menos funcionalidades do que especificado no início do projeto. Dentre os projetos que não foram finalizados de acordo com os prazos e custos especificados, a média de atrasos foi de 222% e a média de custo foi de 189% a mais que o previsto.

Ainda hoje diversos projetos de desenvolvimento de software fracassam.

Considerando todos os projetos que foram entregues, além do prazo e com custo maior, na média, apenas 61% das funcionalidades originais foram incluídas. Mesmo os projetos cuja entrega é feita respeitando os limites de prazos e custos possuem qualidade suspeita, uma vez que, provavelmente, foram feitos com muita pressão sobre os desenvolvedores, o que pode quadruplicar o número de erros de *software*, segundo a mesma pesquisa.

Será que a crise acabou?

Uma pergunta que devemos nos fazer atualmente é: será que a crise acabou? Para respondê-la é necessário fazer uma série de análises.

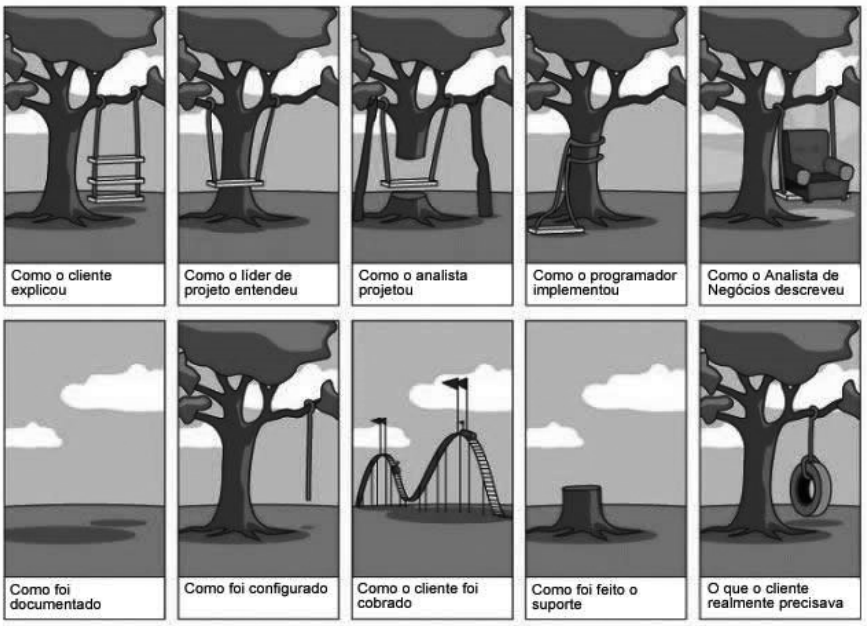

Figura 3: Dificuldades durante o desenvolvimento de software.

Ainda hoje diversos projetos de desenvolvimento de software fracassam.

A Figura 3 apresenta uma ilustração muito conhecida na área de Engenharia de *Software*. Ela apresenta um grande problema relacionado à falta de entendimento entre os grupos que fazem parte de um projeto. O quadro inicial da figura apresenta uma ilustração tentando expressar como o cliente explicou o que necessitava para a equipe de desenvolvimento. O quadro seguinte exibe como o líder do projeto entendeu. Podemos notar que existe uma diferença significativa entre as figuras. Isso não deveria acontecer ou deveria haver o mínimo possível de diferenças.

Desse ponto adiante existe uma série de entendimentos diferentes, algumas vezes causados por falhas na comunicação, outros por comodidade, uma vez que certos entendimentos favorecem alguns grupos. O resultado de tudo isso é que o produto resultante tende a ser algo totalmente discrepante do que foi solicitado e ainda mais diferente do que realmente era necessitado. Observe que o último quadro na figura apresenta o que o cliente realmente desejava e que é bastante diferente do que ele relatou que necessitava. Mais adiante, quando falarmos de requisitos, e na próxima disciplina relacionada à Engenharia de *Software*, que trata exclusivamente de requisitos, vamos entender isso com muito mais clareza.

Embora problemas durante o desenvolvimento de *software* aconteçam e com certa frequência, os processos, métodos e ferramentas existentes

auxiliam muito o desenvolvimento. Uma vez aplicados por pessoas com os conhecimentos adequados, podemos ter certeza do sucesso em um projeto. Por conta disso, existem diversos projetos grandes com sucesso absoluto. Para isso, é necessário aplicar corretamente a Engenharia de *Software*! Ou seja, respondendo à questão inicialmente formulada nesta seção: a crise acabou, mas apenas para aqueles que utilizam a Engenharia de *Software* como base para a construção de produtos de *software*.

Um grande problema a ser superado é a falha na comunicação entre as equipes.

Assim como na construção civil, muitas empresas estouram custos e prazos, mesmo tendo métodos e técnicas bem estabelecidos. No desenvolvimento de software a situação é a mesma: aqueles que utilizam os princípios corretamente têm grandes chances de sucesso.

O que é Engenharia de *Software*?

Já comentamos sobre as origens da Engenharia de *Software*, mas até agora não fizemos uma definição precisa do que seja exatamente isso. Nesta seção faremos isso. Mas antes da definição, vamos fazer uma analogia muito interessante: você sabe como funciona o processo de fabricação de um carro?

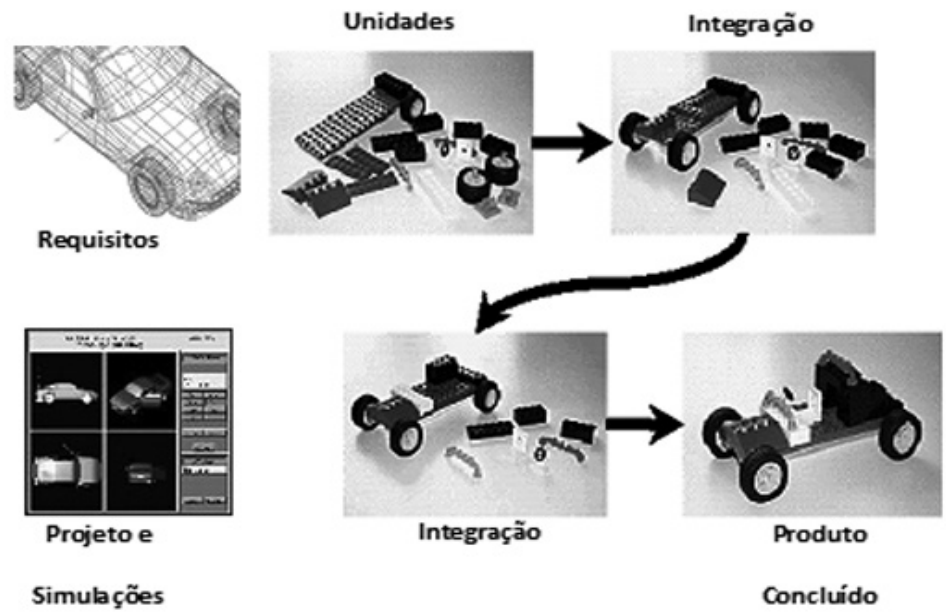

Figura 4: Passos para desenvolvimento de um carro

A Figura 4 apresenta os passos para desenvolvimento de um carro. Sua construção inicia a partir do momento que se tem a ideia de iniciar essa tarefa. A partir disso, é iniciado o levantamento das características, necessidades e desejos relacionados ao produto. Tudo isso irá compor a especificação de requisitos do carro. São exemplos desses requisitos como porta-malas com capacidade de 500l, motor 1.6, câmbio automático, etc.

Com a identificação dos requisitos é possível iniciar a montagem de um protótipo, avaliando diversos pontos que necessitam de esclarecimento antes da construção efetiva do carro. Essa prototipação poder ser feita via ferramentas de simulação, como as famosas ferramentas CAD existentes, ou a partir da criação de versões em miniatura do produto (Figura 4, Projeto e Simulações).

O modelo criado, seja qual for, normalmente é utilizado para responder diversas questões, para em seguida iniciar o desenvolvimento das partes que irão compor o automóvel. Essas partes são chamadas de unidades, como por exemplo, a barra de direção, a porta, o motor, as rodas, etc (Figura 4, Unidades). Cada componente do carro possui especificações e precisa ser verificado com relação a isso. Por exemplo, a barra de direção deve ser retilínea, com 1,80m de extensão, e deve suportar até 500 kg em seu ponto central. A verificação consiste em analisar se isso realmente foi atendido pelo processo de fabricação do componente. Se o item não passar nessa verificação, não podemos continuar com a construção do produto, pois certamente haverá mais problemas quando utilizarmos o componente dessa forma.

Caso os componentes sejam aprovados na verificação, podemos iniciar a combinação desses componentes, integrando-os, para em seguida verificar se eles continuam atendendo às especificações (Figura 4, Integração). Como exemplo, imagine que a barra de direção do carro foi integrada às rodas. Ela deveria sustentar os 500 kg em seu ponto central. Embora isso já tenha sido verificado após a fabricação do item, uma nova verificação é necessária após sua integração com os demais elementos, uma vez que agora o ponto de falha pode não ser a barra de direção em si, mas os pontos de conexão entre a barra e as rodas.

Caso os componentes agrupados continuem funcionando conforme o esperado, poderemos continuar o agrupamento até que tenhamos o produto completo. Uma vez tendo terminado o produto, teremos que submetê-lo a uma série de avaliações para verificar o seu nível de qualidade. No caso de um carro, são necessários testes de desempenho, consumo, segurança, etc. Após essa verificação poderemos afirmar que o produto foi concluído com sucesso.

Mas por que conversar sobre carros se o objetivo desta seção é apresentar o conceito de Engenharia de *Software*? Justamente porque o

processo de fabricação de um carro tem tudo a ver com a Engenharia de *Software*. De modo geral, podemos dizer que Engenharia de *Software* poderia ser resumida à utilização de princípios de engenharia para o desenvolvimento de *software*, ou seja, levantar os requisitos associados, construir modelos para representar a solução a ser desenvolvida, implementar as diversas unidades que irão compor o produto, verificando se tais unidades atendem aos requisitos identificados, realizar a integração entre as unidades, também verificando seu funcionamento, até que tenhamos o produto por completo, que deve passar por uma série de verificações (testes funcionais, desempenho e estresse, usabilidade, etc.) para que possamos concluir o desenvolvimento.

Em resumo e utilizando termos mais formais, podemos dizer que a Engenharia de *Software* é a aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento de *software*.

A construção de um carro e a construção de um software são projetos similares, mas que exigem diferentes habilidades na sua condução e execução.

Na literatura, podem-se encontrar mais definições para Engenharia de *Software*, dentre elas destacamos:

"O estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um *software* que seja confiável e que funcione eficientemente em máquinas reais". (PETER, 1969)

"A aplicação prática do conhecimento científico para o projeto e a construção de programas computacionais e a documentação necessária à sua operação e manutenção". (BOEHM, 1976)

"Conjunto de métodos, técnicas e ferramentas necessárias à produção de *software* de qualidade para todas as etapas do ciclo de vida do produto". (KRAKOWIAK, 1985)

Num ponto de vista mais formal, a Engenharia de *Software* pode ser definida como sendo a aplicação da ciência e da matemática através das quais os equipamentos computacionais são colocados à disposição do homem por meio de programas, procedimentos e documentação associada. De modo mais objetivo, pode-se dizer que a Engenharia de *Software* busca prover a tecnologia necessária para produzir *software* de alta qualidade a um baixo custo.

Os dois fatores motivadores são, essencialmente, a qualidade e o custo. A qualidade de um produto de *software* é um parâmetro cuja quantificação não é trivial, apesar dos esforços desenvolvidos nesta direção. Por outro lado, o fator custo pode ser facilmente quantificado desde que os

procedimentos de contabilidade tenham sido corretamente efetuados.

Um grande problema facilmente percebido hoje é justamente o fato de boa parte das organizações não encararem o desenvolvimento de *software* como um projeto de verdade, aplicando as técnicas, métodos e ferramentas necessários. Por conta disso, boa parte desses projetos fracassa. Na próxima seção vamos discutir alguns dos aspectos que fazem isso acontecer.

Problemas no desenvolvimento de *software*

Embora a Engenharia de *Software* seja uma área consolidada e existam várias empresas que a utilizam com muito sucesso em projetos de desenvolvimento de *software*, isso não é verdade na maioria dos casos. A crise continua em muitos locais, não mais por ausência de métodos, técnicas e ferramentas, mas pela falta do seu uso!

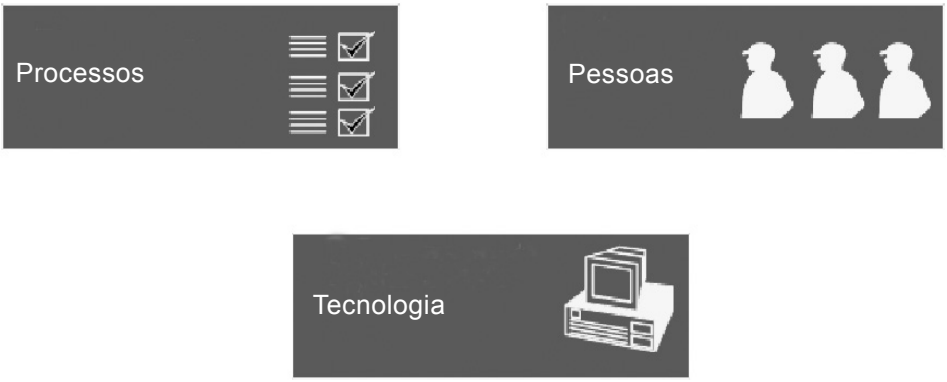

Figura 5: Tripé da Engenharia de Software.

A Figura 5 apresenta o tripé no qual a Engenharia de *Software* é baseada em processos, pessoas e tecnologia. Não adianta termos os melhores profissionais do mundo se não possuímos boas tecnologias para uso ou se não possuímos um processo que guie o desenvolvimento de *software*.

Da mesma forma, não adianta possuir as tecnologias mais avançadas se as pessoas não conseguem utilizá-las. Além disso, mesmo que pareça inconcebível para alguns, de nada adianta termos a melhor tecnologia e as melhores pessoas se não existe um processo que guie as atividades dessas pessoas utilizando tais tecnologias. Existem grandes chances de termos problemas relacionados à falta de controle ou desorganização, caso não tenhamos um processo que discipline as tarefas das pessoas.

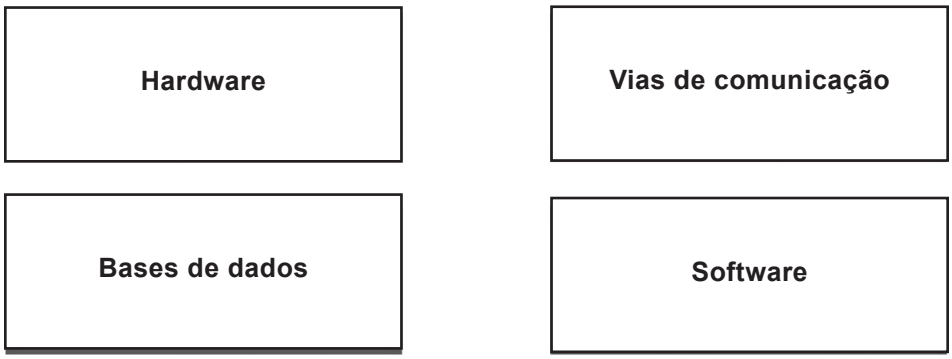

Figura 6: Componentes de um sistema.

Um conceito que muitas vezes causa confusão é o de sistema, que muito se confunde com o conceito de *software*. Pode-se definir o *software*, numa forma clássica, como sendo: "um conjunto de instruções que, quando executadas, produzem a função e o desempenho desejados, estruturas de dados que permitam que as informações relativas ao problema a resolver sejam manipuladas adequadamente e a documentação necessária para um melhor entendimento da sua operação e uso".

É preciso um equilíbrio entre pessoas, processos e tecnologias para o sucesso em um projeto de desenvolvimento de software.

Entretanto, no contexto da Engenharia de *Software*, o *software* deve ser visto como um produto a ser "vendido". É importante dar esta ênfase, diferenciando os "programas" que são concebidos num contexto mais restrito, onde o usuário ou "cliente" é o próprio autor. No caso destes programas, a documentação associada é pequena ou (na maior parte das vezes) inexistente e a preocupação com a existência de erros de execução não é um fator maior, considerando que o principal usuário é o próprio autor do programa, este não terá dificuldades, em princípio, na detecção e correção de um eventual "bug". Além do aspecto da correção, outras boas características não são também objeto de preocupação como a portabilidade, flexibilidade e a possibilidade de reutilização.

Um produto de *software* (ou *software*, como vamos chamar ao longo da disciplina), por outro lado, é sistematicamente destinado ao uso por pessoas diferentes dos seus programadores. Os eventuais usuários podem ter formações e experiências diferentes, o que significa que uma grande preocupação no que diz respeito ao desenvolvimento do produto deve ser a sua interface, reforçada com uma documentação rica em informações para que todos os recursos oferecidos possam ser explorados de forma eficiente. Ainda, os produtos de *software* devem passar normalmente por uma exaustiva bateria de testes, dado que os usuários não estarão interessados (e nem

terão capacidade) de detectar e corrigir os eventuais erros de execução.

Resumindo, um programa desenvolvido para resolver um dado problema e um produto de *software* destinado à resolução do mesmo problema são duas coisas totalmente diferentes. É óbvio que o esforço e o consequente custo associado ao desenvolvimento de um produto serão muito superiores.

Um sistema é bem mais que o *software*. Na verdade, o sistema é o conjunto de elementos, coordenados entre si e que funcionam como uma estrutura organizada. Embora o *software* seja uma parte importante de um sistema, ele não é o único. Se não existir o hardware para execução do *software*, de nada servirá. Da mesma forma, é necessário existir base de dados, uma vez que praticamente todos os sistemas com algum tipo de utilidade devem armazenar dados. Atualmente, com o advento da Internet, dificilmente um sistema seja útil se não tiver certos mecanismos de comunicação associados. Tudo isso junto forma o sistema. Por diversas vezes tendemos a utilizar *software* e sistema como algo similar, mas é importante ressaltarmos suas diferenças, de forma a deixar claro o que representa cada um desses termos.

Os produtos de *software* desenvolvidos utilizando a Engenharia de *Software* sempre estão envolvidos em algum processo de negócio, seja ele simples ou complexo. Assim, é fundamental entender esse processo de negócio para que seja possível informatizá-lo. Embora isso seja óbvio, em muitas ocasiões isso é simplesmente ignorado. O desenvolvimento de *software* muitas vezes acontece sem sabermos o que deve ser feito. Logicamente isso tem grande chance de resultar em fracasso.

É importante ressaltar que isso é inconcebível em qualquer área, mas parece que muitos acreditam que quando tratamos de *software* algo é diferente. Imagine adentrar em uma construtora de respeito e perguntar: Qual o custo de uma construção de uma casa? Se a empresa for séria, certamente não vai dar nenhuma estimativa de custo nem de prazo, uma vez que nada foi dito sobre o projeto. Uma casa pode ser feita gastando-se R$1.000,00 ou R$1.000.000,00. Tudo depende dos requisitos relacionados a casa. Com o *software* é exatamente o mesmo. Se não soubermos quais são os requisitos, é simplesmente impossível fazer qualquer estimativa de tempo e custo. Qualquer ação nesse sentido é irresponsabilidade ou extrema ignorância!

Em resumo, os sistemas informatizados normalmente não fazem

O termo bug está associado ao conceito de falha. Bug signifca inseto em Inglês. Diz-se que o termo foi criado por Thomas Edson quando um inseto causou problemas de leitura em seu fonógrafo em 1878, mas pode ser que o termo seja mais antigo. A invenção do termo é atribuída erroneamente a Grace Hopper, que publicou em 1945 que a causa do mau funcionamento no Mark II (um dos primeiros computadores) seria um inseto preso nos contatos de um relê.

o que deveriam fazer porque os problemas têm que ser minuciosamente definidos para que possam ser resolvidos. Se não entendermos bem sobre o domínio do problema, certamente não desenvolveremos uma boa solução. É fundamental levantarmos os requisitos de um *software* antes de procedermos com sua construção. O custo para o levantamento desses requisitos é um fator a ser considerado, mas não fazer tal levantamento provavelmente seja bem mais caro! Além disso, é importante que os usuários do produto participem desse levantamento, caso contrário, os dados obtidos no levantamento não deverão expressar aquilo que realmente é importante para o processo de negócio.

