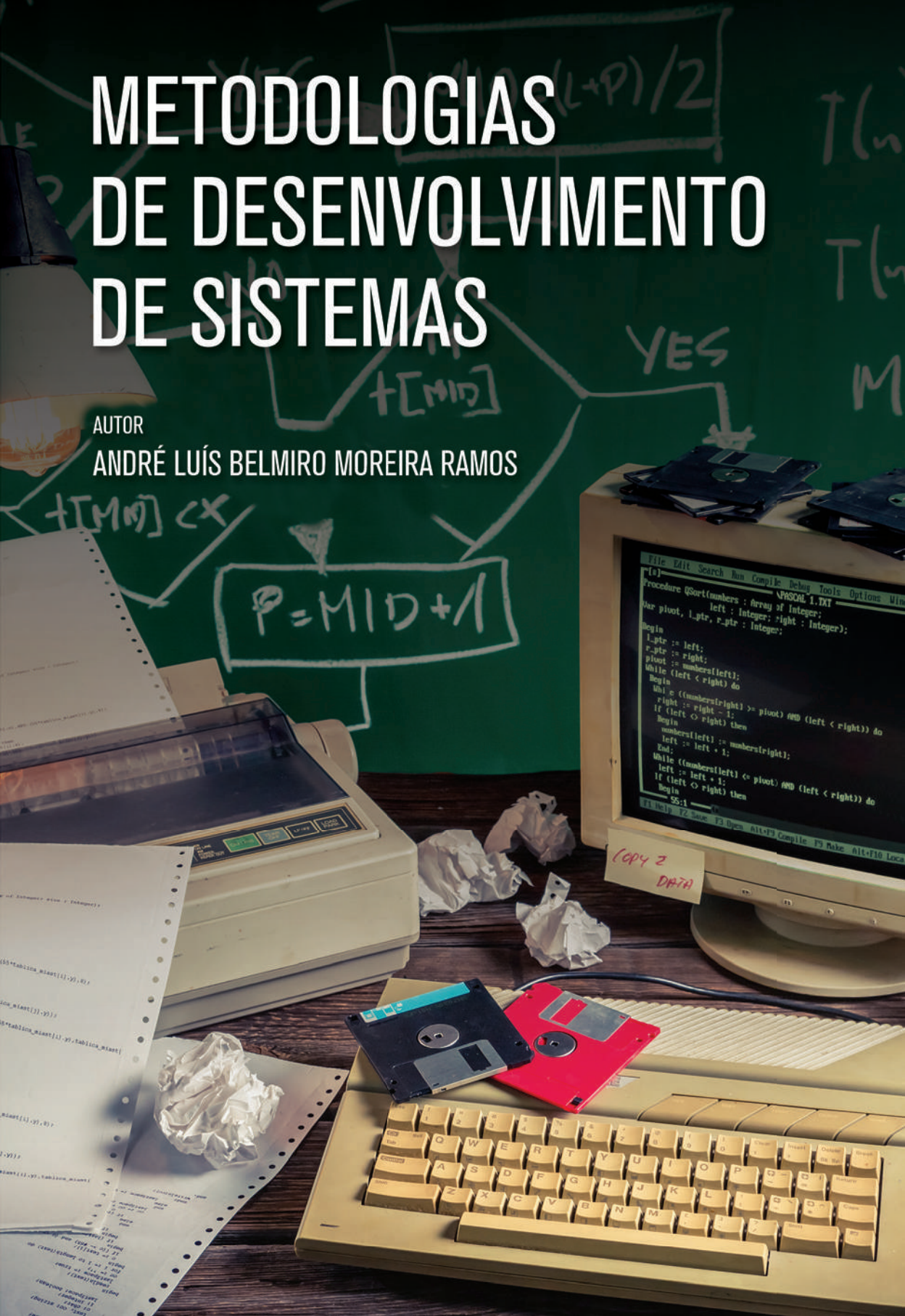
METODOLOGIAS DE DESENVOLVIMENTO DE SISTEMAS

AUTOR
ANDRÉ LUÍS BELMIRO MOREIRA RAMOS



METODOLOGIAS DE DESENVOLVIMENTO DE SISTEMAS

AUTOR
ANDRÉ LUÍS BELMIRO MOREIRA RAMOS

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2017

Conselho editorial ROBERTO PAES E LUCIANA VARGA

Autor do original ANDRÉ LUÍS BELMIRO MOREIRA RAMOS

Projeto editorial ROBERTO PAES

Coordenação de produção LUCIANA VARGA, PAULA R. DE A. MACHADO E ALINE KARINA RABELLO

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística BFS MEDIA

Revisão de conteúdo LUIZ ROBERTO MARTINS BASTOS

Imagem de capa SHAIITH | SHUTTERSTOCK.COM

Dados Internacionais de Catalogação na Publicação (CIP)

R175M RAMOS, ANDRÉ LUÍS BELMIRO MOREIRA

Metodologias de desenvolvimento de sistemas. / André Luís Belmiro Moreira Ramos.

Rio de Janeiro: SESES, 2017.

152 P: IL.

ISBN: 978-85-5548-434-6

1. PROCESSOS. 2. EXTREMING PROGRAMMING. 3. RATIONAL UNFIED PROCESS. 4. AGILIDADE. 5. SCRUM. I. SESES. II. ESTÁCIO. CDD 004

Diretoria de Ensino — Fábrica de Conhecimento

Rua do Bispo, 83, bloco F, Campus João Uchôa

Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

No desenvolvimento de sistemas, vários fatores influenciam na qualidade final do produto. Para que possamos sempre resolver da melhor forma os problemas que necessitam de automatização, precisamos conhecer as diversas metodologias que existem na área de produção de *software*. Precisamos ter em mente que não existe uma única melhor forma de construção de sistemas, mas sim melhores processos em relação a determinados contextos.

Assim, este livro apresentará os conceitos básicos e avançados das principais metodologias de desenvolvimento de sistemas com ênfase nos seus aspectos teóricos e práticos, com exemplos e discussão de cada uma de suas etapas. Consequentemente, com os conhecimentos adquiridos, o estudante terá uma visão clara e prática das principais metodologias estudadas, podendo adaptá-las em situações reais de sua vida profissional.

Diante de todos os pontos descritos anteriormente, acreditamos que com o estudo atencioso do material aqui presente, você, com certeza, terá ferramentas necessárias para escolher a melhor forma de desenvolver sistemas.

Bons estudos!

1

Introdução às metodologias de desenvolvimento de sistemas

Introdução às metodologias de desenvolvimento de sistemas

Ao longo do tempo, sistemas de computador têm desempenhado funções cada vez mais importantes no cotidiano das pessoas. Percebemos que *softwares* estão presentes em diversos setores da sociedade, fazendo com que o seu funcionamento correto seja imprescindível para a vida das pessoas. Entretanto, a construção de sistemas é complexa, sendo necessária a adoção de diferentes metodologias de desenvolvimento para garantir a qualidade do que é produzido. Diante deste cenário, nesse capítulo, estudaremos o funcionamento das principais metodologias de desenvolvimento de sistemas e como elas evoluíram no decorrer dos anos.

OBJETIVOS

- Entender a importância de seguir uma metodologia de desenvolvimento de sistemas;
- Discutir os principais aspectos de diferentes metodologias de desenvolvimento de sistemas.

Introdução aos modelos de desenvolvimento de sistemas

Desenvolver sistemas não é uma tarefa fácil. Cada vez mais a necessidade dos usuários por produtos eficientes e eficazes aumenta, fazendo com que problemas não sejam tolerados. Porém, mesmo com o avanço da tecnologia, percebemos que grande parte das equipes de desenvolvimento ainda entregam produtos que falham em algum momento de sua operação. Neste sentido, podemos afirmar que quando um sistema falha, raramente o problema é técnico. Na maioria das vezes, os erros são consequência da falha na adoção de metodologias de desenvolvimento.

A discussão sobre formas sistemáticas de construção de sistemas teve início com a crise do *software*, na década de 70. O termo em questão faz alusão às dificuldades enfrentadas pelos desenvolvedores de *software* da época em consequência do rápido crescimento da demanda e da complexidade de sistemas, além da inexistência de técnicas de desenvolvimento. Nesta época, ocorreu a Conferência da OTAN sobre Engenharia de *Software*, que marcou o nascimento da engenharia

de *software*. O objetivo da reunião foi o estabelecimento de melhores práticas na construção de sistemas.

Figura 1.1 – Conferência da OTAN sobre Engenharia de *Software*.

Na época, a construção de sistemas era considerada um processo artesanal, onde a dinâmica de produção consistia na implementação de versões do produto a partir de refinamentos sucessivos, com o objetivo de consertar os defeitos até que o cliente se mostrasse satisfeito. Nesta forma de trabalho, com base na tentativa e erro, os requisitos são levantados de maneira informal e dificilmente o problema é modelado. O trabalho é centrado no programador, que a partir do uso de sua criatividade, resulta em produtos únicos. Neste cenário, percebe-se a pouca utilização de documentação e de boas práticas de engenharia. Não há planejamento e o código mostra-se como o artefato mais importante, chegando a ser visto como uma obra de arte.

CONEXÃO

Leia mais sobre a crise do *software* em: <http://cienciacomputacao.com.br/tecnologia/o-que-foi-a-crise-do-software-e-o-inicio-da-engenharia-de-software/>.

Como consequência, um dos problemas enfrentados na época foi o tempo necessário para conclusão de um *software*. Como a forma de trabalho da equipe não era padronizada, o consequente retrabalho implicava em um prazo de desenvolvimento acima do esperado. Com o aumento da demanda por *software*, o mercado passou a não conseguir suprir a necessidade da sociedade da época por sistemas informatizados. Ainda hoje é comum ter atraso no projeto, fazendo com que, muitas vezes, nos acostumemos a aceitá-lo como inevitável. Porém, o devido planejamento e monitoramento do projeto pode minimizar o problema.

Outro problema eram os altos custos atrelados ao desenvolvimento de sistemas: a forma como o sistema era construído impactava no custo do produto final, principalmente por fazer com que recursos não fossem despendidos de forma correta ao longo do processo de desenvolvimento. Neste contexto, podemos também discutir sobre a descoberta de erros antes da entrega do software aos clientes. Com o aumento da complexidade dos produtos, passou a ser comum a entrega de sistemas com muitos defeitos.

Como ponto negativo, também podemos citar a dificuldade em produzir em grande escala, já que o produto era artesanal e o sucesso de um projeto nem sempre significava que outros projetos outros projetos similares também obteriam êxito. Além disto, essa forma de trabalho possui uma forte dependência do talento da equipe e, como consequência, a manutenção se mostra difícil já que cada produto acaba sendo único. Por fim, nota-se uma grande variação na qualidade.

A partir desse cenário, ficou clara a necessidade de desenvolver *softwares* de maneira mais profissional e organizada com o objetivo de minimizar os problemas citados nos parágrafos anteriores.

? CURIOSIDADE

Os irmãos Wright foram os pioneiros na construção de aviões. Porém, até que a invenção fosse concluída com sucesso, diversas tentativas foram realizadas, causando desperdício de recursos. A ideia era a seguinte: os irmãos construíam a sua aeronave por completo e ao final empurrava-a para o despenhadeiro. Se o teste não fosse concluído com sucesso, começava-se tudo outra vez.

Ainda hoje, muitos constroem sistemas como os irmãos Wright construíam aviões. Como consequência, diversos problemas relacionados à construção de sistemas perduram ao logo do tempo, se mostrando como desafios para a obtenção de produtos de qualidade com recursos reduzidos.

Dessa forma, é necessário entender que o desenvolvimento de sistemas não pode ser limitado à implementação do código, sendo necessária a adoção de uma forma padronizada de trabalho que envolva atividades relacionadas ao planejamento, projeto, verificação e validação do que se é produzido. Em suma: faz-se necessário a utilização de um processo de desenvolvimento para construir produtos de qualidade que atendam às necessidades dos usuários.

Um processo de desenvolvimento de sistemas é um conjunto de regras que permite organizar um projeto em particular ao estabelecer o sequenciamento de atividades a serem realizadas. Deve responder perguntas importantes como: Quem realiza a atividade? Como a atividade deve ser feita? Por que se faz? Quando deve ser feito?

Nesse âmbito, Pressman (2006, p. 52) define processo como sendo "um arcabouço para as tarefas que são necessárias para construir software de alta qualidade". Sommerville (2011, p. 18) define um processo de *software* como “um conjunto de atividades relacionadas que levam à produção de um produto de *software*”.

Por sua vez, o Guia PMBOK define processo como sendo “um conjunto de atividades inter-relacionadas realizadas para obter um conjunto específico de produtos, resultados ou serviços” (PMBOK, 2012). Para o CMMI, um processo é definido quando tem uma descrição que é mantida, ou seja, tem documentação que detalha o que é feito (produto), quando (etapas), por quem (papéis), os itens utilizados (insumos) e os itens produzidos (resultados) (CMMI 2006).

A partir dessas definições, podemos perceber que grande ideia de processos de *software* é verificar que caminhos a equipe de desenvolvimento precisa percorrer para ao final ter um produto com qualidade e consequentemente com mais chances de ser o que o cliente espera. Percebe-se ainda que o detalhamento dos processos possui diferentes granularidades, podendo suas etapas serem realizadas de forma paralela. Ainda no contexto dos processos de *software*, diferentes atividades podem ser organizadas seguindo distintos modelos de desenvolvimento.

Sommerville

Um modelo de processo de software é uma representação simplificada de um processo de *software*. Cada modelo representa uma perspectiva particular de um processo e, portanto, fornece informações parciais sobre ele. Por exemplo, um modelo de atividade de processo pode mostrar as atividades e sua sequência, mas não mostrar os papéis das pessoas envolvidas.

FONTE: SOMMERVILLE, I. Engenharia de Software. 9ª Edição. Editora Pearson, 2011, página 18.

Dessa forma, podemos definir modelo de desenvolvimento como uma representação abstrata das atividades de um processo de *software* e suas interdependências, sendo uma versão simplificada de um processo de software, geralmente dividido em fases ou etapas.

Como não existe uma representação de processo perfeita, podemos concluir que um modelo tem que estar adequado à natureza do seu problema. Neste contexto, qualquer forma de desenvolvimento de sistemas deve possuir algumas fases básicas, como:

- **Planejamento e Elaboração**, onde todo o planejamento da construção do *software* deve ser realizado, com objetivo de antecipar possíveis problemas. Nesta fase, os requisitos do sistema são definidos, formando a descrição inicial das características e funções que o sistema deve possuir. Neste momento, a equipe envolvida no projeto deve ter o entendimento do que o usuário necessita, não sendo necessário, nesta etapa, saber como as definições serão implementadas. Além de uma lista de requisitos, protótipos também poderão ser construídos para confirmar os detalhes de negócio;
- **Análise,** onde o modelo conceitual deve ser definido, a partir do refinamento dos requisitos identificados na etapa anterior, para que o domínio do problema seja melhor entendido. Nesta fase, a solução é tratada em alto nível para atender aos requisitos e acaba complementando a especificação, sempre do ponto de vista do usuário, deixando de lado detalhes referentes à implementação;
- **Projeto**, onde a arquitetura do sistema é definida, com o objetivo de descrever o funcionamento do sistema a ser desenvolvido em um baixo nível de abstração, ao explicitar como as partes do sistema interagem entre si. Nesta fase, os

modelos de análise devem ser estendidos e detalhados com o objetivo de servir de insumo para a fase de implementação. Como consequência, módulos e suas interfaces devem ser definidos, e o uso de bibliotecas deve ser planejado. Por fim, o esquema do banco de dados deve ser pensado;

• **Implementação,** onde o código deve ser produzido de acordo com a especificação. Vale salientar que a codificação simples depende de como o sistema foi projetado na fase anterior. Boas práticas de programação devem ser utilizadas, como: padrões de codificação, revisões de código, simplicidade e refatoramento. Nesta etapa, dificilmente novos diagramas surgem;

• **Testes**, onde o produto construído deve ser verificado e validado a fim de que se tenha a garantia do correto funcionamento do sistema. Nesta fase, diversos tipos de testes podem ser aplicados, a saber: testes de unidade, testes de integração, testes de sistema, testes de validação, entre outros; e, por fim,

• **Manutenção**, onde, após a implantação e treinamento dos usuários, é realizada a evolução do *software* com o objetivo de suprir a necessidade de novas funcionalidades do produto e de correção de eventuais defeitos não detectados nas fases anteriores.

Apesar da descrição das fases anteriormente, os modelos de desenvolvimento não devem ser aplicados ao pé da letra e sim adequados à realidade da organização, sendo importante levar em consideração o tipo de sistema que está sendo construído. Os modelos genéricos de processos de software amplamente utilizados são o modelo em cascata, o modelo de prototipação, o modelo espiral, o modelo iterativo e incremental e o modelo de desenvolvimento baseado em componentes. Vale salientar que os modelos de prototipação e baseado em componentes nem sempre são aplicados de forma isolada sendo comumente utilizados em conjunto com outros modelos, principalmente no desenvolvimento de sistemas de grande porte e de alta complexidade.

SAIBA MAIS

Para entender melhor a importância da adoção de um processo de *software*, assista ao vídeo a seguir: <https://www.youtube.com/watch?v=QPiR8jTMLdI>.

Modelo cascata

O modelo em cascata, por ter sido um dos primeiros modelos de desenvolvimento de sistemas, também é conhecido como ciclo de vida clássico. A ideia do modelo em questão é inspirada na engenharia tradicional, onde podemos perceber uma abordagem sistemática ao estabelecer uma sequência entre as atividades envolvidas.

O principal objetivo do modelo cascata é propor uma nova forma de trabalho que minimize os problemas relacionados com o modelo artesanal de construção de sistemas. Vale a pena destacar que os artefatos produzidos em cada fase que compõe o modelo servem como entrada para a fase seguinte, sendo necessário o encerramento da fase corrente para que a próxima tenha início.

Como podemos ver na figura 1.2, o modelo é composto por diferentes fases que englobam desde a concepção do produto até a sua implantação e consequente manutenção.

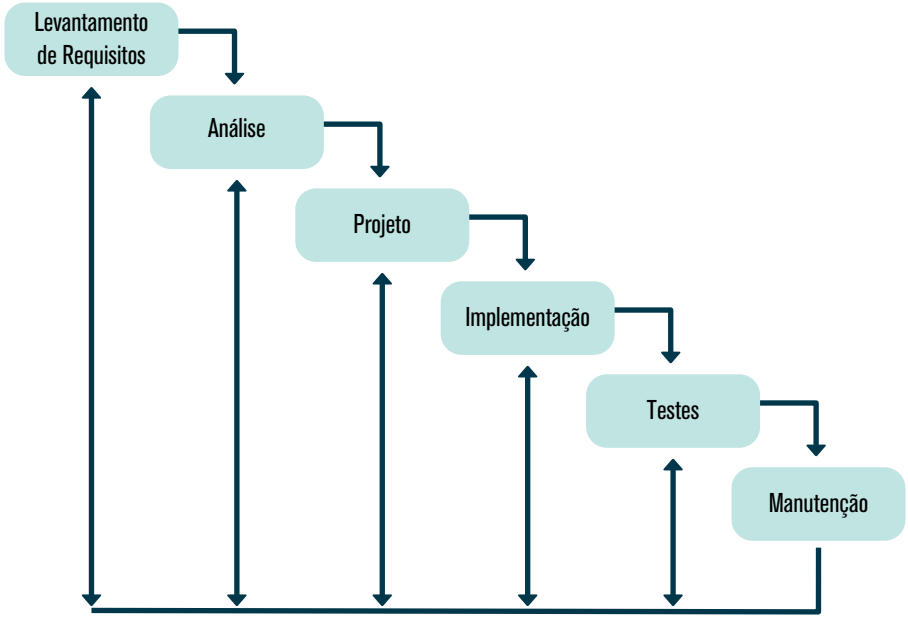


Figura 1.2 – Fases do modelo cascata.

Mesmo estabelecendo uma forma organizada de trabalho, o modelo em questão possui diversos problemas, como podemos ver a seguir.

• **Não prevê prototipação:** As necessidades do cliente são listadas em forma de requisitos, não oferecendo ao cliente uma perspectiva visual. O grande ganho da prototipação seria a possibilidade de o cliente validar os requisitos apontados, como também ajudar na elicitação de novos requisitos que até então estavam apenas na cabeça dele, de forma tácita;

• **Em geral, o cliente não sabe tudo o que quer no início do projeto**: A grande parte dos clientes não entende o domínio do problema, de forma completa, no início do projeto. Muitos deles vão descobrindo o que realmente querem ao longo do processo de desenvolvimento. Como no modelo em cascata as atividades são sequenciais, sendo imperativo que cada fase finalize para que a próxima tenha início, faz-se necessário possuir toda a definição de requisitos ainda no início do projeto;

• **Dificuldade em acomodar mudanças depois que o processo inicia:** A equipe deve garantir que todas as necessidades do cliente e as restrições do sistema sejam detectadas a fase inicial do projeto, o que é difícil de acontecer na prática. O modelo em cascata não acomoda mudanças relacionadas às necessidades dos usuários que são detectadas no decorrer da construção do sistema;

• **Erros graves poderão ser detectados apenas num momento tardio do desenvolvimento:** Como a fase de testes é realizada apenas após toda a codificação do sistema, muitos defeitos podem ser encontrados neste momento, quando é mais caro e difícil de consertá-los;

• **O cliente só conhece o produto na fase final do processo:** Apenas ao final da etapa de validação é que o cliente terá acesso ao produto, o que pode acarretar em inconsistência entre o que foi entregue e o que realmente o cliente queria; e, por fim

• **Projetos reais raramente seguem o fluxo sequencial que o modelo propõe:** na prática, o desenvolvimento de sistema não é realizado da mesma forma que a construção de um produto manufaturado. A natureza de produtos de software faz com que seja natural a paralelização da realização das atividades, onde muitas vezes as etapas de engenharia de requisitos, análise, projeto, implementação e testes são realizadas ao mesmo tempo.

Apesar dos problemas citados, o modelo cascata pode ser utilizado em casos específicos.

Sommerville

Em princípio, o modelo em cascata deve ser usado apenas quando os requisitos são bem compreendidos e pouco provavelmente venham a ser radicalmente alterados durante o desenvolvimento do sistema. No entanto, o modelo em cascata reflete o tipo de processo usado em outros projetos da engenharia. Como é mais fácil usar um modelo de gerenciamento comum para todo o projeto, processo de *software* baseados no modelo em cascata ainda são comumente utilizados.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 20.

Modelo de prototipação

A prototipação é um tipo de modelo evolucionário que representa o desenvolvimento de um modelo vivo do sistema, no qual se enfatiza a interface com o usuário. A prototipação colabora com o entendimento do que o sistema deve fazer, além de ser um mecanismo interessante que faz com que o cliente naturalmente proponha melhorias ao *software* em desenvolvimento. Neste sentido, o processo de prototipação ajuda a minimizar os riscos relacionados ao não atendimento das expectativas do cliente ao final do desenvolvimento do sistema.

Nessa forma de trabalho, um protótipo é construído para experimentação, com o objetivo de se obter requisitos dos usuários e posteriormente uma confirmação sobre as necessidades identificadas anteriormente. Na figura 1.3, podemos verificar as diversas fases do modelo utilizadas para construção de produtos.

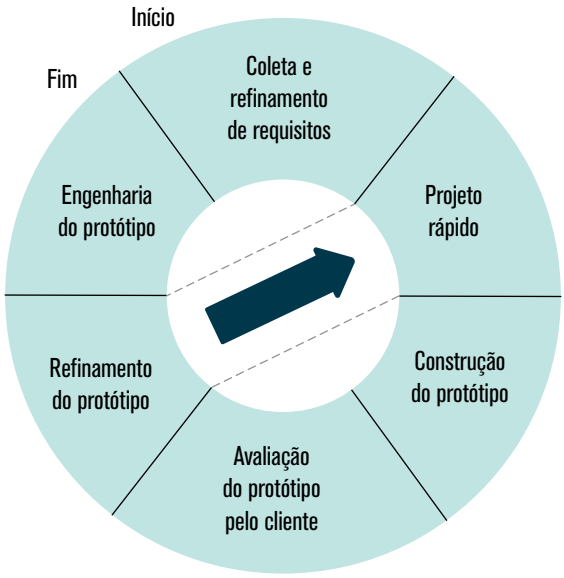


Figura 1.3 – Fases do modelo de prototipação.

A ideia do modelo em questão tem início na **coleta dos requisitos**, onde as necessidades dos clientes serão identificadas e listadas. Na sequência, é realizado um **projeto rápido**, a partir de uma modelagem breve, com o objetivo de diagramar os requisitos elicitados para melhor visualização. Após o projeto inicial, um **protótipo é construído** para o cliente **valide** os requisitos apresentados.

Após a análise do protótipo pelo cliente, o mesmo pode ser refinado para servir de insumo à **engenharia do produto**, onde estão incluídas, de forma implícita, as atividades de implementação e testes de *software*. Percebe-se que o modelo é proposto de forma cíclica, já que o protótipo pode ser refinado sucessivas vezes até estar pronto para ser implementado.

A prototipação apresentada ao cliente pode ser de duas diferentes formas: Prototipação Transitória (prototipação de baixo-nível), onde o nível de detalhamento não é alto e ao final da etapa o mesmo é descartado; Prototipação Evolutiva (prototipação de alto-nível), onde o detalhamento é incrementado até atingir o objetivo final, sendo o mesmo mantido ao longo do projeto.

A maior vantagem do modelo proposto, em relação ao modelo cascata, é a disponibilização para o cliente do aspecto visual do sistema ainda nas fases iniciais, com o objetivo de obter informações e apresentá-las aos usuários/clientes. Outro ponto positivo é a possibilidade de refinar o protótipo do produto a partir até que o mesmo seja validado pelo cliente para ser construído de fato.

Apesar das vantagens apresentadas, o modelo de prototipação possui algumas limitações, sumariadas a seguir:

• **Pode encorajar a análise superficial:** Como o foco do modelo é a parte visual do sistema, o cliente e a equipe de desenvolvimento podem passar despercebidos por questões importantes relacionadas à construção do produto, como o detalhamento das regras de negócio e outros diferentes aspectos técnicos. Um simples detalhe prototipado pode gerar um trabalho complexo que, no final das contas, talvez não vá agregar tanto valor para assim o cliente;

• **O usuário vê aquilo que pensa ser o *software*, mas não é:** Como consequência, clientes tendem a imaginar que, a partir da validação do protótipo, o sistema está quase pronto, o que não é verdade. Em muitos casos, a negociação em relação ao prazo de construção do software pode se tornar um desafio; e por fim

• **O desenvolvedor pode fazer concessões de implementação a fim de colocar um protótipo em funcionamento rapidamente:** Mais uma vez, a equipe de desenvolvimento e o cliente podem focar excessivamente na interface do sistema, em detrimento de aspectos importantes de negócio e/ou implementação.

Modelo espiral

Proposto por Boehm, o modelo espiral, assim como o de prototipação, é evolucionário, porém com aspectos sistemáticos e controlados do modelo cascata. As atividades são organizadas como uma espiral que tem vários ciclos, cada um representando uma determinada fase, como mostrado na figura 1.4.

O modelo é dividido em quatro fases: planejamento, análise dos riscos, engenharia e avaliação do cliente. Na primeira fase, inicialmente os requisitos iniciais são coletados e o planejamento do projeto é realizado. Para isto, a equipe de projeto, de forma conjunta com o cliente, deve determinar os objetivos, soluções alternativas e restrições do projeto. Conforme o projeto evolui, o mesmo é replanejado com base nos comentários do cliente.

Na fase seguinte, os riscos identificados com base na fase anterior devem ser analisados. Inicialmente os riscos tratados são referentes aos requisitos identificados no começo do projeto. Posteriormente, os riscos passam a serem influenciados pela reação do cliente ao longo da construção do *software*.

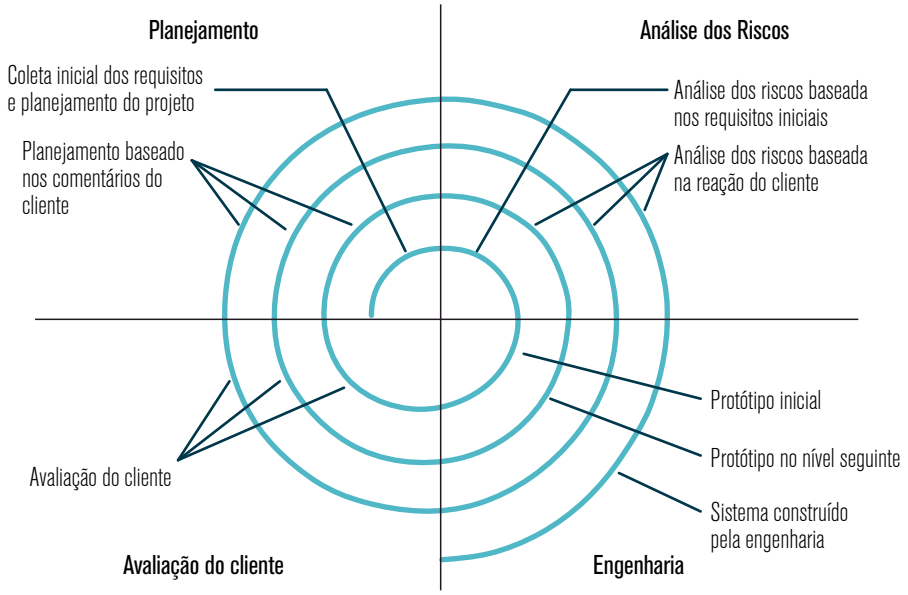


Figura 1.4 – Fases do modelo espiral.

A fase três consiste nas atividades da fase de desenvolvimento, incluindo projeto, especificação, codificação e testes. A ideia do modelo é ter os primeiros ciclos baseados na construção de protótipos, para que, somente no último ciclo, o produto seja construído de fato.

Por fim, a última fase consiste na avaliação do cliente com relação às etapas anteriores, com também no planejamento da próxima fase. A partir deste planejamento, a depender do resultado obtido, pode-se optar por seguir o desenvolvimento no modelo Cascata, caso os requisitos forem completamente especificados e validados. Caso contrário, pode-se optar pela construção de novos protótipos, incrementando-o, avaliando novos riscos e replanejando o processo.

O modelo em questão possui diversas vantagens em relação aos estudados anteriormente. Um dos principais aspectos da abordagem é a adição da análise de riscos, elemento até então desconhecido em outros modelos. Como consequência, essa característica torna o processo de construção de um produto complexo mais seguro.

Sommerville

A **principal diferença** entre o modelo espiral e outros modelos de processos de *software* é o seu reconhecimento explícito do risco. Um ciclo da espiral começa com a definição de objetivos, como desempenho e funcionalidade. Em seguida, são enumeradas formas alternativas de atingir tais objetivos e lidar com as restrições de cada um deles. Cada alternativa é avaliada em função de cada objetivo, e as fontes dos riscos de projetos são identificadas. O próximo passo é resolver esses riscos por meio de atividades de coleta de informações, como análise mais detalhada, prototipação e simulação.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 32.

Outra importante vantagem é que, por ser incremental, novas funcionalidades podem ser adicionadas em cada nova versão. Vale salientar que a evolução implícita ao modelo faz com que não exista distinção entre desenvolvimento e a manutenção do sistema construído. Por fim, podemos comentar que a presença constante de avaliação do cliente faz com que a qualidade seja obtida desde as fases inicias do projeto.

Apesar das vantagens citadas anteriormente, o modelo espiral possui importantes limitações, como:

- **A abordagem deste modelo exige grande experiência na avaliação dos riscos:** Muitas vezes a equipe pode não ter a maturidade suficiente para identificar possíveis riscos associados ao projeto, fazendo com que problemas não sejam identificados a tempo de serem desenvolvidas respostas aos mesmos;
- **Pode ser difícil convencer grandes clientes de que a abordagem evolutiva é controlável:** É necessário definir um prazo final do projeto sob o risco de nunca atingir as expectativas do cliente.

Em suma, modelo espiral é mais adequado para sistemas complexos e que exijam um alto nível de interações com os usuários, a fim de possibilitar a abordagem de todos os problemas desse sistema. Como consequência, a abordagem é utilizada com mais frequência em grandes projetos.

Modelo iterativo e incremental

O modelo iterativo e incremental é uma adaptação da abordagem espiral, onde o desenvolvimento e a entrega são divididos em pequenos pedaços denominados de incrementos. Cada incremento corresponde a parte da funcionalidade

requisitada pelo cliente. Desta forma, a especificação evolui junto com o desenvolvimento do sistema, dando suporte a requisitos parcialmente definidos.

Pfleeger

No **desenvolvimento incremental**, o sistema, como está especificado na documentação de requisitos, é dividido em subsistemas por funcionalidades. As versões são definidas, começando com um pequeno subsistema funcional e, então, adicionando mais funcionalidades a cada versão.

FONTE: PFLEEGER, S.L., Engenharia de *Software*: Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004., página 44.

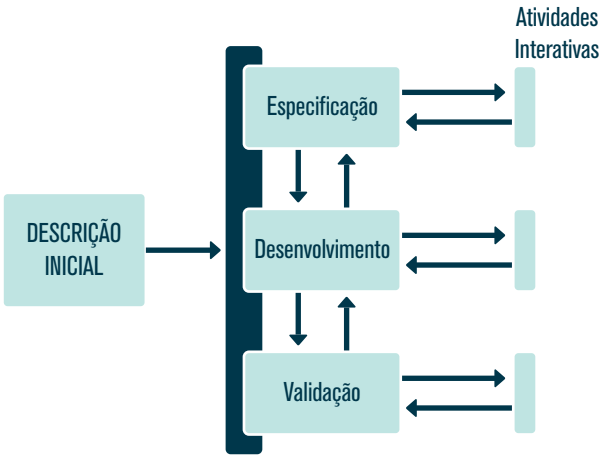


Figura 1.5 – Fases do modelo iterativo e incremental.

Como podemos perceber a partir da figura 1.5, o modelo proposto permite que a equipe de desenvolvimento possa trabalhar a qualidade e os detalhes do produto com refinamentos sucessivos. A partir da descrição inicial, a visão global do sistema pode ser analisada para garantir a viabilidade econômica do produto. Neste momento, pode-se realizar a modelagem inicial e um pouco de implementação, principalmente ligada à construção de protótipos.