Software: mitos e realidade

Muitas causas de problemas relacionados ao desenvolvimento de *software* são provenientes de mitos que surgiram durante a história inicial dessa atividade, conforme descrito por ROGER PRESSMAN (2002). Diferentes dos antigos mitos, que são interessantes histórias e frequentemente fornecem lições humanas merecedoras de atenção, os mitos de *software* propagam desinformação e confusão. Esses mitos tinham certos atributos que os tornavam parecidos com afirmações razoáveis, tendo aspecto intuitivo e, muitas das vezes, eram divulgados por profissionais experientes e que deveriam entender do assunto. Atualmente, boa parte dos profissionais reconhece os mitos por aquilo que são: afirmações enganosas e que já causaram problemas. No entanto, cabe a todos nós conhecê-los, para tentar evitá-los em locais que isso ainda esteja acontecendo.

Mitos de gerenciamento

Mito 1. "Se a equipe dispõe de um manual repleto de padrões e procedimentos de desenvolvimento de *software*, então a equipe será capaz de conduzir bem o desenvolvimento."

Realidade 1. Isso não é o suficiente! É preciso que a equipe aplique efetivamente os conhecimentos apresentados no manual. É necessário que o manual reflita a moderna prática de desenvolvimento de *software* e que este seja exaustivo em relação a todos os problemas de desenvolvimento que poderão aparecer no percurso.

Mito 2. “A equipe tem ferramentas de desenvolvimento de *software* de última geração, uma vez que eles dispõem de computadores modernos.”

Realidade 2. Ter à sua disposição o último modelo de computador pode ser bastante confortável para o desenvolvedor do *software*, mas não oferece nenhuma garantia quanto à qualidade do produto desenvolvido. Mais importante do que ter um hardware de última geração, é ter ferramentas para a automação do desenvolvimento de *software* e saber utilizá-las adequadamente.

Mito 3. “Se o desenvolvimento do *software* estiver atrasado, aumentando a equipe poderemos reduzir o tempo de desenvolvimento.”

Realidade 3. Acrescentar pessoas em um projeto atrasado provavelmente vai atrasá-lo ainda mais. De fato, a introdução de novos profissionais numa equipe em fase de condução de um projeto vai requerer uma etapa de treinamento dos novos elementos da equipe; para isso, serão utilizados elementos que estão envolvidos diretamente no desenvolvimento, o que vai consequentemente implicar em maiores atrasos no cronograma.

Mitos do cliente

Mito 4. “Uma descrição breve e geral dos requisitos do *software* é o suficiente para iniciar o seu projeto. Mais detalhes podem ser definidos posteriormente.”

Realidade 4. Este é um dos problemas que podem conduzir um projeto ao fracasso. O cliente deve procurar definir o mais precisamente possível todos os requisitos importantes para o *software*: funções, desempenho, interfaces, restrições de projeto e critérios de validação são alguns dos pontos determinantes do sucesso de um projeto. O “deixar para depois” pode simplesmente não acontecer, a não ser em casos previstos pelos processos ágeis em que os clientes estão sempre presentes e dentro da organização desenvolvedora. No entanto, é sabido que essa prática é uma das mais difíceis de serem seguidas.

Mito 5. “Os requisitos de projeto mudam continuamente durante o seu desenvolvimento, mas isso não representa um problema, uma vez que o *software* é flexível e poderá suportar facilmente as alterações.”

Realidade 5. É verdade que o *software* é flexível (pelo menos mais flexível que a maioria dos produtos manufaturados). Entretanto, não existe *software*, por mais flexível, que suporte alterações de requisitos significativos sem um

adicional em relação ao custo de desenvolvimento. O fator de multiplicação nos custos de desenvolvimento do *software* devido a alterações nos requisitos cresce em função do estágio de evolução do projeto, como mostra a Figura 7.

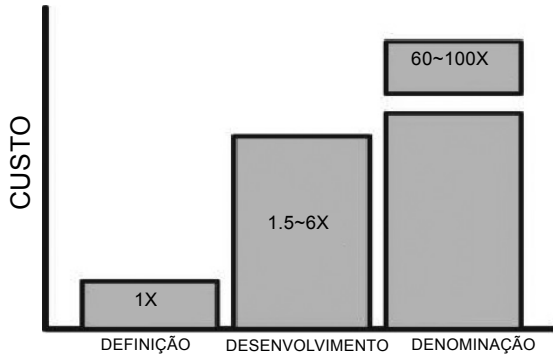

Figura 7: Influência das alterações de requisitos no custo de um sistema.

Mitos do profissional

Mito 6. "Após a finalização do programa e a sua implantação, o trabalho está terminado."

Realidade 6. O que ocorre, na realidade, é completamente diferente disso. Segundo dados obtidos a partir de experiências anteriores, ilustrados no livro de Roger Pressman (2002), 50 a 70% do esforço de desenvolvimento de um *software* é empregado após a sua entrega ao cliente (manutenção).

Mito 7. "Enquanto o programa não entrar em funcionamento, é impossível avaliar a sua qualidade."

Realidade 7. Na realidade, a preocupação com a garantia da qualidade do *software* deve fazer parte de todas as etapas do desenvolvimento. O teste, por exemplo, pode iniciar antes de o produto atingir um estado funcional, a partir do planejamento dos casos de teste.

Mito 8. "O produto a ser entregue no final do projeto é o programa funcionando."

Realidade 8. O programa em funcionamento é um dos componentes do *software*. Além do *software* em si, um bom projeto deve ser caracterizado pela produção de um conjunto importante de documentos. Um produto de *software* sem um manual de operação pode ser tão ruim quanto um *software* que não funciona!

Exercícios de fixação

1. Qual foi a principal causa do surgimento da Engenharia de *software*?

2. Quais eram os problemas associados à Crise do *Software*?

3. A Crise do *Software* realmente acabou? Comente sobre isso.

4. Faça uma comparação entre a Engenharia de *Software* e o desenvolvimento de um carro.

5. Defina Engenharia de *Software*.

6. Qual o tripé em que a Engenharia de *Software* está baseada?

7. O que é um sistema? Quais seus principais componentes?

8. É possível fazer a estimativa de custo e prazo para desenvolvimento de um *software* com apenas alguns poucos minutos de conversa? Comente sobre isso relacionando sua resposta a outras áreas.

9. O que são os mitos do *Software*?

10. Cite alguns mitos relacionados ao gerenciamento, comentando a realidade relacionada aos mitos.

11. Cite alguns mitos relacionados aos clientes, comentando a realidade relacionada aos mitos.

12. Cite alguns mitos relacionados aos profissionais do desenvolvimento de *software*, comentando a realidade relacionada aos mitos.

Processos de *Software*

Conforme comentado no capítulo anterior, a Engenharia de *Software* nada mais é que o tratamento do *software* como um produto, empregando dentro do processo de desenvolvimento todos os princípios da engenharia. Como todo produto, o *software* também possui um ciclo de vida, que pode ser definido como o conjunto de todas as etapas relacionadas à sua existência, desde a sua concepção até o seu desaparecimento. Isso inclui uma série de etapas, dentre elas as destacadas a seguir:

1. A concepção, na qual o produto é idealizado, a partir da percepção de uma necessidade;

2. O desenvolvimento, a partir da identificação dos requisitos e sua transformação em itens a serem entregues ao cliente;

3. A operação, quando o produto é instalado para ser utilizado em algum processo de negócio, sujeita a manutenção, sempre que necessário;

4. A retirada, quando o produto tem sua vida útil finalizada.

No ciclo de vida de um *software*, a codificação representa a escrita do programa utilizando alguma linguagem de programação, é apenas uma parte do ciclo de vida, embora muitos profissionais de informática acreditem, erroneamente, que essa seja a única tarefa relacionada ao desenvolvimento de *software*.

E já que estamos em um capítulo denominado Processo de *Software*, qual a relação com Ciclo de Vida? O Processo de *Software* é um guia de como um produto de *software* deve ser construído do início ao fim. A ligação está no fato de que esse guia depende do modelo de ciclo de vida utilizado. Existem vários modelos e, dependendo deles, as atividades a serem executadas podem variar. Essa é a ligação.

Necessitamos ainda definir o conceito de Processos de *Software* com maior exatidão. Um processo é um conjunto de passos parcialmente ordenados, constituídos por atividades, métodos, práticas e transformações, utilizados para se atingir uma meta. Uma meta está associada a resultados, que são os produtos resultantes da execução do processo.

É importante percebermos outra diferença que muitas vezes é imperceptível para muitos profissionais de informática: enquanto o processo é um guia para se construir um produto, um projeto é o uso de um processo para desenvolvimento de um produto específico. Isso se assemelha muito ao conceito de classe e objeto. Enquanto que classe é uma estrutura que abstrai um conjunto de objetos com características similares, um objeto é uma instância da classe, com valores específicos para cada uma dessas características. Dessa forma, o processo está para classe assim como o projeto está para um objeto. Em resumo, um projeto é a instanciação de um processo para a construção de um produto.

De acordo com Filho (2003), um processo deve ter uma documentação que o descreva, apresentando detalhes sobre o que é feito (produto), quando (passos), por quem (agentes), o que usa como entrada (insumo) e o que é produzido (resultado). Essa documentação pode ser simples e informal, como

é o caso dos Processos Ágeis, o qual verão mais adiante ou completamente detalhado, como é o caso dos processos baseados no Processo Unificado.

Existem processos que abrangem todo o ciclo de vida do *software*, assim como processos específicos para partes do desenvolvimento, como, por exemplo, processos para testes, processos para manutenção, etc., que são considerados subprocessos.

O termo "modelo de ciclo de vida" é utilizado para descrever um grupo de atividades relacionado ao desenvolvimento de software e as formas como elas se relacionam. Para cada modelo de ciclo de vida existe um relacionamento diferente entre as atividades, determinando formas diferentes de se conduzir o processo de construção do produto.

O ponto inicial para seleção de um processo e posterior iniciação de um projeto é entender qual modelo de ciclo de vida ele utiliza. Um modelo de ciclo de vida pode ser apropriado para um projeto, mas não ser apropriado para outro. É necessário entender bem os conceitos relacionados para que as escolhas feitas sejam baseadas em conhecimento e não no acaso. Na próxima seção iremos detalhar os modelos de ciclo de vida existentes, que muitas vezes são tratados como paradigmas de engenharia de *software*. Neste trabalho vamos utilizar o termo modelo de ciclo de vida ao invés de paradigma, por acharmos que ele é mais apropriado.

Modelos de ciclo de vida

Conforme já comentado, ciclo de vida pode ser resumido como sendo todas as atividades relacionadas a um *software*, desde a concepção de suas ideias até a descontinuidade do produto. Existem diversos modelos de ciclo de vida, com características diferentes. Nesta seção iremos apresentar os principais modelos de ciclo de vida.

Para detalhamento dos modelos de ciclo de vida, necessitaremos entender melhor cada um dos subprocessos mais importantes ligados às tarefas de desenvolvimento. Esses subprocessos são organizados de acordo com um tema e são chamados também de fluxos ou disciplinas. A seguir apresentamos uma breve descrição que serão um pouco mais detalhados nos capítulos posteriores:

1. Requisitos: obtenção do enunciado completo, claro e preciso dos desejos, necessidades, expectativas e restrições dos clientes em relação ao produto a ser desenvolvido.

2. Análise: modelagem dos conceitos relevantes do domínio do problema, com o intuito de verificar a qualidade dos requisitos obtidos e detalhar tais requisitos em um nível adequado aos desenvolvedores.

3. Desenho (ou projeto): definição de uma estrutura implementável para um produto que atenda aos requisitos especificados.

4. Implementação: codificação das partes que compõem o *software*, definidas no desenho, utilizando as tecnologias selecionadas.

5. Teste: verificação dinâmica das partes que constituem o *software*, utilizando um conjunto finito de casos de teste, selecionados dentro de um domínio potencialmente infinito, contra seu comportamento esperado.

A literatura cita vários tipos de modelos de ciclo de vida de *software*. Não consideramos que alguns desses modelos devam receber este status. No entanto, iremos comentar sobre eles ao final da seção. São exemplos o modelo de prototipação e modelo de desenvolvimento rápido. Esses modelos são, em nossa visão, apenas o desenvolvimento de *software* baseado em certas técnicas e ferramentas. Embora alguns autores renomados os apresentem como modelos de ciclo de vida, o autor os considera como abordagens para aplicação do ciclo de vida.

Codifica-remenda

O modelo de ciclo de vida mais utilizado é o modelo codifica-remenda. Conforme exibido na Figura 8, partindo de uma especificação incompleta ou mesmo ausente, inicia-se a codificação do *software*, que por sua vez tende a gerar "algo". Esse "algo gerado", na grande maioria das vezes, não é o que o cliente deseja, mas vai sendo alterado e consertado até que o produto atinja um estágio que permita seu uso. Nenhum processo é seguido nessa interação.

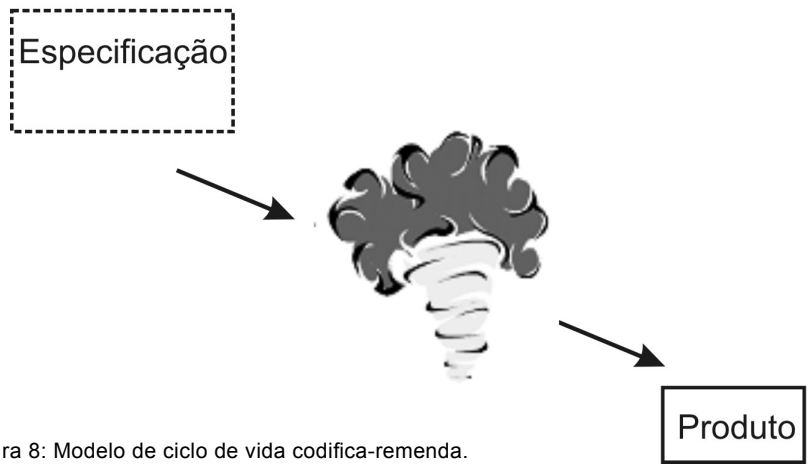

Figura 8: Modelo de ciclo de vida codifica-remenda.

A grande utilização desse modelo se dá em virtude de boa parte dos "profissionais" responsáveis pelo desenvolvimento de *software* não terem qualquer conhecimento sobre a Engenharia de *Software*. É possível que o uso desse modelo de ciclo de vida gere resultados aceitáveis para produtos pequenos e com equipes formadas por poucas pessoas, normalmente por um único desenvolvedor, mas certamente ele só gerará problemas em qualquer projeto envolvendo produtos que requeiram equipes maiores.

O aspecto "positivo" desse modelo é que não exige nenhum conhecimento técnico ou gerencial. Basta iniciar sem muito preparo sobre o tema a ser abordado e trabalhar para gerar algo como resultado. Por outro lado, é um modelo de alto risco, praticamente impossível de se gerenciar, tornando-se impossível estabelecer qualquer estimativa de custo e prazo.

O modelo cascata ainda é o mais comum, pela falta de profissionais habilitados para o desenvolvimento de software. De forma similar, existem muito mais obras sendo tocadas por profissionais sem qualquer formação técnica que obras tocadas por engenheiros. O resultado disso é que dificilmente uma obra sem um engenheiro possuirá custo e prazo conhecidos!

Cascata

O modelo em cascata, também conhecido como modelo sequencial linear, sugere que os principais subprocessos relacionados ao desenvolvimento de *software* sejam executados em sequência, conforme apresentado na Figura 9.

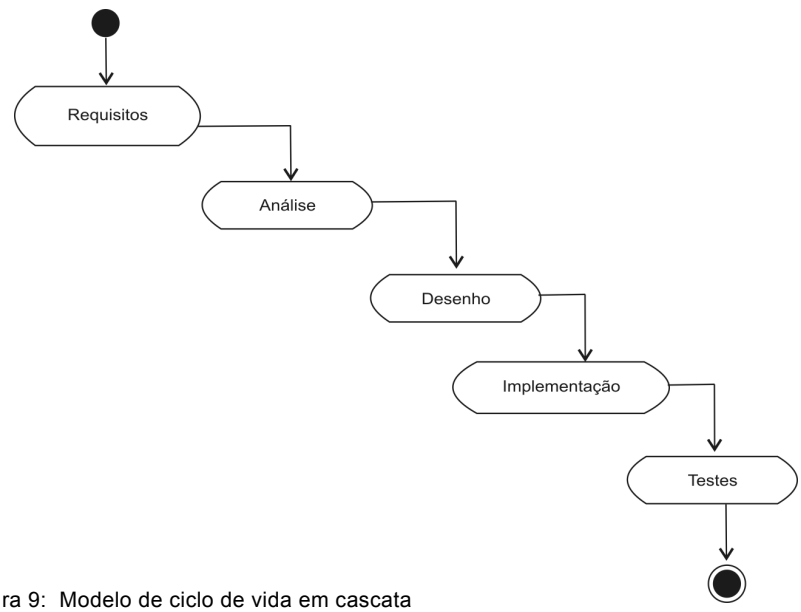

Figura 9: Modelo de ciclo de vida em cascata

Esse modelo de ciclo de vida (ou paradigma) foi muito utilizado no passado. No entanto, devido a falhas durante sua execução, ele vem sendo

preterido por outros modelos. Dentre essas falhas, destacam-se as relatadas por Pressman (2002):

1. Projetos reais raramente seguem o fluxo sequencial, conforme sugerido pelo modelo. Isso pode causar grandes problemas quando temos mudanças em um projeto em andamento;

2. Em geral, é difícil para os clientes identificarem todos os requisitos explicitamente. Esse modelo exige isso e tem dificuldade em acomodar a incerteza natural que existe em grande parte dos projetos;

3. O cliente precisa ter paciência, uma vez que a primeira versão executável do produto somente estará disponível no fim do projeto. Erros grosseiros podem ser desastrosos e causar perda de praticamente todo o trabalho. Esse é, talvez, o maior problema: a baixa visibilidade por parte do cliente.

Espiral

Um modelo bem diferente do apresentado anteriormente (cascata) é o modelo em espiral. Esse modelo foi proposto originalmente por Barry Boehm. A ideia básica é desenvolver um produto a partir de pequenas versões incrementais, que podem iniciar com um modelo em papel e evoluir até versões do sistema completamente funcionais.

O modelo em cascata é, em teoria, uma maravilha. Na prática, é rígido e burocrático, tendo sido considerado inviável pelas organizações que o utilizam.

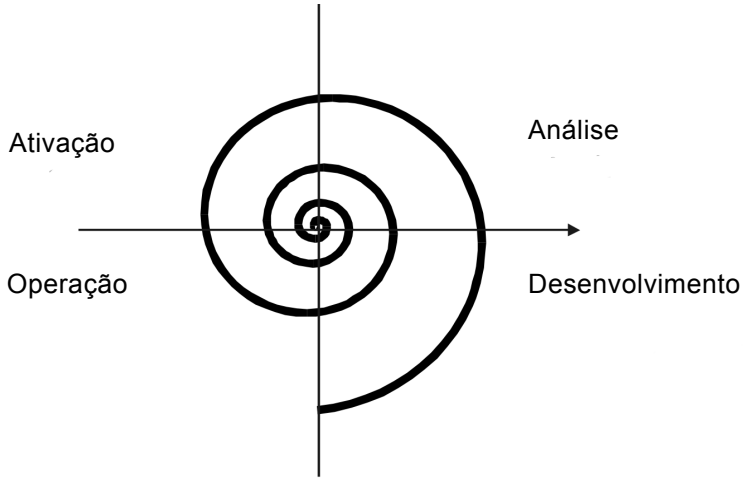

Figura 10: Modelo de ciclo de vida em espiral

Podemos fazer uma comparação com o modelo em cascata: uma volta na espiral equivale à execução do modelo em cascata para uma

pequena parte do *software*. Durante essa volta na espiral, deve ser realizado o levantamento de requisitos para a pequena parte do *software* que desejamos abordar: a modelagem desses requisitos (análise), o projeto das partes que serão desenvolvidas (desenho), a codificação dessas partes (implementação) e sua verificação (teste). Quando isso acontece temos o início de um novo ciclo e tudo se repete, até que tenhamos todo o produto desenvolvido.

É importante ressaltar que as diversas voltas na espiral são utilizadas para se construir as partes do produto, mas essas partes intermediárias e ainda incompletas do produto não são entregues ao cliente. Essa abordagem é utilizada apenas para a redução dos riscos e para o enfoque maior naquilo que é mais importante/complexo/crítico no sistema. O próximo modelo de ciclo de vida que apresentamos é baseado nesse modelo, porém apresenta uma leve e sutil diferença: a entrega das partes para o cliente.

A ideia básica por trás do modelo em espiral é: dividir para conquistar. Ao invés de querer resolver todos os problemas de uma vez, é interessante resolver parte deles e então partir para o restante.

O fato de dividir um sistema para tratá-lo por partes favorece o entendimento e facilita a própria vida dos clientes, que podem emitir opiniões e identificar requisitos com mais propriedade. Isso facilita o uso desse tipo de modelo de ciclo de vida. No entanto, como apenas uma parte do produto é considerada, é possível que uma decisão tomada seja incompatível com uma nova parte sendo desenvolvida. Isso acontece porque a tomada de decisões é feita apenas com o conhecimento do produto que existe até o momento. O ideal seria que todo o produto já fosse conhecido, o que nos remete ao modelo em cascata novamente. Esse ponto é justamente o problema do modelo em espiral: sua dificuldade de gerir, para que tenhamos estimativas em um projeto que seja previsível e confiável.

Incremental

O modelo incremental, também denominado de prototipagem evolutiva, é baseado no modelo em espiral. No entanto, a grande diferença para o modelo anterior é que as versões parciais do produto são efetivamente entregues aos clientes para que sejam verificadas, validadas e utilizadas no ambiente operacional. É importante ressaltar essa diferença: o modelo incremental não é utilizado apenas para construir o produto por completo, mas uma série de versões provisórias, denominadas protótipos, que cobrem cada vez mais requisitos e que são efetivamente entregues aos clientes.

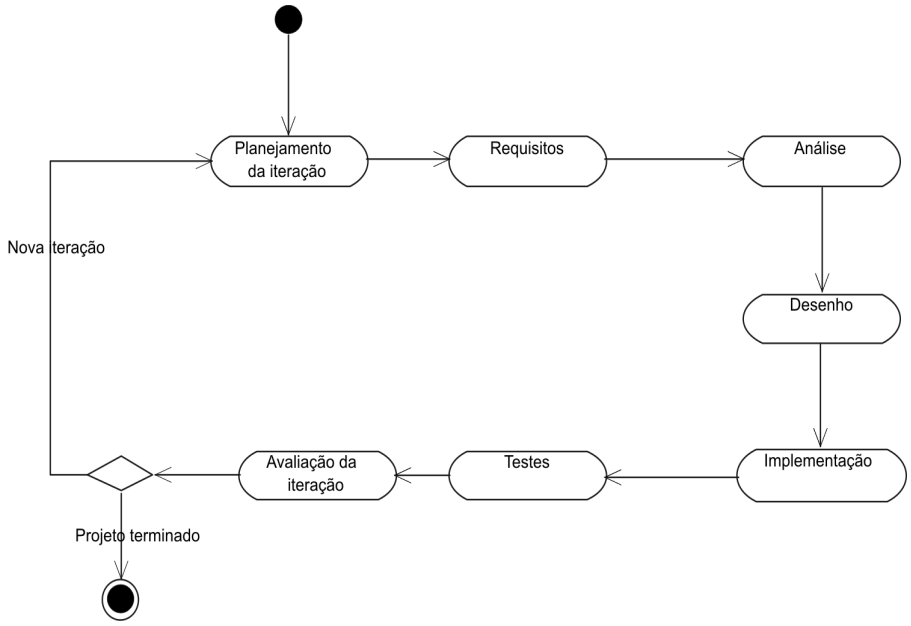

Figura 11: Modelo de ciclo de vida incremental (prototipagem evolutiva)

No modelo incremental, um projeto é dividido em várias interações, os quais que são divisões do produto que serão consideradas durante a construção. Escolhe-se uma dessas interações para iniciar o desenvolvimento, começando pelo planejamento da mesma, levantamento de requisitos, análise, desenho, implementação, testes e avaliação, geralmente realizada com a participação do cliente, conforme apresentado na Figura 11. Essa versão do *software* normalmente é liberada para uso e, caso o produto não tenha sido concluído, uma nova interação é planejada para prosseguir com o desenvolvimento.

O modelo incremental sofre dos mesmos problemas do modelo em espiral, no entanto, apresenta como grande vantagem o fato de os requisitos serem definidos progressivamente com alta flexibilidade e visibilidade para os clientes. No entanto, requer gerência sofisticada e uma arquitetura forte para que o produto inicialmente concebido não esbarre em problemas que impeçam sua continuidade a partir dos direcionamentos (decisões arquiteturais) inicialmente concebidos.

Processos de *software* famosos, como o XP e Scrum, utilizam esse modelo de ciclo de vida. Esses processos são conhecidos como Processos (ou Métodos ou Metodologias) Ágeis, em virtude de darem mais importância

Embora os processos ágeis prescrevam pouca burocracia, eles prescrevem alguma burocracia. Não segui-las pode significar anarquia total no desenvolvimento. Existem muitas organizações que dizem seguir os preceitos ágeis, apenas porque não seguem nada, o que é um completo absurdo e podem manchar as metodologias ágeis.

ao *software* funcionando do que qualquer outro artefato gerado durante o desenvolvimento.

Os métodos ágeis surgiram como uma resposta aos métodos mais estruturados, especialmente aos métodos baseados no modelo em cascata. Processos orientados a documentação para o desenvolvimento de *software*, como o utilizado no modelo em cascata, são de certa forma fatores limitadores aos desenvolvedores. Muitas organizações não possuem recursos ou inclinação para processos burocráticos para a produção de *software*, baseados em intensa documentação. Por esta razão, muitas organizações acabam por não usar nenhum processo, o que pode levar a efeitos desastrosos em termos de qualidade de *software*, conforme descrito por Cockbun (2003). Como alternativa a esse cenário, surgiram os métodos ágeis que apesar de possuírem documentação e alguma burocracia, são mais flexíveis, orientados a entregas e possuem uma maior interatividade no processo de desenvolvimento e codificação do produto.

A maioria dos métodos ágeis não possui nada de novo em relação aos processos tradicionais. O que os diferencia dos outros métodos são o enfoque e os valores. A ideia dos métodos ágeis é o enfoque nas pessoas e não em processos ou algoritmos. Além disso, existe a preocupação de gastar menos tempo com documentação e mais com a implementação. Uma característica dos métodos ágeis é que eles são adaptativos ao invés de serem preditivos. Com isso, eles se adaptam a novos fatores decorrentes do desenvolvimento do projeto, ao invés de procurar analisar previamente tudo o que pode acontecer no decorrer do desenvolvimento.

O termo “metodologias ágeis” tornou-se popular em 2001, quando diversos especialistas em processos de desenvolvimento de *software* representando as metodologias Extremem Programming (XP), SCRUM, DSDM (Dynamic Systems Development Methodology), Crystal e outros estabeleceram princípios comuns compartilhados por todos esses métodos. O resultado foi a criação da Aliança Ágil, e o estabelecimento do “Manifesto Ágil” (Agile Manifesto). Os conceitos chaves do Manifesto Ágil são:

1. Indivíduos e interações ao invés de processos e ferramentas;

2. *Software* executável ao invés de documentação;

3. Colaboração do cliente ao invés de negociação de contratos;

4. Respostas rápidas a mudanças ao invés de seguir planos.

O “Manifesto Ágil” não rejeita os processos, as ferramentas, a documentação, a negociação de contratos ou o planejamento, mas simplesmente mostra que eles têm importância secundária quando comparados com os indivíduos e interações com o *software* estar executável, com a colaboração do cliente e as respostas rápidas a mudanças e alterações. Esses conceitos aproximam-se melhor à forma com que pequenas companhias de Tecnologia da Informação trabalham e respondem a mudanças.

Para ser realmente considerada ágil, a metodologia deve aceitar a mudança, ao invés de tentar prever o futuro. O problema não é a mudança em si, mesmo porque ela ocorrerá de qualquer forma. O problema é como receber, avaliar e responder às mudanças.

Enquanto as metodologias ágeis variam em termos de práticas e ênfases, elas compartilham algumas características, como desenvolvimento interativo e incremental; comunicação e redução de produtos intermediários e documentação extensiva. Desta forma, existem maiores possibilidades de atender aos requisitos do cliente, que muitas vezes são mutáveis.

Entrega evolutiva

Uma combinação muito interessante entre o modelo em cascata e o modelo incremental é o modelo de entrega evolutiva, apresentado na Figura 12. Ele é uma combinação dos modelos de Cascata e Prototipagem Evolutiva. Nesse modelo, o levantamento de requisitos é feito como um todo, para que seja possível entender todas as questões que envolvem o produto a ser construído, porém, em pontos bem definidos os usuários podem avaliar as partes do produto já desenvolvidas, fornecendo

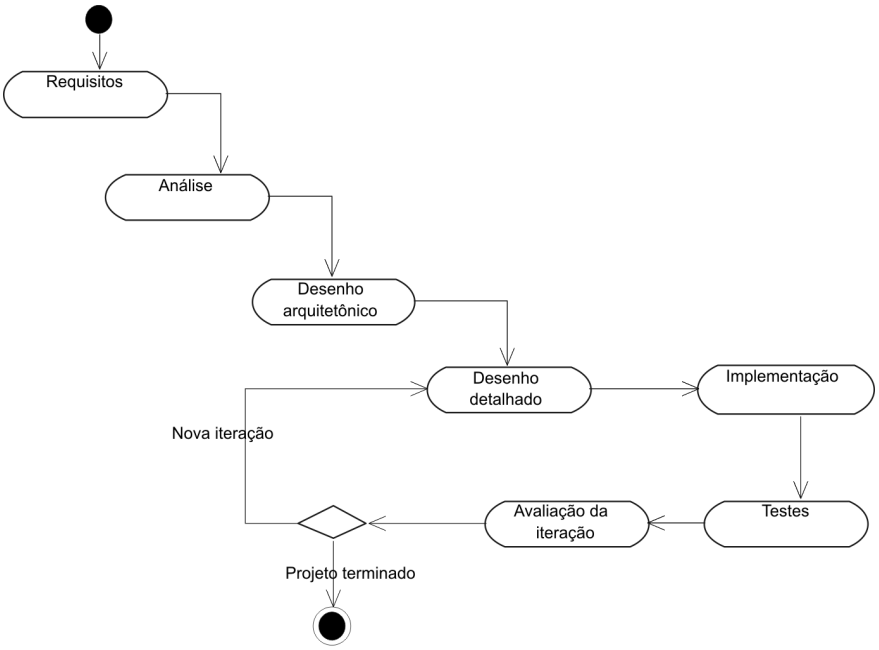

Figura 12: Modelo de ciclo de vida entrega evolutiva.

realimentação quanto às decisões tomadas, de forma bem similar ao que acontece na Prototipagem Evolutiva. Isso facilita o acompanhamento dos projetos, tanto por parte dos gerentes, quanto por parte dos clientes, que podem acompanhar o que está sendo realizado.

A principal dificuldade associada ao modelo de entrega evolutiva é o levantamento de requisitos, que deve ser bem-feito para que a partir disso seja possível a definição da arquitetura do produto, o qual deve ser robusta, mantendo-se íntegra ao longo dos ciclos de liberações parciais. De uma forma geral, a arquitetura é chave nesse modelo. Um dos grandes representantes desse modelo de ciclo de vida é o Processo Unificado (PU). Vários processos, tais como o RUP (Rational Unified Process) e a PRÁXIS utilizam o PU como processo base. Esses processos são muito utilizados no mundo com vários casos de sucesso.

Um erro muito comumente cometido pelos defensores dos métodos ágeis é afirmar que processos como o RUP e PRÁXIS são baseados no modelo em cascata. Esse é um erro grosseiro e frequente. Embora os Processos Ágeis sejam muito interessantes, é comum a existência de "xiitas" que defendem tais métodos sem avaliar os prós e contras de cada um dos modelos de ciclo de vida associados.

O modelo de entrega evolutiva é atual e possui inúmeros casos de sucesso. Muitos dos defensores dos processos ágeis confundem a entrega evolutiva com o modelo em cascata, o que representa um erro absurdo.

Normalmente, onde os requisitos são conhecidos e relativamente estáveis os processos baseados no modelo de entrega evolutiva são mais apropriados; para projetos envolvendo equipes pequenas e cujos requisitos não sejam bem conhecidos, ou seja, muito instáveis, os quais os processos baseados no modelo incremental são mais apropriados.

De forma geral, o bom senso é o melhor termômetro para definição do que é mais apropriado em uma situação, mas um conselho a todos que possam se deparar com essa situação é: **nunca acredite em pessoas que defendem cegamente esse ou aquele processo; como tudo na vida, o contexto da situação pode determinar o que é mais apropriado.**

Outros modelos de ciclo de vida

Alguns autores, erroneamente, em nossa opinião, consideram como sendo outros modelos de ciclo de vida o uso de certas ferramentas e tecnologias específicas. Esse é o caso, por exemplo, de considerar que o Desenvolvimento Baseado em Componentes ou que o uso de ferramentas

RAD sejam modelos de ciclo de vida. No nosso ponto de vista, isso é apenas a definição de uso de certas tecnologias e que não geram ciclos de vida, diferenciados do que já foi exposto neste capítulo.

Exercícios de fixação

1. Quais são as principais fases do ciclo de vida de um produto de *software*?

2. Qual a ligação entre Ciclo de Vida e Processo de *Software*?

3. Qual a definição para Processo de *Software*?

4. Quais são os principais subprocessos (também conhecidos como fluxos ou disciplinas) ligados ao desenvolvimento de *software*?

5. Descreva o modelo de ciclo de vida denominado Codifica-Remenda.

6. Quais são os problemas relacionados ao modelo de ciclo de vida em cascata?

7. Como podemos relacionar o modelo em cascata e o modelo em espiral?

8. Qual a grande diferença entre o modelo em espiral e o modelo incremental?

9. Os métodos ágeis são baseados no manifesto ágil. Quais são os conceitos chaves desse manifesto?

10. Como funciona o modelo de entrega evolutiva?

11. Os dois modelos de ciclo de vida recomendados para uso e mais utilizados atualmente são o modelo incremental e o modelo de entrega evolutiva. Quando um é mais apropriado que outro?

UNIDADE 02

As Principais Disciplinas da Engenharia de Software

Resumindo

Nesta unidade apresentamos os principais fluxos (ou subprocessos) que guiam a execução de projetos de desenvolvimento de software baseados em princípios de Engenharia. Este capítulo é totalmente baseado no livro do prof. Wilson de Pádua Filho (2003) e seu Processo Práxis, uma vez que o mesmo possui exemplos ilustrativos de todas as partes que compõem o desenvolvimento de software, aliados a uma definição precisa das atividades, métodos e técnicas relacionados a cada fluxo.

2 As principais disciplinas da Engenharia de *Software*

Requisitos

Nesta unidade iremos apresentar os principais fluxos da Engenharia de *Software*. Conforme comentamos, utilizaremos o livro do Prof. Wilson de Pádua Filho (2003) como base nessa explicação. Esse livro descreve o Processo Práxis, que será brevemente apresentado a seguir, para então darmos início ao detalhamento dos fluxos ligados à Engenharia de *Software*.

Processo Práxis

O processo Práxis é um processo de *software* baseado no modelo de entrega evolutiva. O termo significa PRocesso para Aplicativos eXtensíveis e InterativoS e constitui um processo de desenvolvimento de *software* desenhado para projetos com duração de seis meses a um ano, realizados individualmente ou por pequenas equipes. Ele é voltado para o desenvolvimento de aplicativos gráficos interativos, baseados na tecnologia orientada a objetos.

Seguindo a arquitetura definida no Processo Unificado, a Práxis propõe um ciclo de vida composto por fases, divididas em uma ou mais interações e fluxos, que correspondem a disciplinas da Engenharia de *Software*.

Durante a execução das fases, diversos fluxos podem ser necessários, de acordo com o que é descrito nos scripts das interações.

A Tabela 1 apresenta a divisão em fases e iterações, enquanto a organização em fluxos é descrita na Tabela 2.

NATUREZA	FLUXO	DESCRIÇÃO
TÉCNICOS	Requisitos	Fluxo que visa a obter um conjunto de requisitos de um produto, acordado entre cliente e fornecedor.
	Análise	Fluxo que visa a detalhar, estruturar e validar os requisitos de um produto, em termos de um modelo conceitual do problema, de forma que eles possam ser usados como base para o planejamento e controle detalhados do respectivo projeto de desenvolvimento.
	Desenho	Fluxo que visa a formular um modelo estrutural do produto que sirva de base para a implementação, definindo os componentes a desenvolver e a reutilizar, assim como as interfaces entre si e com o contexto do produto.
	Implementação	Fluxo que visa a detalhar e implantar o desenho através de componentes de código e de documentação associada.
	Testes	Fluxo que visa a verificar os resultados da implementação, através do planejamento, desenho e realização de baterias de testes.
	Gestão de Projeto	Fluxo que visa a planejar e controlar os projetos de *software*.
GERENCIAIS	Gestão da Qualidade	Fluxo que visa a verificar e assegurar a qualidade dos produtos e processos de *software*.

Tabela 1: Principais fluxos técnicos e gerenciais do Práxis

Apesar de ter sido originalmente destinado à utilização em projetos didáticos das disciplinas de Engenharia de *Software*, o processo Práxis vem sendo usado com sucesso também em projetos maiores, alguns envolvendo equipes de até dezenas de pessoas. Conforme comentado anteriormente, nesta unidade vamos descrever os principais fluxos técnicos e gerenciais do processo, uma vez que eles são similares em quase todos os processos existentes.

Fluxo de requisitos

O fluxo de requisitos possui atividades que visam a obter o enunciado completo, claro e preciso dos requisitos de um produto de *software*. Os requisitos correspondem a qualquer desejo, necessidade, restrição ou expectativa dos clientes em relação a um produto de *software*. Estes requisitos devem ser levantados pela equipe do projeto, em conjunto com representantes do cliente, usuários chaves e outros especialistas da área de aplicação.

O conjunto de técnicas empregadas para levantar, detalhar, documentar e validar os requisitos de um produto forma a Engenharia de Requisitos. O

resultado principal do fluxo de requisitos é um documento de Especificação de Requisitos de *Software* (que abreviaremos ERSw). Esse documento possui uma caracterização do problema do cliente e que deve ser utilizado para criação de um produto.

Projetos de produtos mais complexos geralmente precisam de mais investimento em Engenharia de Requisitos que projetos de produtos mais simples, por questões óbvias. A Engenharia de Requisitos é também mais complexa no caso de produtos novos. O desenvolvimento de uma nova versão de um produto existente é muito ajudado pela experiência dos usuários com as versões anteriores, uma vez que permite identificar de forma rápida e clara aquilo que é o mais importante, que não pode ser mudado e tudo aquilo que não funciona e precisa de uma nova solução urgente. No caso de um novo produto, é difícil para os usuários identificarem o que é mais importante, tornando igualmente difícil para os desenvolvedores entenderem claramente o que os usuários desejam.

Uma boa Engenharia de Requisitos é um passo essencial para o desenvolvimento de um bom produto, em qualquer caso. Neste capítulo descrevemos brevemente as atividades do fluxo de Requisitos do Processo Práxis. Para garantir ainda mais a qualidade, os requisitos devem ser submetidos aos procedimentos de controle previstos no processo e devem ser verificados através das atividades de Análise.

Requisitos de alta qualidade são claros, completos, sem ambiguidade, implementáveis, consistentes e testáveis. Os requisitos que não apresentem estas qualidades são problemáticos: eles devem ser revistos e renegociados com os clientes e usuários.

A Especificação dos Requisitos do *Software* é o documento oficial de descrição dos requisitos de um projeto de *software*. Ela pode se referir a um produto indivisível de *software* ou a um conjunto de componentes de *software*, que formam um produto quando usados em conjunto (por exemplo, um módulo cliente e um módulo servidor).

As características que devem estar contidas na Especificação dos Requisitos do *Software* incluem:

1. Funcionalidade: O que o *software* deverá fazer?

2. Interfaces externas: Como o *software* interage com as pessoas, com o hardware do sistema, com outros sistemas e com outros produtos?

3. Desempenho: Qual o tempo de resposta máximo permitido dado a um contexto de funcionamento, como por exemplo, 100 usuários realizando empréstimo de um livro?

4. Outros atributos: Quais as considerações sobre portabilidade, manutenibilidade e confiabilidade que devem ser observadas?

5. Restrições impostas pela aplicação: Existem padrões e outros limites a serem obedecidos, como linguagem de implementação, ambientes de operação, limites de recursos etc.?