As iterações posteriores devem ser organizadas com o objetivo de garantir que, ao final do processo, de forma incremental, todas as atividades de análise, projeto, implementação e validação foram devidamente realizadas. A ideia central da abordagem é que os produtos finais de todo o processo devem ser amadurecidos e finalizados ao longo do tempo, sendo organizados em iterações.

Pfleeger

O **desenvolvimento interativo** entrega um sistema completo desde o começo e então muda a funcionalidade de cada subsistema a cada nova versão.

FONTE: PFLEEGER, S.L., Engenharia de *Software:* Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004., página 44.

Em suma, podemos perceber que não existe uma sequencialidade entre as atividades realizadas, como no modelo cascata, já que a cada iteração são realizadas as etapas de **análise** (refinamento de requisitos, refinamento do modelo conceitual), projeto (refinamento do projeto arquitetural, projeto de baixo nível), implementação (codificação e testes) e transição **para produto** (documentação, instalação, ...), como explicitado na figura 1.6

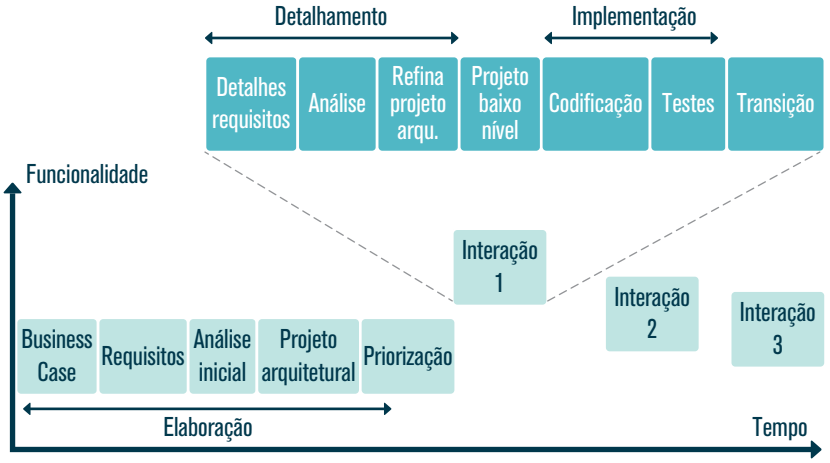


Figura 1.6 – Detalhamento das iterações no modelo interativo e incremental.

Em relação às abordagens anteriores, o modelo iterativo e incremental traz diversas vantagens relacionadas, principalmente, à possibilidade de avaliação prévia de riscos e pontos críticos do projeto, fazendo com que o monitoramento e consequente preparação adequada das respostas a estes riscos sejam realizadas a tempo.

Outro ponto interessante é que, a abordagem minimiza o risco de falha do projeto como um todo, já que como os requisitos mudam, um processo iterativo mantém frequentes contatos com o cliente o que ajuda a manter os requisitos sincronizados e as expectativas entre equipe e cliente alinhadas.

Como o projeto é quebrado em incrementos menores, o sistema é posto em funcionamento mais rapidamente. Como consequência, a equipe sempre tem, ao final de cada iteração, algo para entregar para o cliente.

Por fim, podemos citar que, o aumento da motivação da equipe de desenvolvimento, em razão da visualização prévia do funcionamento do sistema, resulta em uma aceleração do tempo de desenvolvimento do projeto como um todo.

Comparando o modelo iterativo e incremental e o modelo cascata, percebemos que a maior diferença entre os dois se dá na forma de organização das atividades.

Enquanto o modelo cascata define uma sequencialidade de tarefas, sendo a conclusão das tarefas obrigatória para que as posteriores tenham início, o interativo e incremental oferece um mecanismo de paralelização de tarefas, como observado na figura 1.7.

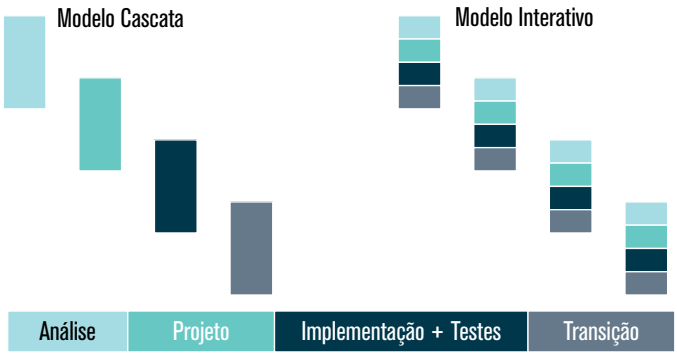


Figura 1.7 – Diferenças entre os modelos iterativo, incremental e cascata.

SAIBA MAIS

Leia um pouco mais sobre o conceito de iteração em: <https://engenhariasoftware.wordpress.com/2012/10/10/iteracao-de-projeto-ou-de-produto/>.

Modelo baseado em componentes

Cada vez mais, as organizações anseiam por metodologias de desenvolvimento de sistemas que maximizem a produtividade e tenham como consequência a obtenção de um alto retorno sobre o investimento. Problemas relacionados ao

cumprimento do prazo e custo previstos no início do projeto, e à diminuição do escopo acordado com o cliente, devem ser minimizados.

Neste cenário, o modelo em questão apresenta uma abordagem alinhada com o reuso de componentes, visando uma de melhoria de produtividade focada na reutilização de artefatos já construídos.

A modelagem baseada em componentes faz com que o produto seja construído por partes, distribuído em pequenos módulos, focado em apenas uma funcionalidade ou um conjunto de funcionalidades semelhantes, com o objetivo de minimizar a complexidade envolvida no desenvolvimento. Como módulo acaba sendo específico, o mesmo pode vir a ser reutilizado em diversas aplicações através do acesso ao componente.

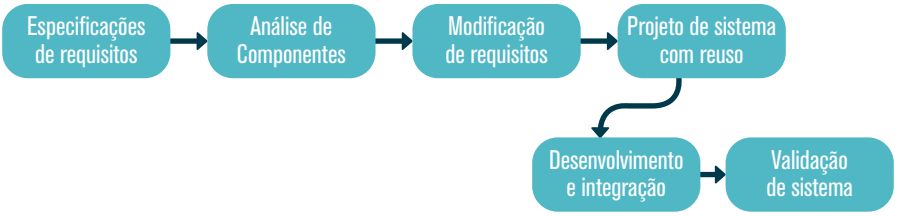


Figura 1.8 – Fases do modelo baseado em componentes.

Na figura 1.8, visualizamos as etapas do modelo baseado em componentes. Como nos demais modelos de desenvolvimento de sistemas, no início é necessária a especificação dos requisitos do *software* para que as necessidades do cliente/usuário se estabeleçam de forma clara e objetiva. A partir deste momento, uma análise é realizada em componentes existentes com o objetivo de verificar se o conjunto analisado supre as necessidades apontadas na etapa anterior.

Dificilmente a equipe de desenvolvimento consegue estabelecer um relacionamento singular entre o que foi especifica e os componentes disponíveis. Desta forma, os requisitos podem modificados para fins de adequação. Na sequência, o projeto do sistema é realizado com base no reuso dos componentes selecionados. Na etapa de desenvolvimento e integração, os componentes são interligados e novos componentes desenvolvidos para compor a solução final. Por fim, o cliente valida o sistema com base nas especificações iniciais.

Ao utilizar a abordagem descrita, percebemos diversos benefícios, tais quais:

- Aumento da produtividade em razão da economia de tempo de desenvolvimento, dependendo do conjunto de componentes pré-existentes. Neste sentido, o modelo permite que organizações assumam um número maior de projetos em

relação à abordagem tradicional de engenharia de *software*, já que a equipe de desenvolvimento, ao reutilizar código que já foi desenvolvido, acaba tendo mais tempo para ser envolvida em projetos adicionais;

• Aumento da robustez em razão da reutilização de componentes que já foram amplamente verificados e validados em projetos anteriores, garantindo uma maior qualidade no produto final. Neste contexto, os engenheiros estão trabalhando na implementação de um produto previamente testado e conhecido. Desta forma, depois que *software* estiver concluído, espera-se que menos problemas sejam apresentados;

• Utilização de um padrão de desenvolvimento orientado ao desenvolvimento nos moldes da componentização.

Em suma, as vantagens descritas acima fazem com que o prazo e o custo das aplicações construídas a partir da abordagem de componentes sejam reduzidos. Como consequência, o tempo e custo adicional pode ser utilizado pelas organizações na busca de outros projetos que aumentem a sua vantagem competitiva frente ao mercado.

Na tabela a seguir, podemos perceber algumas diferenças entre o modelo baseado em componentes e os demais apresentados nas seções anteriores.

	MODELOS TRADICIONAIS	MODELOS BASEADOS EM COMPONENTES
CONCEITO	O reuso existe, mas ocorre na fase de desenvolvimento. Geralmente envolve a reutilização de trechos de código.	O reuso é feito na montagem, sem custos de desenvolvimento, sem mudar a implementação dos componentes. Atualização e extensão do sistema são feitas de forma dinâmica por adição/integração de novos componentes.
ESPECIFICAÇÃO	Na especificação, são realizadas a análise das necessidades do cliente, a especificação do sistema e a validação dos mesmos pelo cliente.	Na especificação, são realizadas a análise das necessidades do cliente, a especificação do sistema e a validação dos mesmos pelo cliente.

	MODELOS TRADICIONAIS	MODELOS BASEADOS EM COMPONENTES
PROJETO	No projeto, é realizada a modelagem do sistema e a implementação da arquitetura do sistema.	No projeto, é realizada a busca e seleção de componentes.
IMPLEMENTAÇÃO	Implementação do código.	Desenvolvimento de partes não atendidas por componentes já existentes. Integração de componentes.
IMPLANTAÇÃO	Disponibilização da aplicação para uso em produção.	Disponibilização da aplicação para uso em produção.

Tabela 1.1 – Tabela referente às diferenças entre os modelos tradicionais e o modelo baseado em componentes.

A abordagem apresenta duas limitações: Muitas vezes os componentes adquiridos prontos não disponibilizam o código fonte, se comportando como uma caixa preta para os desenvolvedores. Tais componentes podem apresentar problemas não funcionais que impeçam a correta utilização do produto. Outro problema está relacionado a necessidade dos componentes possuírem interfaces suficientemente genéricas, sob pena de tornar futuras integração difíceis ou até mesmo impossíveis.

Apesar dos problemas, podemos afirmar que o modelo baseado em componentes é importante no cenário de reuso de *software*, evitando retrabalho, aumentando a qualidade e impactando positivamente na produtividade da equipe e controle de prazos e custos. Além disto, questões inerentes à programação, como confiabilidade e escalabilidade passam a ser tratadas de forma natural e contínua.

SAIBA MAIS

Leia um pouco mais sobre os conceitos de modelos baseados em componentes em: <http://dspace.bc.uepb.edu.br/jspui/handle/123456789/8169>.

Custos relacionados a modelos de desenvolvimento de sistemas

O desenvolvimento de *software* está atrelado a custos. Ao mesmo tempo em que a indústria cada vez mais constrói sistemas maiores e mais complexos, torna-se um desafio para as organizações a entrega de produtos com qualidade, a custos esperados e dentro do prazo.

Uma das formas de mitigar o risco em relação à extrapolação do prazo esperado é realizar um planejamento do projeto que leve em considerações o modelo de desenvolvimento adotado. Neste âmbito, Pressman (2006) afirma que a equipe deve organizar o trabalho e a ser feito e estimar os recursos necessários, com base nas atividades estipuladas, com o objetivo de antecipar possíveis problemas e prever um tempo de encerramento do projeto a partir da base histórica da organização.

Dessa forma, Sommerville (2007) afirma que estimar o desenvolvimento de um software não é tarefa fácil, já que variáveis como a qualidade do produto, recursos humanos envolvidos e riscos atrelados podem interferir na previsão. Em suma, não existe uma resposta única para elucidar essa questão. Contudo, podemos afirmar que, em linhas gerais, os custos relacionados a tempo seguem uma tendência de acordo com a figura 1.9 descrita a seguir.

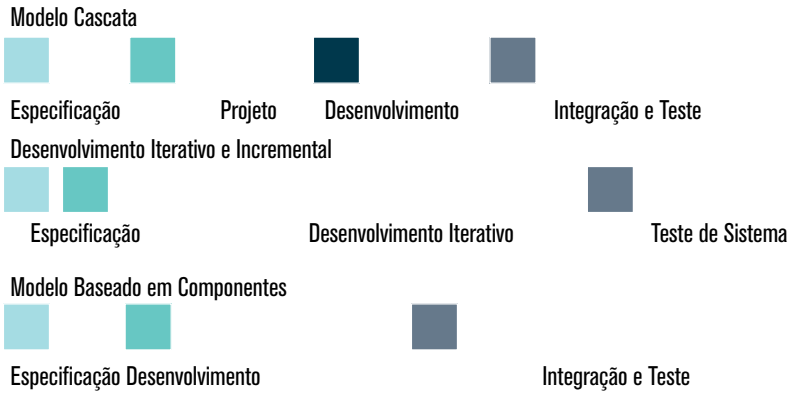


Figura 1.9 – Relação de tempo gasto entre as atividades de diferentes modelos de desenvolvimento.

Podemos perceber que, em modelos tradicionais, como o modelo cascata, o tempo envolvido na construção do produto é de 60% para as atividades

relacionadas ao desenvolvimento (especificação, projeto e desenvolvimento) e de 40% para as atividades relativas à integração do sistema e aos testes do sistema. O tempo entre as atividades é praticamente dividido de forma igual já que o modelo é caracterizado por uma sequência, onde cada atividade só deve começar quando a anterior finaliza.

Por sua vez, o desenvolvimento iterativo e incremental tem como base a paralelização das tarefas, o que faz com que a especificação inicial não ocupe um espaço significativo no processo, já que é esperado que mudanças ocorram e que sejam acomodadas durante construção do produto. Desta forma, temos a maior parte do modelo sendo dedicada ao processo iterativo.

Por fim, no modelo baseado em componentes temos uma situação a parte: Metade do processo de construção é destinada à integração e testes dos componentes do sistema, já que assumimos que a codificação do produto não é realizada do zero e sim com base no reuso de componentes construídos anteriormente. Desta forma, a equipe de desenvolvimento deve realizar o seu planejamento levando em consideração o tempo necessário para integrar os componentes e testá-los individualmente e em conjunto.

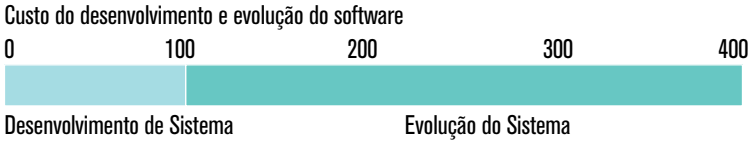


Figura 1.10 – Relação de tempo gasto entre o desenvolvimento e evolução do *software*.

Em relação à manutenção dos produtos, como visto na figura 1.10, o tempo de evolução do sistema acaba sendo bem maior do que o de desenvolvimento. A resposta para isso é simples: a partir do momento em que o produto é finalizado e disponibilizado para produção, ele tende a se estabilizar, porém, com o tempo ele deteriora em razão a novas necessidades dos usuários ou de correção de falhas encontradas. O tempo de evolução do *software* se dá até o momento onde a sua manutenção se torna mais cara do que a produção de um novo sistema que o substitua. A figura 1.11 explicita a distribuição dos custos nas tarefas atreladas ao desenvolvimento de *software*.

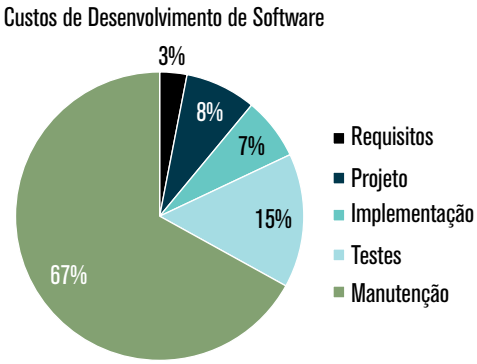


Figura 1.11 – Relação do tempo entra atividades de um modelo de desenvolvimento.

Ainda vale salientar que, como mostrado na figura 1.12, quanto maior o custo distribuído nas atividades de desenvolvimento, menor será o custo necessário para evoluir o sistema construído. A ideia é que quanto melhor desenvolvidas as etapas iniciais, como especificação e projeto, além de quanto mais testado for o sistema, menor a chance de o cliente propor novas funcionalidades e descobrir erros no futuro.

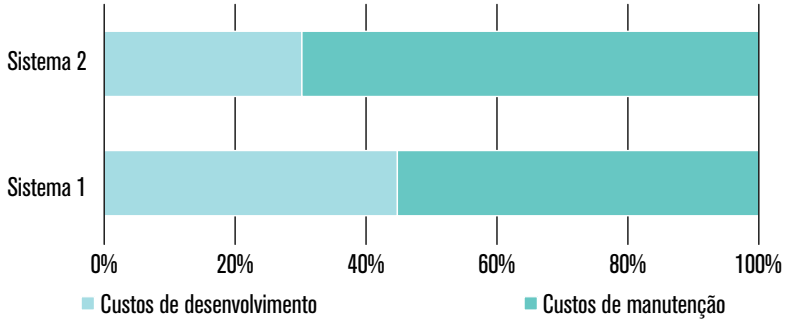


Figura 1.12 – Relação do tempo entre o desenvolvimento e a manutenção do produto.

SAIBA MAIS

Leia o artigo a seguir para entender melhor o desafio em relação ao cálculo do susto de um *software*: <http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php?conteudo=1332>.

ATIVIDADES

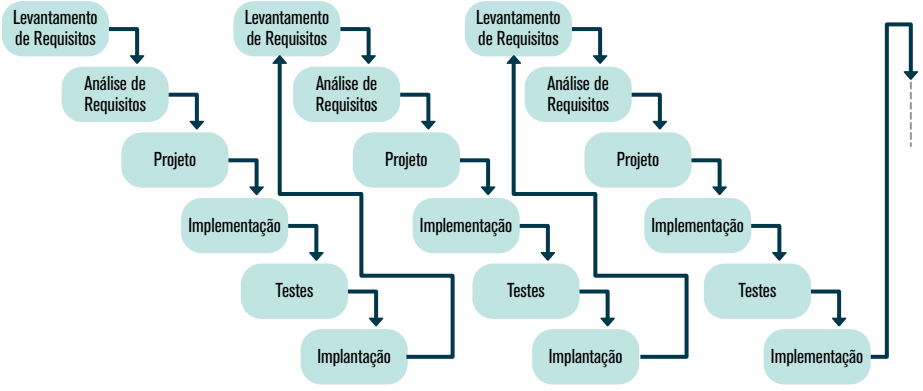
01. Qual a diferença entre modelos e processos de desenvolvimento de sistemas?

02. Defina, a partir das suas características, o modelo cascata. Você adotaria o modelo cascata em que tipo de projeto?

03. Em relação aos modelos de processos de *software*, pode-se dizer que o modelo iterativo e incremental acomoda melhor as mudanças. Assinale a alternativa que melhor descreve um modelo de produção de *software* iterativo.

a) As etapas são sequenciais, necessitando que a anterior finalize para que a próxima tenha início.
b) Os incrementos de um *software* são entregues ao cliente de uma só vez.
c) As etapas não acontecem de forma paralela.
d) É uma adaptação da abordagem espiral, onde o desenvolvimento e a entrega são divididos em pequenos pedaços denominados de incrementos.
e) É o único modelo a tratar questões referentes aos riscos do projeto.

04. Observe um modelo de ciclo de vida para desenvolvimento de sistemas. Nessa abordagem, o desenvolvimento do produto de software é dividido em ciclos, sendo identificadas em cada ciclo, as fases de análise, projeto, implementação e testes.



Este modelo é conhecido como ciclo de vida.

a) Cascata
b) Prototipação
c) Espiral
d) Iterativo e incremental
e) Baseado em componentes

05. Dos diferentes modelos para o ciclo de vida de desenvolvimento de um *software* é correto afirmar que:

a) O modelo em espiral é o mais simples e o mais antigo.
b) O modelo em cascata é o menos flexível e mais simples.
c) A fase de especificação de requisitos pode estar ausente do modelo.
d) A fase de implementação é sempre a última do modelo.
e) O modelo em cascata é um dos primeiros modelos de desenvolvimento de sistemas, também conhecido como ciclo de vida clássico.

REFLEXÃO

Neste capítulo, aprendemos de forma geral, a importância da adoção de métodos para desenvolver sistemas. Especificamente, estudamos as características dos principais modelos de desenvolvimento de sistemas e suas fases, que servirão como base para o aprofundamento na sequência de nossos estudos.

Entendemos que a escolha da forma de construção de um *software* é dependente do seu contexto, não existindo um modelo único a ser utilizado. Vimos também como os custos estão atrelados às tarefas envolvidas nos modelos.

Sugerimos que você faça todos os exercícios propostos e pesquise outras fontes para aprofundar seus conhecimentos. Em caso de dúvidas, retorne aos tópicos e faça a releitura com bastante atenção.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de modelos de desenvolvimento de sistemas consulte a sugestão de capítulos de livros abaixo:

- Capítulo 4 (Processos de *Software*) do livro de SOMMERVILLE, I. Engenharia de *Software*. 8ª Edição. Editora Pearson, 2007
- Capítulos 1 (Desenvolvimento de *software* para o valor de negócios) e 2 (Processos de desenvolvimento de software) do livro de ENGHOLM Júnior, Hélio. Engenharia de *software* na prática. São Paulo: Novatec Editora, 2010

REFERÊNCIAS BIBLIOGRÁFICAS

SOMMERVILLE, I. **Engenharia de *Software***. 9ª Edição. Editora Pearson, 2011.

PRESSMAN, R. S. **Engenharia de *Software*.** 6ª Edição. Editora McGraw Hill, 2006.

ENGHOLM Júnior, Hélio. **Engenharia de *software* na prática.** São Paulo: Novatec Editora, 2010

PFLEEGER, S.L., **Engenharia de *Software***: Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004., página 44.

PMBOOK. **Um Guia do Conhecimento em Gerenciamento de Projetos** (Guia PMBOK), Quinta Edição em Português. Project Management Institute (PMI). Global Standard, dezembro 2012, EUA.

CHRISSIS, M. B., KONRAD, M, SHRUM, S. CMMI: **Guidelines for Process Integration and Product Improvement**, Addison-Wesley, 2003.

2

Fases do desenvolvimento de sistemas

Fases do desenvolvimento de sistemas

Como visto no capítulo 1, independente da abordagem utilizada, qualquer metodologia de desenvolvimento de sistemas deve representar de forma abstrata as atividades de um processo de *software* e suas interdependências.

Para que fique claro o que é necessário realizar para entregar um produto, dividimos os modelos em etapas com o objetivo de deixar claro o que é necessário para definir, desenvolver, testar, operar e manter um sistema. Diferentes modelos, possuem diferentes etapas, porém normalmente estas são classificadas, de forma geral, como: planejamento e elaboração, análise, projeto, implementação, testes e manutenção.

OBJETIVOS

- Conceituar requisitos de sistemas e discutir as tarefas envolvidas na engenharia de requisitos;
- Aprender as diferenças entre as etapas de análise e de projeto;
- Verificar a importância da fase de implementação;
- Entender o objetivo da fase de testes e aprender sobre a classificação dos testes;
- Analisar os diferentes tipos de manutenção de *software*.

Introdução às fases de desenvolvimento de sistemas

O que é suficiente para construir sistemas de computador? Muitos podem imaginar que para desenvolver *software* basta conhecer uma determinada linguagem de programação ou dominar tecnologias envolvidas na construção do produto. De fato, ser um bom programador é uma característica importante no contexto de desenvolvimento de sistemas, porém não é a única habilidade requerida, já que a implementação é apenas uma das etapas do processo.

Para se construir um software, devemos realizar o levantamento e documentação das necessidades dos usuários e das características do sistema, modelar a análise e projeto de *software*, definir as estratégias de implementação, testar se

o *software* atende às expectativas do cliente/usuário e evoluir o produto com o objetivo de corrigir defeitos remanescente e incluir/alterar novas funcionalidades. Resumindo: precisamos entender o processo de desenvolvimento do sistema e adequá-lo ao contexto do produto. Desta forma, mais do que programação, é necessário construir um produto com qualidade, que atenda às expectativas do cliente e que esteja de acordo com a especificação.

Dessa forma, a partir do entendimento e utilização de um determinado processo de *software*, podemos amenizar o risco do produto não possuir atributos de qualidade, como:

- **Facilidade de manutenção:** o *software* deve ser escrito de modo que possa evoluir para atender às necessidades mutáveis dos clientes e usuários;
- **Nível de confiança compatível com o uso:** o nível de confiança envolve confiabilidade, proteção e segurança;
- **Eficiência:** o *software* não deve desperdiçar os recursos do sistema computacional no qual é executado;
- **Facilidade de uso:** o s*oftware* deve ser de fácil utilização, deve ter uma interface apropriada e documentação adequada.

Fase de Planejamento e Elaboração

No início de um projeto de *software* geralmente o domínio do problema é desconhecido. Mesmo que a equipe de projeto conheça parte do negócio a ser abordado, a visão das pessoas envolvidas na construção do sistema é sempre diferente da visão do usuário. Para aumentar o desafio, é comum não termos um referencial, como o sistema já existente. Além disso, cada vez mais as organizações demandam por sistemas com um alto grau de complexidade.

Neste âmbito, é necessário realizar atividades que tem como objetivo coletar, de forma clara e precisa, as necessidades dos usuários e as características do sistema a ser construído. Esse trabalho deve ser realizado pela equipe do projeto em conjunto com representantes dos clientes e usuários. Além disso, especialistas da área de aplicação do projeto podem interferir, levantando questões técnicas que são invisíveis aos demais participantes. Ao conjunto de todas essas tarefas relativas ao levantamento, detalhamento, documentação e validação dos requisitos de um produto damos o nome de **Engenharia de Requisitos**.

Sommerville

Os **requisitos** de um sistema são as descrições do que um sistema deve fazer, os serviços que oferecem e as restrições a seu funcionamento. Esses requisitos refletem a necessidades dos clientes para um sistema que serve a uma finalidade determinada, como controlar um dispositivo, colocar um pedido ou encontrar informações. O processo de descobrir, analisar, documentar e verificar esses serviços e restrições é chamado **de engenharia de requisitos**.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 56.

O sucesso do projeto está diretamente associado à engenharia dos requisitos. Neste sentido, é importante garantirmos que a maior parte das necessidades foi compreendida, caso contrário, teremos clientes/usuários insatisfeitos. Um dos grandes desafios da engenharia de requisitos é estabelecer os requisitos implícitos, ou seja, as expectativas dos clientes e usuários, que são cobradas por estes, embora não documentadas. Requisitos implícitos são indesejáveis, porém mesmo requisitos explícitos (documentados) e normativos (decorrente de lei ou normas) podem apresentar problemas em relação a sua completude, consistência e clareza.

Dessa forma, podemos perceber que requisitos mal-entendidos, imprecisos ou incorretamente especificados representam um grande risco para o desenvolvimento de *software*. O problema torna-se maior ao passo que o desenvolvedor tende a simplificar a interpretação de um requisito ambíguo.

Do ponto de vista de especificação, podemos classificar os requisitos de acordo com o seu nível de detalhe:

- **Requisitos do Usuário:** São as necessidades, característica e restrições do sistema descritas de forma abstrata, em um nível menor de detalhamento. O objetivo é comunicar ao cliente/usuário as funções que o sistema deve conter. Desta forma, devem ser escritos em linguagem natural e não indicar a solução para o problema abordado;
- **Requisitos de Sistema:** São as necessidades, característica e restrições do sistema descritas de forma concreta, em um nível maior de detalhamento. O

objetivo é comunicar ao time de desenvolvimento o que o sistema deve conter. Neste sentido, os requisitos de sistema podem incluir alguns detalhes de projeto.

Sommerville

De acordo com Sommerville, os **requisitos do usuário** são declarações, em uma linguagem natural com diagramas, de quais serviços o sistema deverá fornecer a seus usuários e as restrições com as quais este deve operar. Já os **requisitos de sistema**, são descrições mais detalhadas das funções, serviços e restrições operacionais do sistema de *software*, podendo ser parte do contrato entre o comprador do sistema e os desenvolvedores de *software*.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 58.

SAIBA MAIS

Para ver exemplos dos tipos de requisitos estudados até o momento, acesse: <https://prezi.com/hrq9wuxvzr7j/requisitos-de-usuario/>.

Do ponto de vista funcional, dividimos os requisitos em três grupos: requisitos funcionais, requisitos não-funcionais e requisitos de domínio.

Os requisitos funcionais descrevem as funções que o *software* deve conter, ou seja, as funcionalidades que representam os anseios dos clientes e usuários. Neste sentido, esse tipo de requisito explicita as funcionalidades e serviços do sistema ao passo que documenta como o sistema deve reagir a entradas específicas e como deve se comportar em determinadas situações. Além disso, os requisitos funcionais podem também indicar o que o sistema não deve fazer. A seguir, podemos verificar alguns exemplos de requisitos funcionais:

- O sistema deve emitir relatórios de vendas diárias de forma automática;
- O sistema deve possuir uma tela de cadastro de funcionários;
- O sistema deve permitir a transferência de funcionários entre diferentes setores;
- O sistema deve permitir emissão de nota fiscal.

Pfleeger

Um **requisito funcional** descreve uma interação entre o sistema e seu ambiente. Por exemplo, para se determinar os requisitos funcionais, decidimos quais estados são aceitáveis para o sistema. Além disso, os requisitos funcionais descrevem como o sistema deve se comportar, considerando um estímulo.

FONTE: PFLEEGER, S.L., Engenharia de *Software*: Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004, página 114.

Por sua vez, os requisitos não-funcionais são as características e restrições de um *software*. Estão relacionados a questões de qualidade, descritas na figura 2.1.

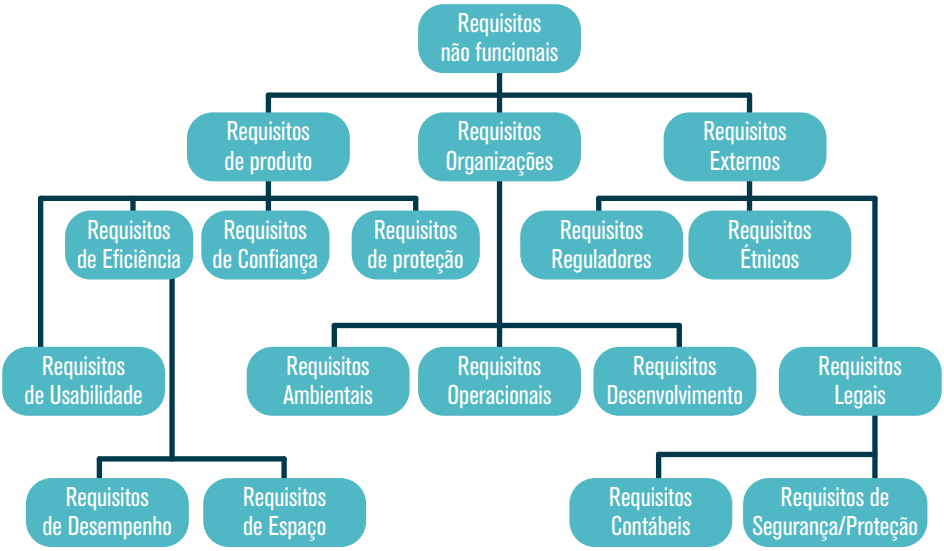


Figura 2.1 – Tipos de requisitos não-funcionais.

Um grande desafio em relação a este tipo de requisitos é mantê-los alinhados entre si. Por exemplo, facilmente requisitos de usabilidade podem se contrapor a requisitos de segurança.

REFLEXÃO

Vamos pensar um pouco: quando utilizamos um sistema bancário muitas vezes podemos nos chatear com exigências do tipo confirmar a senha ao final da operação ou inserir uma

chave de segurança. Isso acontece porque estes requisitos diminuem a **usabilidade** do sistema, porém aumentam a **segurança**. Ambos são requisitos não-funcionais que, neste caso, se contrapõem. E agora? Sabendo disso você prefere que? Uma menor quantidade de passos (usabilidade) ou uma maior verificação (segurança). A resposta ficou fácil! Neste caso quanto maior a segurança, melhor para o usuário, mesmo que pagamos o preço da facilidade de uso.

Pfleeger

Em vez de informar o que o sistema fará, os **requisitos não-funcionais** colocam restrições no sistema. Isto é, os requisitos não-funcionais ou restrições descrevem uma restrição no sistema que limita nossas opções para criar uma solução para o problema.

FONTE: PFLEEGER, S.L., Engenharia de *Software:* Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004, página 115.

A seguir, podemos verificar alguns exemplos de requisitos não-funcionais:

- O sistema deve permitir que apenas pessoas autorizadas acessem o produto;
- O sistema deve ser fácil de ser utilizado;
- As principais consultas do sistema devem ter um tempo de resposta de até 3 segundos;
- O sistema deve estar disponível 7 dias por semana e 24 horas por dia.

Por fim, os requisitos de domínio são definidos pelo negócio e não pelo usuário, sendo expressos através de linguagens específicas do domínio. Este tipo de requisito pode ser de difícil entendimento, já que nem sempre a linguagem do domínio de aplicação é compreendida pela equipe de desenvolvimento. Outro desafio é que, muitas vezes, os especialistas em domínio compreendem a área tão bem que não pensam em tornar os requisitos de domínio explícitos. A seguir, podemos verificar alguns exemplos de requisitos de domínios:

- Para cada dia de atraso de pagamento, será acrescida uma multa de 1% do valor total pago;
- O cálculo da média final de cada aluno é dado pela fórmula: (AV1 + AV2)/2;
- Um aluno pode se matricular em uma disciplina desde que ele tenha sido aprovado nas disciplinas consideradas pré-requisitos.

Como discutido anteriormente, a engenharia de requisitos é o processo de elicitação, análise, especificação e validação das funcionalidades, características e restrições do *software*.