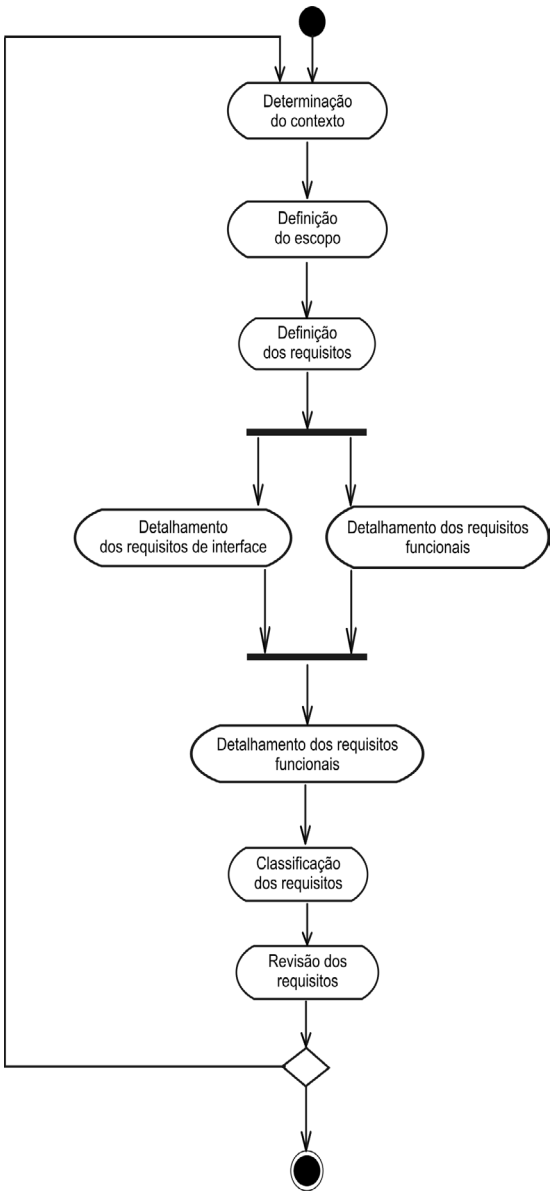

Figura 13: Fluxo de requisitos.

A Figura 13 apresenta as atividades do fluxo de Requisitos do Processo Práxis. O fluxo é iniciado através da "Determinação do contexto" que levanta aspectos dos processos de negócio ou de um sistema maior que sejam relevantes para a determinação dos requisitos do produto. A "Definição do escopo" delimita os problemas que o produto se propõe a resolver.

A "definição dos requisitos" produz uma lista de todos os requisitos funcionais e não funcionais. Estes requisitos são descritos de forma sucinta, ainda sem entrar em detalhes.

A Tabela 2 abaixo exibe um exemplo de requisitos definidos para um produto responsável pelo controle de leitos hospitalares. Normalmente, utilizamos o conceito de Caso de Uso associado às funcionalidades de um produto.

São também identificados os grupos de usuários do produto, também denominados **Atores** e as demais restrições aplicáveis. Um diagrama de casos de uso exibe os relacionamentos entre atores e casos de uso, apresentado na Figura 14. É recomendável, até este ponto, que os tomadores de decisão do cliente participem do levantamento dos requisitos.

As atividades seguintes cobrem aspectos

mais detalhados, sendo provavelmente mais adequado que participem os usuários chaves e não necessariamente pessoal gerencial do cliente. Ela será revista nesta fase, normalmente com a participação de um número maior de partes interessadas.

NR	CASO DE USO	DESCRIÇÃO
1	**Gestão de Hospitais**	Cadastro dos hospitais com leitos controlados pelo **SRLEP.**
2	**Gestão de Leitos**	Cadastro dos leitos hospitalares controlados pelo **SRLEP.**
3	**Autorização de envio para lista de espera**	Registro da autorização da solicitação para constar na lista de espera por leitos.
4	**Controle de Internações**	Controle das internações em leitos, com possibilidade registro de alta, e consequente liberação do leito, além de transferências para outros leitos.
5	**Consulta Escala**	Visualização de quadro de escala hospitalar.

Tabela 2: Exemplo de requisitos para um produto de software

Um Caso de Uso é uma fatia de funcionalidade do sistema, sem superposição ou lacunas, que representa algo de valor para os usuários finais.

Um **Ator** é representação dos usuários e outros sistemas que interagem com o produto.

As três atividades posteriores correspondem ao detalhamento dos requisitos de interface, funcionais e não funcionais.

Os dois primeiros são mostrados como atividades paralelas, pois existem interações fortes entre os requisitos de interface e os requisitos funcionais.

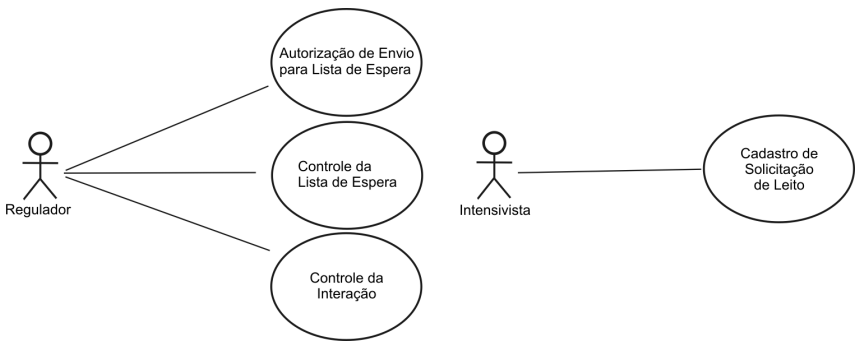

Figura 14: Exemplo de diagrama de caso de uso.

Durante o “detalhamento dos requisitos de interface” são detalhados os aspectos das interfaces do produto que os usuários consideram requisitos. Normalmente, é feito um esboço das interfaces de usuário, levantado através de um protótipo executável ou de estudos em papel. Esses esboços, entretanto, não devem descer a detalhes de desenho, mas apenas facilitar a visualização dos verdadeiros requisitos (por exemplo, que informação a interface deve captar

e exibir).

A Figura 15 exibe um esboço de uma interface de usuário sem amarração a qualquer tecnologia de implementação. São também detalhadas as interfaces com outros sistemas e componentes de sistema. No Processo Práxis, o "detalhamento dos requisitos funcionais" utiliza a notação de casos de uso. O fluxo de execução das funções é descrito de forma padronizada, dentro da Especificação dos Requisitos do *Software*.

Um Diagrama de Casos de Uso exibe os relacionamentos entre atores e casos de uso, deixando claro quais grupos utilizam determinadas funções.

Gestão de Condições de Internação	
Pesquisa por Condições de Internação	
Condição	Insuficiência hepática grave (até 50 caracteres)
<Pesquisar>	
Condições de internação recuperadas	
Condição	**Tipo**
Insuficiência respiratória grave	Primária
Coma mixedematoso	Secundária
<Novo> <Alterar>	
Condições de Internação	
Condição*	Insuficiência hepática grave (até 50 caracteres)
Tipo	[Primária; Secundária]
<Salvar>	

Figura 15: Exemplo de um protótipo de tela independente de tecnologia.

O "detalhamento dos requisitos não funcionais" completa os requisitos, descrevendo os requisitos de desempenho e outros aspectos considerados necessários para que o produto atinja a qualidade desejada. Inclui-se aqui também o detalhamento de **requisitos derivados de outros tipos de restrições (por exemplo,** restrições de desenho).

A "classificação dos requisitos" determina as prioridades relativas dos requisitos e avalia a estabilidade e a complexidade de realização. Os requisitos aprovados são lançados no Cadastro dos Requisitos do *Software* para que sejam posteriormente registrados e rastreados seus relacionamentos com seus derivados, em outros artefatos do projeto.

Finalmente, a "revisão dos requisitos" determina se todos eles satisfazem os critérios de qualidade de requisitos e se a Especificação dos Requisitos do *Software* está clara e bem entendida por todas as partes interessadas. Na disciplina Requisitos de *Software* iremos detalhar com bastante profundidade este fluxo, juntamente com os métodos e técnicas associados.

Exercícios de fixação

1. Qual o objetivo do fluxo de requisitos?

2. Por que o levantamento de requisitos para produtos novos geralmente é mais difícil que para produtos já existentes?

3. O que deve conter uma "especificação de requisitos"?

4. Quais as atividades de requisitos?

5. Em qual atividade é feita uma descrição dos requisitos de forma sucinta?

6. Como são representados os grupos de usuários do produto?

7. Que notação é utilizada para descrição dos requisitos funcionais?

8. Por que é aconselhado fazer um esboço das telas de forma independente de tecnologia?

É aconselhável a utilização de um formato independente de tecnologia para descrição das telas de um sistema. Isso evita uma redução no espaço das soluções. Utilizar um componente do tipo Select ou Radio Button pode bloquear a visão de desenvolvedor para soluções mais interessantes!

Um Diagrama de Casos de Uso exibe os relacionamentos entre atores e casos de uso, deixando claro que grupos utilizam quais funções.

Na disciplina Requisitos de Software será detalhado o fluxo de requisitos e análise. Iremos realizar diversas práticas relacionadas ao tema.

Fluxo de análise

O fluxo da análise tem como objetivos:

1. Modelar de forma precisa os conceitos relevantes do domínio do problema, identificados a partir do levantamento de requisitos;

2. Verificar a qualidade dos requisitos identificados;

3. Detalhar esses requisitos o suficiente para que atinjam o nível de detalhe adequado aos desenvolvedores.

No processo Práxis, os Métodos de Análise são baseados na Tecnologia Orientada a Objetos, resultando em um Modelo de Análise do *Software*, expresso na notação UML.

O modelo de análise deve conter os detalhes necessários para servir de base para o desenho do produto, mas se deve evitar a inclusão de detalhes que pertençam ao domínio da implementação e não do problema.

Quando se usa um Modelo de Análise orientado a objetos, os requisitos funcionais são tipicamente descritos e verificados através dos seguintes recursos de notação:

1. Os casos de uso descrevem o comportamento esperado do produto. Seus

diagramas descrevem os relacionamentos dos casos de uso entre si e com os atores;

2. As classes representam os conceitos do mundo da aplicação que sejam relevantes para a descrição mais precisa dos requisitos, exibindo os relacionamentos entre essas.

As realizações dos casos de uso mostram como objetos das classes descritas colaboram entre si para realizar os principais roteiros que podem ser percorridos de

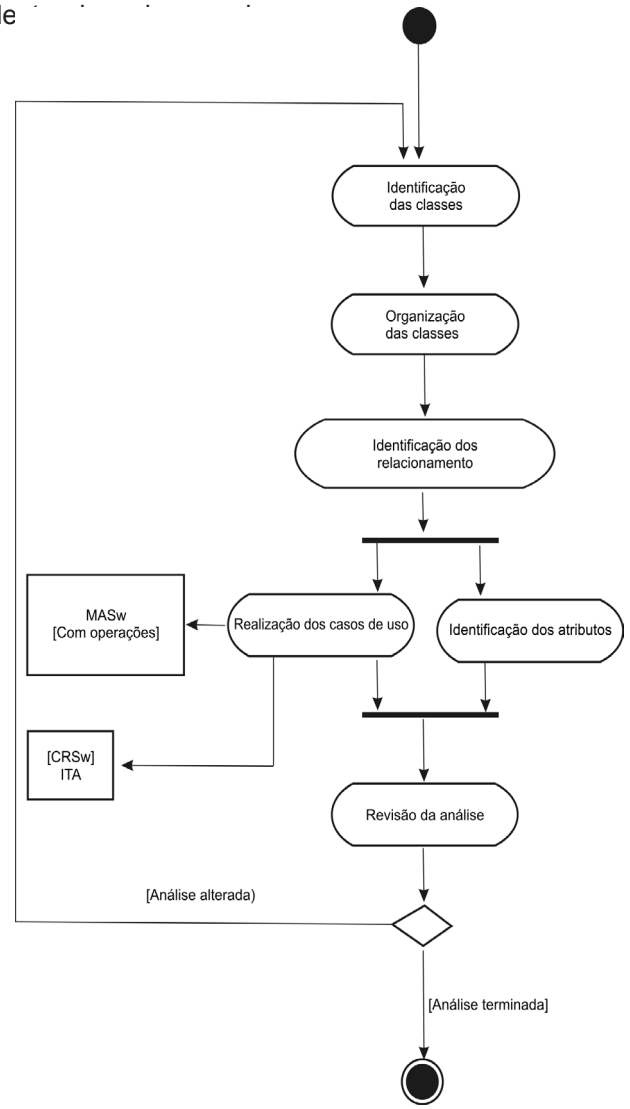

Figura 16: O Fluxo de Análise

A Figura 16 exibe as atividades da Análise. Ela inicia pela “Identificação

das classes", na qual são analisados os fluxos dos casos de uso e outros documentos relevantes em relação ao produto desejado. Os conceitos candidatos a classes são localizados e filtrados de acordo com vários critérios. Prossegue com a "Organização das classes" que organiza as classes em pacotes lógicos (agrupamentos de classes correlatas) e lhes atribui os estereótipos de entidade, fronteira e controle, dependendo do papel que desempenham no modelo. A "Identificação dos relacionamentos" determina os relacionamentos de vários tipos que podem existir entre os objetos das classes identificadas.

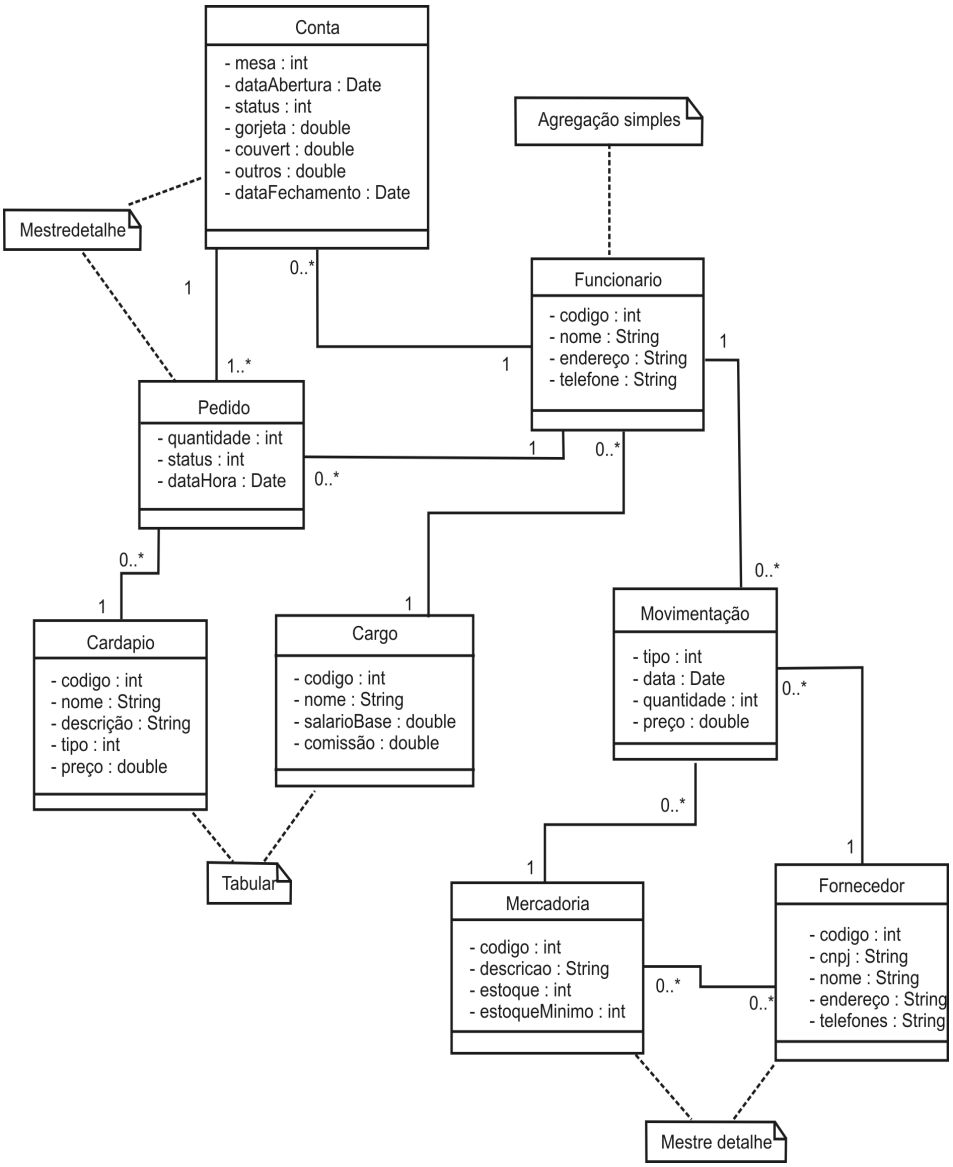

Figura 17: Exemplo de diagrama de classes exibindo classes, atributos e relacionamentos

Em seguida, a “Identificação dos atributos” levanta os atributos de análise, isto é, propriedades que fazem parte do conceito expresso pela classe. Todas as atividades descritas anteriormente dão origem ao diagrama de classes contendo todos os detalhamentos de atributos, divisão em pacotes e com especificação dos relacionamentos. Um exemplo de diagrama contendo essas informações é exibido na Figura 17.

O diagrama de classe fornece uma visão geral do sistema e representa um artefato imprescindível no desenvolvimento.

Em paralelo, a “realização dos casos de uso” verifica os fluxos dos casos de uso, representando-os através de diagramas de interação. Estes mostram como os fluxos são realizados através de trocas de mensagens entre os objetos das classes encontradas. Isso ajuda a localizar imprecisões e outros tipos de problemas nos fluxos e permite definir as operações das classes de análise.

Por fim, a “Revisão da análise” valida o esforço de Análise e o correspondente esforço de Requisitos. Podem acontecer várias revisões informais, individuais ou em grupo (por exemplo, dentro de uma oficina de detalhamento dos requisitos).

Exercícios de fixação

1. Qual o objetivo do fluxo de análise?

2. uais recursos da UML são utilizados para descrever os requisitos em um modelo de análise?

3. O que é descrito no diagrama de classes?

4. O que é a realização dos casos de uso?

5. Que locais são consultados para se tentar identificar classes?

Fluxo de desenho

O fluxo de desenho (design ou projeto) tem por objetivo definir uma estrutura implementável para um produto de *software* que atenda aos requisitos associados. Sua principal diferença para a análise está justamente no foco: a Análise visa modelar o conceito relacionado ao domínio do problema, enquanto o Desenho visa modelar o conceito relacionado a uma solução para o problema.

Alguns processos de *software* prescrevem que a Análise e o Desenho devam ser um único fluxo, com o intuito de modelar a solução para o problema. Conforme detalhado aqui, o Processo Práxis separa bem esses dois conceitos. Nesse caso, então, o desenho de um produto de *software* deve considerar os seguintes aspectos:

1. O atendimento dos requisitos não funcionais como os requisitos de desempenho e usabilidade;

2. A definição de classes e outros elementos de modelo em nível de detalhe suficiente para a respectiva implementação;

3. A decomposição do produto em componentes cuja construção seja relativamente independente, de forma que eventualmente possa ser realizada por pessoas diferentes, possivelmente trabalhando em paralelo;

4. A definição adequada e rigorosa das interfaces entre os componentes do produto, minimizando os efeitos que problemas em cada um dos componentes possam trazer aos demais elementos;

5. A documentação das decisões de desenho, de forma que estas possam ser comunicadas e entendidas por quem vier a implementar e manter o produto;

6. A reutilização de componentes, mecanismos e outros artefatos para aumentar a produtividade e a confiabilidade;

7. O suporte a métodos e ferramentas de geração semiautomática de código.

O foco do fluxo de desenho é a concepção de estruturas robustas, que tornem a implementação mais confiável e produtiva. Tipicamente, as atividades de Desenho são realizadas por grupos bastante pequenos de profissionais experientes nas técnicas deste fluxo; a implementação correspondente pode ser delegada a profissionais não necessariamente proficientes em Desenho, mas conhecedores do ambiente de implementação e treinados nas respectivas técnicas.

A arquitetura de um produto corresponde a todas as tecnologias empregadas para sua construção, aliadas a forma de interconexão entre tais tecnologias.

Uma empresa com diversos produtos normalmente segue a arquitetura já existente para produtos novos.

Quando há a necessidade de mudarmos de tecnologia, é necessário muito estudo e testes para se obter o nível de entendimento e maturidade necessários para implementar um novo projeto.