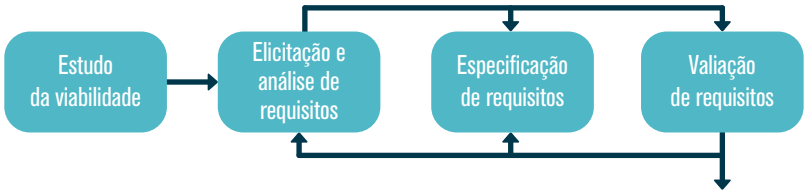


Figura 2.2 – Tipos de requisitos não-funcionais.

O processo em questão se dá a partir do **estudo da viabilidade** do projeto. Mas por que é tão importante avaliar a viabilidade da construção do produto ainda na fase inicial da coleta dos requisitos? A resposta é simples: é necessário que a equipe de desenvolvimento tenha maturidade para analisar se é capaz de entregar o *software* da forma como os clientes e usuários esperam. Neste sentido, o estudo da viabilidade deve ter sempre como objetivo dar insumos para as seguintes tomadas de decisão em relação ao sistema a ser produzido:

- O produto trará benefícios aos usuários interessados?
- Existem soluções alternativas melhores?
- O projeto pode ou não ser desenvolvido?

Após a decisão em relação à viabilidade do projeto, é realizada a **elicitação e análise dos requisitos**. Esta etapa é a coleta propriamente dita das necessidades dos usuários em relação ao *software*, onde teremos o entendimento do domínio da aplicação, do problema, do negócio e das necessidades e limitações dos *stakeholders*.

SAIBA MAIS

Entenda melhor quem são os *stakeholders* do projeto no vídeo a seguir: <https://www.youtube.com/watch?v=HGBXPkrPAps>.

Diversas técnicas podem ser utilizadas, como: entrevistas, aplicação de questionários, *brainstorm*, leitura de documentos, cenários, observações e análise sociais (etnografia), reuso de requisito, prototipação dentre outras.

Em seguida, todos os requisitos coletados devem ser **especificados**, ou seja, descritos em forma de linguagem natural para que, na sequência, sejam **validados** pelos *stakeholders*. Pela figura 2.2, podemos perceber que as etapas de elicitação e análise, especificação e validação de requisitos podem ser realizadas em paralelo, ou seja, não é necessário finalizar a etapa anterior para partir para a próxima fase. O artefato produzido ao final do processo é chamado de **documento de requisitos**. Segundo o Guia PMBOK®, "a documentação descreve como cada requisito atende à(s) necessidade(s) do negócio".

Em relação à complexidade da engenharia de requisitos, produtos mais complexos geralmente demandam por um investimento maior nas fases em comparação com produtos mais simples. A mesma situação é verificada quando temos a relação entre um *software* novo e outro que tem como objetivo o aprimoramento de um produto existente.

SAIBA MAIS

Documentação de *software* vale a pena? Descubra na leitura a seguir: <http://blogdosamueldiniz.blogspot.com.br/2008/05/documentao-de-software-vale-pena.html>.

Fase de Análise e de projeto

Após a geração do documento de especificação de requisitos, a equipe de desenvolvimento já pode fazer a análise e o projeto do produto. Nestas etapas, o principal objetivo é disponibilizar de forma visual as funcionalidades, características e restrições do sistema.

Pfleeger

Para transformar um requisito em um sistema funcional, os projetistas devem satisfazer os clientes e os construtores de sistema da equipe de desenvolvimento. Os clientes sabem o que o sistema deve fazer. Ao mesmo tempo, os construtores do sistema devem saber como o sistema funcionará.

FONTE: PFLEEGER, S.L., Engenharia de *Software*: Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004, página 160.

Como consequência da definição acima de Pfleeger, podemos afirmar que é necessária a divisão entre duas distintas etapas: a **fase de análise**, onde devemos apresentar ao cliente o que o sistema fará, gerando um projeto conceitual, e a **fase de projeto**, onde é realizado um detalhamento para que a equipe de desenvolvimento saiba quais são o *hardware* e *software* necessários para solucionar o problema, gerando um projeto técnico.

Neste sentido, a fase de análise se limita a descrever **o que** o sistema deve fazer, em detrimento de **como** os requisitos devem ser implementados, descritos na fase de projeto. O resultado da etapa de análise é um conjunto de modelos e diagramas que permitem fazer a transição para a fase de projeto, enquanto que o resultado da fase de projeto é um conjunto de diagramas e documentos que permitem que a codificação seja realizada de forma mais planejada.

Na prática, as fases de análise e de projeto são realizadas de forma paralela, em ciclos, onde parte da especificação, identificada na etapa anterior, é analisada e posteriormente detalhada na fase de projeto.

Neste contexto, podemos ter duas definições distintas para ambas as fases, como descritas nas figuras 2.3 e 2.4.

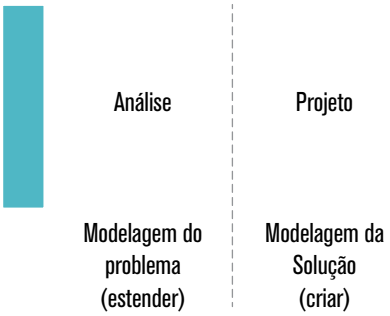


Figura 2.3 – Primeira definição das fases de análise e projeto.

A partir desta primeira definição, temos o limite entre as fases de análise e de projetos bem definido, onde a fase de análise é a etapa de modelagem do problema, ou seja, contém as atividades necessárias para o entendimento do domínio do problema. Por sua vez, a fase de projeto é a etapa de modelagem da solução, ou seja, contém as atividades necessárias para resolver o problema.

Dessa forma, podemos resumir a definição exposta na figura 2.3 como sendo a análise uma tarefa de investigação, onde **o que deve ser feito** é respondido, e o projeto uma tarefa de criação, onde a preocupação vai ser **como deve ser feito**.

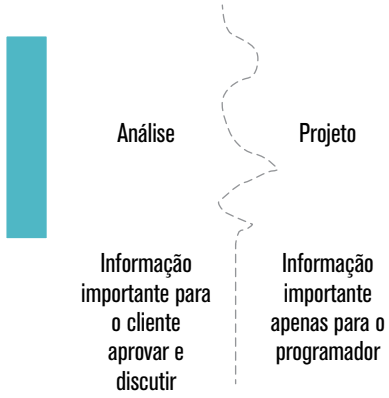


Figura 2.4 – Segunda definição das fases de análise e projeto.

A partir da segunda definição, descrita na figura 2.4, temos a fase de análise como sendo o conjunto de atividades realizadas com a ciência do cliente, ou seja, os usuários devem discutir e validar a informação produzida. Já a fase de projeto representa o conjunto das atividades que produzem informação destinada aos programadores.

No segundo cenário, percebemos que a análise acaba ultrapassando os seus limites e invade as atividades de projeto, já que o cliente acaba tendo que discutir alguns detalhes de implementação, que teoricamente seria de interesse somente dos programadores. Um exemplo clássico está ligado ao detalhamento da interface do usuário, que, apesar de ser do interesse da equipe de desenvolvimento, é necessário ter o aval dos usuários.

A partir das definições apresentadas, podemos entender a fase de análise como sendo a responsável pelo detalhamento dos requisitos obtidos no planejamento elaboração e fase de projeto como sendo responsável pelo detalhamento dos modelos produzidos na fase de análise.

Além disso, a etapa de projeto também é responsável pelas seguintes tarefas:

- **Definir da arquitetura do sistema:** O produto deve ser dividido em diferentes módulos e as interfaces de comunicação entre estes módulos devem ser pensadas;
- **Projetar arquitetura de acordo com os requisitos não-funcionais:** o produto deve ser preparado para atender às características e restrições identificadas na etapa de requisitos, tais quais, desempenho, tolerância a falhas, segurança entre outras;

• **Projetar arquitetura de acordo com as boas práticas e/ou padrões:** o produto deve ser planejado de acordo com padrões de projeto que garantam boas práticas de programação, como reuso, facilidade de manutenção, alta coesão, baixo acoplamento, separação do *software* em camadas dentre outras;

• **Escolher ferramentas e tecnologias que serão utilizadas:** é nesse momento onde é pensado quais ferramentas e tecnologias vão ajudar na solução do problema, tais quais, SGBD, servidores de aplicação, ferramentas de teste, suíte de desenvolvimento, linguagem de programação, dentre outras; e, por fim

• **Escolher estratégia(s) de desenvolvimento:** é nesse momento que a equipe decide como se dará o desenvolvimento do produto, por exemplo, se será necessária a aquisição de módulos e o reuso de componentes.

Em suma, percebemos que o planejamento da arquitetura do sistema, realizado na etapa de projeto, é essencial para que possamos identificar componentes do sistema, garantir o atendimento aos requisitos não funcionais e reaproveitar os modelos em projetos semelhantes.

SAIBA MAIS

Leia o artigo a seguir para aprender um pouco mais sobre arquitetura de sistemas: <http://www.devmedia.com.br/arquitetura-de-sistemas-de-informacao-uma-visao-geral/25326>.

A importância da modelagem visual

Vimos que o principal objetivo das fases de análise e projeto é modelar visualmente o sistema, a partir dos seus requisitos. Mas o que é um modelo?

Um modelo é uma simplificação da realidade. Modelos são úteis na descrição de um sistema a partir de uma perspectiva particular e auxiliam no entendimento e na formulação de problemas. Como exemplo, temos: um mapa, planta de uma casa, desenho de estilista etc.

Sommerville

Os modelos são usados durante o processo de engenharia de requisitos para ajudar a extrair os requisitos do sistema; durante o processo de projeto, são usados para descrever o sistema para os engenheiros que o implementam; e, após isso, são usados para documentar a estrutura e a operação do sistema.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 82.

A partir da definição anterior de Sommerville, a modelagem visual pode também ser utilizada em outras fases do processo de desenvolvimentos, já que colabora com a comunicação entre as partes envolvidas no projeto. Quanto mais complexo for um produto, maior a necessidade em utilizar técnicas de modelagem.

Além da vantagem citada anteriormente, a modelagem também apresenta como ganhos: Auxílio na visualização do que está sendo construído, foco na especificação da estrutura e o comportamento do sistema, melhor entendimento dos riscos envolvidos no projeto e documentação das decisões tomadas ao longo do desenvolvimento.

Com relação aos princípios de modelagem, temos que:

- A escolha dos modelos a serem criados tem uma grande influência em como um problema é atacado e como uma solução é desenvolvida;
- Todo modelo pode ser expresso em diferentes níveis de precisão;
- Nenhum modelo único é suficiente. Todo sistema não-trivial é melhor abordado através de um conjunto pequeno de modelos relacionados;
- Um sistema, geralmente, tem muitas classes (dezenas, centenas...) e nem sempre estas classes são vistas por uma só pessoa;
- Dependendo do nível hierárquico desta pessoa, formas diferentes de apresentar um diagrama de classes devem existir.

Neste contexto, para termos modelos interessantes, é importante que eles sejam independentes da implementação, ou seja, uma implementação pode ser trocada por outra sem afetar o modelo.

Além disso, modelos devem ser descritos de acordo com uma linguagem. No desenvolvimento de sistemas, a linguagem gráfica utilizada para visualização, especificação, construção e documentação de artefatos de um sistema de *software* é a UML (*Unified Modeling Language*).

SAIBA MAIS

Para ler mais sobre a linguagem UML, acesse o site: <http://www.uml.org/>.

Fase de Implementação

A fase de implementação consiste na codificação do *software*, ou seja, na etapa de escrita do código propriamente dita. Esta etapa deve garantir a execução bem-sucedida de todos os aspectos discutidos na fase de projeto.

Sommerville

O projeto e implementação de *software* é um estágio do processo no qual um sistema de *software* executável é desenvolvido. Para alguns sistemas simples, o projeto e implementação de *software* é a engenharia de *software*, e todas as outras atividades são intercaladas com esse processo. No entanto, para grandes sistemas, o projeto e implementação de *software* é apenas parte de um conjunto de processos (engenharia de requisitos, verificação e validação etc.) envolvidos na engenharia de *software*.

FONTE: SOMMERVILLE, I. Engenharia de *Software.* 9ª Edição. Editora Pearson, 2011, página 124.

Por mais que a etapa anterior tenha sido bem realizada, nem sempre a codificação é uma tarefa fácil, já que os projetistas podem não ter conhecimento suficiente em relação à linguagem de programação utilizada e, como consequência, o projeto pode não ser passível de implementação da forma como foi planejado.

Outro problema frequente está relacionado com o trabalho em equipe, necessário para esta etapa. A integração do código, produzido por diferentes pessoas, pode se tornar um desafio. Em razão disto, os codificadores devem seguir regras de implementação da empresa, que muitas vezes já possuem uma arquitetura básica pré-definida, com padrões adotados. Além do mais, regras básicas de programação, como nomes de variáveis, formato de cabeçalhos de programas e formato de comentários, recuos e espaçamento entre outras, devem ser utilizadas.

Falando nisso, segundo Pfleeger (2004, p. 262), cabeçalhos de programas são importantes na exposição de informações, como as seguintes: Qual o propósito do código em relação ao projeto como um todo, quem escreveu o código, quando foi escrito e revisado, entradas e saídas esperadas. O objetivo destas informações é de servir como insumo para tarefas como a integração com outras partes do sistema, testes, manutenção e reuso.

Em relação aos comentários realizados ao longo do código, a sua importância é relativa ao melhor entendimento por parte de todos os envolvidos no trabalho de produção e revisão do produto. Muitos acreditam que um *software* bem codificado não necessita de comentários, porém esta afirmação acaba se tornando um mito, já que, principalmente pela natureza coletiva da tarefa, dificilmente as equipes atingem um nível de qualidade de código que faz com que comentários adicionais sejam dispensáveis.

Por fim, Pfleeger (2004, p.264) chama atenção para a necessidade de termos nomes significativos para variáveis, indicando sua correta utilização. Da mesma forma, o uso adequado de recuo e espaçamento entre linhas de código, que ajudam a visualizar a estrutura de controle do programa.

Devemos prestar atenção também na documentação externa do programa, tão importante quanto os comentários realizados dentro do código. Na documentação externa, deve ser incluída a visão geral dos componentes do sistema, dos diversos grupos de componentes e da inter-relação entre eles.

Pfleeger

Ao passo que a documentação interna é concisa é escrita em um nível apropriado para o programador, a documentação externa será lida, também, por pessoas que nunca verão o código real. Por exemplo, os projetistas podem revisar a documentação externa, quando considerarem modificações ou melhorias. Além disso, a documentação externa oferece uma chance de explicar tudo de uma maneira mais ampla do que pode ser razoável em seus comentários de programa.

FONTE: PFLEEGER, S.L., Engenharia de *Software:* Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004, página 265.

Vale ressaltar que muitas vezes, por ansiedade em ver o *software* funcionando rapidamente, os integrantes do projeto acabam se precipitando e chegando rápido demais nesta etapa, fazendo com que o risco de se ter retrabalho aumente. As etapas anteriores atuam como guia, ainda que o codificador tenha liberdade para propor melhorias. Para que esta etapa seja concluída com sucesso, as unidades

de *software* devem ser codificadas e critérios de verificação das mesmas devem ser definidos. A tarefa de revisão de código também faz parte da fase.

SAIBA MAIS

Uma importante dica para obter código de qualidade é seguir padrões de projeto. Conheça um pouco mais sobre padrões de projeto em: <http://www.devmedia.com.br/conheca-os-padroes-de-projeto/957>.

Fase de Testes

A etapa de testes de *software* consiste na verificação e validação do produto. Segundo Pressman, "a atividade de teste é o processo de executar um programa com a intenção de descobrir um erro". Por sua vez, Sommerville (2011, p. 144) afirma que "os testes são destinados a mostrar o que um programa faz, o que pretende fazer e para descobrir os defeitos do programa antes desse ser colocado em uso".

A partir dessas definições, podemos afirmar que o objetivo desta etapa consiste, além da procura por erros, o mais cedo possível, na garantia de que o programa realiza o que os usuários esperam. Mas o que seriam erros no software? É considerado um erro, todas as situações a seguir:

- O sistema falha em relação ao cumprimento de algum item do documento de especificação do *software*;
- O sistema realiza alguma função ou possui alguma característica/restrição que claramente o documento de especificação do *software* diz que não deveria fazer/ter;
- O sistema possui requisitos que não estão contidos no documento de especificação do *software*;
- O sistema não realiza alguma função ou não apresenta alguma característica/ restrição que deveria estar no documento de especificação do *software*, mas foi esquecida pela equipe de requisitos;
- O sistema não possui uma boa usabilidade, deixando o usuário com um sentimento de frustração em relação ao seu uso.

CONEXÃO

Um mito importante na construção de produtos consiste na equipe adicionar funcionalidades que o usuário não pediu para melhorar o *software*, é o famoso "*gold plating*".

Aprenda mais sobre *gold plating*: <http://www.radardeprojetos.com.br/2015/02/gold-plating-em-projetos.html>.

Na prática, os processos de verificação e validação devem ser realizados em paralelo com as demais etapas do desenvolvimento de *software*, desde o início a partir da revisão dos requisitos, passando pelas revisões de projeto e inspeções de código até os testes do sistema e de manutenção.

Sommerville

Os processos de verificação e validação objetivam verificar se o *software* em desenvolvimento satisfaz suas especificações e oferece a funcionalidade esperada pelas pessoas que estão pagando pelo *software.* Esses processos de verificação iniciam-se assim que os requisitos estão disponíveis e continuam em todas as fases do processo de desenvolvimento.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 144.

Os processos de validação são essenciais já que é necessário que os usuários atestem que o produto final é exatamente o que eles esperam. O maior objetivo é garantir que o *software* é confiável e atende ao propósito estabelecido na etapa de requisitos.

Com relação às técnicas de verificação e validação, podemos classificá-las em dois grandes grupos:

- **Técnicas estáticas:** Utilizadas no contexto onde não é necessário ter o software sendo executado. O objetivo é analisar e verificar modelos do sistema como documento de requisitos, diagramas de análise e projeto, código-fonte do programa. Como exemplos, temos: inspeção de *software* e revisão por pares;
- **Técnicas dinâmicas:** Utilizadas no contexto onde é necessário que o software seja executado, utilizado dados de testes. O objetivo é examinar as saídas produzidas pelo sistema a fim de analisar o seu comportamento. Como exemplo, temos os testes funcionais.

Testes nas etapas anteriores à codificação

Como vimos, as atividades de testes devem ser realizadas em paralelo com as demais etapas do processo de desenvolvimento. Desta forma, começamos a testar ainda na etapa de requisitos, quando temos apenas as descrições do que o usuário necessita em relação ao produto.

Mas como podemos testar se o sistema ainda não está implementado? Vale a pena lembrar que o produto não se resume ao código! Podemos iniciar o processo de verificação e validação a partir da revisão dos requisitos. Neste sentido, as revisões de *software* são como um filtro para a gestão da qualidade do produto. O objetivo é revelar erros e defeitos que podem ser eliminados. Assim, podemos usar este tipo de teste para:

- Verificar o que podemos adicionar para melhorar o produto;
- Confirmar partes do produto os aperfeiçoamentos são desnecessários; e
- Aumentar a qualidade do que se é produzido, a partir de verificação de padronização.

A revisão pode ser informal ou formal. Em uma **revisão informal**, a equipe não realiza um planejamento antecipado em relação à aplicação do teste e os erros encontrados não são formalizados. Apesar de revelar problemas, é menos eficaz que as revisões formais. Podemos aumentar a eficácia das revisões informais a partir da utilização de listas de verificação (*checklists*) para cada artefato produzido pela equipe de desenvolvimento do sistema.

Por sua vez, em uma **revisão formal**, a equipe realiza uma preparação prévia e a aplicação do teste se dá de forma controlada. Além do mais, uma revisão formal se concentra em uma parte específica do sistema. Por questões de limitação de recursos e tempo, as revisões podem ser feitas por amostragem.

Testes na etapa da codificação

Durante a fase de codificação do *software*, podemos separar os testes em três grupos:

- **Testes unitários:** onde os testes são realizados individualmente, nas unidades do programa;
- **Testes de componentes:** onde são testadas as integrações entre as unidades do sistema e suas interfaces;

• **Testes de sistema:** onde o sistema é testado como um todo. Neste momento, as interações entre os componentes do sistema são verificadas.

Sommerville

O teste unitário é o processo os componentes do programa, como métodos ou classes de objeto. As funções individuais ou métodos são o tipo mais simples de componentes. Quando você está testando as classes de objeto, deve projetar os testes para fornecer uma cobertura de todas as características do objeto.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 148.

Dessa forma, fica claro que, a partir de testes unitários, devemos testar todas as operações relacionadas ao objeto, definindo e verificando os valores dos seus atributos, simulando todos os eventos que possam modificar o estado do objeto. Neste tipo de teste:

• O esforço de verificação é aplicado na menor unidade de projeto do *software*;

• Os erros encontrados são limitados ao às estruturas de controle de cada módulo;

• A complexidade e os erros encontrados são limitados pelo escopo de teste estabelecido.

Por sua vez, os testes de componentes, de acordo com Sommerville, "devem centrar-se em mostrar que a interface de componente se comporta de acordo com a sua especificação". Desta forma, os testes de componentes atuam como uma forma sistemática de construção da arquitetura do *software* ao mesmo tempo em que descobrem erros associados com as interfaces. Testes de componentes são importantes já que:

• Informações podem ser perdidas através de uma interface;

• Um componente, ao integrar-se com os demais, pode ter seu funcionamento alterado;

• Componentes combinados podem não apresentar o resultado principal desejado;

• Erros tolerados em componentes podem ter seu efeito amplificado;

• Estruturas de dados globais podem apresentar problemas.

Por fim, os testes de sistema verificam a combinação do *software* com outros elementos do sistema, como *hardware*, pessoal e bancos de dados, realizando uma análise do desempenho global do *software*. Os principais tipos de testes de sistemas são: teste de recuperação, teste de segurança, teste de estresse e teste de desempenho.

SAIBA MAIS

Unidade, integração ou sistema? Qual teste fazer? Leia um pouco mais sobre estes tipos de teste no artigo a seguir: <http://blog.caelum.com.br/unidade-integracao-ou-sistema-qual-teste-fazer/>.

Testes na implantação do sistema

Após a implementação do sistema, é necessário realizar testes como foco nos *stakeholders*. O objetivo é garantir que o *software* atende às necessidades dos usuários.

Sommerville

Teste de usuário ou de cliente é um estágio no processo de teste em que os usuários ou clientes fornecem entradas e conselhos sobre o teste de sistema. Isso pode envolver o teste formal de um sistema que foi aprovado por um fornecedor externo ou processo informal em que os usuários experimentam um produto de *software* novo para ver se gostam e verificar se faz o que eles precisam.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 158.

Os testes com usuários começam com fim do teste de sistema, onde os componentes individuais já foram testados, o *software* está montado como um pacote e os erros de interface já foram descobertos e corrigidos.

A validação é bem-sucedida quando o *software* funciona do modo esperado pelo cliente. Para o teste, são utilizados os requisitos descritos ao longo do projeto. Neste sentido, os testes são projetados para garantir que os requisitos funcionais e não-funcionais, principalmente usabilidade, sejam satisfeitos, e que a documentação esteja completa e correta.

Podemos classificar os testes que utilizam usuários em duas categorias:

- **Teste Alfa:** onde a condução do teste é realizada na instalação do desenvolvedor, utilizando os usuários finais.
- **Teste Beta:** onde a condução do teste é realizada na instalação do usuário, sem a presença da equipe de desenvolvimento.

SAIBA MAIS

Entregar produtos com erros pode resultar em catástrofes inimagináveis. Veja algumas falhas que marcaram a história: <http://crowdtest.me/10-falhas-software-marcaram-historia/>.

Fase de Manutenção

A fase de manutenção tem início a partir do momento em que o desenvolvimento do sistema se encerra e o produto é colocado em produção. Manutenções no sistema ocorrem pela necessidade de garantir que o produto continue atendendo às necessidades do usuário, seja através de correções de possíveis erros encontrados apenas após a entrega, seja a partir de inclusões ou alterações de requisitos propostos pelos clientes. Nesse sentido, percebemos que as atividades de manutenção estão relacionadas ao reparo de defeitos, adaptação de *software* e adição ou modificação de funcionalidade.

Pfleeger

A manutenção enfoca, simultaneamente, quatro aspectos principais da evolução do sistema: manter o controle sobre as funções do dia a dia do sistema, manter o controle sobre as modificações do sistema, aperfeiçoar as funções aceitáveis já existentes, e tomar medidas preventivas para que o desempenho do sistema não diminua para níveis inaceitáveis.

FONTE: PFLEEGER, S.L., Engenharia de *Software*: Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004, página 386.

Podemos classificar as manutenções de sistemas em:

- **Manutenção corretiva:** onde são tratadas as mudanças relativas ao reparo de defeitos do *software*;

• **Manutenção adaptativa:** onde são tratadas as mudanças relativas às adaptações do *software* a outro ambiente;

• **Manutenção perfectiva:** onde são tratadas as mudanças relativas à melhora de alguns aspectos do sistema, mesmo que não estejam atreladas a defeitos;

• **Manutenção preventiva:** onde são tratadas as mudanças relativas à prevenção de possíveis falhas em decorrência de defeitos no produto;

• **Manutenção evolutiva:** onde são tratadas as mudanças relativas à adição de novas funcionalidades no sistema.

Como podemos ver na figura 2.5, o custo de manutenção é geralmente muito maior que o custo de desenvolvimento. Desta forma, é importante garantir que o processo utilizado no desenvolvimento do sistema tenha sido bem aplicado, para que os custos de manutenção sejam reduzidos.

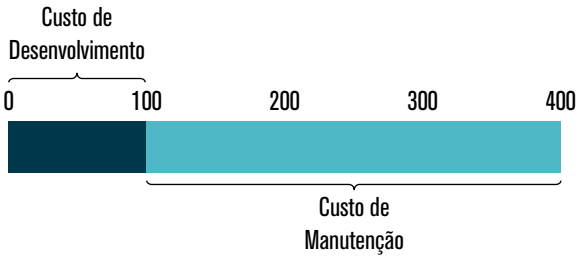


Figura 2.5 – Comparação entre os custos de desenvolvimento e de manutenção.

Um maior custo atrelado à manutenção pode ser explicado a partir dos seguintes fatores:

• Nem sempre a equipe que mantém o sistema é a mesma que o desenvolveu, fazendo com que o esforço envolvido no entendimento do produto seja maior;

• Os desenvolvedores de um sistema podem não ter responsabilidade contratual pela manutenção;

• Por ser uma atividade muitas vezes subvalorizada, a equipe de manutenção é geralmente menos experiente do que a equipe de desenvolvimento, além de possuir um conhecimento limitado do domínio do problema; e

• Com o passar do tempo, os programas têm a sua estrutura degradada, tornando-os mais difíceis de serem entendidos e alterados.

ATIVIDADES

01. Qual o principal objetivo da engenharia de requisitos? Comente sobre as atividades realizadas neste processo.

02. Qual a diferença entre as etapas de análise e projeto?

03. Por que é necessário padronizar a escrita do código?

04. O que você entende por verificação e validação de *software*?

05. Quais os motivos que levam ao aumento do custo em relação à manutenção de produtos?

REFLEXÃO

Neste capítulo detalhamos as fases gerais de modelos de desenvolvimento de sistemas, que servirão como base para o aprofundamento na sequência de nossos estudos. Para tanto, vimos os conceitos relacionados às etapas de planejamento e elaboração, análise e projeto, implementação, testes e manutenção, seus objetivos e seus principais artefatos de saída. Além disso, problemas comuns no desenvolvimento de produtos foram abordados, como a questão da relação entre o custo de desenvolvimento e o custo da manutenção.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de planejamento e elaboração, análise e projeto, implementação, testes e manutenção em projetos e demais assuntos deste capítulo, consulte a sugestão de *link* a seguir:

VACARI, I.; PRIKLADNICKI, R. Desenvolvimento de *software* na administração pública: uma revisão sistemática da literatura. Disponível em: <http://www.pucrs.br/facin-prov/wp-content/uploads/sites/19/2016/03/tr082.pdf>. Acesso em: Ago. 2016.

REFERÊNCIAS BIBLIOGRÁFICAS

SOMMERVILLE, I. **Engenharia de *Software***. 9ª Edição. Editora Pearson, 2011.

PRESSMAN, R. S. **Engenharia de *Software***. 6ª Edição. Editora McGraw Hill, 2006.

PFLEEGER, S.L., **Engenharia de *Software*:** Teoria e Prática, São Paulo: Prentice Hall, 2ª edição, 2004., página 44.

3

Rational Unified Process - RUP

Rational Unified Process - RUP

Nos dias atuais, é necessário que as organizações adotem uma forma padronizada de construir seus sistemas, sob pena de comprometer negativamente as propriedades do *software*. Desta forma, a utilização de um processo bem definido de desenvolvimento, passa a ser essencial na garantia da qualidade do processo e do produto final.

Nesse contexto, o *Rational Unified Process* (RUP) se apresenta como uma ferramenta bastante utilizada na comunidade de desenvolvimento de *software* atual, sendo um arcabouço de processo que pode ser adaptado a diversos tipos de projetos, independente do tamanho e complexidade. Por ser uma ferramenta complexa, a sua adoção pode se tornar burocrática, não sendo indicada a sua utilização em alguns contextos específicos, como em projetos de pequeno porte ou que apresentam uma equipe com pouca experiência.

OBJETIVOS

- Apresentar uma visão geral sobre o RUP;
- Descrever as visões, princípios e elementos do RUP;
- Analisar o ciclo de vida do RUP.

A visão geral do RUP

O RUP foi lançado oficialmente na década, pela *Rational*, em 1998, porém a sua ideia tem como base pesquisas que datam os anos 60, a partir dos estudos de Ivan Jacobson na *Ericsson*. Em 1987, após a saída de Jacobson da *Ericsson* e vários anos de estudo da linguagem *Objectory*, Grady Booch e James Rumbaugh se juntam ao projeto, denominados de "os três amigos", e surge o ***Rational Objectory Process***, como resultado da unificação das metodologias desenvolvidas pela *Rational* na década de 80 com a de Jacobson. Em paralelo, a linguagem de modelagem UML (*Unified Modeling Language*) foi desenvolvida, como também foram adquiridas pela *Rational* diversas companhias que produziam ferramentas para apoiar o desenvolvimento de *software*.

O RUP é um arcabouço de processo que utiliza os conceitos do Modelo em Espiral como alternativa para resolução dos problemas encontrados no então atual modelo de desenvolvimento de *software*, o modelo "cascata". Neste sentido, percebemos que a ideia principal da ferramenta é possibilitar a organização das tarefas de forma customizável, iterativa e incremental, em detrimento da sequencialidade abordada no modelo cascata. O RUP busca organizar as atribuições de tarefas e responsabilidade de forma coerente e coesa, visando a produção de *software* de qualidade a partir de um processo que aceite melhor as mudanças e faça com que o projeto se mantenha dentro do orçamento e cronograma planejados.

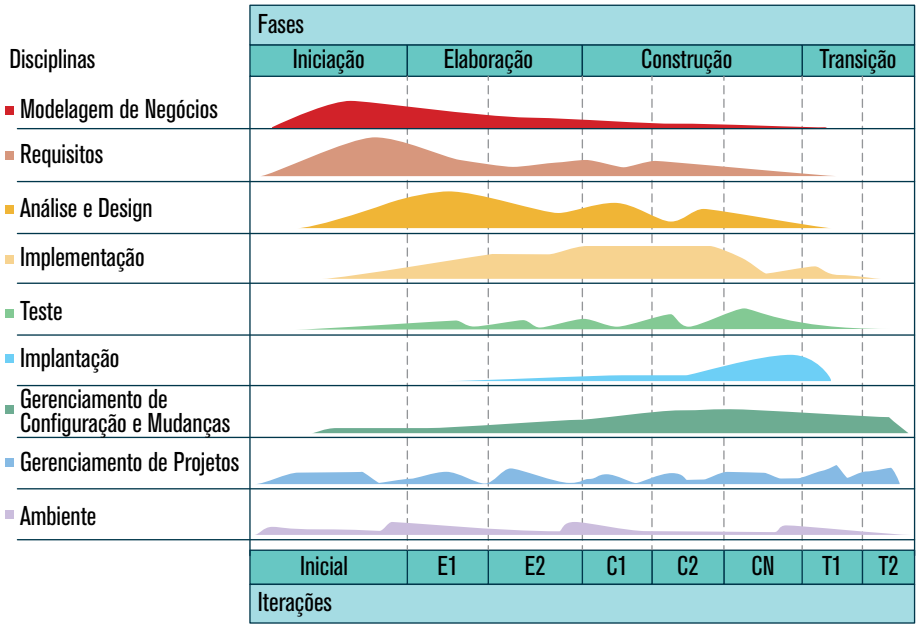


Figura 3.1 – Ciclo de vida do RUP.

Agora que conversamos sobre a história do RUP e as razões para o seu surgimento, podemos entender melhor o seu ciclo de vida, descrito na figura 3.1. O arcabouço de processo em questão apresenta duas perspectivas: Uma **dimensão dinâmica**, que mostra as fases do modelo ao logo do tempo, e uma **dimensão estática**, que mostra as atividades realizadas no processo, em conjunto com os papéis, artefatos e fluxos.

Vamos primeiro nos concentrar no entendimento dos aspectos dinâmicos do modelo, relacionados às fases de concepção, elaboração, construção e transição.

Neste contexto, de acordo com Sommerville (2011, p. 34), "O RUP é um modelo constituído de fases que, ao contrário do modelo em cascata, no qual as fases são equalizadas com as atividades do processo, as fases do RUP são estreitamente relacionadas ao negócio, e não a assuntos técnicos".

A **fase de concepção (iniciação)** tem como objetivo estabelecer o entendimento do negócio relacionado ao *software* a ser construído, além de realizar a análise de viabilidade do projeto. Nesta etapa, todas as entidades externas que irão interagir com o sistema devem ser identificadas e as interações devem ser descritas.

A **fase de elaboração** é utilizada para o entendimento do domínio do problema, com o estabelecimento dos requisitos, a partir dos modelos da UML, como também da definição da arquitetura e do plano de projeto. Nesta etapa, uma lista de riscos deve ser identificada.

Na **fase de construção**, o projeto, codificação e os testes do produto devem ser realizados. A implementação deve ser realizada em partes, tendo, ao final, os módulos integrados e funcionando como um sistema único. É nessa etapa onde a documentação interna e externa do *software* é produzida.

Com a **fase de transição**, temos a finalização do processo do RUP, sendo marcada pela entrega do sistema produzido para os clientes em um ambiente real. O objetivo desta etapa é realizar as atividades necessárias para permitir a disponibilização do *software* em um ambiente operacional

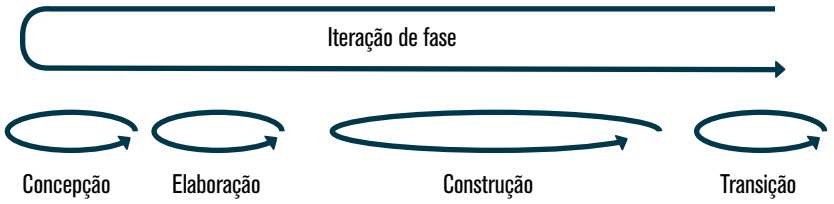


Figura 3.2 – Fases do RUP.

Na figura 3.2 fica claro que as fases descritas são realizadas de forma iterativa com os resultados desenvolvidos de forma incremental, ou seja, o trabalho relacionado à produção do *software* pode ser dividido em partes menores, como se fossem miniprojetos. Assim, podemos dizer que a abordagem iterativa e incremental consiste na repetição de atividades pré-estabelecidas que gera um acréscimo de funcionalidade ao sistema. Vale salientar que nem todo incremento consiste em

uma adição de funcionalidades ao sistema, já que podemos ter incrementos que são resultados de refinamento de funcionalidades, ou até mesmo de retirada de algo que já estava rodando em produção.

As principais características do desenvolvimento iterativo são:

• Permitir que riscos sejam pensados de forma antecipada, a partir do *feedback* constante do usuário; e

• Realização da integração e do teste de forma contínua, tornando possível a disponibilização de implementações parciais.

A maior vantagem relacionada a essa forma de trabalho está ligada ao fato do produto poder ser validado pelo usuário enquanto o projeto ainda está em desenvolvimento. Nesse contexto, em casos de falha, o sistema pode ser interrompido nas fases iniciais, poupando tempo e investimento.

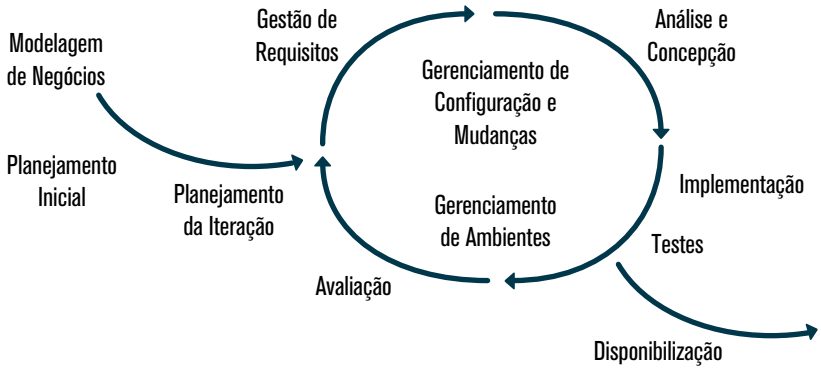


Figura 3.3 – O desenvolvimento iterativo e incremental do RUP.

SAIBA MAIS

Para entender melhor um modelo iterativo e incremental, assista ao vídeo a seguir:<https://www.youtube.com/watch?v=q6_Pq5zByME>.

Agora que entendemos a visão dinâmica do RUP, vamos estudar a sua perspectiva estática, focada nas suas disciplinas.

Sommerville

A visão estática do RUP prioriza as atividades que ocorrem durante o processo de desenvolvimento. Na descrição do RUP, estas são chamadas de *workflows*. Existem seis workflows centrais, identificados no processo, e três *workflows* de apoio. O RUP foi projetado em conjunto com a UML, assim, a descrição do *workflow* é orientada em torno de modelos associados à UML, como modelos de sequência, modelos de objeto etc.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 34.

Voltando para a figura 3.1, podemos verificar que as disciplinas descritas de forma estática não estão diretamente atreladas às fases, organizadas ao longo do tempo do projeto. O que temos é que todas as disciplinas podem, em tese, estar presentes em todas as fases descritas. Porém, é comum haver uma prevalência de certos *workflows* em determinadas fases, como por exemplo, na fase de concepção é comum que as disciplinas de modelagem de negócios e requisitos sejam realizadas de forma mais intensa. Por sua vez, na fase de transição já não faz mais sentido estar trabalhando intensamente com o foco na descoberta de novos requisitos ou na modelagem do negócio, desta forma, os principais *workflows* desta etapa são os relacionados à implantação, gerenciamento de configuração e mudança, gerenciamento de projetos e ambiente.

Os nove *workflows* descritos na ferramenta RUP são:

- **Modelagem de Negócio (*Business Modeling*):** O objetivo desta disciplina é analisar e entender o negócio envolvido, garantindo o alinhamento entre as expectativas de todos os *stakeholders* do projeto e os objetivos da organização. A disciplina contém tarefas relacionadas ao entendimento do que construir, a identificação de funcionalidades chaves, determinação de pelo menos uma solução possível e entendimento dos custos e riscos;
- **Requisitos (*Requirements*):** O objetivo desta disciplina é definir os limites do sistema, de acordo com os requisitos estabelecidos. É neste momento que os casos de uso são criados para que os *stakeholders* tenham um melhor entendimento em relação ao negócio;
- **Análise e *Design* (*Analysis & Design*):** O objetivo desta disciplina é modelar de forma visual os requisitos especificados nas disciplinas anteriores. A arquitetura do sistema é definida neste momento. Além disto, devem ser construídos os modelos da UML que servirão de base para a implementação do produto;

• **Implementação (*Implementation*):** O objetivo desta disciplina é codificar a solução adotada, tendo como insumo os modelos produzidos na etapa anterior. É nesse momento que os testes unitários são executados para garantir que cada módulo programado funciona de forma isolada;

• **Testes (*Test*):** O objetivo desta disciplina é verificar e validar o sistema construído, visando a detecção, documentação e endereçamento dos defeitos encontrados a partir da comparação do que foi construído com o documento de requisitos e com a perspectiva obtida pelo usuário. Essa disciplina é realizada em conjunto com a implementação;

• **Implantação (*Deployment*):** O objetivo desta disciplina é garantir que o *software* produzido fique disponível para seus usuários finais, em um ambiente real;

• **Gerenciamento de Configuração e Mudança (*Configuration & Change Management*):** O objetivo desta fase é permitir o controle das mudanças que ocorrem ao longo do projeto, além de manter a integridade dos artefatos trabalhados. Neste momento, cada artefato deve ser identificado, auditados e ter níveis de configuração e manutenção definidos;

• **Gerenciamento de Projeto (*Project Management*):** O objetivo desta disciplina é realizar as atividades relacionadas à gestão do projeto, visando o controle sob os riscos, cronograma, custos, pessoas, aquisições, entre outros, sempre com o foco no atendimento das expectativas dos clientes e usuários;

• **Ambiente (*Environment*):** O objetivo desta disciplina é manter a configuração do ambiente para que o processo e suas atividades possam ser executados. As ferramentas necessárias para o correto desenvolvimento do produto devem ser providas e disponibilizadas.

Na tabela 3.1 temos um resumo das disciplinas do RUP abordadas, fechando assim o nosso estudo sobre os aspectos estáticos da ferramenta.

WORKFLOW	DESCRIÇÃO
Modelagem de negócios	Os processos de negócio são modelados por meio de casos de uso de negócios.
Requisitos	Atores que interagem com o sistema são identificados e casos de uso são desenvolvidos para modelar os requisitos do sistema.

WORKFLOW	DESCRIÇÃO
Análise e projeto	Um modelo de projeto é criado e documentado com modelos de arquitetura, modelos de componentes, modelos de objetos e modelos de sequência.
Implementação	Os componentes do sistema são implementados e estruturados em subsistemas de implementação geração automática de código a partir de modelos de projeto ajuda a acelerar esse processo.
Teste	O teste é um processo iterativo que é feito em conjunto com a implementação. O teste do sistema segue a conclusão de implementação.
Implantação	Um *release* de produto é criado, distribuído aos usuários e instalado em seu local de trabalho.
Gerenciamento de configuração e mudanças	Esse workflow de apoio gerencia as mudanças do sistema (veja os capítulos 25).
Gerenciamento de projeto	Esse workflow de apoio gerencia as desenvolvimento do sistema (veja os capítulos 22 e 23).
Meio ambiente	Esse workflow está relacionado com a disponibilização de ferramentas apropriadas para a equipe de desenvolvimento de *software*.

Tabela 3.1 – Disciplinas do RUP.

Sommerville

As inovações mais importantes do RUP são a separação de fases e workflows e o reconhecimento de que a implantação de *software* em um ambiente de usuário é parte do processo. As fases são dinâmicas e tem metas. Os *workflows* são estáticos e são atividades técnicas que não são associadas a uma única fase, mas podem ser utilizadas durante todo o desenvolvimento para alcançar as metas específicas.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 35.

Além das perspectivas dinâmica e estática, o RUP apresenta a perspectiva prática, que sugere boas práticas a serem seguidas ao longo do projeto:

• **Desenvolvimento iterativo de *software:*** como já discutido, consiste na divisão do projeto em miniprojetos, denominados iterações;

• **Gerenciamento de requisitos:** visando o alinhamento com as perspectivas dos *stakeholders*, a gestão dos requisitos é essencial para controlar as tarefas de elicitação, especificação e documentação de requisitos ao longo do projeto, mesmo em fases finais no processo. A ideia é que mudanças possam ocorrer em qualquer momento e o projeto tem que se adaptar a elas. Para esta atividade, são utilizadas noções de casos de uso e cenários para facilitar a comunicação com usuários e garantir que a equipe esteja resolvendo o problema certo e desenvolvendo o sistema correto;

• **Arquitetura baseada em componentes:** é importante que a equipe projete uma arquitetura flexível, que leve em consideração o reuso e customização de componentes. Desta forma, esta boa prática permite que o software evolua de forma incremental, a partir da utilização de componentes disponíveis no mercado. Como consequência, teremos um maior encapsulamento;

• **Modelagem visual do *software*:** consiste na utilização de diagramas estruturais e comportamentais da UML, com o objetivo de modelar visualmente o sistema. O maior ganho desta boa prática é a manutenção da consistência entre concepção e implementação do produto. Como consequência da sua natureza visual, a modelagem de software promove uma comunicação não ambígua entre a equipe e os *stakeholders* e dentro do próprio time de desenvolvimento;

• **Verificação da qualidade de *software:*** faz alusão à preocupação com relação às seguintes dimensões de qualidade do produto: funcionalidade, usabilidade, confiança, suportabilidade e performance. A dimensão funcionalidade testa cada cenário de uso da aplicação. A dimensão usabilidade testa o software a partir da perspectiva do usuário. A dimensão confiabilidade testa a consistência e previsibilidade do produto. A dimensão suportabilidade testa a capacidade de manutenção do sistema em produção. Por fim, a dimensão de desempenho realiza os testes de carga. Na figura 3.4 podemos observar as dimensões de qualidade do produto e os seus respectivos tipos de testes, atrelados de forma específica a cada dimensão.

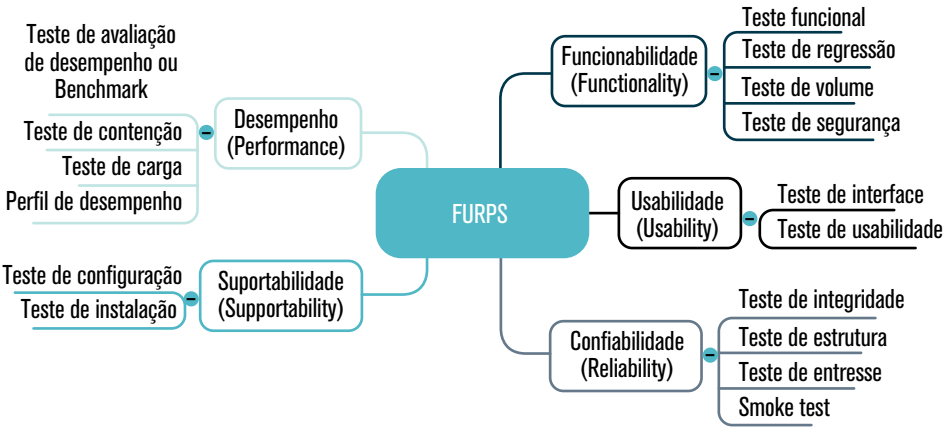


Figura 3.4 – Testes atrelados às dimensões de qualidade.

• **Controle de mudanças do *software:*** visa garantir o controle de novos requisitos, características e conserto de defeitos que surgem ao longo do processo de desenvolvimento do sistema. Como percebido na figura 3.5, as solicitações de mudanças podem vir de várias fontes durante o ciclo de vida do produto, o que torna o seu controle é essencial para o sucesso na construção do produto.

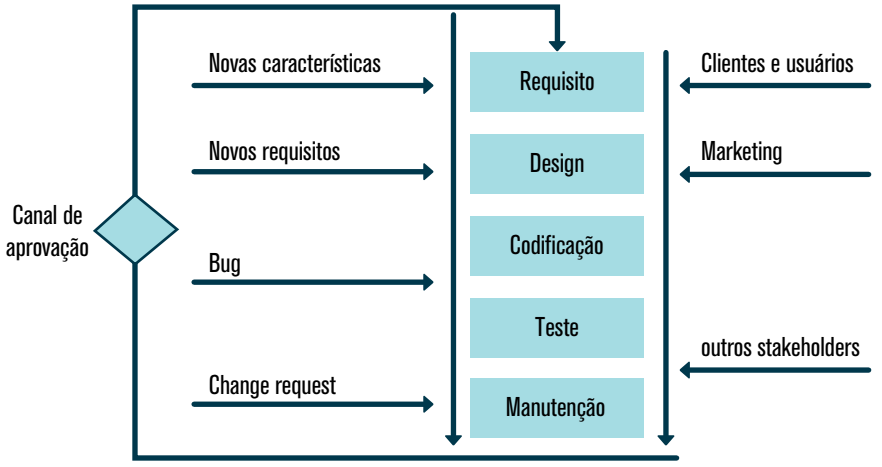


Figura 3.5 – Controle de mudanças do *software.*

Agora que já conhecemos as boas práticas encorajadas pelo RUP, podemos definir a ferramenta em relação a três importantes aspectos: o processo deve ser Guiado pelos Casos de Uso, Centrado na Arquitetura, e Iterativo e Incremental.

Os casos de uso no RUP são vistos como a força condutora do desenvolvimento, a partir do momento que são utilizados em diferentes disciplinas para captar novos requisitos dos *stakeholders* e para aceitação do produto final. Este aspecto é importante na garantia da perspectiva do usuário dentro do processo de desenvolvimento do sistema.

O segundo aspecto destaca a importância de se definir uma arquitetura no início do projeto para que as mudanças possam ser facilmente acomodadas. Neste sentido, a arquitetura é vista como um importante artefato na prova de conceitos, construção e evolução do sistema. Um dos pontos principais da arquitetura é a visualização preliminar do sistema antes da etapa de codificação. É nesse momento em que questões importantes, como a componentização, são abordadas.

A arquitetura de um sistema é descrita através de diferentes visões do sistema: a visão de lógica, a visão de implementação, a visão de processo, a visão de implantação e a visão de caso de uso.

CONCEITO

Aprenda mais sobre arquitetura de *software* acessando o *link* disponibilizado a seguir : <http://mds.cultura.gov.br/core.base_rup/guidances/concepts/software_architecture_4269A354.html>.

Visões do RUP

No dia a dia, a perspectiva dos envolvidos no projeto é diferente. Cada stakeholder observa o produto que está sendo construído sob uma ótica própria, ressaltando propriedades que lhe interessa e omitindo outras que não são relevantes para o seu contexto.

No RUP, os arquitetos de software utilizam a separação em visões com o objetivo de gerenciar a complexidade do projeto, a partir da organização de diferentes aspectos em visões distintas. Dessa forma, o arquiteto reduz a quantidade que informação tratada em contextos específicos. Com essa abordagem, cada componente da arquitetura de *software* pode ser observada de forma diferente, a depender do interessado, como visto na figura 3.6.

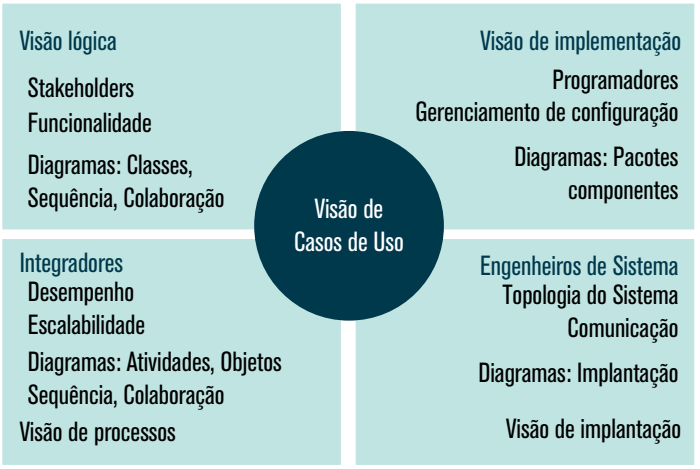


Figura 3.6 – Visões do RUP.

A **visão lógica** é realizada sob a perspectiva dos *stakeholders* e descreve os requisitos comportamentais e a decomposição do sistema em um conjunto de abstrações. Podemos afirmar que o objetivo é entender o produto do ponto de vista de problema de negócio, o que torna a perspectiva independente de decisões de projeto. Nessa ótica, os principais elementos apresentados são as classes e os objetos, tendo como principal base a modelagem dos diagramas de classes, sequência e colaboração (comunicação). As figuras 3.7, 3.8 e 3.9 apresentam exemplos desses tipos de diagramas da UML.

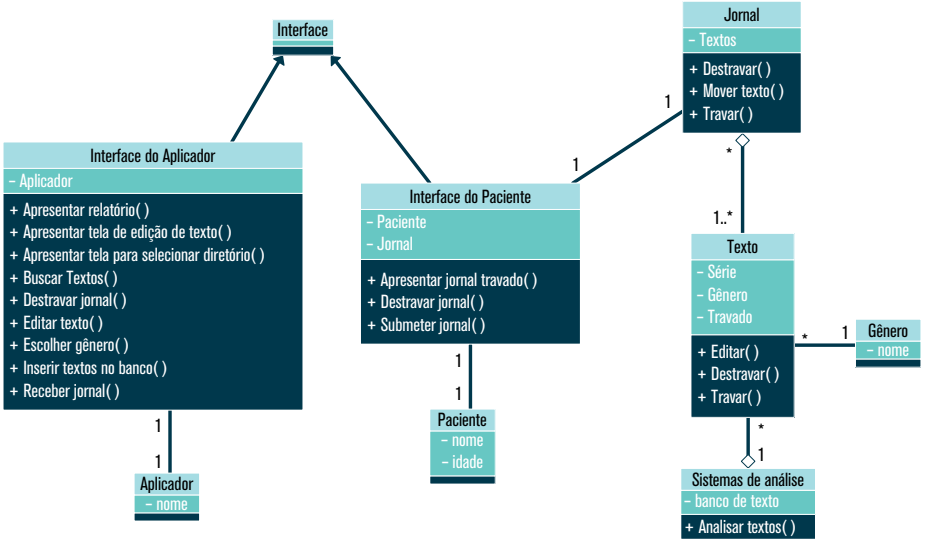


Figura 3.7 – Exemplo de diagrama de classes.

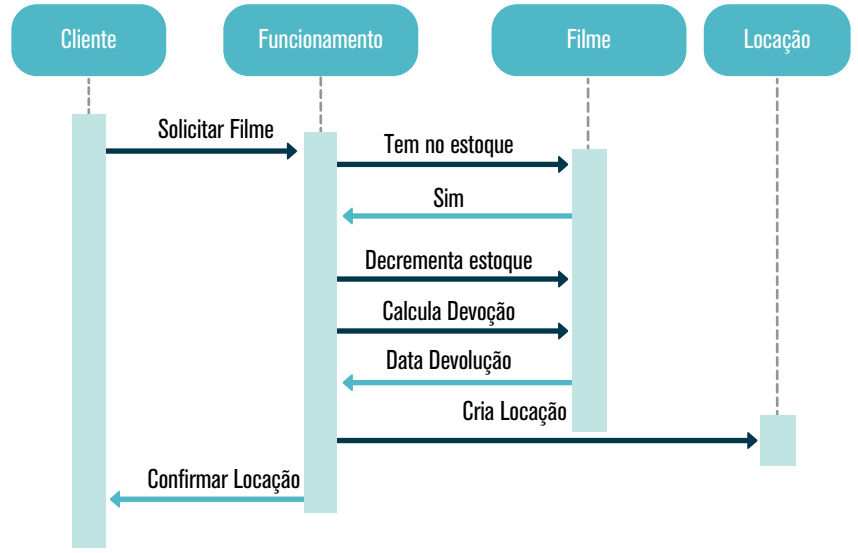


Figura 3.8 – Exemplo de diagrama de sequência.

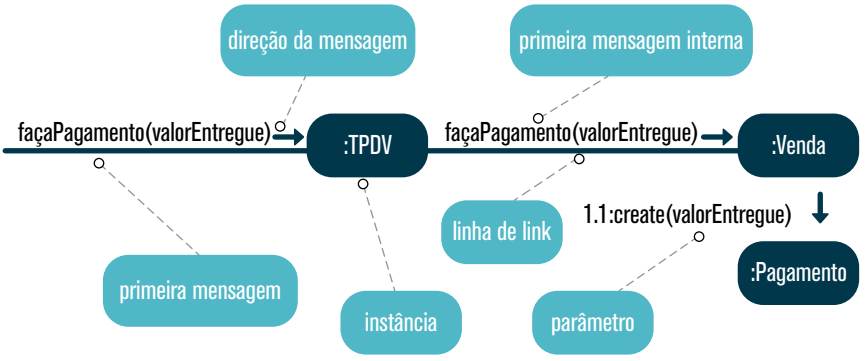


Figura 3.9 – Exemplo de diagrama de colaboração (comunicação).

A **visão de implementação** é realizada sob a perspectiva dos programadores, sendo utilizada para descrever os módulos do sistema e seus componentes. Nesse sentido, o objetivo é descrever detalhes do trabalho de codificação para toda a equipe, considerando aspectos de reuso, subcontratação e aquisição de *software*. Os diagramas da UML que representam essa visão são os diagramas de pacotes e de componentes.

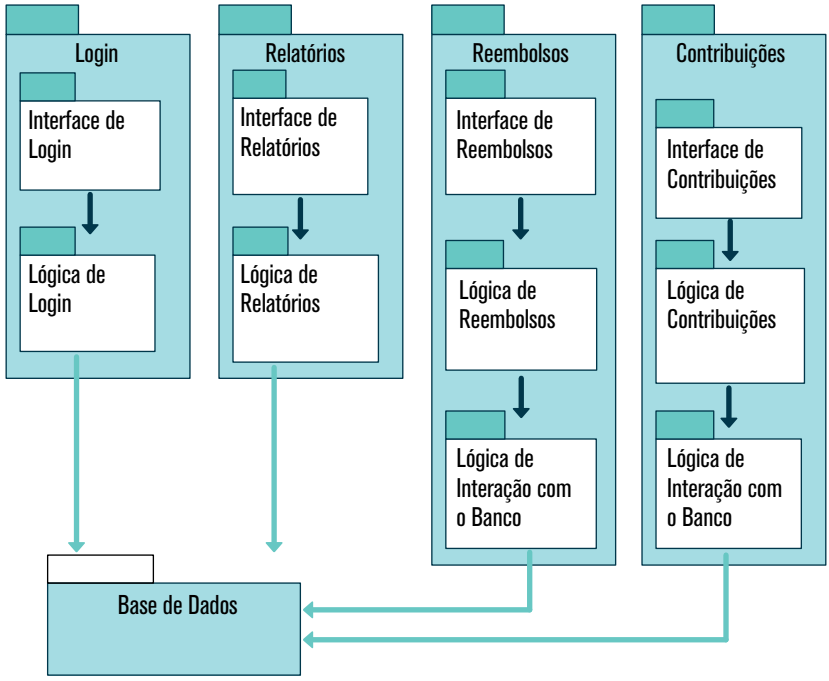


Figura 3.10 – Exemplo de diagrama de pacotes.

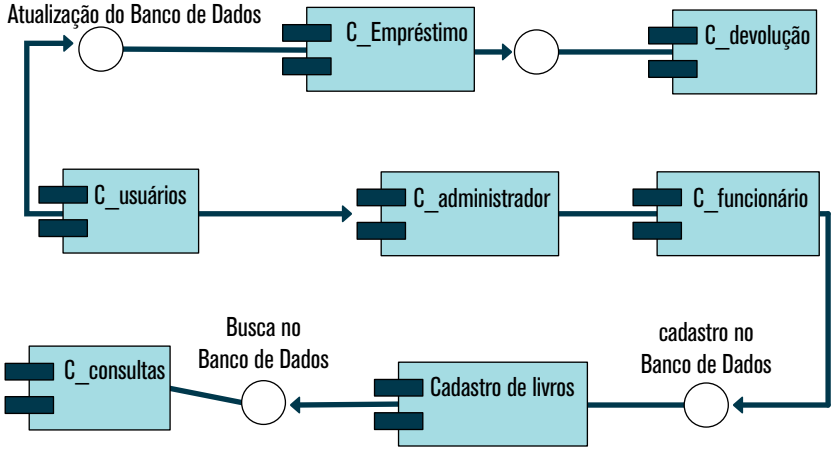


Figura 3.11 – Exemplo de diagrama de componentes.

A **visão de implantação** é realizada sob a perspectiva dos desenvolvedores, integradores e testadores, sendo utilizada para descrever como a aplicação é instalada. Esta visão permite avaliar requisitos não funcionais, como: desempenho, disponibilidade, confiabilidade e escalabilidade. A visão de implantação também se apresenta com o objetivo de modelar o sistema do ponto de vista da organização física, explicitando os computadores, periféricos e como eles se conectam entre si. Esta visão é representada pelo diagrama de implantação da UML.

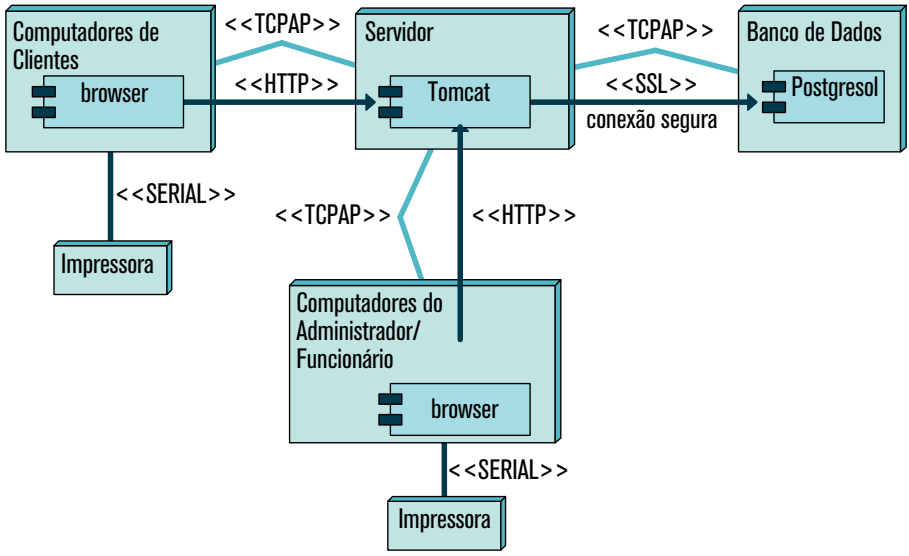


Figura 3.12 – Exemplo de diagrama de implantação.

A **visão de processo** é realizada sob a perspectiva dos integradores, sendo utilizada para descrever os processos do sistema e como eles se comunicam. Um dos objetivos desta visão é apresentar modelos que permitam avaliar requisitos não funcionais relacionados à execução e comunicação, como desempenho e disponibilidade. Os diagramas da UML que representam essa visão são os diagramas de atividades, objetos, sequência e colaboração.

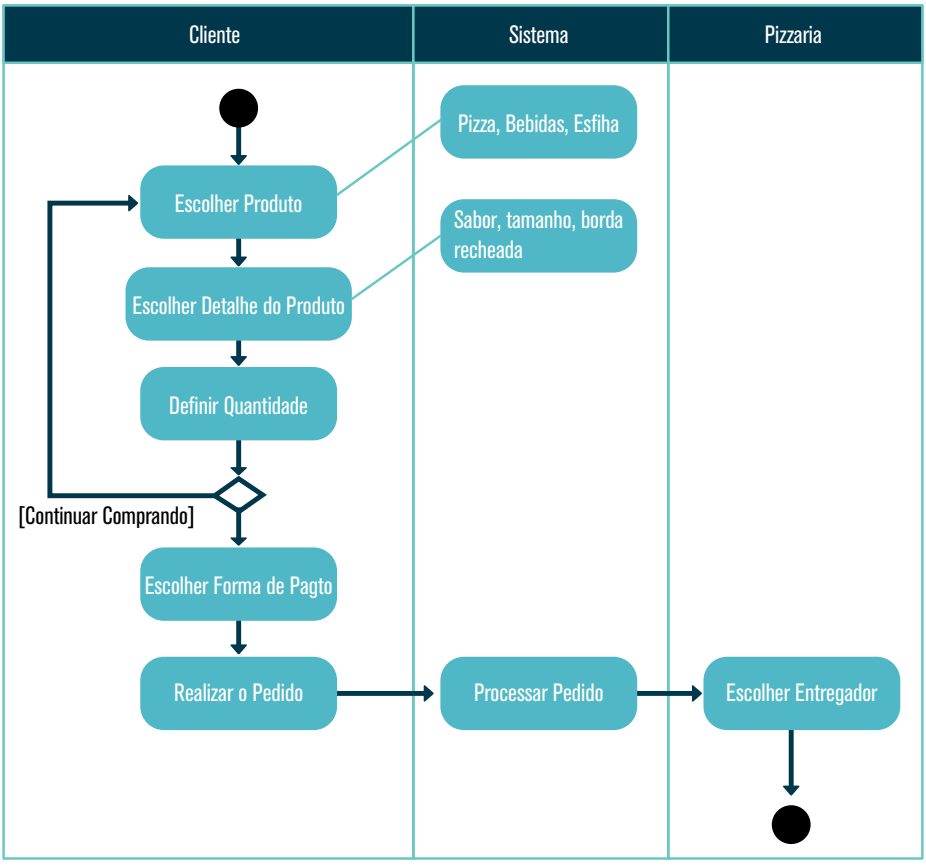


Figura 3.13 – Exemplo de diagrama de atividades.

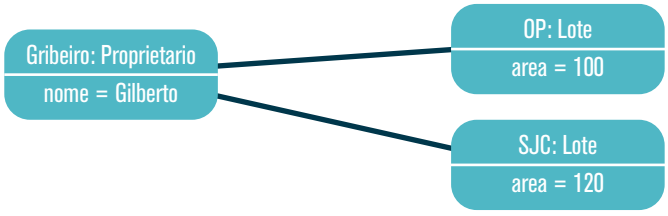


Figura 3.14 – Exemplo de diagrama de objetos.

A **visão de caso de uso** é realizada sob a perspectiva dos usuários, sendo utilizada para descrever a funcionalidade do sistema. Um dos objetivos desta visão é apresentar casos de uso e cenários que colaborem com o entendimento do que é necessário ser feito do ponto de vista funcional. Essa visão também mapeia o

relacionamento das demais visões ao mostrar com os seus elementos interagem. O diagrama de caso de uso representa essa visão.

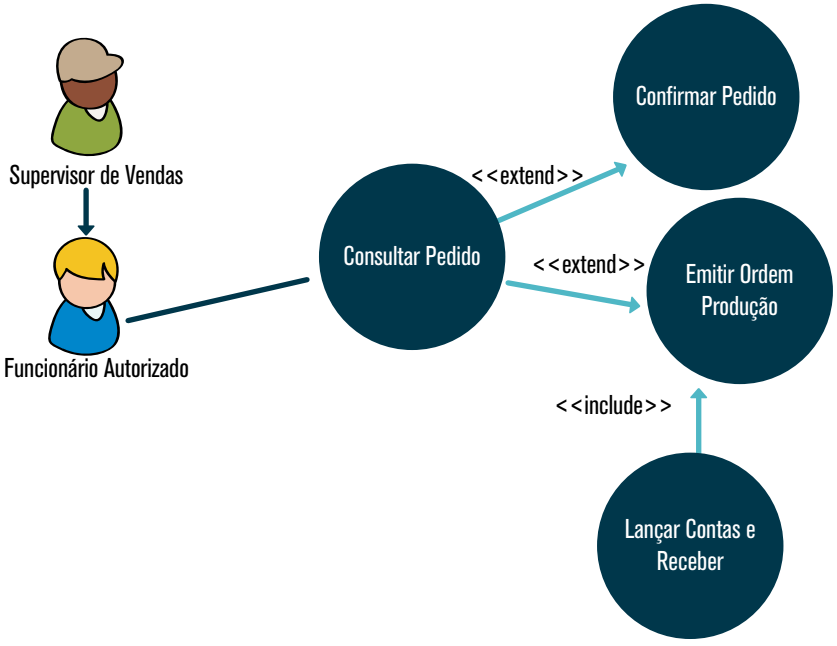


Figura 3.15 – Exemplo de diagrama de caso de uso.