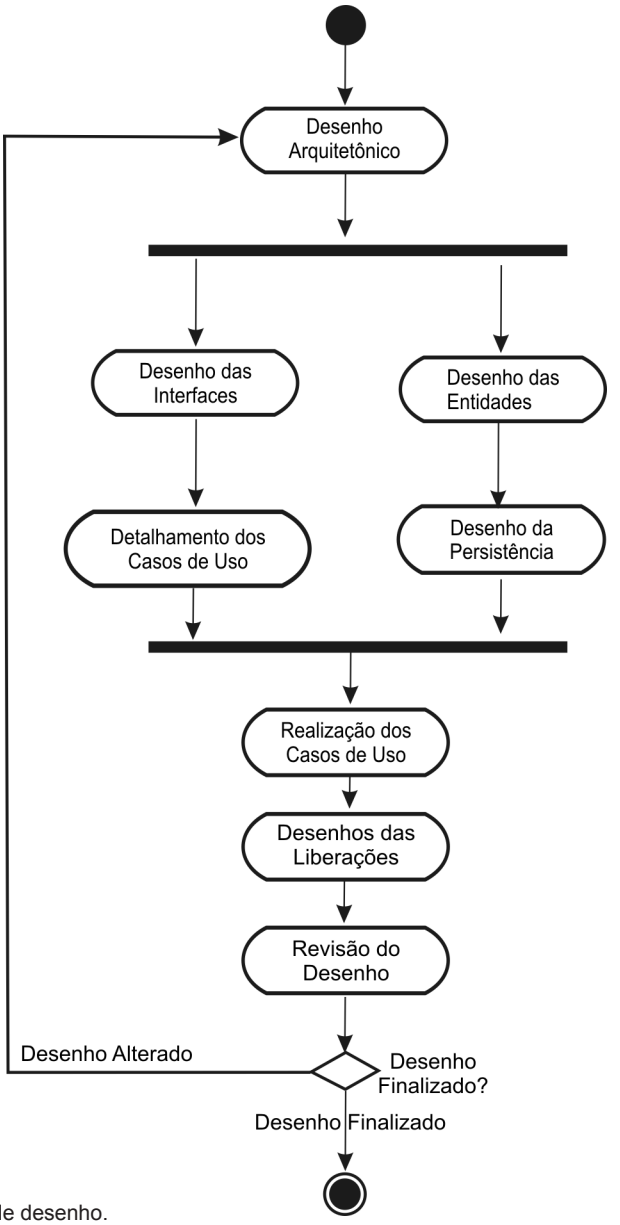

Figura 18: Fluxo de desenho.

A Figura 18 exibe as atividades do fluxo de desenho. Ele inicia pela atividade de “desenho arquitetônico”, que trata de aspectos estratégicos de desenho. Ela define a divisão do produto em subsistemas, escolhendo-se as tecnologias mais adequadas. As decisões sobre tecnologia devem viabilizar o atendimento dos requisitos não funcionais e a execução do projeto dentro do

prazo e custos estimados. Uma decisão fundamental é a escolha do ambiente definitivo de implementação, se isso já não fizer parte das restrições de desenho constantes da Especificação dos Requisitos do *Software*. Faz parte dessa atividade, também, a definição estratégica e estrutural da documentação para usuários.

Um exemplo da escolha do ambiente definitivo de implementação seria: vamos utilizar Java, na versão 1.6, com o Hibernate 2.1, utilizando o Framework Struts 2.0, que será executado no container Tomcat 5.5. Todas essas decisões devem ser registradas e comunicadas à equipe.

Um *framework* ou arcabouço é uma abstração que une códigos comuns entre vários projetos de software, provendo uma funcionalidade genérica e facilitando o seu uso.

Figura 19: Exemplo de uma tela no ambiente final de implementação

O “desenho das interfaces” trata do desenho das interfaces reais do produto em seu ambiente definitivo de implementação. Essa atividade é baseada nos requisitos de interfaces constantes da Especificação dos Requisitos do *Software*. Esses requisitos são detalhados, levando-se em conta as características do ambiente de implementação e os resultados dos estudos de usabilidade. Parte do código fonte das classes correspondentes figura como artefato resultante desta atividade, pois em muitos ambientes é mais fácil desenhar os elementos gráficos das interfaces de usuário dentro do ambiente de implementação, extraindo desse código o respectivo modelo por meio de ferramentas. Um exemplo do detalhamento da interface de usuário independente de tecnologia, exibido na Figura 15, é apresentado na Figura 19 utilizando a

tecnologia escolhida para a implementação do *software*.

O "detalhamento dos casos de uso" resolve os detalhes dos fluxos dos casos de uso de desenho, considerando os componentes reais das interfaces. Todos os fluxos alternativos devem ser detalhados, inclusive os que consistem de emissão de mensagens de exceção ou erro. Se esta atividade for feita de forma completa, pode-se deixar a cargo de profissionais de redação técnica a confecção da documentação para usuários e até das mensagens aos usuários. Os fluxos dos casos de uso de desenho são também a base para o desenho dos testes de aceitação e a referência para os implementadores quanto ao comportamento desejado para o produto.

A atividade de "Desenho das entidades" transforma as classes de entidade do Modelo de Análise nas classes correspondentes do Modelo de Desenho. Inclui-se aqui a consideração de possível reutilização dessas classes, ou de transformação delas em componentes fisicamente distintos. Muitos detalhes irrelevantes para a Análise devem ser então decididos como a visibilidade das operações e a navegabilidade dos relacionamentos. Inclui-se aqui também a tomada de decisões sobre como serão representados os relacionamentos; no caso de multiplicidades maiores que valor um, geralmente essa representação envolve o desenho de coleções de objetos.

A atividade de "desenho da persistência" trata da definição de estruturas externas de armazenamento persistente como arquivos e bancos de dados. Dois problemas devem ser resolvidos: a definição física das estruturas persistentes externas como o esquema de um banco de dados e a realização de uma ponte entre o modelo de desenho orientado a objetos e os paradigmas das estruturas de armazenamento que, geralmente, são de natureza bastante diferente. Esta ponte é feita pela camada de persistência. As classes desta camada são desenhadas nesta atividade; entretanto, como uma camada de persistência bem desenhada, é altamente reutilizável, muitas vezes a atividade se reduz a adaptações de componentes já existentes.

A "Realização dos casos de uso" determina como os objetos das classes de desenho colaborarão para realizar os casos de uso. As classes da camada de controle conterão o código que amarra essas colaborações, de maneira a obter o comportamento requerido pelos casos de uso de desenho. Diagramas de interação são usados para descrever as colaborações. Esta atividade permite validar muitas das decisões tomadas nas atividades anteriores. Classes utilitárias, agrupadas em uma camada de sistema, ajudam a abstrair aspectos

No fluxo de desenho devemos identificar os padrões a serem seguidos no projeto. Padrões são soluções bem conhecidas para problemas recorrentes. São exemplos de problemas recorrentes: como criar uma classe e garantir que haverá somente uma instância dela ativa no sistema? O padrão *Singleton* nos ensina como resolver esse problema de forma simples, sem termos que perder tempo pensando nessa solução. Nas referências citamos o livro mais famoso relacionado a padrões, escrito por quatro profissionais altamente reconhecidos no mundo e citados como sendo a Gangue dos Quatro (Gang Of Four – GoF).

Scott Ambler discute sobre camadas de persistência em um artigo referenciado no mundo inteiro. Confira na Web-bibliografia!

que sejam comuns às realizações de diferentes casos de uso.

Casos de uso e classes de desenho são repartidos entre as liberações, procurando-se mitigar primeiro os maiores riscos, obter realimentação crítica dos usuários a intervalos razoáveis, e possivelmente dividir as unidades de implementação de forma conveniente entre a força de trabalho.

Finalmente, a "revisão do desenho" valida o esforço de Desenho, confrontando-o com os resultados dos Requisitos e da Análise. Dependendo da interação, o Processo Práxis padrão prevê revisões técnicas da Descrição do Desenho ou inspeções que focalizam principalmente o Modelo de Desenho.

Exercícios de fixação

1. Qual a principal diferença entre a Análise e o Desenho?

2. Quais são os principais aspectos a considerar no Desenho?

3. Quais as características dos grupos que realizam Desenho em uma organização desenvolvedora de *software*?

4. O que vem a ser a arquitetura de um produto?

5. O que é uma camada de persistência?

6. O que é feito de adicional no desenho das entidades em relação ao que existe na Análise?

Implementação

O fluxo de Implementação detalha os componentes previstos no desenho, descrevendo todos os componentes de código fonte e código binário, em nível de linguagem de programação ou de acordo com as tecnologias escolhidas, que podem até não incluir programação diretamente. A implementação inclui as seguintes tarefas:

1. Planejamento detalhado da implementação das unidades de cada iteração;

2. Implementação das classes e outros elementos do modelo de desenho, em unidades de implementação, geralmente constituídas de arquivos de código fonte;

3. Verificação das unidades, por meio de revisões, inspeções e testes de unidade;

4. Compilação e ligação das unidades;

5. Integração das unidades entre si;

6. Integração das unidades com componentes reutilizados, adquiridos de terceiros ou reaproveitados de projetos anteriores;

7. Integração das unidades com os componentes resultantes das liberações anteriores. A integração comentada acima nada mais é que a ligação de um componente desenvolvido com outro que necessite dos seus serviços. Consideramos que durante a implementação é gerada a documentação de uso do produto. Esta documentação inclui manuais de usuários, ajudas on-line, sítios Web integrada ao produto, material de treinamento, demonstrações e outros recursos. O fluxo de Implementação contém um grupo de atividades de implementação normal e duas atividades de categoria à parte, que são as atividades de “Documentação de usuário” e “Prototipagem”.

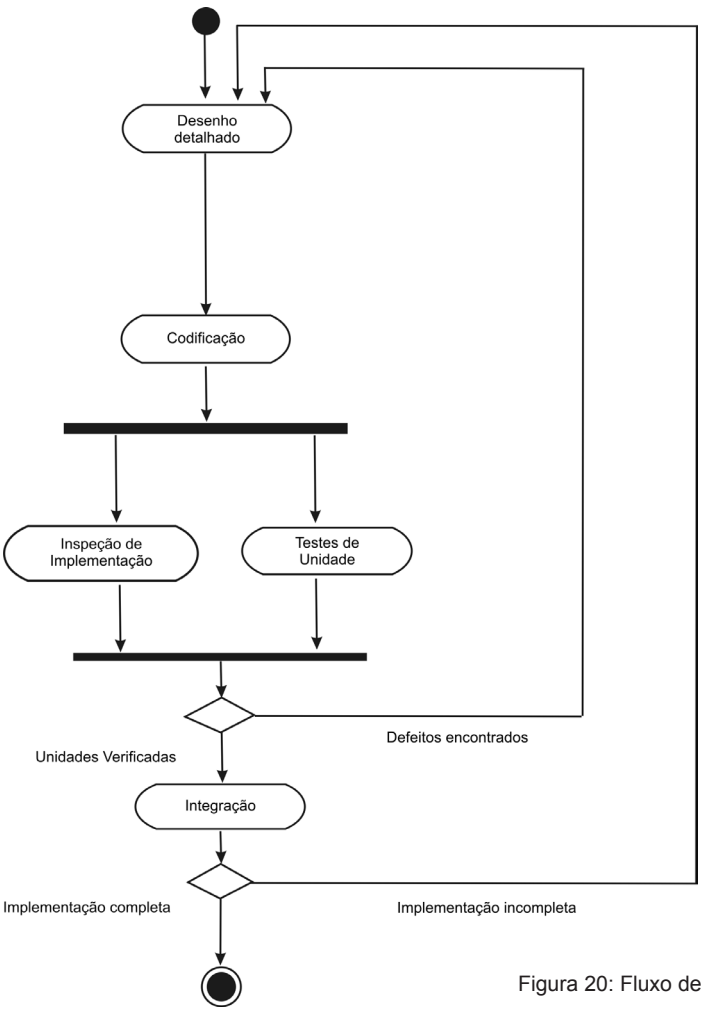

Figura 20: Fluxo de implementação.

A Figura 20 exibe as atividades do Fluxo de Implementação. Ele inicia com o “Desenho detalhado” que preenche os detalhes restantes do Modelo de Desenho em nível necessário para guiar a codificação. Nessa revisão são conferidos os detalhes de interfaces, lógica e máquinas de estado das operações.

O uso de linguagens orientadas a objetos virou um padrão mundial de programação atualmente. Linguagens como C e Java são disparadas como as linguagens mais utilizadas para o desenvolvimento de novos sistemas.

```
12 public class Compra implements Serializable {
13
14     private Long idCompra;
15
16     private Date dataCompra;
17
18     private BigDecimal valorTotal;
19
20     private int numeroNotaFiscal;
21
22     private Cliente cliente;
23
24     private Funcionario funcionario;
25
26     private Set<ItemCompra> itensCompra;
27
28     public Compra() {
29         this.itensCompra = new HashSet<ItemCompra>();
30     }
31
32     public Long getIdCompra() {
33         return idCompra;
34     }
35
36     public void setIdCompra(Long idCompra) {
37         this.idCompra = idCompra;
```

Figura 21: Exemplo de uma entidade codificada na linguagem Java.

A tradução do desenho detalhado no código de uma ou mais linguagens de programação é feita na atividade de “Codificação”. A Figura 21 exibe o exemplo de uma entidade codificada na linguagem Java. Essa codificação é normalmente feita utilizando uma IDE (Integrated Development Enviroment), que são ferramentas para desenvolvimento incluindo uma série de bibliotecas, atalhos, wizards e outras funcionalidades integradas, de forma a tornar mais fácil a vida do desenvolvedor. As unidades de código fonte produzidas são submetidas a uma revisão de código, para eliminar os defeitos que são introduzidos por erros de digitação ou de uso da linguagem. Essa revisão também deve ser feita pelos autores, possivelmente com a ajuda de outros programadores. As unidades de código fonte são transformadas em código objeto por meio da compilação.

As próximas atividades são: a “inspeção de implementação” e os “Testes de unidade”. A “Inspeção de implementação” verifica, de forma rigorosa, o desenho detalhado e o código, por meio de um grupo de revisores pares dos autores.

Os “testes de unidade” são feitos usando-se componentes de teste que devem também ter sido desenhados, revistos, codificados e compilados nas atividades anteriores. Um arcabouço para teste muito utilizado atualmente é a família xUnit. Existem arcabouços específicos para Java (JUnit), para C (CUnit), para Delphi (DUnit), dentre outros. Esses arcabouços provêm facilidades para se criar classes de teste, disponibilizando mecanismos para execução dos testes a partir de interfaces gráficas, facilidades para comparação de valores, agrupamento de testes e etc.

Um teste de unidade nada mais é que uma classe que invoca métodos de outra classe e verifica se essa ativação, com certos parâmetros, gera a resposta correta, previamente conhecida. Um exemplo de um teste de unidade é exibido na Figura 22. Essa classe é um teste para a classe triângulo, que possui um método, denominado “classifica” que retorna o tipo do triângulo. Podemos facilmente notar que em cada teste existe a criação de um objeto triângulo, com certos valores para seus vértices, seguido da chamada ao método que retorna o tipo do triângulo e a posterior comparação desse resultado com o valor esperado.

Embora bem menos formais que os testes de integração e aceitação, os resultados desses testes devem, pelo menos, ser registrados em relatórios simplificados. A ordem de execução dessas atividades é definida pela conveniência de cada projeto.

A “Integração”, finalmente, liga as unidades implementadas com os componentes construídos em liberações anteriores, reutilizados de projetos anteriores ou adquiridos comercialmente. O código executável resultante passa ao fluxo de Testes para realização dos testes de integração.

A atividade de “documentação de usuário” tem como principal objetivo produzir uma documentação que auxilie o uso do sistema por parte dos usuários. Em certos tipos de aplicativos como os sistemas baseados em Web, esta atividade não gera documentos separados e sim gabaritos para a geração de páginas que são misturadas aos relatórios produzidos, de acordo com a tecnologia empregada.

```
/**
 * Testa a classe triangulo.
 */
public class TestTriangulo extends TestCase {

  /**
   * Construtor padrao.
   */
  public TestTriangulo() {
  }

  public void testeEquilatero()
  {
    Triangulo t = new Triangulo(3,3,3);
    String result = t.classifica();
    assertEquals(result, "EQUILATERO");
  }

  public void testeIsosceles() {
    Triangulo t = new Triangulo(3, 3, 4);
    String result = t.classifica();
    assertEquals(result, "ISOSCELES");
  }

  public void testeEscaleno()
  {
    Triangulo t = new Triangulo(3,4,5);
    String result = t.classifica();
```

Figura 22: Um exemplo de teste de unidade.

A atividade de “Prototipagem” também representa uma categoria à parte. Ela executa as mesmas atividades da implementação normal, mas de maneira informal e sem maiores preocupações com padronização e documentação, pois o código resultante é sempre descartável.

Exercícios de fixação

1. Qual o objetivo do fluxo de implementação?

2. O que significa integrar no fluxo de implementação?

3. Em qual atividade é feita a programação utilizando alguma linguagem ou tecnologia?

4. O que são as IDEs?

5. O que é um teste de unidade?

6. O que é a documentação de usuário? Em qual atividade ela é desenvolvida?

Na disciplina Qualidade de Software será detalhado o fluxo de teste e realizadas diversas práticas relacionadas ao tema.

TESTES

O Teste de *Software* é a verificação dinâmica do funcionamento de um programa utilizando um conjunto finito de casos de teste, adequadamente escolhido dentro de um domínio de execução infinito, contra seu comportamento esperado. Nesse conceito existem alguns elementos chaves que merecem mais detalhamento:

1. Dinâmica: o teste exige a execução do produto, embora algumas de suas atividades possam ser realizadas antes de o produto estar em operação;

2. Conjunto finito: o teste exaustivo é geralmente impossível, mesmo para produtos simples;

3. Escolhido: é necessário selecionar testes com alta probabilidade de encontrar defeitos, preferencialmente com o mínimo de esforço;

4. Comportamento esperado: para que um teste possa ser executado, é necessário saber o comportamento esperado, para que ele possa ser comparado com o obtido.

No processo Práxis existem diversos conjuntos de teste, também conhecidos como baterias de teste. Cada um desses conjuntos está relacionado com uma determinada iteração do processo. As principais baterias de testes existentes são: testes de aceitação, testes de integração e testes de unidade.

Um caso de teste especifica como deve ser testada uma parte do *software*. Essa especificação inclui as entradas, saídas esperadas e condições sob as quais os testes devem ocorrer. Um exemplo para um caso de teste para um login inválido é apresentado na:

Um procedimento de teste detalha as ações a serem executadas em um determinado caso de teste. A Tabela 4 exibe o Procedimento de Teste de Login. Observe que um caso de teste só faz sentido se for especificado o(s) procedimento(s) de teste associado(s). Sem essa associação não é possível determinar o que deve ser feito com as entradas especificadas no caso de teste. Observe também que o procedimento de teste é independente dos dados utilizados nos casos de teste.

IDENTIFICAÇÃO	PROCEDIMENTO DE TESTE LOGIN
OBJETIVO	EFETUAR O LOGIN DE UM USUÁRIO NO MERCI.

REQUISITOS ESPECIAIS	A TELA PRINCIPAL DEVE ESTAR NO ESTADO SEM USUARIO.
FLUXO	1. PREENCHER O CAMPO LOGIN E SENHA 2. PRESSIONAR O BOTÃO LOGIN

IDENTIFICAÇÃO	Login inválido com caracteres não permitidos	
ITENS A TESTAR	Verificar se a tentativa de login utilizando um identificador inválido, contendo caracteres não permitidos, exibe a mensagem apropriada.	
ENTRADAS	Campo	Valor
	Login	Pasn!
	Senha	Aaaa
SAÍDAS ESPERADAS	Campo	Valor
	Login	Pasn!
	Senha	aaaa (exibida com *'s)
ESTADO ALCANÇADO	SEM USUÁRIO	
AMBIENTE	Banco de dados de teste.	
PROCEDIMENTOS	Procedimento de Teste Login	
DEPENDÊNCIAS	Não aplicável.	

Tabela 4: Exemplos de procedimento de teste

O fluxo de teste tem dois grandes grupos de atividades para cada bateria a ser desenvolvida: preparação e realização. O plano da bateria é elaborado durante a preparação, juntamente com o desenho das especificações de cada teste. Durante a realização, os testes são executados, os defeitos encontrados são corrigidos e os relatórios de teste são redigidos. A preparação e realização de cada bateria correspondem a uma execução completa do fluxo de Testes. As atividades do Fluxo de Teste são exibidas na Figura 23.

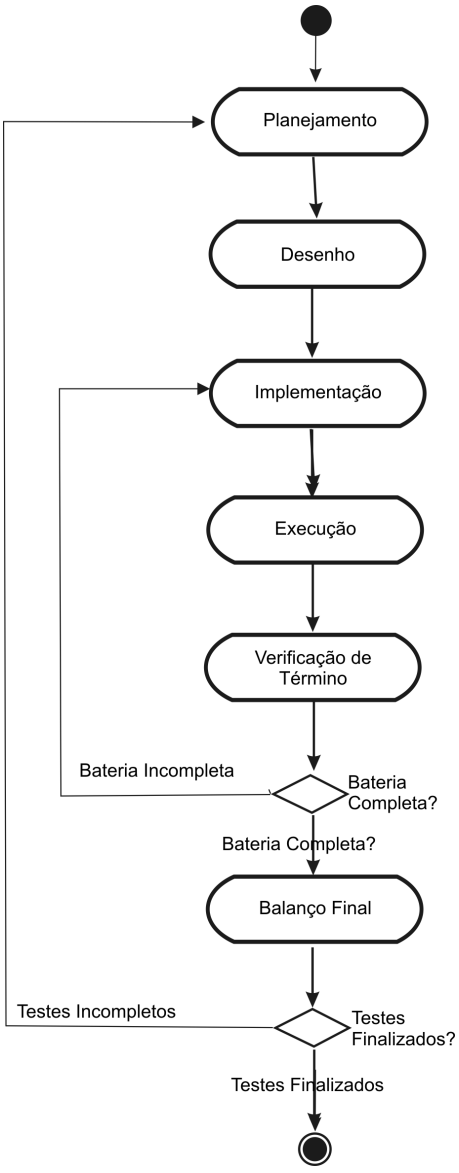

Figura 23: Fluxo de teste.