VISÃO	LÓGICA	PROCESSOS
Componentes	Classes	Tarefas (processos e *threads*)
Conectores	Associação, herança, contenedores	Mensagem, RPC, *rendez-vous, broadcast*
Containers	Categoria de Classe	Processo
Envolvidos (*Stakeholders*)	Usuário-final	Projetista do sistema, integrador
Interesses (*concems*)	Funcionalidade	Desempenho, disponibli-dade e, integridade

IMPLEMENTAÇÃO	IMPLANTAÇÃO	CASOS DE USO
Módulos e sub-sistemas	Nó processador	*Step, Scripts*
Dependência, "*includes*" em C, "*with" clause*.	Rede de comunicação- *-lan, wan*,	
Sub-sistema (biblioteca)	Sub-sistema físico	*Web*
Desenvolvedor, gerente	Projetista do sistema	Usuário-final, desenvolvedor
Organização, reuso, portabilidade	Escalabilidade, desempenho, disponibilidade	Compreensibilidade, funcionalidade

Tabela 3.2 – Resumo das visões do RUP.

SAIBA MAIS

Falando em visões de arquitetura, o artefato que reúne todas é o documento de arquitetura de *software*. O documento de arquitetura de software fornece uma visão geral de arquitetura abrangente do sistema de *software*. Serve como um meio de comunicação entre o arquiteto de *software* e outros membros de equipe de projeto, com relação a decisões arquiteturalmente significativas tomadas sobre o projeto. Aprenda mais sobre este artefato no *link* a seguir:

<http://mds.cultura.gov.br/core.base_rup/workproducts/rup_software_architecture_document_C367485C.html>.

Elementos do RUP

A ferramenta RUP possui uma série de elementos que são utilizados para compor um processo de desenvolvimento de sistemas com o objetivo de explicitar quem faz o que e em que momento esta tarefa é realizada ao longo da construção do produto. São eles: papéis, atividades, artefatos, fluxos de trabalho e disciplinas.

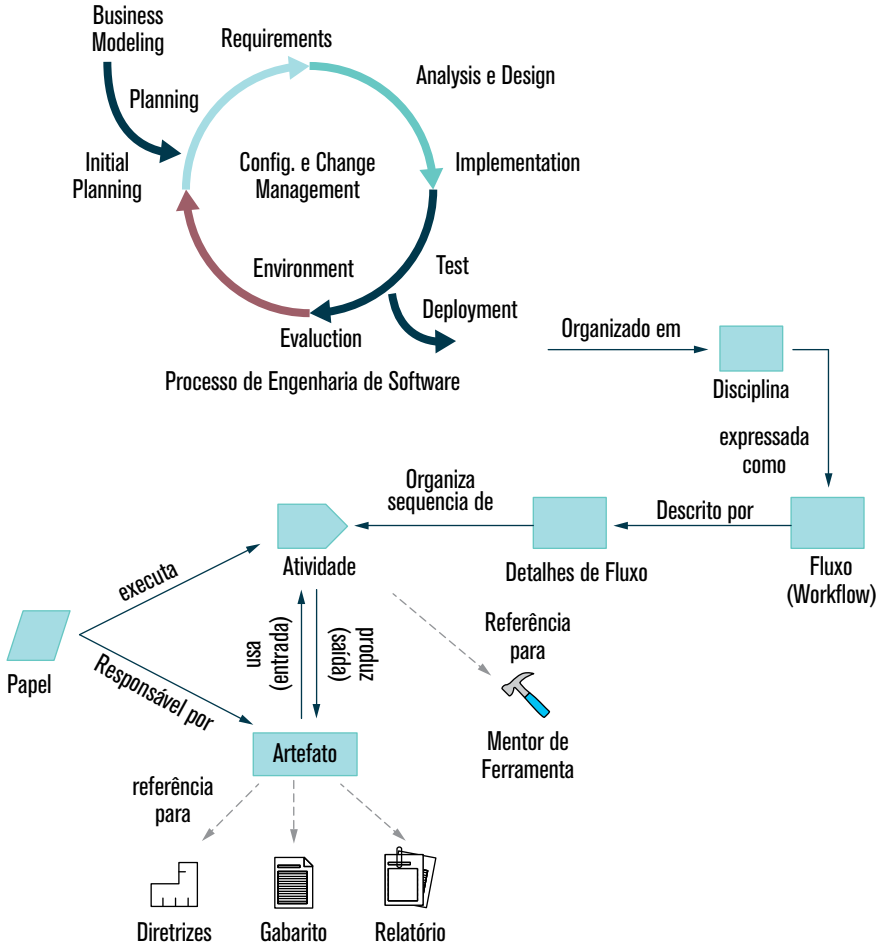


Figura 3.16 – Estrutura básica do RUP.

Papel

Define o comportamento e as responsabilidades de um indivíduo ou grupo de indivíduos trabalhando em equipe. O comportamento é determinado pelas atividades que o papel pode desempenhar. Por sua vez, as responsabilidades são explicitadas através dos artefatos que o papel tem que produzir. São exemplos de papéis do RUP: programador, testador, gerente de projeto, arquiteto projetista, analista de sistema dentre outros.

Atividade

É um conjunto de passos, organizados em uma tarefa, realizado para produzir um resultado esperado no processo, geralmente tendo como resultado a produção de um artefato. Cada tarefa é realizada por indivíduos que representam, naquele momento, um papel determinado. São exemplos de atividades no RUP: Planejar uma iteração, encontrar casos de uso e atores, rever o projeto, executar um teste de performance, dentre outras.

Artefato

É o resultado final da atividade executado por um determinado papel, podendo ser um documento, um modelo, um código-fonte, um programa-executável etc. É importante deixar claro que cada artefato possui apenas um papel como sendo responsável pela sua produção, porém diferentes papéis podem usá-lo como insumo para a construção de outros artefatos. São exemplos de artefatos no RUP: plano de gerenciamento de requisitos, visão, glossário, documento de arquitetura dentre outros.

Fluxo de Trabalho

Não conseguirmos montar um processo de desenvolvimento de sistemas apenas com atividades, papéis e artefatos. É necessário pensar na sequência em que tudo deve ser realizado e produzido. Desta forma, é importante definirmos o fluxo de trabalho, que consiste na ordem em que as atividades são executadas. Fluxos de trabalho podem ser representados por diagramas de sequência, diagramas de colaboração e diagramas de atividades da linguagem UML.

O RUP utiliza três tipos de fluxos de trabalho: fluxos de trabalho principais, associados com cada disciplina; fluxos de trabalho de detalhe, para detalhar cada fluxo de trabalho principal; e planos de iteração, que mostram como a iteração deverá ser executada.

Disciplina

Como já discutido anteriormente no capítulo, uma disciplina é o conjunto de atividades que se inter-relacionam. O RUP possui nove disciplinas, divididas em

disciplinas do processo e de suporte. As disciplinas de processo são: modelagem de negócios, requisitos, análise e projeto, implementação, teste e distribuição. As de suporte são: configuração e gerenciamento de mudanças, gerenciamento de projeto, e ambiente.

Ciclo de vida do RUP

O ciclo de vida do RUP é dividido em quatro fases e nove disciplinas.

Fase de concepção (iniciação):

Nesta fase, o foco da equipe de desenvolvimento será nas disciplinas de Modelagem de negócios e Requisitos. Isso acontece porque as disciplinas acima estão relacionadas com a definição do problema e com a elicitação nas necessidades dos usuários frente ao *software* que irá ser produzido.

Apesar da maior parte do esforço se concentrar nas atividades de coleta e especificação de requisitos, neste momento as discussões acerca de uma arquitetura inicial são pertinentes para que seja possível analisar a viabilidade do projeto. É importante decidir de vale a pena continuar o projeto do ponto de vista financeiro e técnico. Da mesma forma, por exemplo, parte da camada de apresentação pode ser codificada e testada para colaborar com a descoberta de requisitos.

A principal meta da fase de concepção é o alinhamento das expectativas entre os *stakeholders*, onde o escopo é mapeado e os riscos iniciais identificados. Para projetos evolutivos, a fase pode ocupar um espaço menor no cronograma do projeto, porém, para novos produtos, esta etapa deve realizada de forma detalhada.

Objetivos da fase:

- **Definir o escopo do produto:** a partir do levantamento dos casos de uso, a equipe deve definir, junto aos usuários e clientes, quais são as necessidades, do ponto de vista funcional e não funcional, que devem estar presentes no sistema. É importante também deixar claro que não vai estar contido na solução final, chamado de não escopo;
- **Estimar os custos do projeto:** a partir da definição do escopo, a equipe deve estimar os custos de forma inicial já que neste momento ainda não há

insumos suficientes para uma definição mais precisa de quanto vai ser gasto com o projeto. O custo calculado pode ser revisto a cada iteração;

- **Estimar tempo do projeto:** da mesma forma, após a definição do escopo, a equipe deve planejar o cronograma do projeto, tendo em mente que será uma atividade contínua, onde o cronograma Serpa atualizado a cada iteração;
- **Estimar riscos do projeto:** a equipe deve analisar os casos de uso e cenários mais significativos do ponto de vista arquitetural para que riscos sejam levantados e documentados;
- **Propor arquitetura inicial:** a equipe deve propor pelo menos uma arquitetura básica para que a viabilidade técnica seja analisada.

Principais artefatos produzidos:

- **Documento de Visão (disciplina de Requisitos):** Documento de alto nível que descreve as necessidades, característica e restrições mais importantes do sistema, sem entrar em detalhes técnicos. Serve de base para o contrato;
- **Caso de Negócio (disciplina de Gerenciamento de Projetos):** Documento que contém informações do ponto de vista do negócio para que seja analisada a viabilidade financeira a partir do retorno de investimento (ROI). Contém as estimativas de custos;
- **Plano de Desenvolvimento do Software (disciplina de Gerenciamento de Projetos):** Documento análogo ao plano de projeto do PMBOK. Reúne todas as informações necessárias para o gerenciamento do projeto ao longo do processo de desenvolvimento do sistema;
- **Modelo de Caso de Uso:** Documento que determina o escopo do sistema a partir do mapeamento dos casos de uso do sistema;
- **Glossário:** Documento que reúne as definições ambíguas que podem acarretar em problemas de entendimento e clareza ao longo da construção do sistema.

Fase de Elaboração:

Na fase de elaboração, todas as disciplinas podem ser executadas, porém o foco vai ser a disciplina de Análise e *Design*. Neste momento do projeto, muitas atividades das disciplinas de modelagem de negócios e requisitos ainda estão sendo executadas. As atividades de implementação ocupam um espaço maior do que na fase anterior, já que neste momento é necessário que se tenha uma arquitetura

estabilizada. A meta principal da Elaboração é implementar os requisitos que possuem um grande impacto na arquitetura, além de desenvolver uma arquitetura executável.

Objetivos da fase:

- **Definir uma arquitetura executável e estável:** a partir desta fase, é importante que haja um mapeamento entre a solução escolhida e os requisitos identificados até o momento, inclusive os requisitos não funcionais;
- **Tratar os riscos identificados do ponto de vista de projeto:** neste momento, muitos dos riscos identificados são tratados, já que a solução para o problema abordado começa a ser projetada. Essa ação é realizada a partir da estabilização da arquitetura e a implementação dos casos de uso mais significativos;
- S**elecionar componentes:** a equipe deve selecionar componentes a serem utilizados visando o reuso;
- **Criar planos de iterações para a fase de Construção:** a equipe deve planejar como os requisitos serão implementados e testados ao longo das iterações.

Principais artefatos produzidos:

- **Protótipos:** são utilizados para modelar visualmente os requisitos. Como consequência, atua na validação de requisitos já especificados e na descoberta de novas necessidades.
- **Documento de Arquitetura de *Software*:** documento que une as visões do RUP discutidas anteriormente neste capítulo.
- **Modelo de Projeto:** são os diagramas comportamentais e de interação da UML que descrevem a realização dos casos de uso, mostrando como implementá-los. É uma abstração do modelo de implementação e código-fonte.
- **Modelo de Dados:** modelo de implementação que descreve a representação conceitual, lógica ou física dos dados persistentes no sistema.

Fase de Construção:

A fase de construção é marcada pelas disciplinas de implementação e testes. Apesar das demais disciplinas poderem ser executadas, neste momento espera-se

que a maior parte dos requisitos já tenha sido identificada e modelada. Da mesma forma, é importante que a arquitetura do sistema já tenha sido definida.

Pela natureza da fase, percebe-se um alto paralelismo da equipe a partir do uso de diversas iterações. Por este motivo, a disciplina de gerenciamento de configuração e mudanças também é muito utilizada.

A principal meta da fase é prover a implementação e os testes do projeto. Caso haja indefinições nos requisitos, este é o momento para tirar dúvidas com os clientes/usuários. A arquitetura pode ser evoluída.

Como estudado em capítulos anteriores, a documentação externa é de fundamental importância para o entendimento do produto. Desta forma é importante que manuais de treinamento sejam construídos. Apesar destes artefatos terem início na fase de elaboração, é na construção que são produzidos de fato.

Objetivos da fase:

- **Minimizar custos de desenvolvimento, otimizar recursos e evitar retrabalho:** a codificação deve ser realizada com base nas boas práticas de programação do RUP;
- **Disponibilizar versões executáveis do programa:** a equipe deve liberar versões de testes intermediárias para validar a codificação realizada. São comuns versões de testes alfa, por exemplo;
- **Concluir a análise, o projeto, o desenvolvimento e o teste de todas as funcionalidades:** a fase de construção marca o fim do desenvolvimento do produto, sendo o momento certo para encerrar atividades que servem de insumo ou que são geradas pela codificação;
- **Verificar e decidir se o *software* está pronto para implantação:** ao final da construção do produto, é necessário que a equipe garanta que o sistema apresenta os requisitos elicitados com a ajuda dos usuários, tanto do ponto de vista funcional quanto do ponto de vista das características e restrições que o *software* deve ter.

Principais artefatos produzidos:

- **Código executável:** ao final da fase, a equipe deve disponibilizar o sistema executável, pronto para ser testado;
- **Plano de Implantação:** a equipe deve produzir a versão inicial de um plano que orienta a implantação do sistema;

• **Testes:** a equipe deve produzir e executar um conjunto de testes que garantam a estabilidade do sistema construído.

Fase de Transição

Na fase de transição, a meta principal é disponibilizar o sistema em um ambiente real do usuário. Desta forma, a principal disciplina executada é a de implantação. As demais disciplinas do RUP acabam sendo pouco utilizadas nesta fase, já que se espera que o produto esteja todo codificado e testado. Apesar da ênfase ser na disciplina de implantação, a disciplina de gestão de configuração e mudança também é bastante utilizada.

A duração desta fase tem uma grande variação a depender da complexidade do sistema construído. Normalmente projetos maiores, com requisitos instáveis, devem alocar mais tempo no cronograma para a correção de defeitos encontrados e a disponibilização de uma versão estável para o usuário.

Objetivos da fase:

• **Disponibilizar o *software* ao usuário final:** a equipe deve garantir que o produto construído seja implantando no ambiente real dos usuários;

• **Obter *feedback* do usuário:** essa ação faz referência à lapidação do sistema, devendo os ajustes serem mínimos, normalmente referentes a questões de usabilidade e desempenho. O aceite final pode ser obtido a partir de testes Beta;

• **Treinar os usuários e a manutenção:** com o sistema pronto e implantado, a depender da complexidade do negócio, os usuários deverão ser treinados. Da mesma forma, os detalhes referentes ao desenvolvimento de todo o sistema devem ser repassados para a equipe de manutenção, que ficará responsável por adicionar futuras funcionalidades além de corrigir erros encontrados após o fim do projeto.

Principais artefatos produzidos:

• **Notas de Release:** este artefato é importante na transparência das informações relacionadas às mudanças que ocorrem entre as versões do sistema. O objetivo é deixar claro o que mudou de uma versão para outra, quais os erros que foram corrigidos e quais são as novas funcionalidades;

• **Artefatos de Instalação:** a equipe deve construir uma orientação para que a instalação do sistema seja bem-sucedida;

• **Material de Treinamento:** a equipe deve prover material de treinamento, guias de e manuais do sistema para que os usuários possam utilizar o produto de forma satisfatória.

O RUP nos dias de hoje

Nunca se ouviu tanto falar em desenvolvimento ágil como nos últimos tempos. A ideia é que metodologias como o XP e Scrum podem resolver problemas persistentes na produção de *software* com qualidade. Nesse universo, falar sobre o RUP pode aparentar um retrocesso, mas precisamos analisar melhor o panorama para não tirar conclusões precipitadas.

É preciso fazer a seguinte pergunta: o RUP de fato é uma metodologia ultrapassada ou não estamos utilizando a ferramenta da forma correta? O fato é que nenhum arcabouço de processo deve prometer a resolução de todos os problemas que as organizações vêm enfrentando desde a década da crise do *software.* Certamente as metodologias ágeis também não são a solução para todos os problemas.

Como estudado neste capítulo, a ferramenta RUP é baseada em três conceitos básicos: desenvolvimento incremental, utilização de iterações e customização. Muitas equipes não utilizam o poder do RUP ao, por exemplo, utilizar de forma errada o conceito de incrementos. Não basta dividir o projeto em entregas se você não utiliza os usuários para validar os módulos construídos. Também é importante planejar o que vai ser entregue neste período de tempo para que a release seja caracterizada por uma entrega de valor, ou seja, que o pedaço do sistema entregue represente parte dos sistemas que o usuário deseja utilizar.

Nesse contexto, o conceito de iterações também não é bem utilizado por grande parte da indústria de TI. A ideia é que utilizando esse mecanismo, o projeto seja dividido em miniprojetos, onde é mais fácil gerenciar riscos e aceitar as fatídicas mudanças. Mais uma vez o arcabouço acaba perdendo força, e pode não se mostrar tão eficiente quanto esperamos.

O conceito de customização também é quase sempre esquecido, o que faz com que as organizações acreditem que tenham que produzir todos os artefatos descritos no RUP, tornando o processo burocrático para muitos contextos. A ideia

é que, como arcabouço de processo, as equipes analisem o que é necessário ser realizado para cada tipo de projeto. Certamente em projetos complexos, muito do que está descrito no RUP será utilizado. O mesmo não deve acontecer em projetos menores, onde provavelmente o número de artefatos necessários será menor.

O RUP é uma ferramenta poderosa e continua podendo ser utilizada, desde que seja utilizada da forma correta, nos contextos adequados. Além disto, o RUP pode ser aplicado em conjunto com metodologias ágeis, mesclando os conceitos, fortalecendo o conceito de documentação, buscando o atendimento das necessidades dos usuários.

SAIBA MAIS

Para entender melhor a comparação entre as novas metodologias ágeis e o RUP, leia o artigo a seguir: <http://www.ateomomento.com.br/scrum-rup/>.

ATIVIDADES

01. Quais as principais características do arcabouço de processo RUP?

02. A visão estática do RUP prioriza as atividades que ocorrem durante o processo de desenvolvimento. Na descrição do RUP, essas são chamadas de *workflows*. Existem seis workflows centrais, identificadas no processo e três de apoio, dentre os quais é possível citar os *workflows* de:

a) Meio ambiente e Gerenciamento de projeto.
b) Concepção e Construção.
c) Transição e Iteração.
d) Plano de desenvolvimento e Conceito de operação.
e) Plano de desenvolvimento e Conceito de operação.

03. O RUP usa a abordagem da orientação a objetos em sua concepção e é projetado e documentado utilizando a notação UML *(Unified Modeling Language*) para ilustrar os processos em ação. O objetivo da disciplina de análise e projeto é:

a) modelar como o sistema deve ser implementado.

b) mostrar como o sistema pode estabelecer requisitos.
c) controlar a execução do desenvolvimento.
d) estabelecer metodologias de análise decorrentes de projetos.
e) mostrar como o sistema pode especificar o projeto.

04. No RUP (*Rational Unified Process*), casos de uso são
a) casos de usuários unificados em processos de racionalização.
b) cenários de utilização do sistema por usuários.
c) cenários de racionalização de aplicações.
d) casos de utilização do RUP para maior racionalidade na aplicação dos recursos.
e) cenários de utilização compartilhada de soluções por usuários de maior racionalidade.

05. Você usaria um processo baseado no RUP para desenvolver um sistema *Web*?

REFLEXÃO

Neste capítulo, estudamos o arcabouço de *software* RUP. Foi apresentado todo o seu ciclo de vida, a partir da definição das suas fases e dimensões, e a descrição dos seus elementos básicos. Além disto, discutimos sobre as boas práticas de programação difundidas pela ferramenta.

Entendemos que a ferramenta em questão não é obsoleta, podendo ser utilizada em diversos contextos, inclusive em parceria com metodologias ágeis. Todos estes conhecimentos, certamente serão imprescindíveis para sua vida profissional.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de RUP, consulte a sugestão de artigos a seguir:

▪ Souza, R., Vasconcelos, A. Uma Extensão do Fluxo de Análise e Projeto do RUP para o Desenvolvimento de Aplicações Web. 2003. Disponível no site:< http://www.lbd.dcc.ufmg.br/colecoes/sbes/2003/0017.pdf>.

- Descrição do RUP pela IBM disponível no site:<ftp://public.dhe.ibm.com//software/pdf/br/RUP_DS.pdf>.
- Capítulo 2 do livro PRESSMAN, R. S. Engenharia de Software. 6ª Edição. Editora McGraw Hill, 2011.

REFERÊNCIAS BIBLIOGRÁFICAS

SOMMERVILLE, I. **Engenharia de *Software***. 9ª Edição. Editora Pearson, 2011.

PRESSMAN, R. S. **Engenharia de *Software***. 6ª Edição. Editora McGraw Hill, 2011.

4

Introdução à metodologia ágil

Introdução à metodologia ágil

Cada vez mais percebemos um aumento da pressão de mercado nas organizações em relação à necessidade por inovação, maximização da produtividade, entregas mais rápidas, minimização dos custos, dentre outros fatores que fazem com que as empresas apresentem uma maior vantagem competitiva frente a seus concorrentes. Nesse contexto, dominado por projetos que geralmente não obtém sucesso, seja por atrasos ou até mesmo cancelamento, surgem as metodologias ágeis, com o propósito de modificar a forma de desenvolvimento de sistemas, apostando na satisfação do cliente como maior prioridade.

A partir de agora estudaremos os principais conceitos das metodologias ágeis, dando ênfase neste capítulo ao *framework* XP.

OBJETIVOS

- Entender a importância do manifesto ágil;
- Mostrar como os princípios Lean podem ser aplicados em abordagens de desenvolvimento de software ágil;
- Apresentar os valores, papéis, princípios e práticas da metodologia XP;
- Entender o ciclo de vida do XP.

O manifesto ágil

As abordagens ágeis são baseadas no modelo iterativo e incremental, onde o projeto é dividido em miniprojetos, como discutido no capítulo anterior.

Sommerville

Os métodos ágeis são métodos de desenvolvimento incremental em que os incrementos são pequenos e, normalmente, as novas versões do sistema são criadas e disponibilizadas aos clientes a cada duas ou três semanas. Elas envolvem os clientes no processo de desenvolvimento para obter *feedback* rápido sobre as evoluções dos requisitos. Assim, minimiza-se a documentação, pois se utiliza mais a comunicação informal do que reuniões formais com documentos inscritos.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 39.

Dessa forma, a ideia é dividir o problema em problemas menores, visando um aumento da interação com o cliente a partir da entrega contínua de *software* funcionando. Por esta razão, as metodologias ágeis apresentam um forte ganho às organizações que estão situadas em um contexto caótico, como apresentado na figura 4.1.

Um ambiente caótico é caracterizado por requisitos e tecnologia não totalmente conhecidos. Neste contexto, as mudanças ao longo do projeto são inevitáveis, sendo necessária a adoção de metodologias que aceitem melhor essas mudanças, como o *framework Scrum*.

Por sua vez, o ambiente previsível é caracterizado pela presença de requisitos e tecnologia conhecidos e estáveis. Projetos nestes contextos são facilmente adaptáveis às metodologias com processo definido, como o *framework* RUP.

Vale a pena destacar que, em um ambiente anárquico, marcado por requisitos e tecnologia desconhecidos, mesmo a utilização de um processo ágil não irá garantir o sucesso do produto desenvolvido.

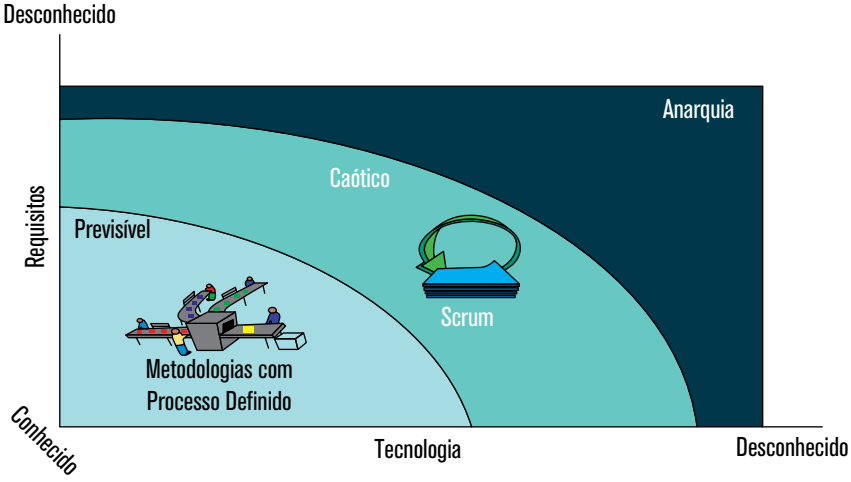


Figura 4.1 – Contextos de projetos versus processos de *software*.

Mas como as metodologias ágeis diminuem os problemas relacionados à um ambiente caótico de desenvolvimento de *software*? A resposta está atrelada a duas questões: aumento da comunicação e entrega de valor.

Nas abordagens ágeis a comunicação deve ser contínua e face a face, fazendo com que os riscos associados às incertezas do projeto sejam minimizados. A entrega de valor faz com que em um curto espaço de tempo a equipe disponibilize

software funcionando com base nas prioridades do cliente. Como consequência, a resposta à mudança torna-se natural, aumentando a satisfação do cliente. Em resumo, podemos afirmar que o principal objetivo de qualquer metodologia ágil é entregar um produto que seja útil ao cliente e que tenha qualidade.

Além do aumento de comunicação e da entrega de valor, podemos perceber diversas outras vantagens das metodologias ágeis, tanto para o cliente, quanto para as equipes de desenvolvimento, como mostrado a seguir.

- Entrega de produto de forma contínua e rápida;
- Aumento da transparência dentro do projeto;
- Facilidade na adequação de novos requisitos e na priorização dos requisitos já identificados;
- Aumento da qualidade do produto final;
- Melhora da produtividade;
- Minimização dos riscos atrelados ao projeto;
- Escopo do projeto claro e detalhado no momento correto;
- Equipes autogerenciáveis, comprometidas e mais motivadas;
- Maior adaptação do processo ao contexto do negócio;
- Aumento da verificação e validação do produto;
- Antecipação dos problemas e maior agilidade na tomada de ações.

Então a discussão sobre agilidade é recente? Não! De acordo com Sommervile (2011, p. 40), a insatisfação com a utilização de metodologias baseadas fortemente em especificação data da década de noventa, sendo que em 2001 diversos autores assinaram o **manifesto ágil,** que consiste em uma declaração que lista doze princípios do *software* ágil.

A partir da análise do manifesto ágil, podemos derrubar alguns mitos que rondam a cultura ágil de desenvolvimento. Muitos desenvolvedores acreditam que, em uma abordagem que prioriza entregas rápidas, os processos, ferramentas, documentação, contratos e planos são dispensáveis. Nesta ótica, a utilização desses itens é atrelada à diminuição da produtividade da equipe e deve ser evitada. Porém essa afirmação é falsa! Como mostrado na tabela 4.1, os processos, ferramentas, documentação, contratos e planos continuam sendo importantes para o projeto, mas não se mostram mais importantes do que os indivíduos e interações entre eles, *software* (ou produto) em funcionamento, colaboração com o cliente e responder a mudanças.

Indivíduos e interações	sobre	Ferramentas e processos
Software funcionanado	sobre	Documentação extensa
Colaboração com o cliente	sobre	Negociação de contratos
Responder à mudanças	sobre	Seguir um plano

Tabela 4.1 – Valores de abordagens ágeis.

Manifesto ágil

Estamos descobrindo maneiras melhores de desenvolver *softwares*, fazendo-o nós mesmos e ajudando outros a fazerem o mesmo. Através deste trabalho, passamos a valorizar: Indivíduos e interações mais que processos e ferramentas, *software* em funcionamento mais que documentação abrangente, colaboração com o cliente mais que negociação de contratos e responder a mudanças mais que seguir um plano. Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

FONTE: AGILE MANIFESTO, Manifesto *for Agile Software Development*, 2001. Disponível em: <http://agilemanifesto.org/>. Acesso em: ago. 2016.

Valorizar os indivíduos e interações significa enfatizar a importância da cultura organizacional das pessoas envolvidas no projeto em detrimento do processo e ferramentas utilizadas. Neste ponto de vista, fica claro que apesar da importância da organização das tarefas que levam à construção do produto final, os membros da equipe, com suas características e talentos individuais, é que geram o *software*. O indivíduo é visto como algo único, que não pode ser simplesmente trocado sem acarretar em perdas para o projeto.

Neste contexto, uma maior interação entre os indivíduos que formam o time de desenvolvimento é essencial para que o objetivo do projeto seja alcançado. O aumento da comunicação faz com que haja uma maior transparência e, como consequência, seja necessária a elaboração de um número menor de artefatos, em comparação a *frameworks* como o RUP. De acordo com Larman (2003) e Highsmith (2002), as ferramentas utilizadas ao longo do desenvolvimento devem ser simples e eficazes, assim como o processo deve ser guiar e apoiar o trabalho do time, porém o mesmo deve se adaptar à equipe e não o contrário.

Com relação à documentação, podemos verificar que as abordagens ágeis priorizam o ***software* em funcionamento** em detrimento da criação e manutenção de artefatos. Este fato é consequência da importância da entrega contínua para o cliente. A documentação se mostra útil ao desenvolvimento do produto, porém a sua excessiva elaboração pode acarretar numa menor produtividade da equipe como resultado de uma difícil manutenção. No ponto de vista ágil, a satisfação do cliente é mais facilmente atingida a partir da entrega de resultados (código executável), e não da geração de documentação. Highsmith (2002) afirma que a constante entrega de *software* funcionando faz com que o *feedback* do cliente seja parte importante do desenvolvimento, o que não acontece na mesma proporção com documentação.

Por falar em *feedback*, outro importante valor presente é a **colaboração com o cliente**. No desenvolvimento ágil o cliente faz parte do time de desenvolvimento, colaborando com o cumprimento do objetivo final, a entrega do produto. O seu envolvimento deve ser constante, já que ele é o maior conhecedor do negócio e deve ajudar a equipe a produzir *software* de valor. Desta forma, a participação do cliente deve ser colaborativa e o seu poder de decisão deve ser forte, ao ponto de tornar contratos desnecessários. Em certos contextos, marcados por volatilidade e incerteza, contratos continuam sendo interessantes, porém a negociação amigável será sempre a melhor saída. Transformar o cliente em um inimigo e se apoiar apenas em contratos não é a melhor estratégia para obter sucesso no projeto.

Ainda sobre os principais valores ágeis da tabela 4.1, **responder às mudanças** é mais interessante do que seguir um plano. O planejamento do projeto continua sendo importante, porém temos que ter a maturidade de entender que as necessidades dos clientes mudam com muita frequência. Além disso, a maioria dos contextos é marcada por incerteza que fazem com que premissas sejam adotadas ao longo do desenvolvimento que nem sempre se mostram verdadeiras. Desta forma, podemos afirmar que mudanças são inerentes ao desenvolvimento de *software*, e a equipe deve ser capaz de acomodá-las de forma natural. Sempre é melhor mudar a rota do que chegar ao destina errado. É isso que fazendo o tempo todo quando estamos construindo *software*.

Além dos valores discutidos, o manifesto ágil apresenta também doze princípios importantes no contexto de desenvolvimento rápido de software, que podem ser vistos na tabela a seguir.

PRINCÍPIOS DO MANIFESTO ÁGIL
Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de *software* de valor.
Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adéquam a mudanças, para que o cliente possa tirar vantagens competitivas.
Entregar *software* funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos.
Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto.
Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho.
O Método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara.
Software funcional é a medida primária de progresso
Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes.
Contínua atenção à excelência técnica e bom *design*, aumenta a agilidade.
Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito.
As melhores arquiteturas, requisitos e designs emergem de times auto-organizáveis.
Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo.

Tabela 4.2 – Princípios do manifesto ágil.

O primeiro princípio está ligado à satisfação do cliente a partir da entrega contínua. Como já discutimos anteriormente, a principal forma de verificar se o cliente está satisfeito com o projeto é fazendo com que ele receba de forma antecipada pequenas partes executáveis do programa. Essa entrega deve ser de forma contínua para permitir a constante validação do produto e assim acomodar melhor possíveis mudanças e correções. Essa estratégia faz com que a qualidade do software seja constantemente monitorada.

A necessidade de adequação às mudanças está preconizada no segundo princípio. Na maioria dos contextos de desenvolvimento de sistemas a necessidade dos clientes não permanece a mesma no início até o final do projeto. É comum que novas funcionalidades sejam percebidas, como também funcionalidades já especificadas sejam revistas. Fazer com que o produto se adapte a estas mudanças faz com que o *software* esteja sempre em sintonia com as necessidades reais dos *stakeholders*, fazendo com que os mesmos obtenham vantagem competitiva frente aos concorrentes.

Da mesma forma que a adaptação às mudanças, entregar frequentemente software funcionando faz com que haja um aumento de vantagem competitiva a partir do momento que o cliente torna-se parte da equipe ao avaliar constantemente o produto que está sendo construído. Prováveis problemas no *software* podem ser antecipados e novos requisitos podem ser elicitados, mantendo sempre um alinhamento de expectativas entre a equipe o cliente. A necessidade de uma maior interação entre todos os envolvidos no projeto está clara no quarto princípio. Todos têm o mesmo objetivo, que é obter um software com qualidade e utilidade. Assim, os envolvidos devem trabalhar como um time, diariamente.

Como consequência da necessidade do trabalho em equipe, o quinto princípio fala sobre a motivação dos envolvidos ao longo do projeto. É importante que cada integrante saiba que o seu papel é importante e faz diferença no trabalho em conjunto. Neste contexto, podemos verificar que, em projetos ágeis, cada envolvido é visto como uma peça única que produz resultados únicos que formam o conjunto final.

No trabalho em equipe, é necessário que a comunicação seja constante e clara. As informações devem ser passadas com transparência e nenhum meio se mostra mais eficaz do que a conversa cara a cara. Por mais que hoje em dia temos as facilidades proporcionadas por diversas tecnologias, a comunicação presencial será sempre mais a melhor escolha. Por esta razão, preferencialmente os projetos ágeis devem ser realizados com equipes geograficamente não distribuídas.

Por sua vez, o sétimo princípio determina que *software* funcionando é a melhor métrica de sucesso do projeto. Desta forma podemos afirmar que por mais

que a documentação tenha valor, entregas constantes de partes executáveis do programa se configuram como importantes insumos para a medição do progresso do trabalho realizado. Atrelado a esse fato, o oitavo princípio afirma que o ambiente constante de trabalho promovido por abordagens ágeis faz com que todos os *steakholders* trabalhem com mais segurança e tranquilidade.

O nono princípio aborda a importância da excelência técnica e bom *design* para o aumento da agilidade. Quanto mais a equipe se torna madura em relação às boas práticas de desenvolvimento, mais *software* funcionando consegue ser entregue ao cliente, maximizando muitas dos princípios ágeis discutidos até o momento. Como consequência da excelência técnica, muito trabalho que não se mostra útil em relação às necessidades dos usuários deixa de ser realizado, o que aumenta a produtividade da equipe ao concentrar os esforços nos requisitos realmente importantes. Este fato está descrito no décimo princípio.

O décimo primeiro princípio aborda a questão de a importância da equipe possuir um time auto-organizado, onde todos possuem importância e são livres para desempenhar suas habilidades para que o objetivo do projeto seja cumprido. Neste sentido, não é obrigatório que todos os integrantes do time dominem todos os aspectos do desenvolvimento do sistema em questão. Porém, é importante que a equipe consiga resolver o problema sem necessitar de ajuda externa. A forma de trabalho individual e coletiva pode ser revista de tempos em tempos, como explicitado no décimo segundo princípio.

Lean Software Development

O desenvolvimento enxuto de *software* é uma prática ágil, focada em estratégias de negócio e de gerenciamento de projetos, que tem como base os conceitos desenvolvidos na manufatura pela empresa japonesa *Toyota* de automóveis, chamado de "*Lean Manufacturing*". O termo "*Lean Software Development"* vem do livro " *Lean Software Development:* Na *Agile Toolkit*", escrito por Tom & Mary Poppendieck em 2003. O livro apresenta sete princípios e vinte e duas ferramentas que encapsulam o processo de Lean.

Franco

A história da produção enxuta iniciou-se no Japão com a família *Toyoda* e a empresa *Toyota Motor Company*, fundada em 1937. Após ter passado por períodos difíceis no final da década de 1930, época em que foi obrigada pelo governo a produzir caminhões militares, com métodos em grande parte artesanais, a *Toyota* resolveu ingressar na fabricação em larga escala de carros e caminhões comerciais, porém deparou com uma série de problemas.

FONTE: FRANCO, Eduardo Ferreira. Um modelo de gerenciamento de projeto baseado nas metodologias ágeis de desenvolvimento de *software* e nos princípios da produção enxuta. São Paulo, 2007. Página 39.

O sistema *Toyota* de produção foi criado no Japão após a segunda guerra mundial. O seu maior objetivo era organizar a produção japonesa em um cenário onde a produtividade estava em baixa e os recursos estavam escassos, o que impossibilitava a produção em massa. O princípio fundamental desta abordagem tem como base a eliminação de desperdício na manufatura de um determinado produto. Os outros princípios são: amplificar o aprendizado, tomar decisões o mais tarde possível, fazer entregas o mais rápido possível, tornar a equipe responsável, construir integridade e visualizar o todo.

Eliminar perdas

Podemos entender desperdício como tudo que não gera valor para o cliente, como fazer alguma atividade que não é necessária no momento, movimentação, transporte, espera, processamento extra, dentre outros. No contexto de manufatura, a perda pode ocorrer, por exemplo, a partir do momento que uma fábrica produz mais do que é necessário. No contexto de desenvolvimento de sistemas, o desperdício pode ser percebido nas situações em que a equipe desenvolve funcionalidades que não serão úteis para o cliente.

Levando em consideração essa problemática, Shingo (1981) identificou sete tipos de desperdícios em manufaturas: estoque, processamento extra, superprodução, transporte, espera, movimentação e defeitos. Por sua vez, Mary e Tom Poppendieck (2003) traduziram os princípios para o contexto de desenvolvimento de sistemas, como exposto na tabela a seguir.

PERDAS NA MANUFATURA	PERDAS NO DESENVOLVIMENTO DE SOFTWARE
Estoque	Trabalho parcialmente pronto
Processamento extra	Processo extra
Superprodução	Funcionalidades extras
Transporte	Chaveamento de tarefas
Espera	Espera
Movimento	Movimento
Defeitos	Defeitos

Tabela 4.3 – Desperdícios na manufatura e no desenvolvimento de *software*.

O primeiro desperdício catalogado por Mary e Tom Poppendieck (2003) faz referência ao ***software* parcialmente pronto** no desenvolvimento de sistemas. De acordo com os autores, sistemas inacabados tendem a não terem utilidade para o cliente, ficando obsoletos. Com esse cenário, a equipe não consegue garantir que o sistema vai funcionar da forma como esperam e, consequentemente, se irá atender às necessidades de negócio do cliente. Além disto, *softwares* inacabados podem minar com os recursos que poderiam estar sendo melhores utilizados. Desta forma, diminuir a quantidade de sistemas inacabados reduz riscos e perdas para o projeto.

Outro problema levantado pelos autores é o excesso de documentação (**processos extra**) requerida por processos de desenvolvimento de sistemas. Documentação consome recursos, diminui o tempo de resposta, esconde problemas de qualidade, se perde, torna-se degradada e obsoleta. Desta forma, somente documentos que sejam impeditivos na realização de tarefas devem ser produzidos. Estes devem ser vistos como artefatos que agregam valor ao produto e não trabalho a mais que ninguém irá consumir. Como exemplo, podemos citar a especificação de requisitos: uma alternativa à escrita do documento de requisitos é a escrita de testes com o foco nas necessidades do cliente. Desta forma, os testes passam a ser a própria documentação do que o cliente espera que o sistema realize e se comporte.

SAIBA MAIS

Para entender melhor a utilização de testes integrados às regras de negócio, leia o artigo a seguir sobre BDD:

<http://www.devmedia.com.br/desenvolvimento-orientado-por-comportamento-bdd-artigo-java-magazine-91/21127>.

Não é muito incomum encontrar desenvolvedores que acrescentam requisitos, que o cliente não pediu, no sistema desenvolvido. Acrescentar **características extras** ao *software* em desenvolvimento parece inofensivo, mas configura perdas para o projeto. Cada pedaço de código inserido na aplicação adiciona mais complexidade e pode se tornar mais um ponto de falha. Todo o código deve ser inspecionado, compilado, integrado e testado. Dessa forma, acrescentar requisitos funcionais ou não funcionais ao produto, impacta na produtividade da equipe. Além disto, o código extra pode se tornar obsoleto antes de ser usado, já que não era uma necessidade do cliente.

Outro motivo de perdas apontado por Mary e Tom Poppendieck (2003) é a constante **mudança de tarefas** dos integrantes da equipe. Quando um desenvolvedor muda de tarefas, ele leva um tempo para assimilar o que deve feito, além de interromper algo que já estava encaminhado. Desta forma, os autores acreditam que atribuir pessoas a múltiplos projetos é uma fonte de perdas, sendo que o caminho mais rápido para completar dois projetos, com a mesma equipe, é fazê-los sequencialmente.

Com relação à **espera**, muito tempo de um projeto pode ser perdido enquanto a equipe aguarda que as coisas aconteçam, como: atraso para iniciar um projeto, atraso na alocação das pessoas envolvidas, atraso devido à excessiva documentação de requisitos, atraso nos testes e atrasos na implantação. A consequência dos atrasos que podem ocorrer no projeto é a demora na entrega de valor para o cliente, que como já discutido anteriormente, deve acontecer o mais rápido possível. Para alguns projetos, os atrasos podem não ter maiores problemas, mas para os projetos que tem como objetivo a construção de produtos competitivos, os atrasos podem acarretar perdas incalculáveis. Em um mercado altamente competitivo, se o concorrente lança um produto na frente, todo o projeto que sofreu atraso pode perder a viabilidade.

A figura 4.2 mostra a cadeia de valor e os atrasos no modelo de desenvolvimento cascata. Na parte superior, podemos perceber a linha do tempo referente ao trabalho que agrega valor ao projeto, representado pelas atividades de

requisitos, análise, projeto, codificação, testes e operação, além de atividades de interação com o cliente. Na parte inferior, vemos a linha do tempo relacionado com o desperdício.

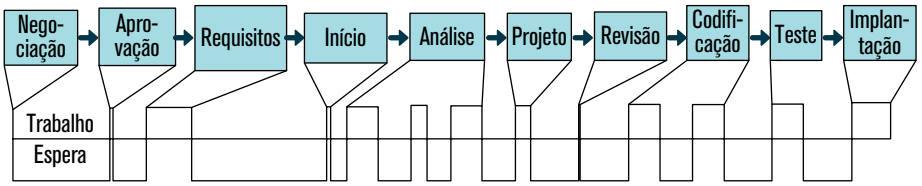


Figura 4.2 – Mapa de cadeia de valor do modelo cascata. (POPPENDIECK, M.; POPPENDIECK, T., 2003).

Podemos perceber que o período de espera apresentado na figura 4.2 é grande quando comparado à linha de trabalho que agrega valor ao projeto. O desperdício de tempo relacionado à espera é justificado pelas transições entre as etapas no modelo cascata, que exige a verificação dos artefatos produzidos e um encerramento formal.

Com o objetivo de diminuir o tempo de espera e maximizar o tempo de trabalho, o processo do desenvolvimento enxuto de *software* organiza as etapas de análise, projeto, codificação e testes dentro de iterações que devem ser realizadas pelas mesmas pessoas, não necessitando de um encerramento formal. A figura 4.3 representa a cadeia de valor e os atrasos presentes no desenvolvimento enxuto. A linha do tempo está dividida em duas regiões: a de trabalho e a de espera. Agora, podemos verificar que os tempos de espera são menores e menos frequentes, e os períodos de geração de valor são maiores, me comparação com a cadeia de valor do modelo cascata.

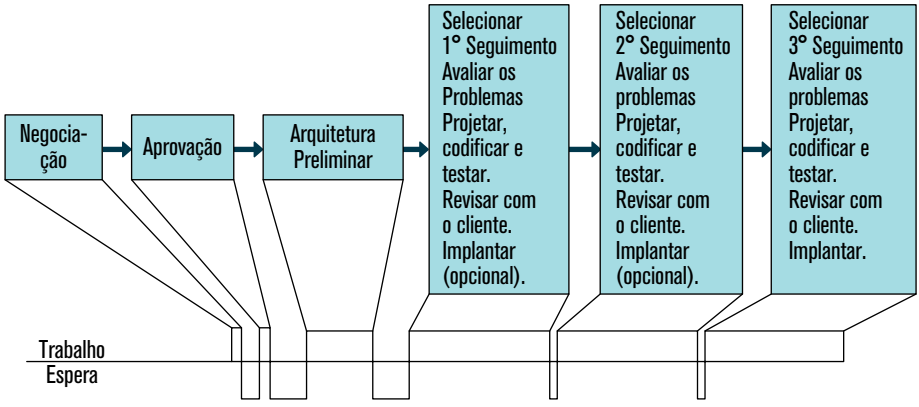


Figura 4.3 – Mapa da cadeia de valor do desenvolvimento enxuto de *software* – mapa de cadeia de valor do modelo cascata. (POPPENDIECK, M.; POPPENDIECK, T., 2003).

Por sua vez, o desperdício relacionado com o **movimento** faz alusão ao tempo de resposta para questões cotidianas do projeto. A equipe pode perder tempo se movimentando para obter as informações necessárias para o desenvolvimento do produto. Para evitar este problema, algumas questões devem estar resolvidas ainda nas fases inicias do projeto, como: Temos as pessoas certas para responder a perguntas técnicas? O cliente está facilmente acessível para discutir características do produto? Em projetos ágeis, este problema é minimizado a partir da utilização de ciclos curtos de entrega e aumento da comunicação entre os envolvidos.

Franco

As pessoas não são os únicos recursos que se movimentam, muitos artefatos também se movimentam. Cada movimentação de artefatos está repleta de oportunidades de desperdícios, o maior desperdício nestas movimentações é que os artefatos não possuem todas as informações que a próxima pessoa que o utilizará precisa para conduzir seu trabalho. Grande parcela do conhecimento tácito é mantida pelo criador do artefato e não é entregue ao receptor.

FONTE: FRANCO, Eduardo Ferreira. Um modelo de gerenciamento de projeto baseado nas metodologias ágeis de desenvolvimento de *software* e nos princípios da produção enxuta. São Paulo, 2007. Página 57.

Por fim, o desperdício ligado aos defeitos é determinado pelo impacto da detecção tardia de problemas no sistema. Temos que entender que um defeito crítico detectado em minutos não é um grande desperdício. Já um defeito menor não descoberto por semanas é um desperdício muito maior. Logo, a ideia é que defeitos sejam detectados o mais cedo possível no processo de desenvolvimento de *software*, a partir de iterações curtas.

Amplificar o aprendizado

Na manufatura, o aprendizado é obtido a partir da repetição de tarefas. Quanto mais o trabalhador realiza aquele trabalho de forma repetitiva, mais ele se torna especialista. No desenvolvimento de sistemas, a ideia é diferente, já que o produto final não é realizado a partir de repetições pré-definidas. Para que o aprendizado seja amplificado, é necessário utilizar de técnicas, como o *feedback* constante, utilização de iterações curtas e sincronização a partir de testes automatizados.

Tomar decisões o mais tarde possível

Este princípio do desenvolvimento enxuto está ligado à necessidade de retardar o máximo possível a tomada de decisão em projetos que são construídos em ambientes de incerteza. Quanto mais cedo as decisões são tomadas, maximiza-se a chance destas decisões terem resultados com base em especulação e não em fatos. No desenvolvimento de *software*, a melhor saída para a questão da tomada de decisão é projetar o sistema focando na adaptação às mudanças. Desta forma, decisões podem ser revistas a partir de alteração nos requisitos.

Fazer entregas o mais rápido possível

A entrega mais rápida de *software* faz com que a qualidade seja revista de forma contínua, além de proporcionar um mecanismo eficaz de realimentação de informações confiável, fazendo com que o ciclo de descoberta se mantenha também continuamente.

Além disso, a velocidade da entrega permite adiar possíveis decisões que ainda não estejam amadurecidas, fazendo com que partes críticas do produto sejam deixadas para serem executadas em um momento mais coerente, onde existam mais fatos para justificar as decisões, diminuindo os riscos associados.

Tornar a equipe responsável

Toda a equipe deve ser se envolver nas tomadas de decisões técnicas. Dificilmente líderes podem responder pelos membros da equipe, já que cada integrante possui o seu conhecimento técnico individual que se mostra necessário para que determinados domínios de problemas sejam resolvidos.

Franco

Pelo fato das decisões serem tomadas tardiamente e a execução ser conduzida de forma rápida, não é possível gerenciar as atividades dos trabalhadores através de uma autoridade central. As práticas enxutas utilizam as técnicas de produção puxada **(pull)** para agendar o trabalho e possuem mecanismos de sinalizações locais, de forma a permitir que outros trabalhadores identifiquem o trabalho que necessita ser realizado.

FONTE: FRANCO, Eduardo Ferreira. Um modelo de gerenciamento de projeto baseado nas metodologias ágeis de desenvolvimento de **software** e nos princípios da produção enxuta. São Paulo, 2007. Página 49.

No desenvolvimento enxuto de *software*, o envolvimento da equipe pode ser explicitado nos acordos de entregas de versões do produto, em datas predefinidas e regulares. De acordo com Franco (2007, p.49), "A sinalização local é feita através de gráficos visuais, reuniões diárias, integrações frequentes e testes automatizados".

A figura 4.4 exemplifica um quadro que pode ser utilizado para sinalização local, que pode também ser utilizado para nivelar e controlar o fluxo de produção.

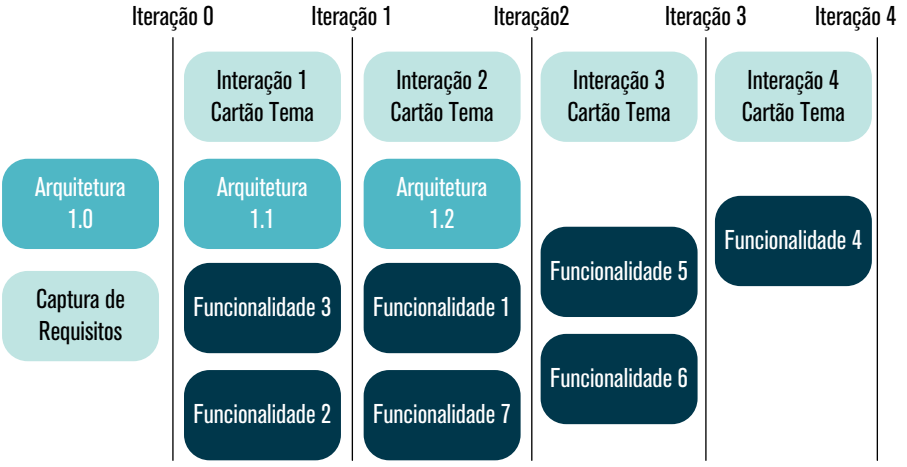


Figura 4.4 – Quadro de cartões de funcionalidades. (POPPENDIECK, M.; POPPENDIECK, T., 2003).

A figura 4.5 traz um exemplo de gráfico *burndown*, cujo objetivo é controlar o andamento das tarefas contidas na execução do projeto. Para este gráfico, a curva azul representa a estimativa de esforço necessário para finalizar as tarefas remanescentes na iteração atual e a linha amarela corresponde à quantidade de funcionalidades remanescentes para serem codificadas e incorporadas ao incremento do produto a ser entregue no fim da iteração.

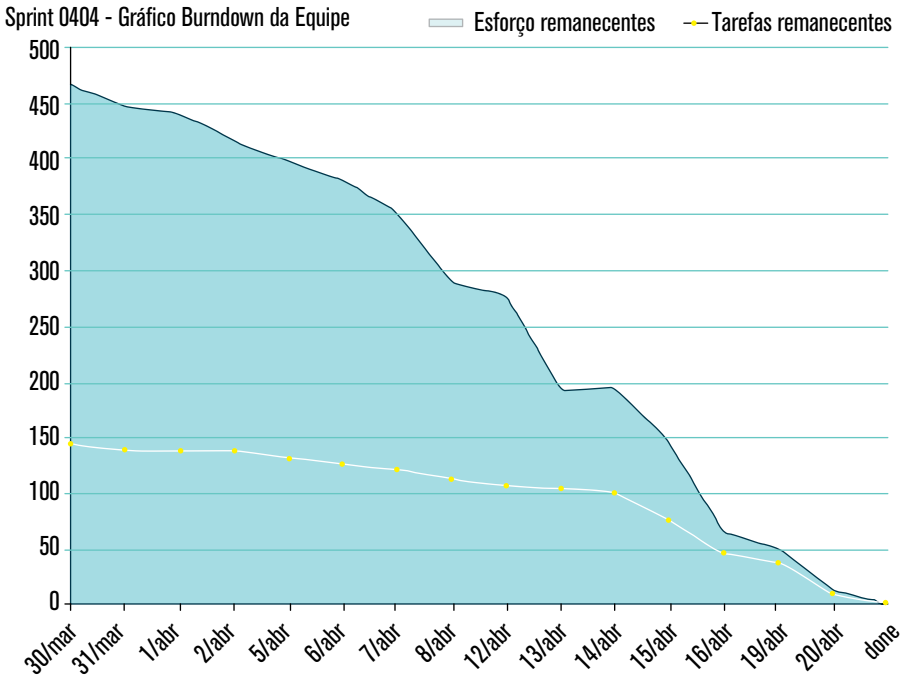


Figura 4.5 – Quadro de cartões de funcionalidades. (POPPENDIECK, M.; POPPENDIECK, T., 2003).

Construir integridade

A integridade do sistema é percebida a partir do funcionamento regular dos conceitos centrais do sistema, como arquitetura bem estruturada, boa usabilidade, utilidade, adaptabilidade e flexibilidade.

Visualizar o todo

O último princípio proposto pelo desenvolvimento enxuto de *software* está atrelado à necessidade de ver o produto como um todo e não como partes isoladas. É comum que os especialistas trabalhem com o foco apenas nas nuances que fazem parte do seu entendimento, o que pode dificultar a manutenção da integridade do sistema. O foco deve ser a busca pelo melhor desempenho do sistema como um todo e não de partes isoladas do sistema.

XP

Conceitos e Definições

Extreme Programming (XP) é uma metodologia ágil de desenvolvimento adaptativa e flexível, sendo indicada para ambientes caóticos, onde os requisitos mudam com frequência e tecnologia é desconhecida. Além disto, por ser dividida em pequenos ciclos de algumas semanas, que geram resultados capazes de agregar valor ao negócio do cliente, XP é uma boa saída para projetos que precisam realizar entregas para os clientes em curtos espaços de tempo.

Sommerville

Em *Extreme Programming*, os requisitos são descritos como cenários (chamados de histórias do usuário), que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes de escreverem o código. Quando o novo código é integrado ao sistema, todos os testes devem ser executados com sucesso. Há um curto intervalo entre os releases do sistema.

FONTE: SOMMERVILLE, I. Engenharia de *Software*. 9ª Edição. Editora Pearson, 2011, página 44.

A figura 4.6 mostra o ciclo de um *release* em *Extreme Programming*. Na abordagem em questão, inicialmente as histórias de usuários previamente especificadas são escolhidas para compor a release. As histórias de usuário são as descrições textuais das funcionalidades e devem ser definidas e priorizadas, com a ajuda do cliente, de acordo com as necessidades do negócio e o tempo e o custo da implementação.

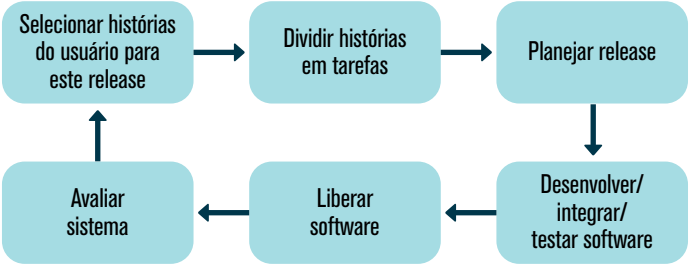


Figura 4.6 – Ciclo de um *release* em *Extreme Programming*. (SOMMERVILLE, I., 2011)

Na sequência, as histórias de usuário são detalhadas na forma de uma lista de atividades necessárias para seu funcionamento. No planejamento, as tarefas são estimadas e distribuídas entre as duplas da equipe para que sejam desenvolvidas, integradas e testadas. Se necessário, o código é refeito ou corrigido. Por fim uma nova versão executável já atualizada é disponibilizada para toda a equipe e posteriormente, avaliada pelos clientes.

Valores, Papéis, Princípios e Práticas do XP

No desenvolvimento de sistemas, é necessário que a equipe tenha em mente sempre o que realmente importa para que o produto seja produzido de forma satisfatória. Mas o que realmente importa na construção de um *software*? Nesse quesito podemos compilar diferentes opiniões sobre o tema: algumas pessoas da equipe podem achar que o que realmente importa é a documentação excessiva do projeto. Outros podem afirmar que o código é o artefato mais importante e deve ser o foco. Falando ainda sobre código, cada um na equipe pode revelar um estilo de programação que seja completamente diferente dos demais. Toda essa problemática pode fazer com que o time não caminhe em uma direção única na condução do projeto.

Para minimizar o problema, o arcabouço XP se baseia em cinco valores (comunicação, *feedback*, coragem, simplicidade e respeito) para guiar o desenvolvimento e, consequentemente, fazer com que a equipe se concentre no que realmente importa no contexto ágil.

A importância da **comunicação** em projetos é essencial a partir do momento em que os desenvolvedores devem compreender os anseios do cliente e os clientes devem também entender as limitações e desafios técnicos envolvidos no projeto. Neste contexto, o cliente apresenta um problema que deve ser solucionado com o sistema desenvolvido e possui as ideias sobre que funcionalidades podem fazer parte do sistema. Por sua vez, a equipe de desenvolvimento possui os conhecimentos técnicos que podem influenciar a solução para o problema do cliente.

O sucesso da comunicação entre os cliente e membros da equipe depende do canal utilizado. Preferencialmente, o diálogo presencial deve ser o canal de comunicação mais usado, já que este meio facilita a compreensão entre as partes a partir da utilização de elementos como gestos, expressões faciais, postura, palavras verbalizadas, tom de voz, emoções entre outros. Conversas cara-a-cara são sempre melhores do que videoconferências, telefonemas, *e-mails*, cartas ou *fax*.

Outro importante valor em XP é o ***feedback.*** As respostas às decisões tomadas devem ser rápidas e visíveis. Lavando isso em consideração, projetos desenvolvidos com XP devem observar os resultados obtidos a partir de uma tarefa executada o mais rápido possível. Um exemplo disso é a utilização de releases curtos para que o cliente observe mais rapidamente as funcionalidades pedidas. Além disto, os clientes, ao se manterem próximos, acabam validando e encontrando problemas mais cedo no processo de desenvolvimento.

Em ambientes caóticos, a mudança é uma constante. É muito comum que os clientes revejam as suas necessidades de tempos em tempos e, inevitavelmente, façam com que partes desenvolvidas do produto sejam revistas. Sobre este aspecto, a equipe de desenvolvimento tem duas opções: frear a criatividade do cliente e seguir com o planejado anteriormente ou acomodar a mudança. A primeira opção não é a ideal, já que, ao não adaptar o sistema às novas necessidades dos usuários, o produto final pode ser um *software* sem utilidade. Em outras palavras, o sistema pode se tornar obsoleto antes mesmo do seu uso.

Desta forma, a equipe deve aprender a lidar com o risco de mudanças como algo natural. Desta forma, XP prega o valor da **coragem**, como sendo o sentimento que o time tem em relação a essas mudanças. A equipe confia na acomodação de novas funcionalidades e na alteração de funcionalidades antigas a partir do momento que o arcabouço dispõe de mecanismos de proteção como desenvolvimento orientado a testes, programação em par e integração contínua.

Para que mudanças sejam fáceis de serem realizadas e principalmente para que as necessidades dos usuários sejam mais rapidamente atendidas, o XP prega o valor da **simplicidade**. Este valor está ligado diretamente ao conceito do desenvolvimento enxuto de *software*, baseado no método *Toyota* de produção, que afirma que não se deve produzir mais do que o necessário. *Extreme programming* utiliza o conceito de simplicidade para que a equipe se concentre em realizar o trabalho estritamente necessário, evitando fazer o que ainda não se provou como essencial.

Por fim, o **respeito** se mostra como um valor base, que envolve todos os demais já discutidos. Todos têm sua importância dentro da equipe e devem ser respeitados e valorizados para que juntos consigam atingir o objetivo final, que é a entrega de um produto com qualidade e que atenda às necessidades dos usuários.

O *Extreme Programming* aborda diversas práticas que refletem os princípios da metodologia ágil. Em projetos realizados com XP, o desenvolvimento deve ser incremental para que haja entregas frequentes. No arcabouço de processo, os incrementos são representados por pequenos releases. Além disto, a funcionalidade incluída é baseada em cenários especificados pelos usuários.

Releases curtas permitem que os clientes permaneçam envolvidos de forma contínua, especificando histórias do usuário e contribuindo com os testes de aceitação. As pessoas se tornam mais importantes do que o processo, sendo sustentadas por programação em pares, propriedade coletiva do código do sistema e um processo de trabalho que não envolve uma carga excessiva, como a utilização de horas extras. Como já discutido, as mudanças são acomodadas por meio dos releases contínuas e do desenvolvimento guiado por testes. Por fim, a refatoração constante aumenta a qualidade e promove simplicidade. Todas as práticas do XP estão resumidas na tabela a seguir.

PRÁTICA	DESCRIÇÃO
Planejamento incremental	Os requisitos são gravados em cartões de histórias e as histórias que serão incluídas em um release são determinadas pelo tempo disponível e sua relativa prioridade. Os desenvolvedores dividem essas histórias em tarefas.
Pequenos *releases*	Em primeiro lugar, desenvolve-se um conjunto mínimo de funcionalidades útil, que fornece o valor do negócio. *Releases* do sistema são frequentes e gradualmente adicionam funcionalidades ao primeiro release.
Projeto simples	Cada projeto é realizado para atender às necessidades atuais, e nada mais.
Desenvolvimento *test-first*	Um *framework* de testes iniciais automatizados é usado para escrever os testes para uma nova funcionalidade antes que a funcionalidade em si seja implementada.

PRÁTICA	DESCRIÇÃO
Refatoração	Todos os desenvolvedores devem refatorar o código continuamente assim que encontrarem melhorias de código. Isso mantém o código simples e manutenível.
Programação em pares	Os desenvolvedores trabalham em pares, verificando o trabalho dos outros e prestando apoio para um bom trabalho sempre.
Propriedade coletiva	Os pares de desenvolvedores trabalham em todas as áreas do sistema, de um modo que não se desenvolvam ilhas de expertise. Todos os conhecimentos e todos os desenvolvedores assumem responsabilidade por todo o código. Qualquer um pode mudar qualquer coisa.
Integração contínua	Assim que o trabalho em uma tarefa é concluído, ele é integrado ao sistema como um todo. Após essa integração, todos os testes de unidade do sistema devem passar.
Ritmo sustentável	Grandes quantidades de horas-extras não são consideradas aceitáveis, pois o resultado final, muitas vezes, é a redução da qualidade do código e da produtividade a médio prazo.
Cliente no local	Um representante do usuário final do sistema (o cliente) deve estar disponível todo tempo à equipe de XP. Em um processo de Extreme Programming, o cliente é um membro da equipe de desenvolvimento e é responsável por levar a ela os requisitos de sistema para implementação.

Tabela 4.4 – Práticas do Extreme Programming. (SOMMERVILLE, I., 2011)

Além das práticas enumeradas por Sommerville (2011), podemos acrescentar mais três: Metáfora, Padrões de codificação e Testes do usuário, como exposto na figura 4.7.

Figura 4.7 – Boas práticas em Extreme Programming.

Metáfora faz alusão à definição das histórias do usuário, que deve usar termos do negócio para que os desenvolvedores se comuniquem melhor com o cliente. A padronização da codificação deve ser adotada por toda a equipe para facilitar a compreensão do código e a comunicação entre desenvolvedores. O código deve ser o mais claro possível para minimizar a necessidade de documentação adicional. Por fim, os testes de usuário estão relacionados à validação das funcionalidades pelo cliente, o mais rápido possível.

Papéis do XP

Em projetos ágeis, as equipes envolvidas devem ser autossuficientes. Desta forma, é importante que o grupo responsável por construir o produto consiga reunir as habilidades técnicas e de negócio necessárias para o desenvolvimento. Porém, isso não quer dizer que todos os integrantes devem possuir todas as habilidades,

mas sim que a equipe não deve precisar de ajuda externa ao projeto para realizar as tarefas necessárias.

Outro ponto importante é que os membros da equipe devem ser autogerenciáveis. Neste sentido, a estruturação do grupo em hierarquias é desencorajada, não sendo recomendada a divisão de tarefas. É comum que no início do projeto as pessoas tenham uma inclinação para a realização de tarefas que fazem parte da sua especialidade. Desta forma, pessoas com experiência em especificação, tendem a trabalhar de acordo com esse perfil. Da mesma forma, indivíduos que são reconhecidos como bons programadores irão desempenhar tarefas relacionadas à codificação do produto.

Com o tempo, a partir dos princípios do XP, naturalmente o conhecimento dentro da equipe é disseminado, fazendo com que os membros se sintam à vontade de realizar tarefas até então fora das suas especialidades. Esse fato é relevante para garantir que o conhecimento não seja concentrado em determinados indivíduos dentro do projeto, o que causa dependência.

De forma geral, os principais papéis assumidos em projetos que utilizam *Extreme Programming* são: programador, coach, tracker e testador. Os programadores são a maioria da equipe e tem como principal responsabilidade a produção do código executável, além da elaboração dos testes de unidade. Por sua vez, o coach é o programador mais experiente do grupo e tem a responsabilidade de assegurar que as práticas e princípios do XP estão sendo realizados da forma correta. O *tracker* é o programador que possui a função de manter a equipe atualizada em relação ao progresso do projeto e por mostrar pontos que devem ser melhorados. Por fim, o testador é o integrante responsável por garantir que o produto está bom o suficiente para ser entregue ao cliente, a partir de realização de verificações e validações.

Vale a pena enfatizar que o cliente é parte da equipe na abordagem XP. As suas responsabilidades estão atreladas às atividades necessárias para a compreensão do negócio, priorização das funcionalidades e validação constante do que é construído. O ideal é que a interação entre o cliente e a equipe seja realizada a todo o momento no decorrer do projeto.

PAPÉIS NO XP	
Programador	Responsável por produzir o código executável.

PAPÉIS NO XP	
Coach	Responsável por garantir que as práticas e princípios do XP estão sendo praticados.
Tracker	Responsável por atualizar a equipe em relação ao progresso do projeto.
Testador	Responsável pelas tarefas de verificação e validação dentro do projeto.
Cliente	Responsável por detalhar as questões relacionadas ao negócio, priorizar as funcionalidades e validar as entregas.

Tabela 4.5 – Papéis no *Extreme Programming*.

Princípios do XP

Na abordagem XP, os princípios são utilizados para ligar os valores às práticas, servindo como um guia. São eles: auto-semelhança, benefício mútuo, diversidade, economia, falha, fluidez, humanismo, melhoria, oportunidade, passos de bebê, qualidade, redundância, reflexão e responsabilidade aceita.