O grupo de preparação compreende as atividades de planejamento que produz o plano de testes da bateria e Desenho que produz as respectivas especificações. O grupo de realização é formado pelas demais atividades: Implementação, que monta o ambiente de testes; Execução que os executa efetivamente, produzindo os principais relatórios de testes; Verificação do término que determina se a bateria pode ser considerada como completa; e Balanço final que analisa os resultados da bateria, produzindo um relatório de resumo.

Tipicamente, uma organização começa a implementar o fluxo de teste

pelas baterias de testes de aceitação. Esse tipo de teste tem por objetivo validar o produto, ou seja, verificar se ele atende aos requisitos identificados. Na medida em que a cultura da organização absorva esses testes, passa a preparar e realizar também os testes de integração.

Na Web-bibliografia apresentamos o sítio do Brazilian Software Testing Qualification Board, que é responsável pela Certificação Brasileira de Teste de Software. Essa certificação é atribuída ao profissional que é aprovado na prova e garante, assim, que ele possui conhecimento na área. De acordo com diversas pesquisas de mercado feitas recentemente, os profissionais de teste estão entre os mais procurados e mais bem remunerados profissionais de informática do mundo.

Exercícios de fixação

1. O que é o teste de *software*?

2. O que é um procedimento de teste? O que é um caso de teste? Qual a ligação que existe entre esses dois conceitos?

3. As atividades de teste são divididas em dois grandes grupos. Quais são eles?

4. Quais são os objetivos do teste de aceitação?

Gestão de Projetos

Este tópico trata dos principais métodos de gestão de projetos de desenvolvimento de *software*. Os seguintes tópicos são brevemente apresentados aqui:

1. Os procedimentos de gestão dos requisitos, que regulam o cadastramento, a manutenção e a alteração dos requisitos de um projeto de *software*;

2. Os procedimentos de planejamento de projetos, que tratam da previsão de tamanho, esforços, recursos, prazos e riscos de um projeto de *software*;

3. Os procedimentos de controle de projetos, que tratam do acompanhamento do progresso e dos riscos de um projeto, assim como do replanejamento e do balanço final.

A Figura 24 mostra um diagrama de alto nível de abstração do fluxo de Gestão de Projetos, onde cada subfluxo aparece como uma atividade. Cada subfluxo é expandido em atividades mais detalhadas, mas que não serão apresentadas neste trabalho. Iremos detalhar parte desses subfluxos nas disciplinas subsequentes do curso.

A Gestão de Requisitos refere-se ao processo de acompanhamento dos requisitos durante todo decorrer do projeto, verificando a necessidade de alterações, analisando o impacto causado em casos como esse e calculando tempo e esforço necessários para tal realização. O Planejamento de Projetos

Na Web-bibliografia apresentamos o sítio do Project Management Institute - PMI. O PMI é uma entidade mundial sem fins lucrativos voltada à definição de boas práticas do gerenciamento de projetos.

refere-se ao processo de estimar os recursos, esforço e prazo necessários para se desenvolver um *software* ou parte dele, com base nos dados históricos mantidos pela organização.

O Controle de Projeto refere-se às tarefas de verificação comumente executadas durante o desenvolvimento e ao monitoramento dos riscos e da evolução esperada, sempre prescrevendo ações quando o estimado diferir muito do planejado.

Na próxima unidade apresentamos um exemplo de um processo de *software*, ressaltando suas atividades relacionadas à gestão do projeto. Embora sejam atividades simples, poderemos visualizar como essas atividades podem auxiliar o acompanhamento do projeto, fornecendo uma visão geral do que foi realizado e ainda existe a ser feito.

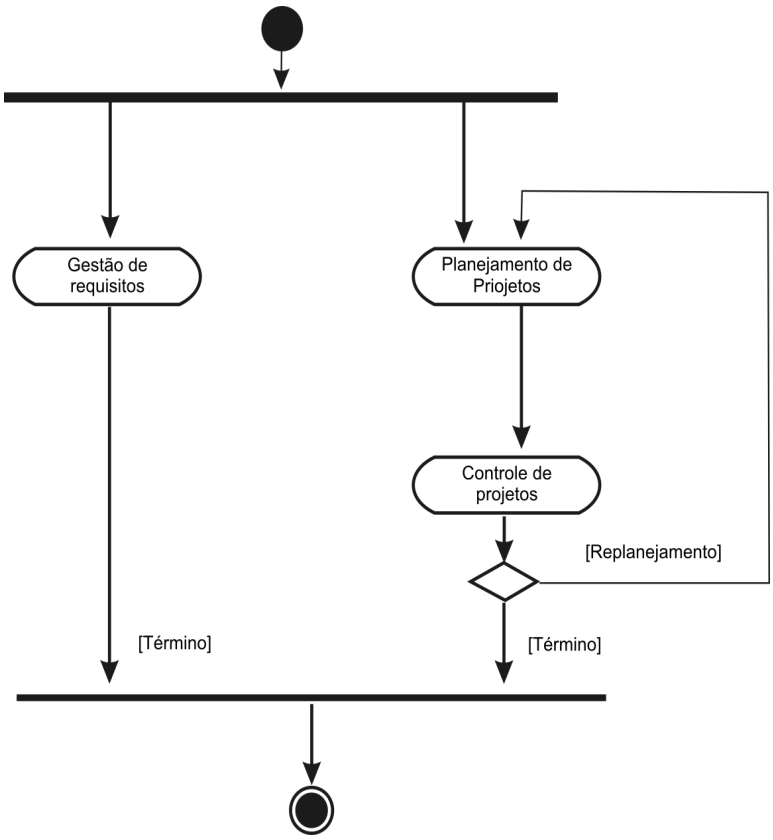

Figura 24: O Fluxo de Gestão de Projetos

Exercícios de fixação

1. O que significa Gestão de Requisitos?

2. O que é o Planejamento de Projeto?

3. Quais são as ações associadas ao Controle de Projeto?

UNIDADE 03

Exemplo de um processo de *software*

Resumindo

Nesta unidade apresentaremos um processo de *software* baseado nos princípios da agilidade, além de apresentarmos seu uso para uma tarefa simples. No capitulo 9 apresentaremos as prescrições do processo SCRUM, que vem se destacando no cenário industrial de desenvolvimento de software, sendo cada dia mais utilizado por diversas organizações. Apresentaremos os principais papéis associados ao seu uso, além das cerimônias previstas durante sua utilização, detalhando os artefatos que devem ser gerados. No capítulo 10 apresentaremos um exemplo da aplicação do SCRUM para guiar a execução de um projeto simples, ilustrando como utilizar o processo para qualquer atividade em que se desejar ter mais controle sobre sua execução.

3 EXEMPLO DE UM PROCESSO DE SOFTWARE

Scrum é um processo ágil para gerenciamento e desenvolvimento de projetos. Esse método foi primeiro descrito por Takeuchi e Nokata em um trabalho intitulado "The New *Product Development Game",* em 1986, pela Harvard Business Review, que definia um método de desenvolvimento de alto desempenho apropriado para pequenas equipes. Jeff Sutherland, John Scumniotales e Jeff McKenna conceberam, documentou e implementou o *Scrum* na empresa *Easel Corporation,* em 1993, incorporando estilos de gerenciamento observados por Takeuchi e Nonaka. Em 1995, Ken Schwaber formalizou a definição de *Scrum* e ajudou a implantá-lo no desenvolvimento de software em todo o mundo.

Scrum é uma disposição de jogadores de um time de Rugby, utilizada para reinício do jogo em certos casos.

O *Scrum* é utilizado com sucesso em diversas empresas de desenvolvimento de software. Porém, este método pode ser utilizado em qualquer contexto em que uma equipe, composta por diversas pessoas, necessite trabalhar unida para atingir um objetivo comum. Empresas como a *Microsoft, Yahoo, Google, Philips, Siemens, Nokia e várias outras usam o Scrum* como método de desenvolvimento na confecção de softwares comerciais, desenvolvimento interno, projetos de preço fixo, aplicações financeiras, videogames, websites e muitas outras aplicações.

O Scrum utiliza o modelo de ciclo de vida incremental, conforme discutido anteriormente.

Ken Schwaber define o *Scrum* como um guia geral de como fazer desenvolvimento de produtos, com a finalidade de fazer a equipe pensar diferente.

Segundo ele, o desenvolvimento de produtos é uma tarefa complexa e não deve depender de um conjunto de regras, pois elas não são suficientes para todas as circunstâncias. A equipe deve descentralizar as decisões porque, provavelmente, não haverá uma única resposta para cada tipo de questionamento.

Scrum é um método interativo e incremental baseado no modelo de ciclo de vida interativo, em que o foco principal está na entrega do produto, no menor tempo possível. Isso permite uma rápida e contínua inspeção do software

em produção. As necessidades do negócio é que determinam as prioridades do desenvolvimento do sistema. As equipes se auto-organizam para definir a melhor maneira de entregar as funcionalidades de maior prioridade. Em tempos ajustáveis é possível analisar o software em produção, decidindo se ele avança no processo ou se deve ser melhorado.

Para uma boa adoção do método *Scrum* em uma empresa é necessário que ela tenha uma cultura de fazer as coisas de modo diferente, mas que necessita também de muita disciplina, uma vez que não haverá muitas regras e nem interferência externa na execução das tarefas.

A evolução do desenvolvimento de um sistema é feita com base em uma série de períodos definidos de tempo, chamados Sprints, que variam, preferencialmente, de 2 a 4 semanas. Durante esse tempo, os requisitos são detalhados, o produto é projetado, codificado e testado. Ao final de cada Sprint é feita uma avaliação do que estava programado, assim o acompanhamento é feito em curtos espaços de tempo.

Embora os Processos Ágeis prescrevam menos burocracia no seu uso, eles exigem alguma burocracia e não segui-la significa anarquia.

O desenvolvimento é feito de forma paralela em detrimento do desenvolvimento sequencial (modelo de ciclo de vida em cascata), ou seja, as equipes fazem um pouco de cada coisa, todo o tempo.

Para a descrição do *Scrum* vamos apresentar os papéis relacionados ao seu uso; em seguida, as principais cerimônias que são prescritas pelo processo. Uma cerimônia é um ritual com formalidades a serem seguidas, lembrando sempre que o objetivo dos processos ágeis é reduzir ao máximo o trabalho burocrático. Apresentaremos juntamente com as cerimônias os principais artefatos produzidos, exibindo alguns exemplos práticos do mesmo.

Papéis

O *Scrum* possui três papéis associados à sua execução: o *Scrum* ***Master, o Product Owner*** e a ***Equipe de Desenvolvimento.***

O *Product Owner* é o membro com maior conhecimento das funcionalidades do produto, decidindo as datas de lançamento e conteúdo. Ele prioriza as funcionalidades de acordo com o valor de mercado, mas para isso deve entender o produto a ponto de priorizar sem gerar problemas. Da mesma forma, ele tem a tarefa contínua de ajustar as funcionalidades e prioridades, revendo descrições e conversando com os membros da equipe sempre que uma dúvida surgir. É fundamental para o sucesso de um projeto a presença de um *Product Owner* com domínio do problema. O ideal seria que a pessoa com tal responsabilidade fosse um representante dos clientes do produto, embora

em casos excepcionais um desenvolvedor possa assumir esse papel, devendo assim manter contato permanente com os clientes reais.

O *Scrum* Master representa a gerência para o projeto, ao mesmo tempo em que é responsável pela aplicação dos valores e práticas do *Scrum*. Ele deve manter uma vigilância contínua sobre o uso correto das prescrições do *Scrum*, garantindo assim a disciplina de todos os membros da equipe. Embora os processos ágeis tenham um mínimo de burocracia, ainda existe burocracia a ser seguida. Não segui-la pode levar ao caos e a mais completa anarquia no desenvolvimento.

O Scrum Master é o papel chave no Scrum. Ele deve garantir o uso do processo e eliminar interferências externas.

Embora o *Scrum* Master seja uma espécie de gerente do projeto, removendo todo e qualquer obstáculo que atrapalhe o trabalho da equipe e devendo ter bom relacionamento com os níveis mais altos da organização, ele também deve ter conhecimento técnico suficiente para entender todo o desenvolvimento e ajudar sempre que necessário.

A Equipe no *Scrum* normalmente é composta de 5 a 9 membros, embora o *Scrum* possa ser aplicado com mais colaboradores ou apenas um colaborador. A diferença é que aplicar o *Scrum* com um membro apenas torna sem sentido algumas das práticas previstas no processo. No entanto, ainda assim o seu uso garante um controle nas atividades sendo executadas e um conhecimento na produtividade, fato esse que só tem a acrescentar a qualquer tipo de trabalho. Os membros da equipe devem ser multifuncionais, ou seja, eles devem ser capazes de executar diversos tipos de tarefas, desde a modelagem até o teste das funcionalidades.

Embora o *Scrum* tenha sido criado para equipes pequenas, existem relatos de projetos com grandes equipes utilizando essa metodologia. Existe o conceito de *Scrum* aplicado justamente nesse caso.

Em alguns casos é possível associar colaboradores a tarefas específicas, mas isso não deve ser utilizado de modo contínuo, pois pode gerar o que chamamos de ilhas de conhecimento, onde apenas uma pessoa entende sobre determinado aspecto. Essa característica é péssima para um projeto e deve ser evitada sempre que possível, sendo uma das premissas básicas da agilidade: o código é coletivo! A equipe deve ser auto-organizada, ou seja, cada um escolhe o que quer fazer. Uma característica importante da equipe é que ela deve sentar junto e se ajudar sempre que necessário, pois isso é a premissa básica para o trabalho em conjunto.

Cerimônias

As cerimônias se referem aos rituais que devem ser seguidos no uso do processo. As principais cerimônias previstas no *Scrum* são as seguintes reuniões: Inicial, Planejamento, Apresentação, Retrospectiva e Reunião Diária.

Reunião inicial

No início do projeto é necessário fazer o levantamento de requisitos do produto a ser desenvolvido. No caso dos processos ágeis, isso é feito em uma série de reuniões, denominada aqui de Reunião Inicial. Nela é que ocorre a definição das estórias do produto. Uma estória pode ser vista como sendo a descrição do cliente sobre uma funcionalidade do produto, descrito em sua própria linguagem. Cabe à equipe de desenvolvimento entender esse linguajar e depois detalhar as atividades necessárias para implementação da estória.

O conjunto de todas as estórias identificadas nessa reunião inicial será usado para compor o Product Backlog, que equivale aos requisitos do produto, descrito na forma de estórias do usuário. A definição dessas estórias é geralmente feita com a participação de toda a equipe, juntamente com o Product Owner e, opcionalmente, outros membros representantes dos clientes. Essa reunião geralmente dura um ou mais dias, com bastante conversa sobre as estórias do produto. É fundamental ressaltar que o mais importante nesse momento não é obter um entendimento detalhado sobre o produto, mas identificar as funções existentes que devem ser tratadas a posteriori.

O levantamento de requisitos durante o uso do *Scrum* acontece durante todo o projeto, mas a identificação desses requisitos, descrevendo-os de forma sucinta, acontece na reunião inicial.

O registro dos dados coletados na reunião é um ponto crucial para o restante do processo. É necessário registrar tudo o que foi discutido, pois cada uma dessas estórias será utilizada no decorrer do projeto. Também é importante que a equipe tente entender melhor sobre o produto a ser desenvolvido, para que a reunião possa ser a mais produtiva possível. Nesse caso, a equipe deve estudar e ler materiais relacionados, analisar soluções similares pré-existentes e, acima de tudo, perguntar bastante para os representantes dos clientes.

Um formato seguido pelo autor deste material para guiar as perguntas é sempre questionar como tudo inicia. Por exemplo, no levantamento de requisitos para uma solução para uma cooperativa foi questionado: De que necessitamos para iniciar os trabalhos da cooperativa? A resposta foi: Inicialmente, precisamos cadastrar os cooperados. Nesse caso foi novamente feita outra pergunta: O que é necessário para cadastrar um cooperado? A resposta foi uma série de dados relacionada ao cooperado, incluindo informações bancárias. Nesse momento, uma nova questão foi formulada: O que é necessário para cadastrar informações bancárias? Tudo então se repete até que tenhamos chegado a um ponto em que tudo que existe de dependência tenha sido esclarecido.

Logicamente, isso dará origem a uma série de estórias que deverão ser registradas. Resolvendo todas as questões, podemos partir para o próximo passo: uma vez tendo cooperados cadastrados, o que mais é necessário? Mais uma vez, as questões seguem novamente.

Um ponto interessante em cada questão é tentar identificar quem é o responsável pelas ações. Isso normalmente está relacionado a papéis dentro do produto, os quais representam responsabilidades e não pessoas. Cada estória normalmente possui um papel associado a sua execução.

As estórias e os papéis podem ser representados utilizando diagramas de caso de uso. Essa notação será abordada com mais detalhes na disciplina Requisitos de Software. Nesse diagrama, os papéis são representados por bonecos e as funcionalidades (estórias) são representadas por elipses, chamadas de casos de uso.

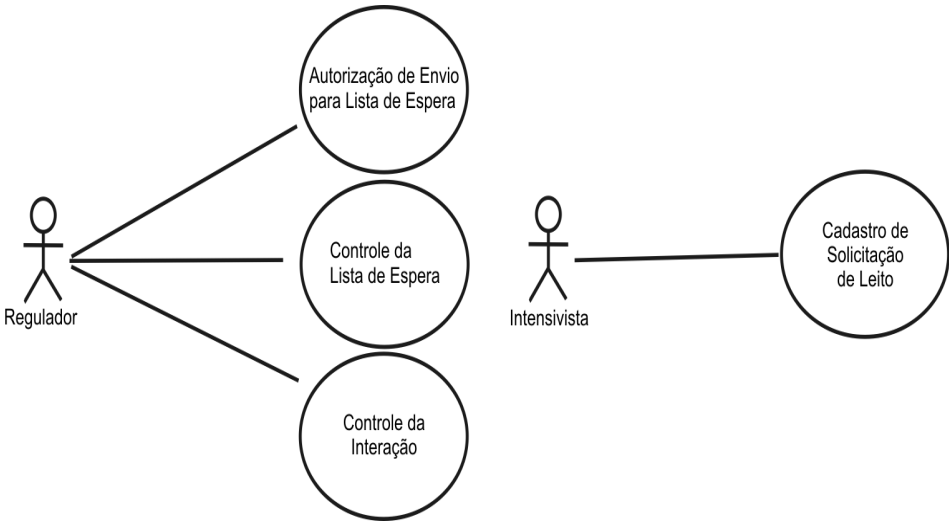

Figura 25: Exemplo de Diagrama de Caso de Uso

Diagramas de caso de uso fornecem uma boa visão dos grupos de usuários do sistema e suas funcionalidades associadas.

A Figura 25 apresenta um exemplo de um Diagrama de Caso de Uso que exibe dois papéis (regulador e intensivista) e algumas funções do produto (autorização de envio para lista de espera, controle da lista de espera, controle de internação e cadastro de solicitação de leito).

Essas funcionalidades foram descobertas a partir de conversas com os usuários do produto. Analisando o diagrama é fácil perceber quais são os papéis relacionados a certas estórias, facilitando assim o entendimento.

É necessário ainda detalhar cada caso de uso e cada um dos atores, para facilitar o entendimento do contexto por qualquer pessoa que ingresse no projeto. Um exemplo do detalhamento dos casos de uso é apresentado na Tabela 4. Um exemplo do detalhamento dos papéis (ou atores) é apresentado na Tabela 5.

Número de ordem	Casos de uso	Descrição
1.	Autorização de envio para Lista de Espera	Registro da autorização da solicitação para constar na lista de espera por leitos.
2.	Controle da Lista de Espera	Controle da lista de espera por leitos, com registro de evolução dos pacientes, além de registro de entrada ou encaminhamento em leitos hospitalares e eventual remoção da lista.
3.	Controle de Internações	Controle das internações em leitos, com possibilidade registro de alta, e consequente liberação do leito, além de transferências para outros leitos.
4.	Cadastro de Solicitação de Leito	Cadastro das solicitações de leitos para pacientes com necessidades de internação.

Tabela 4: Detalhamento dos casos de uso

Número de ordem	Ator	Definição
1.	Intensivista	Médico plantonista que coordena a UTI no hospital e faz a solicitação das vagas.
2.	Regulador	Médico regulador da Central de Leitos, que recebe, avalia e gerencia as solicitações e eventuais internações..

Tabela 5: Detalhamento dos atores

Embora o Diagrama de Caso de Uso seja importante para obtermos um bom entendimento de alto nível do produto, é necessário detalhar um pouco melhor cada estória e registrá-la em um formato adequado para manipulação. Uma planilha eletrônica é um formato que utilizamos com relativo sucesso, mas dependendo do caso, pode não ser o mais adequado. Assim, aconselhamos a utilização de uma planilha contendo todos os dados das estórias, seguindo o modelo de cartões para registro das estórias, detalhado a seguir.

O principal resultado da reunião inicial é o Product Backlog, que é o

conjunto dos requisitos do produto que retrata todo o trabalho desejado no projeto. Cada estória deve ter sua prioridade definida pelo Product Owner e repriorizado em vários momentos do processo.

Reunião de planejamento

Na Reunião de Planejamento a equipe seleciona os itens do Product Backlog com os quais se compromete a concluir dentro do prazo do Sprint. O Sprint é o tempo que a equipe planeja para produzir uma versão funcional do produto com parte dos itens do Product Backlog.

Um tamanho comum para um Sprint gira entre 2 e 4 semanas, mas isso pode ser diferente, dependendo da situação. Esse número pode ser menor, quando precisamos ter respostas mais rápidas e obter dados sobre produtividade, ou maiores, quando as coisas já estão mais estáveis e podemos nos dar ao luxo de adiar a entrega para o cliente para termos mais itens desenvolvidos.

Definido o tamanho do Sprint, é hora de calcular a força de trabalho que a equipe poderá cumprir. Uma forma simples de calcular é multiplicar a soma das quantidades de horas diárias dedicadas ao projeto por parte da equipe pela quantidade de dias úteis do Sprint.

Essa é a força de trabalho bruta. Normalmente utiliza-se um fator de ajuste para encontrarmos a força de trabalho líquida. O fator de ajuste corresponde ao percentual de tempo produtivo efetivamente utilizado para computar carga de trabalho da equipe.

Colaboradores que trabalham 8h/dia dificilmente passam 8h realizando tarefas diretamente relacionadas ao desenvolvimento. Existe tempo para ler e-mails, conversar com colegas, tomar café, ir ao banheiro, etc. Um fator de ajuste muito comum é usar 75%, mas esse número depende de várias condições que devem ser analisadas.

Um ponto importante na definição da dedicação da equipe é nunca considerar o *Scrum* Master nem o Product Owner (caso esse seja um membro da equipe agindo como um cliente) dedicados integralmente às tarefas de desenvolvimento.

Assim, um *Scrum* Master que trabalhe 8h/dia certamente poderá se dedicar apenas 4h/dia para agir como desenvolvedor. O restante das horas deve ser gasto com as tarefas de acompanhamento do processo e de auxílio à equipe.

A reunião de planejamento é a cerimônia mais importante do Scrum e deve ser realizada com muito cuidado.

É necessária muita prudência para definição da dedicação dos colaboradores ao projeto, pois isso pode gerar grandes problemas no Sprint. Um fator comum é superestimar a participação dos membros, esquecendo-se de outras tarefas existentes!

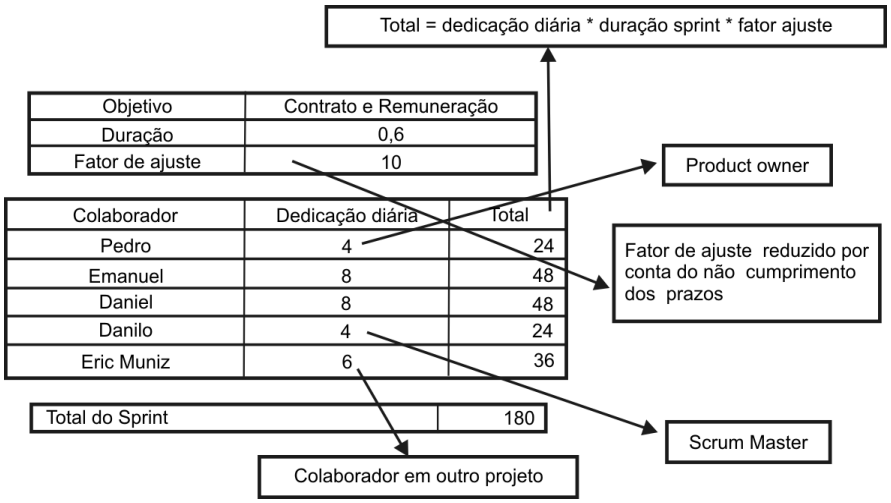

Figura 26: Cálculo da força de trabalho de uma equipe.

A Figura 26 exibe o detalhamento do cálculo da força de trabalho de uma equipe. Existem cinco colaboradores no projeto com sua dedicação diária expressa na figura. Na coluna Total é feito o cálculo das horas disponíveis durante o Sprint, levando em consideração a quantidade de dias e o fator de ajuste. O valor Total do Sprint representa a força de trabalho para essa equipe no Sprint.

Com o cálculo da força de trabalho é possível elaborar o gráfico de acompanhamento, também conhecido como Burndown. Esse gráfico nos ajuda a acompanhar visualmente o andamento do projeto, facilitando a identificação visual de atrasos ou de adiantamentos. Para sua criação, basta traçar uma reta que inicia na quantidade de horas alocadas para o Sprint e que toca em zero no último dia do Sprint. A Figura 27 exibe o gráfico para os cálculos que fizemos anteriormente. Em breve, na cerimônia denominada acompanhamento diário descreveu como o gráfico deverá ser utilizado.

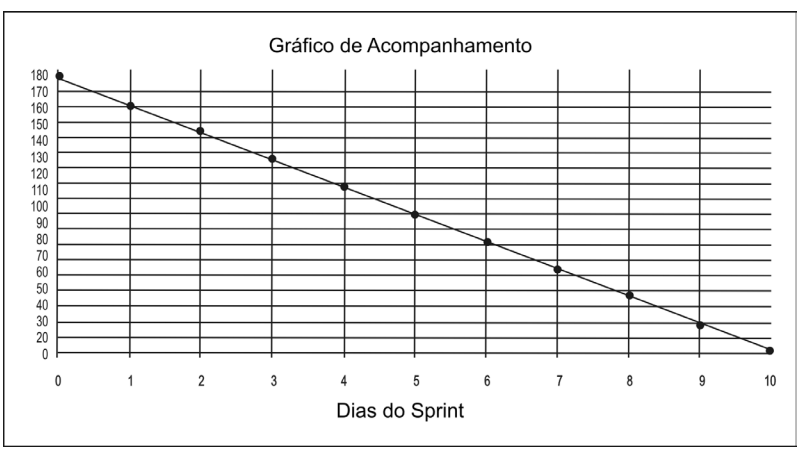

Figura 27: Gráfico de acompanhamento.

O próximo passo é selecionar as estórias que farão parte do Sprint. Elas comporão o que denominamos Sprint Backlog ou os requisitos do Sprint. Para que possamos inserir uma estória no Sprint teremos que conversar sobre ela e tentar descobrir todas as atividades associadas ao seu desenvolvimento. Observe que uma estória pode desencadear várias atividades ou apenas uma. Tudo depende da situação.

Cada estória a ser inserida no Sprint deve ser representada por um cartão, com o formato exibido na Figura 28. Ele possui diversos compartimentos, cada um com um significado específico:

1. ID: identificador da tarefa;
2. Nome: descrição resumida da tarefa para fácil identificação;
3. Descrição: texto simples e objetivo descrevendo a tarefa com mais detalhes;
4. Como verificar: indicação de como deve ser feita a verificação dessa tarefa, quando a mesma estiver concluída;
5. Tempo estimado: tempo especificado pela equipe para realização da tarefa;
6. Tempo real: tempo efetivamente utilizado para realização da tarefa;
7. Responsável: colaborador encarregado de realizar a tarefa.

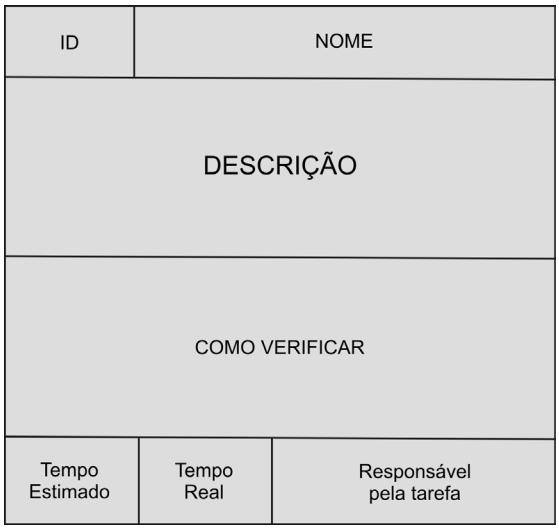

Figura 28: Exemplo de cartão descrevendo uma atividade

Para o cálculo do tempo estimado, a maioria das equipes que usa *Scrum* utiliza o Jogo do Planejamento. Nesse jogo, cada membro da equipe deve opinar com uma estimativa de tempo para uma atividade. Essa estimativa deve ser feita a partir do lançamento de cartas com o valor estimado. Isso ocorre dessa maneira para tentar não influenciar os outros membros da equipe em suas

estimativas. De preferência, as cartas devem ser atiradas viradas, de forma que apenas depois que todos as joguem é que possamos virá-las para então analisar os valores estimados.

A Figura 29 exibe algumas cartas utilizadas durante o Jogo do Planejamento. Pode-se considerar a média dos valores obtidos, o que é bastante comum, ou descartar alguns valores para esse cálculo (maior e menor, por exemplo). O mais importante é estabelecer parâmetros de aceitação, impedindo, por exemplo, que estimativas muito diferentes possam acontecer. Por conta disso, existem algumas definições para as cartas que merecem atenção:

É importante que todos façam suas estimativas sem conhecer as estimativas dos outros membros do grupo para evitar interferências nas opiniões. Mas é fundamental conversar quando grandes diferenças são registradas!

1. Um "?" indica que mais explicações são necessárias;

2. Um "0" indica algo tão pequeno que não merece estimar;

3. Uma xícara indica necessidade de tempo para pensar e hora para o café...

Tarefas acima de 16h merecem ser divididas, pois caso contrário podem gerar problemas no acompanhamento, por serem grandes demais e necessitarem de vários dias para se descobrir que ela está atrasada! Estimativas muito diferentes merecem explicações

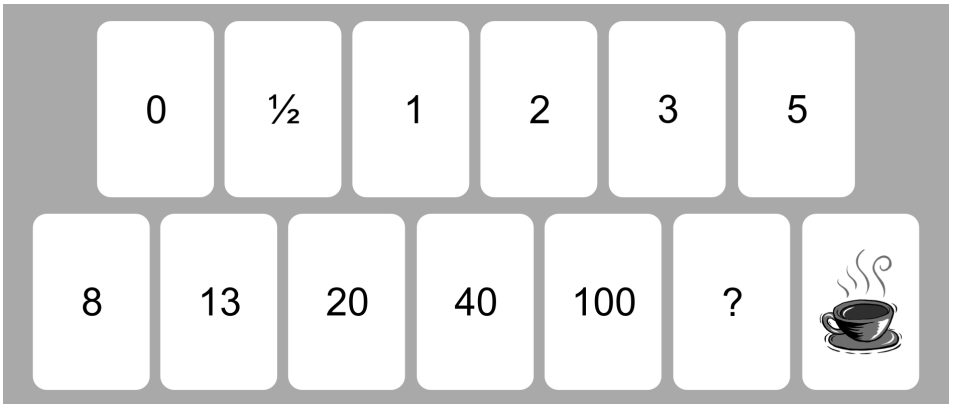

Figura 29: Exemplos de cartas para o Jogo do Planejamento

Assim, para que possamos descobrir que itens devem entrar em um *Sprint,* devemos analisar as prioridades dos itens no *Product Backlog* e selecionar aquele com maior prioridade. Em seguida, devemos conversar bastante sobre a estória, a ponto de descobrirmos as tarefas associadas, para então iniciar a estimativa das tarefas. Cada vez que uma tarefa ingressar no *Sprint,* devemos calcular quanto temos ainda temos, que horas disponíveis, para saber se mais tarefas podem ser introduzidas no *Sprint.*

O tempo real, presente nos cartões, deve ser preenchido com o tempo realmente gasto para a conclusão da tarefa, independente do tempo estimado. A

partir do confrontamento entre o tempo estimado e o tempo real é que poderemos apurar nossas estimativas e assim chegarmos a um ponto onde o erro possa ser considerado desprezível.

A princípio, os responsáveis pelas tarefas não devem ser definidos: cada um escolhe, dentre as tarefas do Sprint, aquilo que for mais interessante em seu ponto de vista, sempre respeitando as prioridades envolvidas. Embora o campo Como verificar pareça "dispensável", quando existem muitas tarefas a verificar, podemos perder o foco no que é importante. Esse campo ajuda a sempre mantermos o foco em como a tarefa deve ser verificada, garantindo assim que ela realmente tenha sido concretizada.

Outro item importante a ser verificado durante o planejamento é a Definição de Feito (DoD). Todo projeto deve ter uma definição clara sobre o que é considerado feito! Para uma tarefa de modelagem, por exemplo, poderíamos considerar uma tarefa efetivamente concluída apenas se existisse:

1. Diagrama representando a modelagem feita na ferramenta de modelagem definida no projeto;

2. Diagrama contido em um repositório do projeto;

3. Revisão em conjunto com o *Scrum* Master.

Sem a definição de feito clara e precisa é comum que as tarefas entrem e saiam da coluna Concluída, retornando para em Execução, por diversas vezes, sem que estejam efetivamente terminadas.

Outro ponto prescrito pelo *Scrum* são as reuniões diárias, que abordaremos a seguir. Durante o planejamento é importante a definição do local e horário dessas reuniões.

Por fim, um quadro com as informações do Sprint precisa ser criado. Um exemplo desse quadro é exibido na Figura 30. O quadro deve ter as seguintes divisões para tarefas: Não iniciadas, Em execução, Concluídas e Verificadas. Tarefas Concluídas são aquelas em que seus executores acreditam ter finalizado todo o trabalho associado.

No entanto, elas podem não estar totalmente finalizadas, por conta do esquecimento de algum item a ser feito, e por conta disso não recebem o status de Verificadas, devendo retornar para em Execução. As possíveis mudanças de estado permitidas são exibidas no diagrama de estados contido na Figura 31. Nessa figura podemos notar que uma tarefa em execução pode ser considerada concluída pelo seu responsável, mas ela pode retornar para em execução, caso o *Scrum* Master não a considere concluída durante a verificação que sempre é exigida nesses casos.

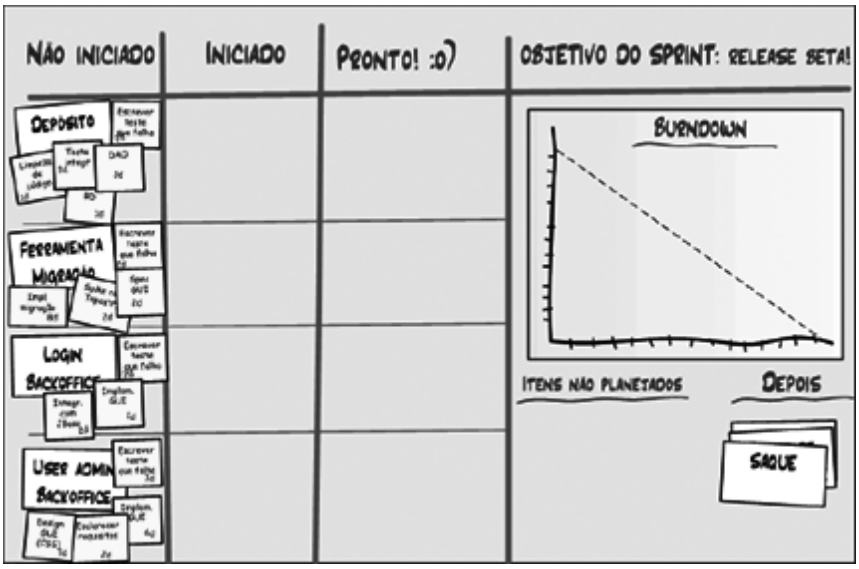

Figura 30: Exemplo da organização do quadro do Sprint.

É importante manter o quadro do projeto com todas as definições registradas aqui!

De um modo geral, o quadro do Sprint é algo fundamental para o processo, devendo conter todos os itens que facilitem a comunicação com a equipe e o restante da organização, sendo elas:

1. Área para Objetivo do Sprint;
2. Área para o Gráfico de Acompanhamento;
3. Área para especificar a equipe;
4. Área para detalhar a data de início e fim do Sprint;
5. Área para especificar dia da apresentação dos resultados;
6. Área para especificar local e horário das reuniões diárias;
7. Área para especificar o dia da retrospectiva do Sprint.

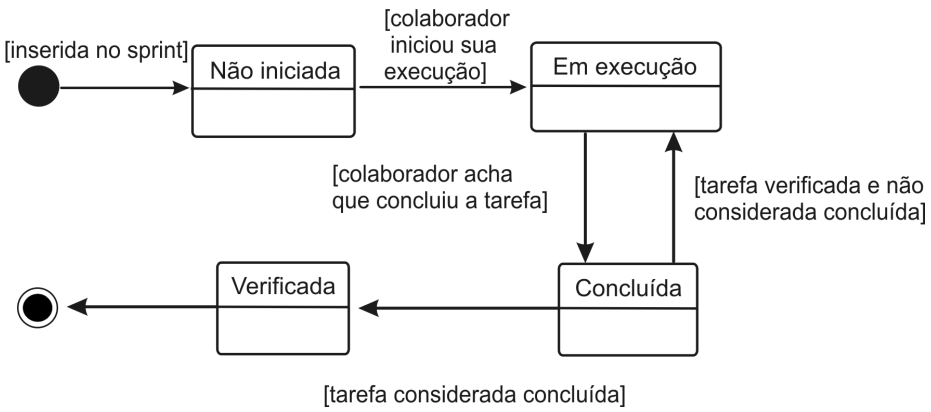

Figura 31: Mudanças permitidas no quadro do Sprint.

Reunião diária

A Reunião Diária é uma prática feita com toda a Equipe, o *Scrum Master* e o *Product Owner,* com duração de no máximo 15 minutos, apenas para uma avaliação das ações do dia anterior e o planejamento do dia. O objetivo principal da reunião consiste em cada membro responder às seguintes perguntas: O que você fez de ontem para hoje? O que fará de hoje até amanhã? Existe algum impedimento?

Na reunião não se resolvem problemas: eles devem apenas ser identificados e os envolvidos em sua resolução devem se reunir posteriormente para tentar chegar a uma possível resolução.

Os impedimentos identificados na reunião devem ser registrados, assim como os itens não planejados. Todos esses elementos precisam ser registrados para acompanhamento. É importante garantir que todos falem, sob a supervisão do *Scrum* Master.

A reunião diária tem que ser rápida, caso contrário, perderá sua efetividade.

Acompanhamento do projeto

O *Scrum* Master possui diversas tarefas no controle diário do projeto, além da reunião diária:

1. Verificar tarefas concluídas;
2. Atualizar gráfico de acompanhamento;
3. Resolver impedimentos;
4. Atuar quando houver desvio no planejamento.

Para a verificação das tarefas concluídas, normalmente é aconselhado a presença do *Scrum* Master e do Product Owner. O que deve ser feito é a utilização da definição de "como verificar" existente no cartão para analisar se realmente o item está concluído. Se algo discordar do que foi acertado para o projeto, isso deve ser indicado ao responsável para correção e efetiva conclusão do item. Conforme será visto na disciplina de Qualidade de Software, é importante ter muito cuidado com os testes para as tarefas que envolvem implementação. Também é necessário incluir a análise do código fonte para evitar surpresas futuras.

A atualização do gráfico de acompanhamento deve seguir a seguinte diretriz: para cada tarefa concluída e verificada, baixar o número correspondente à estimativa da tarefa no gráfico. Muito importante: baixar a estimativa e não o tempo efetivamente gasto! Essa ação provê uma informação visual de fácil interpretação e de grande valia para qualquer gerente de projeto. Talvez seja aquilo que é mais chamativo no *Scrum*.

Para exemplificar, se existe uma tarefa estimada em 8h dentro do backlog do Sprint, mesmo que ela tenha sido feita em 12h, devemos baixar no gráfico de acompanhamento apenas 8h quando ela tiver sido efetivamente concluída e verificada.

Sempre que houver uma diferença muito grande no previsto e no realizado, o *Scrum* Master deve tentar identificar as causas e, se possível, atuar para tentar resolver o problema associado. Da mesma forma, toda vez que um impedimento for registrado, é necessário procurar mecanismo para resolvê-lo, caso contrário, todo o projeto poderá ser comprometido.