PRINCÍPIOS DO XP	
Auto-semelhança	Ao encontrar soluções que funcionem em um contexto, equipes XP devem também procurar adotá-las em outros, mesmo que em escalas diferentes.
Benefício mútuo	As práticas do XP devem ser estruturadas de modo a serem mutuamente benéficas para todos os envolvidos em um projeto de *software*.
Diversidade	Em projetos XP, a diversidade de habilidades, abordagens e opiniões deve ser encorajada.
Economia	XP reconhece que se investe em *software* com a expectativa de que gere retornos para os negócios. Suas práticas são organizadas para antecipar receitas e adiar despesas.

PRINCÍPIOS DO XP	
Falha	Na dúvida, falhe! Desenvolvimento de *software* sempre vem acompanhado de novos problemas, muitos dos quais não temos ideia de como resolver em princípio.
Fluidez	O que se busca em XP é estabelecer um fluxo contínuo de valor. Ao invés de impor obstáculos, através de etapas bem definidas, herdadas de uma adaptação equivocada das práticas da engenharia civil, o que se faz é permitir que o desenvolver aprenda sobre um requisito e avance rapidamente para a implementação do mesmo.
Humanismo	XP coloca as pessoas no centro do esforço de desenvolvimento e suas práticas são voltadas para potencializar o melhor que podem oferecer, bem como suprimir suas falhas.
Melhoria	Não devemos nos preocupar em construir o *software* perfeito, nem o design perfeito, nem o processo perfeito, mas sim em aperfeiçoar esses e outros aspectos dos projetos continuamente.
Oportunidade	Em XP, não esperamos que tudo dê certo no projeto. Temos consciência de que eventos inesperados podem e irão acontecer. Quando esse for o caso, queremos que todos aprendam ao máximo e, juntos, criem as melhores soluções.
Passos de bebê	Passos de bebê determinam que é melhor avançar um pouquinho de cada vez, com segurança, que tentar dar grandes passos sem validar suas consequências.
Qualidade	*Extreme Programming* gera valor rapidamente e evita desperdícios ao máximo. *Software* de má qualidade representa uma enorme perda
Redundância	Os problemas difíceis e críticos em desenvolvimento de *software* devem ser resolvidos de várias formas diferentes. Mesmo que uma solução falhe completamente, as outras soluções irão prevenir um desastre.
Reflexão	Boas equipes não apenas fazem seu trabalho, mas também pensam sobre **como** estão trabalhando e **por que** estão trabalhando. Elas analisam o porquê de terem tido sucesso ou falhado. Elas não tentam esconder seus erros, mas os expõem e aprendem com eles.

PRINCÍPIOS DO XP	
Responsabilidade aceita	Responsabilidade não pode ser atribuída; ela só pode ser aceita. Se alguém tenta te dar uma responsabilidade, só você pode decidir se é responsável ou não.

Tabela 4.6 – Princípios do Extreme Programming.

Ciclo de vida de um Projeto de Programação XP

Um projeto desenvolvido a partir da abordagem XP apresenta diversas fases, como exploração, planejamento, iterações até versão, produção manutenção e morte. As tarefas e artefatos envolvidos podem ser visualizados na figura 4.9.

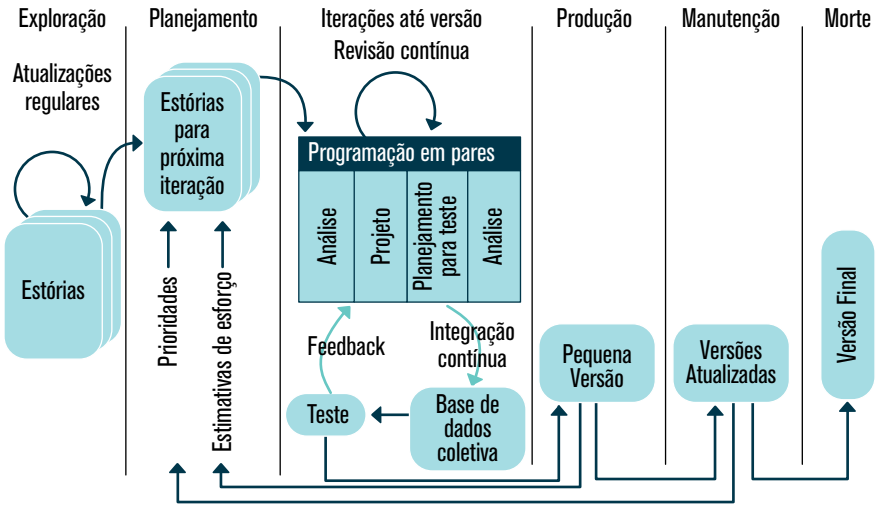


Figura 4.8 – Ciclo de Vida Extreme Programming. Fonte:<http://www2.dc.ufscar.br/~junia/MetAgEds.pdf>.

Fase de Exploração

A fase de exploração tem como objetivo verificar a viabilidade do projeto a partir da análise dos requisitos iniciais. É neste momento que as funcionalidades e características do produto são detalhadas para que todos os envolvidos tenham uma melhor ideia do que deve ser construído.

Fase de Planejamento

Na fase de planejamento as histórias de usuário são definidas e priorizadas. Além disto, cada história é estimada em relação à sua complexidade. Histórias de usuário mais complexas devem ser implementadas primeiro. Neste momento devem ser definidos o tempo de iteração e de release. Cada iteração pode ser realizada de uma a três semanas, enquanto a release deve ter de dois a quatro meses.

Fase de Iteração

A fase de iteração é a etapa de concentração das atividades de engenharia do produto. A equipe deve realizar de forma contínua e iterativa as atividades de análise, projeto, implementação e testes a partir dos princípios, valores e práticas do XP. Normalmente a primeira iteração tende a ser mais longa já que o problema e as tecnologias envolvidas ainda podem ser desconhecidos. Com o tempo de projeto, a equipe vai amadurecendo as estimativas e as iterações podem passar a ter um tempo mais curto.

Fase de produção

Após o final de cada *release*, o produto parcial deve ser integrado e disponibilização em ambiente similar ao de produção. O objetivo é realizar testes em relação ao desempenho e comportamento do *software*. Testes de aceitação podem ser realizados para que o cliente valide a entrega.

Fase de Manutenção

A fase de manutenção é marcada pela acomodação de mudanças necessárias para fazer com que o *software* continue sendo útil ao cliente. Neste momento, novas funcionalidades podem ser adicionadas ao produto, como também erros podem ser corrigidos. Além disto, o código pode ser melhorado ou adaptado a novos contextos. É importante que alterações nesta fase sejam realizadas com cautela, já que erros podem ser inseridos no sistema, causando prejuízo aos clientes.

Fase de Morte

A fase da morte é caracterizada pelo encerramento do projeto. A finalização das atividades em um projeto XP pode ocorrer por duas razões: o cliente pode estar satisfeito ao ponto de não precisar de novas alterações ou o sistema está deteriorado a um ponto em que manutenções já são complexas de ser realizadas, tornando a continuação do projeto inviável economicamente.

ATIVIDADES

01. O desenvolvimento ágil de **software** é guiado por metodologias que compartilham um conjunto comum de valores e de princípios, conforme definido pelo Manifesto Ágil. Assinale a opção que indica um princípio do desenvolvimento ágil.

a) As mudanças nos requisitos devem ocorrer dentro do quadro de tempo estabelecido para a iteração
b) O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é por meio de conversa face a face.
c) Os intervalos regulares devem ser evitados para tornar a equipe mais eficaz e maximizar a quantidade de trabalho realizado.
d) As pessoas de negócio e desenvolvedores devem interagir somente no início de cada iteração.
e) A entrega contínua e adiantada de *software*, mesmo que o conjunto de funcionalidades desenvolvidas não agregue valor, deve ser feita para satisfazer o cliente.

02. Como o desenvolvimento enxuto de *software* atua na construção do desenvolvimento ágil de sistemas?

03. NÃO é uma característica da *Extreme Programming* (XP):

a) simplicidade.
b) agilidade.
c) desenvolvimento orientado a testes.
d) programação em par.
e) documentação extensa e abundante em artefatos.

04. Determinado projeto de *software* utiliza XP (*Extreme Programming*) como metodologia de desenvolvimento. A esse respeito, é INCORRETO afirmar que:

a) o cliente participa ativamente e acompanha os passos dos desenvolvedores diariamente.
b) os integrantes da equipe se reúnem rapidamente no início do dia, de preferência em pé.
c) a equipe de desenvolvimento concentra esforços naquilo que gera maior valor para o cliente.
d) a programação em pares dispensa o desenvolvimento orientado a testes no projeto.
e) as funcionalidades do software são descritas em histórias, da forma mais simples possível.

05. Assegurar que a equipe se concentre em fazer, primeiro, apenas aquilo que é claramente necessário e evite fazer o que poderia vir a ser necessário, mas ainda não se provou essencial. Este é um dos cinco valores fundamentais do XP (*Extreme Programming*), denominado:

a) coragem.
b) respeito.
c) comunicação.
d) simplicidade.
e) *feedback*.

REFLEXÃO

Neste capítulo estudamos os princípios de metodologias ágeis a partir da análise do manifesto ágil. Estes princípios norteiam abordagens como a *extreme programming* e o *Scrum*. Continuamos nossos estudos entendendo a desenvolvimento rápido de *software*, criado a partir da metodologia *Toyota* de produção. Aprofundamos o entendimento de ágil a partir do estudo detalhado sobre XP a partir dos seus valores, papéis, princípios e práticas. Por fim, detalhamos o ciclo de vida XP.

É importante que, baseado nos conceitos apresentados, você seja capaz de identificar as vantagens e desvantagens de metodologias ágeis. Os conhecimentos adquiridos serão importantes para o entendimento da abordagem *Scrum*, foco do nosso estudo no próximo capítulo.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de extreme programming e demais assuntos deste capítulo, consulte o capítulo 3 do livro: PRESSMAN, R. S. Engenharia de *Software*. 6ª Edição. Editora McGraw Hill, 2011.

REFERÊNCIAS BIBLIOGRÁFICAS

SOMMERVILLE, I. **Engenharia de *Software***. 9ª Edição. Editora Pearson, 2011.

PRESSMAN, R. S. **Engenharia de *Software***. 6ª Edição. Editora McGraw Hill, 2011.

LARMAN, Craig. ***Agile & Iterative Development:*** A Manager's Guide. Addison-Wesley, 2003.

HIGHSMITH, Jim. ***Agile Software Development Ecosystems.*** Adisson-Wesley, 2002.

FRANCO, Eduardo Ferreira**. Um modelo de gerenciamento de projeto baseado nas metodologias ágeis de desenvolvimento de software e nos princípios da produção enxuta**. São Paulo, 2007. Disponível em: http://www.teses.usp.br/teses/disponiveis/3/3141/tde- 09012008-155823/pt-br.php Acesso em: 04 de agosto de 2016.

SHINGO, S. ***Study of "Toyota" Production*** System from Industrial Engineering Viewpoint: Produce What Is Needed, When It's Needed. Cambridge: Productivity Press, 1981. 291 p.

POPPENDIECK, M.; POPPENDIECK, T. ***Lean Software Development:*** An Agile Toolkit for Software Development Managers. Primeira Edição. Boston: Addison-Wesley Professional, 2003. 240 p.

5

Scrum

Scrum

O *framework Scrum* surgiu na década de 90, porém a sua popularização é mais recente. Até os anos 2000 as abordagens tradicionais ainda eram imperativas. Somente a partir desse período que os métodos ágeis tornaram-se uma forma bem sucedida de desenvolver sistemas.

Projetos que utilizam o *Scrum* possuem 3,5 vezes mais chances de sucesso do que outros projetos (*The Standish Group*, 2015). Isso não dignifica que a adoção do *framework* garante que problemas não irão acontecer. Muitas das vantagens proporcionadas pela abordagem só poderão ser percebidas caso a ferramenta seja bem utilizada. Neste capítulo vamos estudar os conceitos gerais do *Scrum*, seus papeis, artefatos e eventos. Por fim vamos analisar a abordagem Kanban.

OBJETIVOS

- Aprender os conceitos básicos de *Scrum*;
- Entender os papéis envolvidos no *Scrum*;
- Conhecer os artefatos presentes no *Scrum*;
- Entender o ciclo de vida do XP;
- Aprender os conceitos de Kanban.

Introdução ao *Scrum*

Sabbagh
Scrum e um *framework* ágil, simples e leve, utilizado para a gestão do desenvolvimento de produtos complexos imersos em ambientes complexos. *Scrum* e embasado no empirismo, e usa uma abordagem iterativa e incremental para entregar valor com frequência, assim, reduzindo os riscos do projeto.

FONTE: SABBAGH, R. Scrum: Gestão ágil para projetos de sucesso, 2016, página 19.

O *Scrum* é uma abordagem utilizada principalmente no desenvolvimento de sistemas que apresentam um contexto instável do ponto de vista dos seus requisitos e utilização de tecnologia. Desta forma, o *framework* se encaixa em situações onde as mudanças ocorrem frequentemente. Com o objetivo de reduzir os riscos

inerentes a projetos que necessitam se adequar constantemente a mudanças, a abordagem em questão foca na entrega de valor e na comunicação. Como consequência do seu uso, a qualidade do produto entregue e o aumento da produtividade da equipe pode ser percebida. Vale a pena destacar que muitos dos conceitos da abordagem não são inéditos na literatura. Podemos perceber a influência de práticas ágeis presentes em outras abordagens e no manifesto ágil.

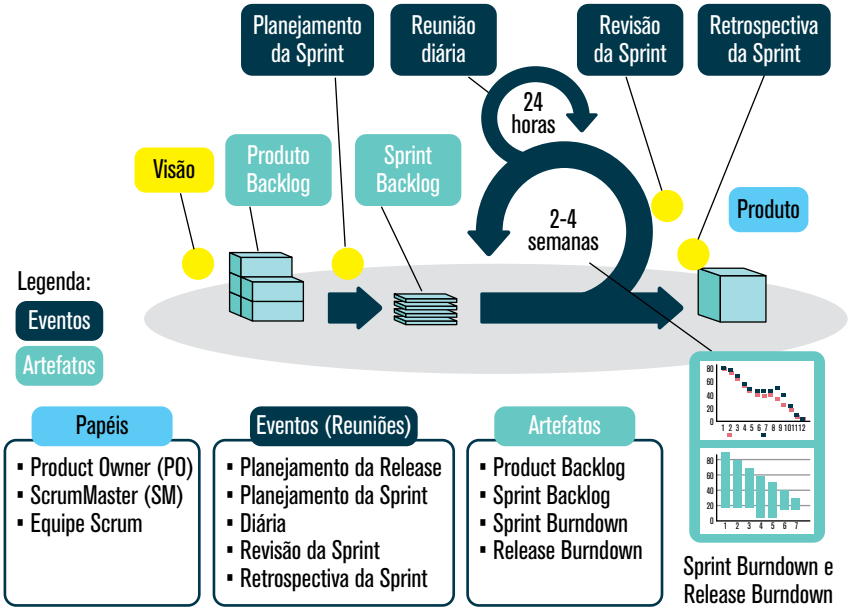


Figura 5.1 – Ciclo do trabalho Scrum.

Pressman

Os princípios do *Scrum* são consistentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades estruturais: requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica ocorrem tarefas a realizar dentro de um padrão de processo chamado *Sprint*. O trabalha realizado dentro de uma *Sprint* (o número de *Sprints* necessários para cada atividade metodológica varia dependendo do tamanho e da complexidade do produto) é adaptado ao problema em questão e definido, e muitas vezes modificado em tempo real, pela equipe *Scrum*.

FONTE: PRESSMAN, R. S. Engenharia de *Software*. 7ª Edição. Editora McGraw Hill, 2011, página 95.

O ciclo do trabalho *Scrum* pode ser visto na figura 5.1. A abordagem tem como ponto de partida as necessidades funcionais e não funcionais dos clientes e usuários. É importante que o produto a ser construído possua uma visão estratégica a partir de um ***Roadmap*** e também uma lista priorizada dos requisitos que forma o ***Backlog* do produto**. O papel responsável por manter estes artefatos é o **dono do produto** (*Product Owner)*.

O *Roadmap* do Produto simboliza o planejamento em um grau de abstração maior do produto. Este artefato mostra como o produto deve evoluir estrategicamente ao longo do tempo a partir da definição de objetivos a serem realizados. Na prática, o artefato pode ser visto como uma linha de tempo que contém marcos e metas do produto a serem realizadas, como exposto na figura 5.2.

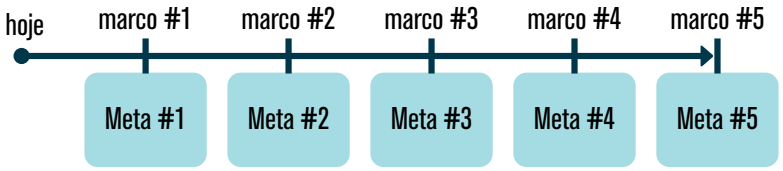


Figura 5.2 – Exemplo de *roadmap* do produto. (SABBAGH, R. Scrum: Gestão ágil para projetos de sucesso, 2016, página 270)

Por sua vez, o *Backlog* do produto é a lista dos requisitos do produto, não sendo necessário estar completo no início do desenvolvimento. O ideal é que a construção do *software* tenha início a partir das necessidades conhecidas. Com o decorrer do projeto, o *Backlog* do produto deve crescer. As necessidades contidas no artefato podem estar definidas a partir de histórias de usuário ou casos de uso.

A partir do momento que o time possui o *Backlog* do produto definido, pode-se realizar o planejamento da *Sprint*. A cerimônia em questão consiste em uma reunião com a presença do dono do produto, do líder do projeto e do time, além de qualquer pessoa interessada que esteja representando a gerência ou o cliente. Durante a reunião, o dono do produto deve detalhar os requisitos de maior prioridade para o time. A proposta é que a equipe consiga quebrar as funcionalidades em tarefas que formarão o ***Backlog* do *Sprint***.

As *Sprints* são formadas por tempos fixos que podem variar de uma a quatro semanas. A partir do momento que, no início do projeto, ficou decidido o tempo da *Sprint*, este número não pode ser alterado durante todo o processo de desenvolvimento. Durante as *Sprints*, o **líder do projeto** (*Scrum Master*) deve conduzir as **reuniões diárias** com o objetivo de disseminar conhecimento sobre o que foi

feito no dia anterior, identificar impedimentos e priorizar o trabalho a ser realizado no dia que se inicia. É também durante as *Sprints* que o **gráfico *burndown*** é atualizado com o objetivo de monitorar o progresso do projeto em relação ao que foi planejado no início. Como podemos ver na figura 5.3, o eixo horizontal de um gráfico *burndown* mostra os *Sprints* e o eixo vertical mostra a quantidade de trabalho que ainda precisa ser feita no início de cada *Sprint*. O trabalho que ainda resta pode ser mostrado na unidade preferencial da equipe: *story points*, dias ideais, *team days* e assim por diante.

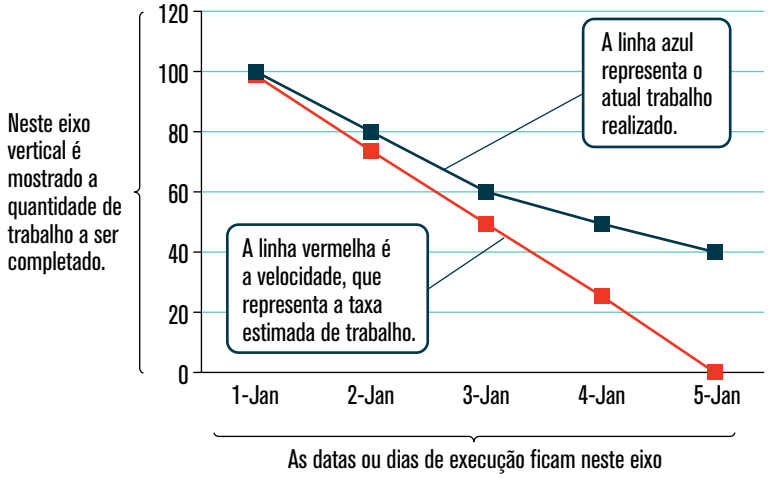


Figura 5.3 – Exemplo de gráfico burndown.

Ao final de todas as *Sprints*, o **time** deve realizar a **revisão** e a **retrospectiva** da Sprint. O objetivo da reunião de revisão é verificar a conclusão dos itens presentes no *Backlog* da *Sprint*. A validação deve ser realizada pelo dono do produto ou pelo cliente, sendo a melhor maneira realizar uma apresentação, ao estilo demonstração, de todos os itens concluídos. Com isso, o dono do produto deve avaliar o que está sendo considerado pronto, levando em conta o produto entregue e também o que não foi disponibilizado.

Por fim, a reunião de retrospectiva tem como objetivo analisar a última *Sprint* em relação às pessoas, relações, processos e ferramentas. Além disto, essa cerimônia deve identificar e ordenar os principais itens que foram bem e as potenciais melhorias, além de criar um plano para implementar melhorias no modo que o time faz seu trabalho.

A figura 5.4 resume todos os conceitos apresentados até o momento. Ainda neste capítulo, iremos detalhar os papéis, artefatos e eventos do *Scrum*.

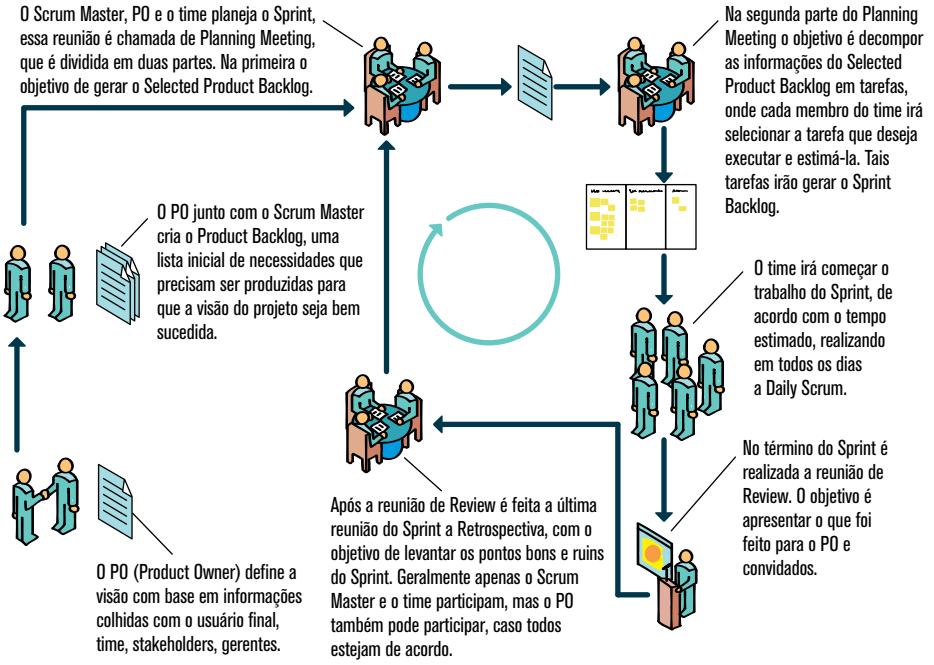


Figura 5.4 – Papéis, artefatos e eventos do *Scrum*.

De acordo com Sabbagh (2016), os benefícios no uso do *Scrum* incluem: entregas frequentes de retorno ao investimento dos clientes; redução dos riscos do projeto; maior qualidade no produto gerado; mudanças utilizadas como vantagem competitiva; visibilidade do progresso do projeto; redução do desperdício; e aumento da motivação e produtividade.

Entregas frequentes são caracterizadas pela quebra do projeto em incrementos, possibilitando um maior envolvimento do cliente com o produto. Muitos projetos ainda utilizam abordagens tradicionais que permitem que o produto seja visualizado apenas no fim do desenvolvimento, quando mudanças são mais caras e difíceis de serem realizadas. Com o desenvolvimento baseado em pequenos incrementos, o cliente interage com o produto mais cedo, podendo perceber o retorno do investimento ainda no começo do projeto. Além disto, a acomodação de possíveis alterações passa a ser um processo natural na construção do produto.

Como consequência da proximidade do cliente e constante validação, **os riscos do projeto são reduzidos**, já que os problemas são percebidos e enfrentados de forma antecipada. Apesar de não estipular um processo formal de gerenciamento de riscos, a abordagem Scrum minimiza a probabilidade de insucesso do projeto a partir do momento que foca na entrega do produto correto, desenvolvido com base em *feedback* constante. Além disto, possíveis atrasos são eliminados através da priorização do escopo: o que realmente importa para o cliente é construído mais cedo. Por fim, os riscos referentes a gargalos no desenvolvimento são minimizados pelo fato da equipe ser multifuncional.

Outro benefício apontado é o **aumento da qualidade do produto gerado**. Essa qualidade é atingida a partir da utilização de testes ao longo do desenvolvimento, principalmente automatizados. Além disto, a revisão do trabalho realizado pela própria equipe e a validação do cliente colaboram com o desenvolvimento de um produto mais robusto.

A qualidade do produto final também pode ser percebida a partir da **acomodação das mudanças** detectadas ao longo do processo. Desta forma, as alterações propostas agregam valor ao produto, passando a oferecer vantagem competitiva.

Por sua vez, a **visibilidade do progresso do projeto** se apresenta como consequência da forma transparente como os membros do time trabalham. Além da adoção de entrega frequentes, que faz com que o cliente acompanhe melhor o progresso do projeto, outras cerimônias como a reunião diária e a retrospectiva fazem com que a equipe tenha uma melhor ideia do que foi feito e do que ainda necessita ser construído.

Outro problema minimizado pela adoção do *Scrum* é a **redução do desperdício**. A abordagem é baseada na simplicidade, tendo como objetivo a utilização apenas dos artefatos que são necessários ao desenvolvimento do produto, a produção apenas das funcionalidades que sejam necessárias aos usuários e o planejamento apenas com o nível de detalhe possível para o momento. Dos problemas citados, podemos destacar a importância da priorização das funcionalidades a serem implementadas. A partir da figura 5.5, percebe-se que cerca de 80% das funcionalidades contidas no produto não são exercidas com frequência, sendo que 45% nunca são utilizadas. Desta forma, fica claro como o desperdício pode ser presente na vida útil de um *software*.

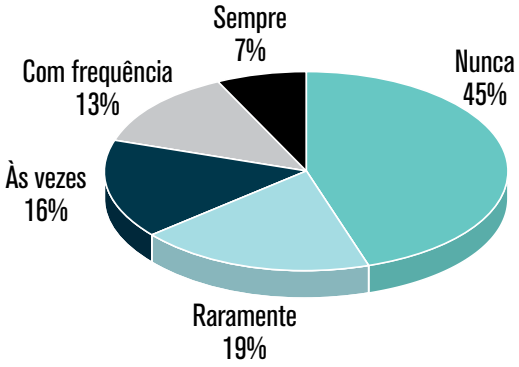


Figura 5.5 – Percentual de uso das funcionalidades em software, de acordo com o presidente do *Standish Group* em 2002. (SABBAGH, R. Scrum: Gestão ágil para projetos de sucesso, 2016)

Por fim, o **aumento da motivação e produtividade** é potencializado a partir de diversos fatores como: o trabalho em equipe e a autonomia do time na realização desse trabalho; a existência de facilitação e de remoção de impedimentos; a melhoria continua dos processos de trabalho; um ritmo sustentável de trabalho e a realização do trabalho de ponta a ponta.

Papéis no *Scrum*

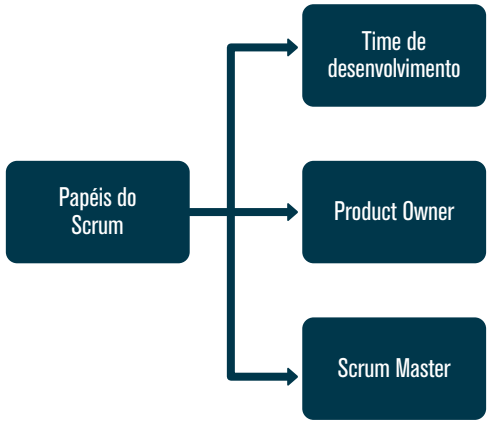


Figura 5.6 – Papéis no *Scrum*. (SABBAGH, R. Scrum: Gestão ágil para projetos de sucesso, 2016)

Diferente de abordagens tradicionais, como o RUP, que possuem diversos papéis especializados dentro do processo de desenvolvimento, o *Scrum* apresenta apenas

três papeis: o time de desenvolvimento, o *Scrum Master* e o *Product Owner* (PO) Todos os papéis são responsáveis pelo sucesso do produto a partir do momento que todos são comprometidos com os resultados dos trabalhos. Os membros do time de desenvolvimento, o *Product Owner* e o *Scrum Master* formam o **time de Scrum**.

Sabbagh

Um time ou uma equipe é um grupo de pessoas que trabalham juntas e colaboram em busca de um objetivo comum. Portanto, o time de *Scrum*, ou seja, time de desenvolvimento, *Product Owner* e *Scrum Master*, devem colaborar lado a lado, em seu dia a dia de trabalho, em busca do objetivo comum de atingirem o sucesso do projeto, em cada passo para chegarem lá.

FONTE: SABBAGH, R. *Scrum*: Gestão ágil para projetos de sucesso, 2016, página 86.

Com relação às atividades, de forma geral, como exposto na figura 5.7, o *Product Owner* participa das tarefas relacionadas ao planejamento e esclarecimento das funcionalidades. Por sua vez, o *Scrum Master* atua no planejamento e no progresso das tarefas. Por fim, o time de desenvolvimento solicita esclarecimentos junto ao PO e também participa do progresso do produto.

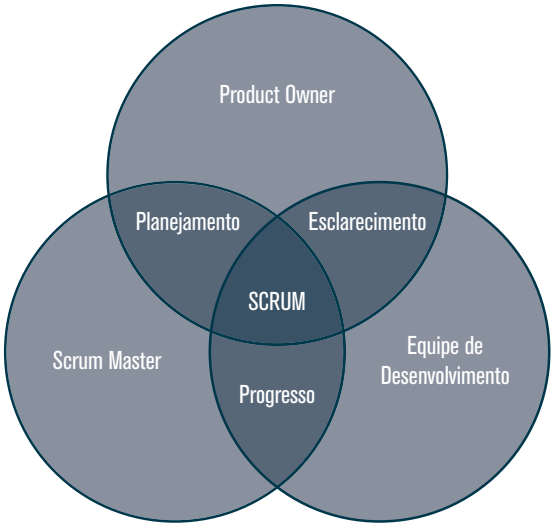


Figura 5.7 – Atividades gerais versus papéis.

Time de desenvolvimento

O Time de Desenvolvimento é o conjunto dos membros de um projeto ágil. De acordo com as definições do *Scrum*, o time de desenvolvimento deve ser autocontido, ou seja, o grupo deve ser multidisciplinar para garantir que todo o trabalho necessário vai ser realizado sem a necessidade de aquisições externas. Além disto, o time deve ser autogerenciável. Nesse sentido, cada indivíduo é responsável por monitorar o seu trabalho, incluindo poder de decisão em relação à definição técnica de como o produto deve ser construído, ao planejamento do trabalho e acompanhamento do seu progresso. Desta forma, podemos dizer que o time é responsável pelos seus resultados.

Sabbagh

O Time de Desenvolvimento gerencia o seu trabalho de desenvolvimento do produto, balizado pelo *framework Scrum*. E ele que, mesmo que sujeito aos limites dados pelo contexto do cliente e organizacional, determina tecnicamente como o produto será desenvolvido, planeja esse trabalho e acompanha seu progresso. Para tal, tem propriedade e autoridade sobre suas decisões e, ao mesmo tempo, e responsável e responsabilizado por seus resultados.

FONTE: SABBAGH, R. *Scrum*: Gestão ágil para projetos de sucesso, 2016, página 88.

As principais responsabilidades do time de desenvolvimento são:

- **Planejar a *Sprint:*** inicialmente o time de desenvolvimento deve participar do planejamento de cada *Sprint* com o objetivo de esclarecer possíveis dúvidas sobre o escopo com o *Product Owner*. Além disto, todo o *Time Scrum* deve estabelecer a meta para o próximo *Sprint*. Como consequência desta cerimônia, é obtido o *Sprint Backlog*.
- **Realizar Execução *Sprint:*** durante as *Sprints*, a equipe é responsável por realizar as atividades de engenharia de *software* necessárias para a conclusão do produto, como a concepção, construção, integração e testes de itens do *Product Backlog*.
- **Realizar Inspeção e Adaptação:** os membros da equipe de desenvolvimento devem participar das reuniões diárias com o objetivo de inspecionar o progresso da *Sprint* e expor possíveis impedimentos em relação às suas atividades. Caso membros do time não participem dessa cerimônia diariamente, o objetivo da *Sprint* pode ser impactado.

• **Refinar o *Product Backlog:*** a equipe de desenvolvimento atua junto ao *Product Owner* no refinamento do *Backlog* do Produto e na estimativa e priorização dos itens e trabalho.

• **Participar da retrospectiva da *Sprint:*** toda a equipe deve participar da reunião de retrospectiva com o objetivo de apontar problemas na execução das atividades durante a *Sprint* corrente, como também ressaltar os pontos positivos do trabalho.

• **Participar da revisão da *Sprint:*** toda a equipe deve participar da reunião de revisão com o objetivo de apresentar o resultado do trabalho realizado para o *Product Owner*.

Para que as responsabilidades discutidas acima sejam assumidas com sucesso, o time de desenvolvimento deve conter algumas características essenciais, conforme a figura 5.8.