Apresentação do Sprint

Ao final de um Sprint, o *Scrum* recomenda que seja feita a apresentação dos resultados. Essa apresentação tipicamente se resume à demonstração de novas funcionalidades do produto ou da sua arquitetura nos casos de implementação. Ela é informal, sem a necessidade de muita preparação e sem slides. Toda a equipe deve participar, assim como outros participantes. Essa reunião é uma forma de forçar o comprometimento de todos com o trabalho realizado e uma forma de estimular a obtenção de resultados efetivos, uma vez que tudo o que foi feito deverá ser apresentado.

Retrospectiva

Durante um *Sprint,* é necessário observar o que funciona e o que não funciona. Na retrospectiva, tudo o que aconteceu e merece algum registro deve ser discutido por todos para que possamos ter melhores resultados em *Sprints* futuros, levando em consideração as experiências passadas. Essa reunião tipicamente é rápida, durando cerca de uma hora, ao final de cada *Sprint,* com todos os envolvidos no projeto. A equipe tenta responder a três questões nessa reunião:

1. O que você achou melhor no Sprint e o que deveria continuar sendo feito?
2. O que você achou ruim e não deveria mais ser feito (ou mudado)?
3. O que não está sendo feito, mas deveria iniciar?

Um formato indicado para conduzir tal reunião é solicitar a cada participante que responda às questões, respondendo com até três alternativas cada questão, com cada uma das respostas em um cartão diferente. Nesse caso, se houver três respostas para cada pergunta, deverá haver nove cartões para um participante. Em seguida, para cada questão devem ser agrupadas as respostas similares, de forma a identificarmos as respostas mais comuns. Com

base nessa identificação é que registraremos o que foi mais consensual para cada uma das três questões. O *Scrum* Master deve ficar atento a esses itens e garantir que eles serão levados em consideração no próximo *Sprint.*

De forma geral, as atividades para se utilizar o *Scrum* em um projeto são as seguintes:

1. Obter a lista do que fazer (product backlog);
2. Priorizar os elementos da lista;
3. Definir o tamanho do *Sprint;*
4. Calcular a força de trabalho para o *Sprint;*
5. Criar o gráfico de acompanhamento;
6. Estimar o tamanho das tarefas a comporem o *Sprint;*
7. Registrar as tarefas utilizando os cartões;
8. Organizar o quadro do projeto, contendo todos os dados identificados neste capítulo;
9. Acompanhar o decorrer do projeto, realizando reuniões diárias, atualizando o gráfico de acompanhamento e verificando as tarefas terminadas, confrontando com a definição de feito e a forma descrita sobre como verificar a tarefa;
10. Realizar uma apresentação dos resultados do *Sprint;*
11. Realizar uma retrospectiva do *Sprint.*

1. Qual é a origem do *Scrum?*
2. Quais são os papéis relacionados ao uso do *Scrum?* Detalhe as responsabilidades de cada um desses papéis dentro do processo.
3. Qual o objetivo da Reunião Inicial no *Scrum*? Qual o principal artefato gerado?
4. O que é um *Sprint?*
5. Como podemos definir a força de trabalho para um *Sprint* no *Scrum?* Que variáveis podem influenciar nesse cálculo?
6. Como sabemos quantas estórias (ou tarefas) podem ser incluídas em um Sprint?
7. Quais informações precisam estar associadas a uma estória (ou tarefa) registrada em um *v*
8. O *Scrum* prescreve a organização de um *Sprint* utilizando um quadro. O que deve ter nesse quadro?
9. Como é criado o gráfico de acompanhamento e como devemos atualizá-lo diariamente?

10. Quais são as divisões do quadro de tarefas e quais as possíveis transições entre essas divisões?

11. Quais são as perguntas básicas durante as reuniões diárias?

12. Quais são as tarefas rotineiras do *Scrum* Master durante o dia a dia dos projetos?

13. O que deve ser feito na reunião de retrospectiva do *Sprint?* Como isso pode ser guiado?

Aplicando o *Scrum*

Neste capítulo apresentamos um exemplo do uso do *Scrum* para o desenvolvimento de um livro sobre Introdução à Engenharia de Software. Como o exemplo não está relacionado ao desenvolvimento de um software e não envolve uma equipe, mas envolve um único executor das tarefas, certamente não poderemos utilizar todas as prescrições do *Scrum,* uma vez que boa parte delas não faz sentido quando o desenvolvedor é o único membro atuante no projeto. No entanto, esse exemplo servirá para mostrar a dinâmica maior do processo e o controle que ele pode nos dar durante o desenvolvimento de qualquer tarefa.

Qualquer trabalho pode ser controlado com auxílio do *Scrum.* Várias equipes de empresas que não trabalham com desenvolvimento de software utilizam Scrum para controlar suas atividades.

Exemplo

Conforme mencionado anteriormente, utilizaremos como exemplo o desenvolvimento de um livro introdutório sobre Engenharia de Software.

Depois de alguns momentos de reflexão fizemos a construção do Backlog desse projeto, que seria composto basicamente pelas seguintes tarefas exibidas na tabela 6. Nela, as prioridades definidas para as atividades já se encontram especificadas.

ID	NOME	PRIORIDADE
1	Estudar conceitos básicos, processos e ciclo de vida	80
2	Escrever capítulo introdutório	60
3	Escrever capítulo sobre processos	60
4	Estudar as disciplinas da Engenharia de Software	50
5	Escrever capítulos sobre as principais disciplinas	40
6	Estudar a metodologia *Scrum*	30
7	Escrever capítulo sobre o *Scrum*	20

Tabela 8 : Product Backlog para o exemplo considerado

Objetivo	Introdução
Duração	10
Fator de ajuste	0,75

Colaborador	Dedicação diária	Total
Pedro	1,2	9

Total do Sprint	9

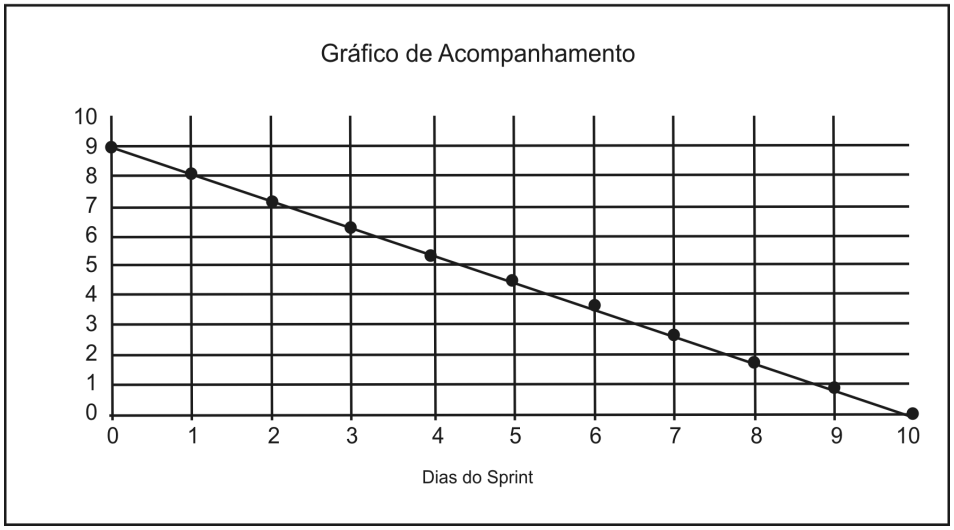

Figura 32: Dados do Sprint e Gráfico de Acompanhamento

Iremos utilizar Sprints de duas semanas (dez dias úteis) para acompanhamento das tarefas. Com essa definição já podemos criar nosso gráfico de acompanhamento, conforme exibido na Figura 32. Podemos notar na Figura a definição da carga de trabalho diária, definida como 1,2h, além do tamanho do Sprint (10 dias) e o fator de ajuste considerado (75%). Com isso chegou-se a uma carga de trabalho do Sprint de 9h.

Antes de começar as tarefas é importante definir como verificá-las para que possamos saber exatamente quando poderemos considerar algo concluído, uma vez que não teremos a ajuda de outras pessoas nessa empreitada por se tratar de um trabalho individual. Isso tem a ver não somente com a definição dos cartões que precisam ter essa informação, como está relacionado também à definição de feito associado ao projeto. Um exemplo para a definição de feito associada às tarefas existentes no Backlog poderia ser:

1. Para tarefas relativas a estudos: breve resumo do item estudado, com detalhamento das referências utilizadas;

2. Para tarefas relativas à escrita de texto: texto escrito, formatado conforme exigido, existência de bibliografia e exercícios relacionados.

O próximo passo é a definição da estimativa de tempo para a realização das tarefas. Normalmente encontramos um erro grande nas tarefas iniciais de um projeto, mas, à medida que o tempo prossegue, conseguimos cada vez mais nos aproximarmos do tempo efetivamente gasto. Fizemos a estimativa de todas as tarefas no intuito de agilizar o processo. Observe que essas estimativas podem ser alteradas, desde que a tarefa com a estimativa a ser alterada não se encontre no Sprint em execução. As tarefas e suas estimativas são apresentadas na Tabela 7.

ID	NOME	PRIORIDADE	TEMPO ESTIMADO
1	Estudar conceitos básicos, processos e ciclo de vida	80	4
2	Escrever capítulo introdutório	60	4
3	Escrever capítulo sobre processos	60	6
4	Estudar as disciplinas da Engenharia de Software	50	8
5	Escrever capítulos sobre as principais disciplinas	40	4
6	Estudar a metodologia *Scrum*	30	4
7	Escrever capítulo sobre o *Scrum*	20	4

Tabela 7: Tarefas com estimativa de execução

Com base em nossa carga de trabalho de apenas 9h e analisando o *Backlog,* sempre atento às prioridades existentes e estimativas definidas, podemos notar que apenas as tarefas 1 e 2 podem ser incluídas no Sprint inicial, ainda assim deixando uma lacuna de 1h em aberto. Resolvemos então iniciar o Sprint com apenas essas duas tarefas.

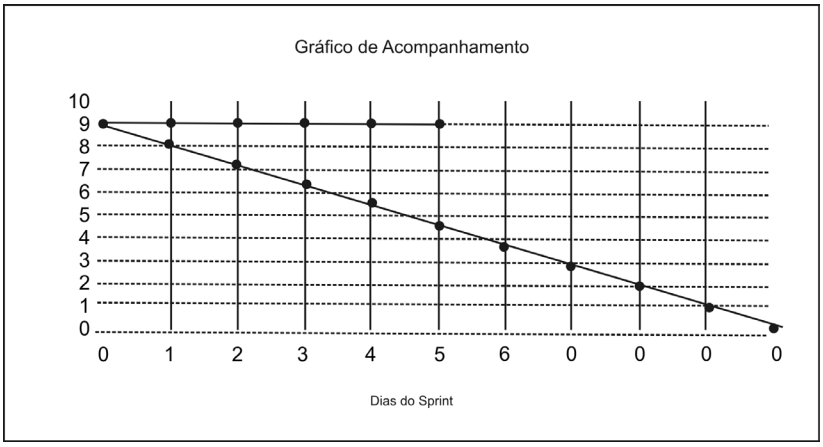

Figura 33: Gráfico de acompanhamento após a primeira semana de trabalho

Na primeira semana de trabalho não conseguimos finalizar nenhuma das tarefas, o que nos gerou um gráfico de acompanhamento conforme exibido na Figura 33. Observe que analisando o gráfico podemos facilmente perceber que estamos atrasados em relação ao que foi previsto nas estimativas de execução das tarefas. A linha de acompanhamento acima da ideal indica atraso, assim como a linha abaixo indica um adiantamento em relação ao previsto.

Ao final de duas semanas e finalizado o Sprint, obtivemos o gráfico exibido na Figura 34. Observe que apenas uma tarefa foi finalizada. Por conta disso, ao final do Sprint o gráfico de acompanhamento indica que produzimos bem menos que o estimado.

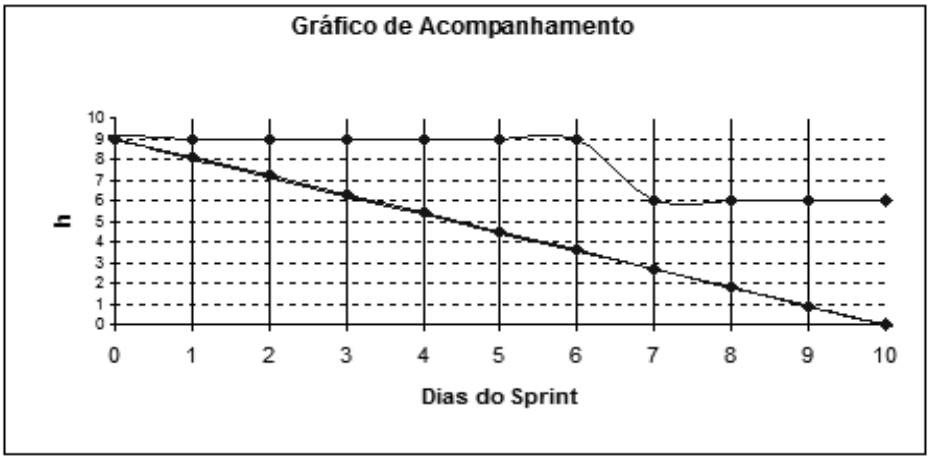

Figura 34: Gráfico de acompanhamento ao final do Sprint

A tarefa foi finalizada com apenas 3h de esforço de acordo com os registros associados à sua execução, exibidos na tabela 8. Fizemos uma breve retrospectiva sobre o que aconteceu e facilmente identificamos que nossa estimativa não foi o problema principal: na verdade, o desenvolvedor não conseguiu se dedicar conforme havia sido planejado.

ID	NOME	PRIORIDADE	TEMPO ESTIMADO	TEMPO GASTO
1	Estudar conceitos básicos, processos e ciclo de vida	80	4	3
2	Escrever capítulo introdutório	60	4	

Tabela 8: Registro dos tempos gastos nas tarefas

Com base nos dados obtidos na retrospectiva, teremos que realizar um novo planejamento para um novo Sprint e continuar com a execução das atividades até que tenhamos finalizado todo o trabalho.

Discussão sobre o uso do *Scrum*

Embora o exemplo apresentado anteriormente seja relativamente simples e não ligado diretamente ao desenvolvimento de software, ele ilustra parte do uso do *Scrum* no controle de uma atividade. É importante ressaltar que o *Scrum* é utilizado hoje nas mais variadas áreas de aplicação, tendo rompido a barreira dos núcleos de tecnologia e se espalhado nos outros setores das empresas há algum tempo.

O gráfico de acompanhamento é uma ferramenta fundamental para qualquer gerente de projeto, pois informa, de maneira simples e direta, como está o andamento das tarefas planejadas para um período de tempo.

Por tudo isso é que o *Scrum* ganha muito destaque no cenário nacional e representa como a Engenharia de Software pode ajudar no desenvolvimento de software de qualidade, além de auxiliar a organização de qualquer trabalho baseado em tarefas a serem executadas.

Exercícios de fixação

1. Utilize o *Scrum* para guiar um projeto bem simples: a leitura de um livro. Esse livro poderá ser livremente escolhido por você. Leia alguns capítulos, inicialmente, apenas para ter noção da sua velocidade de leitura de páginas. Em cima disso, planeje as tarefas (ler os capítulos 1, 2, 3, 4,...), calcule sua força de trabalho para o Sprint, defina o tamanho do Sprint, acompanhe sua execução, sempre de olho no gráfico, conforme exemplo exibido neste capítulo. Faça um breve relato sobre o uso do processo para controlar essa atividade.

WEB-BIBLIOGRAFIA

Manifesto Ágil disponível em: <http://agilemanifesto.org/>

Sítio de apoio ao livro de Wilson de Pádua disponível em: <http://homepages.dcc.ufmg.br/~wilson/>

<http://www.ambysoft.com/>

<http://www.bstqb.org.br/>

<http://www.comp.lancs.ac.uk/computing/resources/IanS/>

<http://www.controlchaos.com/>

<http://www.inf.pucrs.br/~rafael/*Scrum*ming/>

<http://www.infoq.com/br/>

<http://www.mountaingoatsoftware.com/>

STORM (Software Testing Online Resources). Sítio com diversos materiais sobre testes, desde ferramentas, artigos, livros, etc. Disponível em:

Vídeos e tutoriais sobre o *Scrum* disponível em:<http://www.mtsu.edu/~storm/>

Project Management Institute (PMI) disponível em: <http://www.pmi.org/>

<http://www.rspa.com/>

Sítio de apoio ao processo Práxis, com exemplos da aplicação dos fluxos em um exemplo real de desenvolvimento disponível em: <http://www.*Scrum*alliance.org/ >

<http://www.sei.cmu.edu/>

Relatórios do Standish Group, incluindo dados sobre desenvolvimento de software disponível em: <http://www.standishgroup.com>

Página da Universidade Aberta do Brasil- UAB disponível em:

<http://www.uab.gov.br>

Página da Universidade Aberta do Piauí – UAPI disponível em: <http://www.ufpi.br/uapi>

Certificação Brasileira de Teste de Software disponível em:

Empresa com diversos artigos e casos de sucesso no uso de práticas ágeis disponível em:

Instituto de Engenharia de Software do MIT disponível em:

Organização divulgadora do *Scrum* mundialmente disponível em:

Página de Scott Ambler com boas práticas para o desenvolvimento de software disponível em:

Sítio com um livro disponível para download em:

Sítio de apoio ao livro de Ian Sommerville disponível em:

Sítio de apoio ao livro de Roger Pressman disponível em:

Srcumming - Ferramenta Educacional para Apoio ao Ensino de Práticas de *Scrum* disponível em:

Referências

ALISTAIR, Cockburn. **Escrevendo Casos de Uso Eficazes.** Editora Bookman, 2003.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, ISO/IEC 12207 – Tecnologia da Informação – **Processos de ciclo de vida de software.** Rio de Janeiro: ABNT, 1996.

BERTRAND, Meyer. ***Object-oriented Software Construction.*** Prentice-Hall. 2ª edição, 1997.

EDSGER, W. Dijkstra. The humble programmer, ***Communications of ACM,*** Volume 15, número 10, páginas 859-866, outubro de 1972.

FILHO, Wilson de Pádua Paula. **Engenharia de Software:** Fundamentos, Métodos e Padrões. Editora LTC. 2ª Edição, 2003.

FREDERICK, P. Brooks. No silver bullet: essence and accidents of software engineering. ***Computer Magazine,*** Abril de 1987.

GLENFONRD, Myers. ***The art of the software testing.*** John Wiley & Sons, 1979.

GRADY, Booch; JAMES, Rumbaugh; IVAR, Jacobson. ***UML: Guia do Usuário.*** Tradução da 2ª edição. Editora Campus, 2000.

HENRIK, Kniberg H. *Scrum* ***e XP das Trincheiras:*** Como fazermos *Scrum*.

InfoQ, 2007.

HIROTAKA, TAKEUCHI; IKUJIRO, NONAKA. ***The New Product Development Game.*** Harvard Business Review, 1986.

IAN, Sommerville. ***Engenharia de Software.*** 6ª. Edição. Addison-Wesley, 2005.

IEEE. ***Guide to the Software Engineering Body of Knowledge (SWEBOK).*** Editores: Alain Abran e James W. Moore, 2004.

KEN SCHWABER. ***Agile Project Management with*** *Scrum.* Microsoft Press, 2004.

_____. *Scrum* **Development Process,** Burlington, MA. USA, 1996.

KENDALL, Scott. **Processo Unificado Explicado.** Editora Bookman, 2003.

KENT, Beck. ***Extreme Programming Explained:*** embrace change. Boston: Addison–Wesley/Longman, 1999.

RICHARD, Fairley. ***Software Engineering Concepts.*** New York: McGraw-Hill, 1985.

ROGER, S. Pressman. ***Engenharia de Software.*** 5ª. Edição, São Paulo: McGraw-Hill, 2002.

STANDISH, The Standish Group International; CHAOS Demographics and Project Resolution, Dennis, MA, USA, 2004.

STEVE, Macconnell. ***Rapid Development.*** Microsoft Press, 1996.

TELES, Vinicius Manhães. ***Extremme Programming.*** Editora Novatec, 2004.

W. WAYT, Gibbs. Software's chronic crisis, ***Scientific American.*** Setembro de 1994.

Minicurrículo

Pedro de Alcântara dos Santos Neto

Possui graduação em Bacharelado em Ciência da Computação pela Universidade Federal do Piauí (1995), mestrado em Ciência da Computação pela Universidade Federal de Pernambuco (1999) e doutorado em Ciência da Computação pela Universidade Federal de Minas Gerais (2006). Atualmente é professor da Universidade Federal do Piauí. Tem experiência na área de Engenharia de Software, atuando principalmente na automação de testes, desenvolvimento de software, utilizando metodologias ágeis e engenharia de software experimental.

Ministério
da Educação

www.uapi.ufpi.br