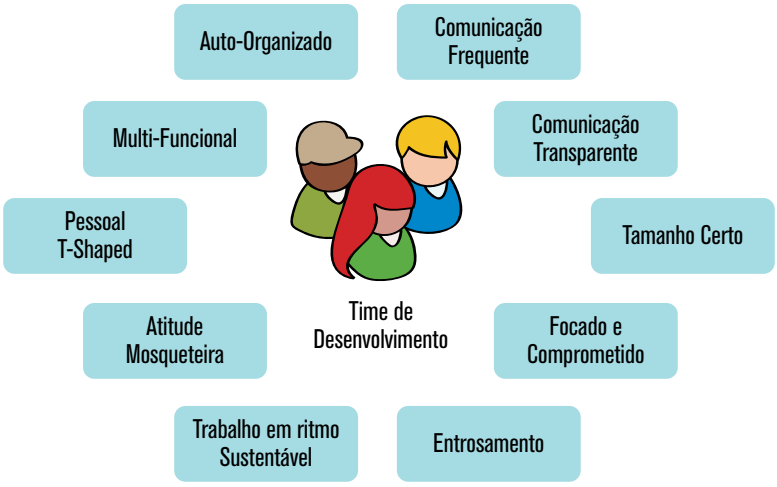


Figura 5.8 – Características essenciais ao time de desenvolvimento.

A equipe de desenvolvimento deve ser **auto-organizada**. Na prática, isso quer dizer que os próprios membros do time devem escolher a melhor forma de cumprir o objetivo da *Sprint*, sem a necessidade de um gerente de projetos estar guiando e controlando o processo. Mas como os indivíduos sabem exatamente o que fazer para que o sucesso seja obtido? Em um ambiente complexo, como o de projetos, o estabelecimento de um objetivo em comum e de regras simples fazem

com que a interação constante e o *feedback* culminem no encontro da melhor forma de chegar ao ponto desejado.

Outro ponto importante é que o time *Scrum* deve ser **multifuncional**. Nesse sentido, os membros devem possuir o conjunto de habilidades necessárias para construir durante a *Sprint* um potencial de entrega. Isso não quer dizer que todas as pessoas envolvidas numa equipe ágil devem saber de tudo, mas que as habilidades do grupo sejam suficientes para o desenvolvimento do sistema sem aquisições externas. Equipes pouco plurais tender a fazer um trabalho incompleto e necessitar de ajuda especializada. A diversidade dentro de um projeto tende a gerar melhores resultados em termos de soluções mais rápidas. Além disto, as soluções propostas tendem a ter uma maior qualidade e inovação.

Nesse contesto, é importante que os membros de um time *Scrum* tenham habilidade em forma de T, ou ***T-shaped***. Indivíduos com essa habilidade são aqueles que possuem um conhecimento sólido em um determinado assunto específico, porém ao mesmo tempo tem a capacidade de compreender outros assuntos. Por exemplo, imagine um indivíduo que é um exímio codificador. Apesar de conhecer a fundo padrões de projetos e dominar linguagens de programação, o membro pode também especificar histórias do usuário. Talvez ele não vá construir as melhores histórias, mas certamente irá ajudar o restante do time com o seu trabalho. Na figura 5.9, fica claro a comparação entre pessoas generalistas, especialistas e *T-shaped*.

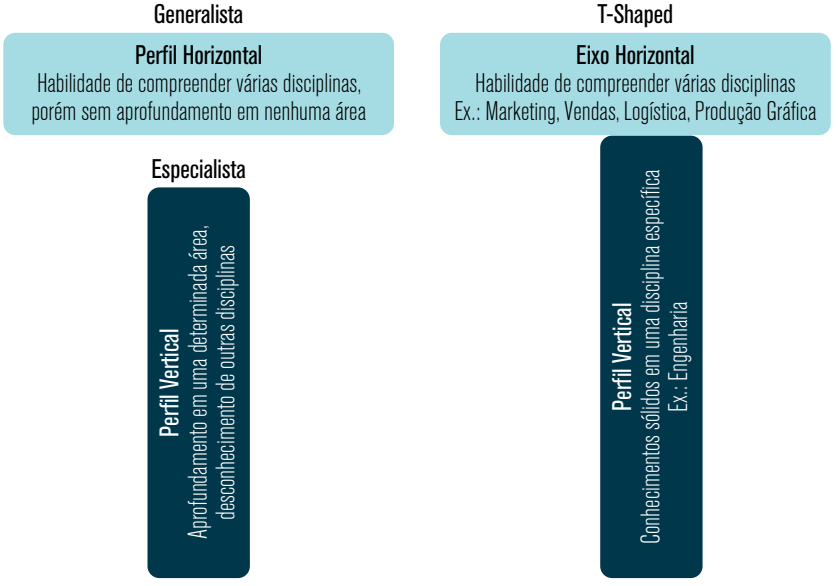


Figura 5.9 – Características de pessoas generalistas, especialistas e T-shaped.

Do ponto de vista de colaboração em grupo, é ideal que todos tenham **atitude mosqueteira**. O objetivo é entender que a responsabilidade da entrega do produto é compartilhada entre os membros, e dessa todos devem colaborar para que o objetivo final seja atingido. Essa característica está intimamente ligada com a necessidade da equipe ser *t-shaped*: a partir do momento que a equipe é formada por pessoas que possuem um perfil variado e, consequentemente, podem trabalhar em diferentes tarefas, a colaboração passa a ser incentivada.

Ainda sobre as principais características de uma equipe *Scrum,* podemos destacar a **habilidade em se comunicar frequentemente**. A informação dentro da equipe deve ser repassada a todo o momento, seja para discutir impedimentos, seja para validar o trabalho realizado. Da mesma forma, a **transparência na comunicação** deve ser permanente. Uma comunicação clara evita surpresas ao longo do desenvolvimento do sistema e estabelece confiança entre os envolvidos.

Para que a comunicação seja eficaz, **o tamanho do time deve ser limitado**. A abordagem *Scrum* preconiza que o tamanho das equipes não deve passar de nove pessoas, sendo o mínimo de cinco. Equipes grandes levam a uma dificuldade de controle e perda de qualidade na comunicação, enquanto que equipes pequenas podem apresentar um conjunto de habilidades insuficientes para solucionar o problema.

Os membros da equipe também devem ser **focados e comprometidos**. Para que isso aconteça, não é indicado que as pessoas trabalhem em mais de um trabalho simultaneamente. Da mesma forma, não é interessante que a equipe seja inserida em um ambiente de trabalho exaustivo, como por exemplo, onde a necessidade de realização de horas extras seja constante. O **ritmo da equipe deve ser sustentável**, sob o risco de diminuir a qualidade dos produtos.

Por fim, é interessante que haja **entrosamento** entre os indivíduos. Um bom entrosamento é consequência de um trabalho a longo prazo. Desta forma, é recomendável que não haja uma forte rotatividade entre membros de equipes.

Sabbagh

O *Product Owner*, também chamado de P.O., é a pessoa responsável pela definição do produto, trabalho que é realizado de forma incremental ao longo de todo projeto. Seu objetivo primário é garantir e maximizar, a partir do trabalho do Time de Desenvolvimento, o retorno sobre o investimento para os clientes do projeto, satisfazendo suas necessidades com relação ao produto em desenvolvimento.

FONTE: SABBAGH, R. *Scrum*: Gestão ágil para projetos de sucesso, 2016, página 110.

A partir da definição de Sabbagh (2016), podemos afirmar que o *Product Owner* representa um dos papéis mais importantes no *Scrum*, já que exerce a liderança sobre o produto, ao fazer a ponte entre o Time de Desenvolvimento e o cliente, conforme a Figura 5.10.

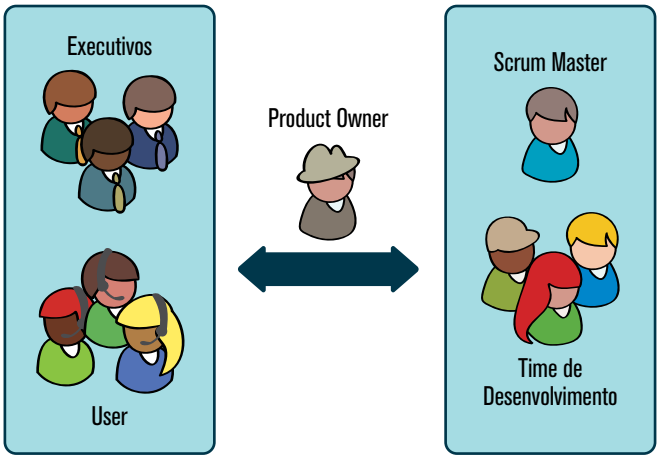


Figura 5.10 – Papel do *Product Owner.*

Primeiramente, o trabalho do P.O. envolve o entendimento das necessidades dos clientes e usuários. Além disso, é importante que ele atue na priorização dos itens do *Backlog*, garantindo que a solução correta seja desenvolvida e que vai ter utilidade. Da mesma forma, o P.O. deve manter uma forte comunicação com o time *Scrum*, detalhando e validando os itens que representam os requisitos funcionais e não funcionais do produto. O *Product Owner* também deve garantir que os critérios para aceitação do produto estão especificados e que os testes que verificam

esses critérios foram executados para determinar que o produto (ou *release*) possa ser considerado como pronto ao final do *Sprint*.

Dessa forma, podemos resumir as principais responsabilidades deste papel na figura 5.11.

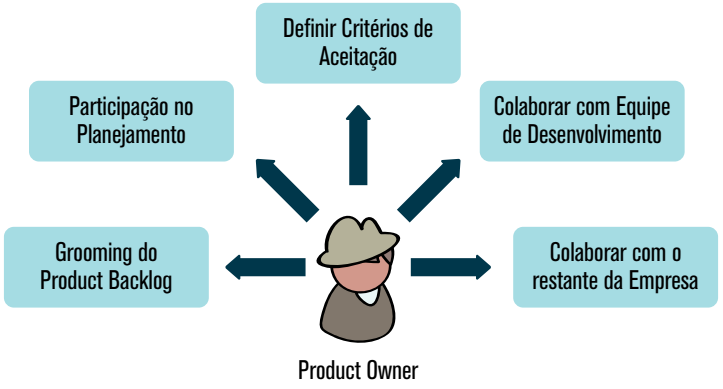


Figura 5.11 – Responsabilidades do *Product Owner*.

O P.O. deve **participar do planejamento** do produto, do projeto e de cada *Sprint*. No planejamento do produto, ele deve manter contato direto com os clientes para garantir que as necessidades vão estar presentes no *Backlog* do produto. No planejamento do projeto, o P.O. deve ajudar o restante do *Time Scrum* a definir o escopo do trabalho e a priorização dos itens do *Backlog*. Por fim, no planejamento da *Sprint*, ele deve detalhar os itens e ajudar na estimativa, além de definir o objetivo da *Sprint*.

Nesse âmbito, o P.O. é o principal responsável por criar, atualizar, estimar e priorizar os itens do *Backlog*, o que chamamos de ***grooming* do *Product Backlog***. Além disto, é papel do P.O. definir **os critérios de aceitação** para cada item do *Backlog*. Ao final de cada *Sprint*, a validação deve ser realizada com base nos critérios definidos.

Para que o trabalho seja realizado com sucesso e o objetivo da *Sprint* seja cumprido, o P.O. deve manter contato frequente com o time de desenvolvimento. A comunicação deve ser constante, diferentemente de abordagens tradicionais onde a comunicação com o cliente prevalece apenas no início e final da release, como mostrado na figura 5.12. O problema da comunicação em projetos tradicionais é que muitas vezes as necessidades dos clientes podem não ser bem entendidas e somente ao final, quando o custo de alterações é maior, é que o erro é percebido.

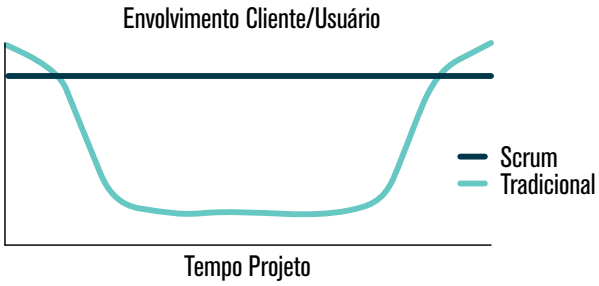


Figura 5.12 – Envolvimento cliente/usuário em projetos.

Por fim, o P.O. deve **colaborar com o restante da empresa**, a partir do momento que uma das suas funções é ser o porta-voz dos *stakeholders*.

Para que as responsabilidades descritas sejam assumidas, o P.O. deve possuir as seguintes habilidades pessoais: conhecimento de negócio, habilidades com pessoas, autoridade e responsabilidade.

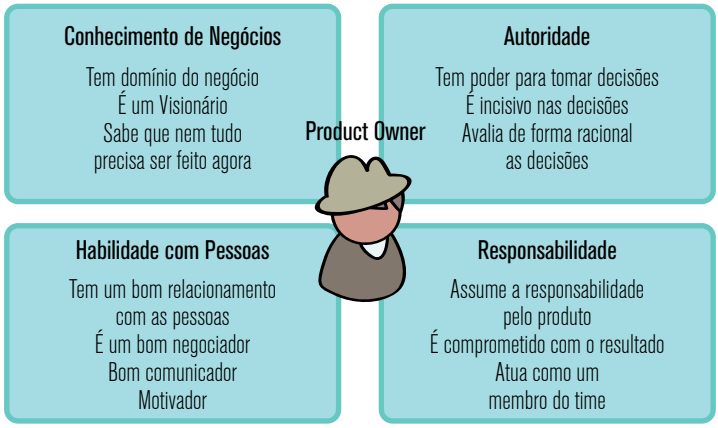


Figura 5.13 – Habilidades pessoais de um P.O.

CONCEITO

Aprenda mais sobre o *Product Owner* no vídeo a seguir: <https://www.youtube.com/watch?time_continue=1&v=L_KX457DwaM>.

Sabbagh

O *Scrum Master* trabalha para facilitar e potencializar o trabalho do *Time de Scrum*. Ou seja, utilizando-se de seu conhecimento de *Scrum*, habilidade de lidar com pessoas, técnicas de facilitação e outras técnicas, ele ajuda o *Product Owner* e Time de Desenvolvimento a serem mais eficientes na realização do seu trabalho.

FONTE: SABBAGH, R. *Scrum*: Gestão ágil para projetos de sucesso, 2016, página 124.

A partir da definição de Sabbagh (2016), podemos afirmar que o *Scrum Master* tem como principal objetivo atuar como um *coach* de processo, ajudando a equipe a compreender e executar da melhor forma os valores, princípios e práticas do *Scrum*. As principais responsabilidades deste papel estão descritas na figura 5.14.

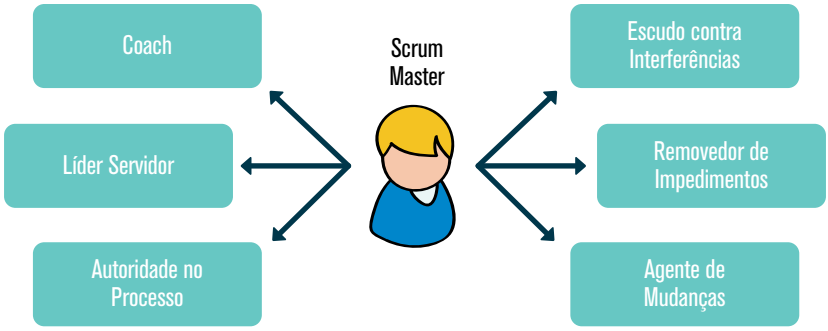


Figura 5.14 – Responsabilidades do *Scrum Master*.

Além de atuar como ***coach*** de processo, o *Scrum Master* deve ser um **escudo contra interferências**, blindando o Time de Desenvolvimento de qualquer evento externo que possa atrapalhar o andamento das *Sprints*. Além disto, ele deve atuar como um **líder servidor**, se comprometendo com o trabalho de toda a equipe.

Outra importante tarefa do *Scrum Master* é a **remoção de impedimentos**. A remoção de qualquer obstáculo que possa inibir a produtividade da equipe deve ser realizada sempre que os próprios membros da equipe não podem removê-los.

Além disso, é importante que o *Scrum Master* **domine o processo *Scrum***, se mostrando autoridade no processo. Por fim, como consequência, os indivíduos que exercem

este papel devem se mostrar como **agente de mudanças** dentro das organizações. Para muitas empresas, o método de trabalho do *Scrum* é uma mudança muito grande na cultura de trabalho. O *Scrum Master* ajuda, então, as pessoas a entenderem essas mudanças, os impactos da adoção do *Scrum* e os benefícios que o *Scrum* pode ajudar a atingir.

Para que o trabalho do *Scrum Master* tenha sucesso, é importante que ele possua determinadas características, como exposto na figura 5.15.

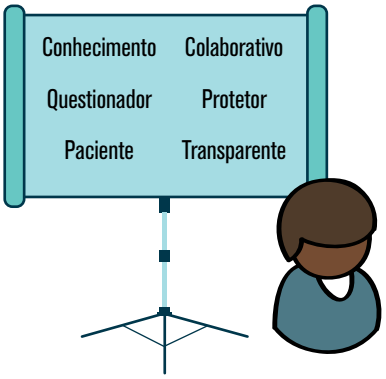


Figura 5.15 – Características do *Scrum Master*.

Artefatos do *Scrum*

Nesta seção vamos discutir acerca dos principais artefatos do *Scrum: Roadmap* do Produto, *Backlog* do Produto e *Backlog* da *Sprint*.

Roadmap do produto

O *Roadmap* do Produto é um artefato que descreve como será o produto a cada período de sua evolução. O seu maior objetivo é auxiliar a organização a traçar uma trajetória para seus produtos com base em objetivos de negócio, por ordem de prioridade. Por ter uma visão estratégica, este artefato ajuda a comunicar aos interessados a visão de futuro que se tem para seus produtos.

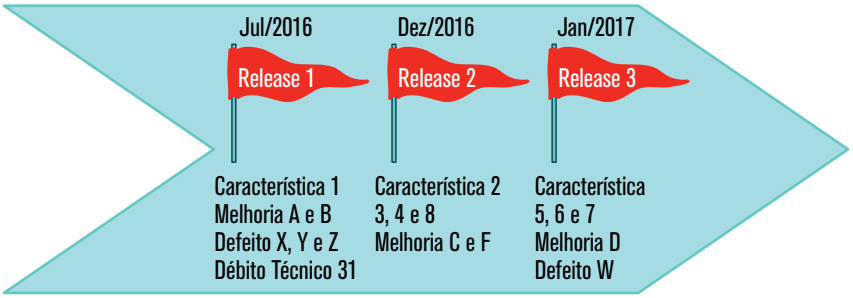


Figura 5.16 – Exemplo de *Roadmap* do Produto.

Para que o *Roadmap* ajude na condução do produto, alguns aspectos devem ser observados na sua elaboração, como: agrupar os itens do *Backlog* do Produto por ordem de prioridade, na quantidade compatível com a capacidade de produção das equipes e no tempo disponível para o desenvolvimento; estabelecer uma cronologia de entregas ou uma periodicidade; organizar reuniões em que os envolvidos participem ativamente da construção do *Roadmap*; e divulgar o *Roadmap* para todos os envolvidos.

Backlog do produto

> SABBAGH
>
> O *Product Backlog* e uma lista ordenada ou priorizada de itens sobre os quais o Time de Desenvolvimento trabalhara no decorrer do projeto, desde os itens do topo da lista, buscando realizar os objetivos do produto, comumente representados por uma Visão de Produto. Ele é atualizado, reordenado e refinado de acordo o nível de detalhes que e possível de se ter em cada momento do projeto.
>
> FONTE: SABBAGH, R. *Scrum*: Gestão ágil para projetos de sucesso, 2016, página 147.

A partir da definição acima, podemos entender o ***Product Backlog*** como sendo uma lista contendo todas as funcionalidades desejadas para um produto. Os itens que compõem o conteúdo desta lista são definidos pelo *Product Owner*. Um ponto interessante para destacar é que este artefato é um documento vivo, sendo

alterado durante todo o projeto. Desta forma, o *Product Backlog* não precisa estar completo no início de um projeto. Pode-se começar com tudo aquilo que é mais óbvio em um primeiro momento. Com o tempo, o *Product Backlog* cresce e muda à medida que se aprende mais sobre o produto e seus usuários (figura 5.17).

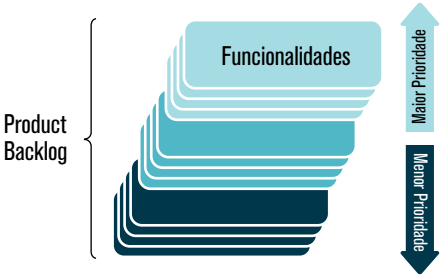


Figura 5.17 – *Backlog* do Produto.

Backlog da Sprint

Sabbagh

O *Sprint Backlog* e uma lista de itens selecionados do alto do *Product Backlog* para o desenvolvimento do Incremento do Produto no *Sprint* (o que), adicionada de um plano de como esse trabalho será realizado (como). O *Sprint Backlog* existe apenas no contexto de seu *Sprint* correspondente. Assim, ele e criado na reunião de *Sprint Planning*, e deixa de existir após as reuniões de *Sprint Review* e *Sprint Retrospective* de seu *Sprint*. O *Sprint Backlog* pertence ao Time de Desenvolvimento e é uma importante ferramenta utilizada por seus membros para organizarem o trabalho durante o *Sprint*. Por essa razão, o *Sprint Backlog* deve ser de alta visibilidade.

FONTE: SABBAGH, R. *Scrum*: Gestão ágil para projetos de sucesso, 2016, página 157.

Como podemos perceber pela figura 5.18, os itens do *Sprint Backlog* são extraídos do *Product Backlog*, pela equipe, com base nas prioridades definidas pelo *Product Owner* e a percepção da equipe sobre o tempo que será necessário para completar as várias funcionalidades. Cabe à equipe determinar a quantidade de itens do *Product Backlog* que serão trazidos para o *Sprint Backlog*, já que é ela quem irá se comprometer a implementá-los. Durante um *Sprint*, o *Scrum Master* mantém o *Sprint Backlog* atualizando-o para refletir que tarefas são completadas e quanto tempo a equipe acredita que será necessário para completar aquelas que

ainda não estão prontas. A estimativa do trabalho que ainda resta a ser feito no *Sprint* é calculada diariamente e colocada em um gráfico, resultando em um *Sprint Burndown Chart*.

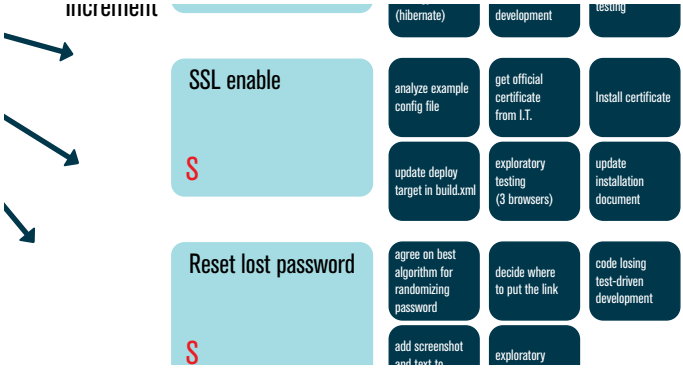


Figura 5.18 – Relação entre o *Backlog* do Produto e o *Backlog da Sprint*.

Eventos do *Scrum*

O ciclo de desenvolvimento do *Scrum* é composto por reuniões e cerimônias que vamos discutir nessa seção. Os principais eventos são: *Sprint*, planejamento do release, planejamento da *Sprint*, reuniões diárias, revisão da *Sprint*, retrospectiva da *Sprint* e refinamento do *Backlog* do produto.

Sprint

Sprints são ciclos que representam uma unidade temporal onde itens de *Backlog* devem ser desenvolvidos a partir de um conjunto de tarefas de engenharia de *software*. No mundo ágil, uma *Sprint* permite a implementação de um conjunto parcial de funcionalidades definido pelo próprio cliente. Por ser parcial, significa que não é um *software* completamente pronto, mas uma parte funcional dele. As *Sprints* possibilitam que a equipe trabalhe dentro de um prazo fixo para a implementação de um número definido de funcionalidades que, por sua vez, podem ser classificadas por prioridade.

É importante ressaltar que uma vez definido o objetivo e o escopo da *Sprint* (*Backlog da Sprint*) isto não deve mudar até o fim da duração estabelecida. O fluxo da *Sprint* deve ser repetido até esgotar todos os itens do *Backlog* da *Release*.

Planejamento do *release*

O planejamento do *release* é o momento de estimar o trabalho inicial para o projeto, devendo acontecer antes do primeiro *Sprint* do projeto ou ao final do último *Sprint* da *Release* anterior. Neste momento, muitos aspectos relacionados ao projeto são desconhecidos, desta forma o planejamento não deve ser detalhado. Um ponto importante é que a quantidade e duração das *Sprints,* o tamanho da equipe, a quantidade de entregas, o valor a ser entregue em cada *release* e a data de liberação de cada um deles devem ser definidos nesta etapa.

Com relação à duração da reunião, não há um prazo estabelecido, mas é importante limitar um espaço de tempo para a sua realização. Para o planejamento do *release*, é necessária a participação do *Product Owner*, Time de Desenvolvimento e do *Scrum Master*.

Planejamento da *Sprint*

A reunião de planejamento da *Sprint* tem como objetivo escolher os itens do *Backlog* com maior prioridade para serem desenvolvidos durante a *Sprint*. A partir do *Backlog* do *release* priorizado, histórias do usuário são escolhidas e quebradas em pequenas tarefas.

Neste momento, o *Product Owner* detalha e prioriza as histórias de usuário. A sua principal função nesta etapa é de auxiliar o Time de Desenvolvimento em relação à duvidas que possam surgir relacionadas aos itens do *Backlog*.

Reuniões diárias

A reunião diária é uma cerimônia curta que deve ser realizada pelo Time de Desenvolvimento. A ideia é que a duração máxima seja de quinze minutos, sendo realizada sempre no mesmo local e hora, preferencialmente. É importante que seja mantido um foco com base nos seguintes pontos:

- O que eu fiz desde a última reunião?
- O que eu pretendo fazer ate a próxima reunião?
- Que impedimentos estão em meu caminho?

O objetivo destas três perguntas é verificar se a meta da *Sprint* está sendo cumprida e o que possivelmente está atrapalhando o não cumprimento da meta.

Revisão da *Sprint*

A revisão da *Sprint* deve ser realizada sempre ao final de cada ciclo de desenvolvimento. Durante esta cerimônia, o *Time Scrum* deve apresentar o que foi produzido durante o *Sprint*. Devem participar da reunião o *Product Owner,* o *Scrum Team*, o *Scrum Master*, gerência, clientes e engenheiros de outros projetos.

Durante o *Sprint Review,* o projeto é avaliado em relação aos objetivos do *Sprint,* determinados durante o *Sprint Planning Meeting*. Idealmente, a equipe completou cada um dos itens do *Product Backlog* trazidos para fazer parte do *Sprint*, mas o importante mesmo é que a equipe atinja o objetivo geral do *Sprint.*

Retrospectiva da *Sprint*

O *Sprint Retrospective* ocorre ao final de um *Sprint* e serve para identificar o que funcionou bem, o que pode ser melhorado e que ações serão tomadas para melhorar o trabalho do Time. Deve ser realizada no último dia de cada *Sprint*, após a reunião de *Sprint Review*, tendo a duração máxima proporcional a três horas para *Sprints* de um mês. Time de Desenvolvimento, *Product Owner* e *Scrum Master* devem participar desta cerimônia.

Refinamento do *Backlog* do produto

O objetivo principal do refinamento do *Product Backlog* é a preparação do *Backlog* para o desenvolvimento nas próximas *Sprints*. Essa atividade deve ser realizada pelo Time de Desenvolvimento em conjunto com o *Prodct Owner*, sempre que necessário.

O trabalho de Refinamento do *Product Backlog* inclui a adição de novas funcionalidades, remoção de itens que não são mais úteis ao produto, o desmembramento de itens maiores em itens menores, a junção de itens menores em um item maior que perdeu prioridade, o detalhamento de itens, seu reordenamento e, possivelmente, a criação de algum tipo de estimativa.

Kanban

O sistema Kanban consiste em uma simbologia visual usada na indústria para registrar ações. A metodologia foi criada pela empresa *Toyota*, sendo parte

do sistema *Toyota* de produção, já discutido em capítulos anteriores. O seu termo, portanto, tem origem japonesa e pode ser traduzido como **"cartão visual"**. Na técnica, os cartões são a necessidade de peças e itens para o processo produtivo. O objetivo principal do método é propiciar a interação entre a gestão do estoque e a produção. Neste sentido a ideia é atuar na entrega de uma determinada quantidade de peças. No momento em que as peças se esgotarem, um aviso é emitido para que novas peças sejam produzidas e entregues. É uma analogia ao serviço "*Just in Time*".

Mas como é o funcionamento do sistema Kanban? O planejamento da produção e o controle de estoque são realizados a partir dos quadros e cartões visuais que integram a técnica. As decisões são tomadas conforme a quantidade de cartões disponíveis nos quadros, priorizando o que é mais importante, realizando *setup* de máquinas e até mesmo as paradas para manutenção. Neste sentido, podemos dividir a técnica em dois tipos: o de produção e o de movimentação.

SAIBA MAIS

Aprenda mais sobre o sistema Kanban no vídeo a seguir: <https://www.youtube.com/watch?feature=player_embedded&v=hpPBUVs9t1s>.

O Kanban de produção é o cartão responsável por autorizar a produção dos itens. Os cartões circulam entre o setor fornecedor e a produção, sendo afixados junto às peças imediatamente após a produção e retirados depois que vai para o cliente. Em seguida, retorna ao processo para autorizar a produção e reposição dos itens consumidos.

SAIBA MAIS

Aprenda mais sobre o Kanban de produção no vídeo a seguir: <https://www.youtube.com/watch?feature=player_embedded&v=Q3x6DblDNbk>.

Kanban de Movimentação é denominado ainda de Kanban de Transporte, sendo um cartão (diferente do Kanban de Produção) que autoriza a movimentação física de peças entre o fornecedor e o cliente. Os cartões são afixados nos produtos, geralmente, o cartão de movimentação é afixado em substituição ao cartão de produção, e levados a outro processo ou local, onde são retirados e voltam à etapa inicial.

ATIVIDADES

01. *SCRUM* é um *framework* baseado no modelo ágil. No *SCRUM*,

a) o *Scrum* team é a equipe de desenvolvimento, necessariamente dividida em papéis como analista, designer e programador. Em geral o *Scrum* team tem de 10 a 20 pessoas.

b) as funcionalidades a serem implementadas em cada projeto (requisitos ou histórias de usuários) são mantidas em uma lista chamada de *Scrum board*.

c) o *Scrum Master* é um gerente no sentido dos modelos prescritivos. É um líder, um facilitador e um solucionador de conflitos. É ele quem decide quais requisitos são mais importantes.

d) um dos conceitos mais importantes é o *Sprint*, que consiste em um ciclo de desenvolvimento que, em geral, tem duração de 4 a 7 dias.

e) o *Product Owner* tem, entre outras atribuições, a de indicar quais são os requisitos mais importantes a serem tratados em cada *Sprint*. É responsável por conhecer e avaliar as necessidades dos clientes.

02. Sobre os princípios do método de desenvolvimento *Scrum*, que são consistentes com o manifesto ágil, julgue as seguintes afirmativas e assinale a alternativa correta.

I . Testes e documentação constantes são realizados à medida que o produto é construído.

II. O processo produz frequentes incrementos de *software* que podem ser inspecionados, ajustados, testados, documentados e expandidos.

III. O trabalho de desenvolvimento e o pessoal que o realiza é dividido em partições claras, de baixo acoplamento, ou em pacotes.

a) Apenas as afirmativas I e II são corretas.

b) Apenas as afirmativas I e III são corretas.

c) Apenas as afirmativas II e III são corretas.

d) Todas as afirmativas são corretas.

e) Nenhuma das afirmativas é correta.

03. Segundo Roger S. Pressman, em seu livro Engenharia de *Software*, 7a edição, os princípios do *Scrum* são consistentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as atividades estruturais de requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica, ocorrem tarefas a realizar dentro de um padrão de processo chamado

a) *process Backlog.*
b) *Scrum Master.*
c) *Product Owner.*
d) *Backlog.*
e) *Sprint.*

04. Nos métodos ágeis XP e *Scrum*, as entregas de partes funcionais do projeto são divididas em ciclos, geralmente compreendidos no período de 2 a 4 semanas. Estes ciclos denominam-se, respectivamente,

a) iterações e *Sprint.*
b) reunião de planejamento e *Backlog*.
c) período de entrega e reunião de revisão.
d) *Backlog* e planejamento da produção.
e) entrega e retrospectiva.

05. Um dos pontos da metodologia *Scrum* é o *Daily Scrum*, que consiste em uma reunião diária com aproximadamente 15 minutos de duração onde são tratados assuntos relacionados ao projeto. Nessa reunião são feitas três perguntas a cada membro do time de desenvolvimento, constando o que foi feito desde a última reunião, o que será feito até a próxima reunião e qual:

a) modelo de testes está sendo utilizado pela tarefa atual.
b) o tempo restante para finalização da tarefa.
c) a relação da tarefa atual com o outro membro da equipe.
d) a tarefa que está sendo executada no momento.
e) obstáculo impede o desenvolvedor de prosseguir com a tarefa.

REFLEXÃO

Neste capítulo aprendemos de forma geral, os conceitos e características da metodologia ágil *Scrum* de desenvolvimento de sistemas. Estudamos conceitos relacionados aos papéis, artefatos e eventos do arcabouço. Ao final, vimos como a metodologia Kanban pode contribuir na produção, inclusive de *software*.

Sugerimos que você faça todos os exercícios propostos e pesquise outras fontes para aprofundar seus conhecimentos. Em caso de dúvidas, retorne aos tópicos e faça a releitura com bastante atenção.

LEITURA

Para você avançar mais o seu nível de aprendizagem envolvendo os conceitos de *Scrum* e demais assuntos deste capítulo, consulte a sugestão de *links* a seguir:

- Gamonar, Flávia. Saiba mais sobre o *Scrum*, um *framework* para desenvolvimento ágil de *software*. Disponível em:<https://www.linkedin.com/pulse/saiba-mais-sobre-o-Scrum-um-framework-para-ágil-de-software-gamonar>.
- Gamonar, Flávia. O *Scrum*, o princípio de Pareto, a produtividade e outras teorias relacionadas. Disponível em:<https://www.linkedin.com/pulse/o-Scrum-princípio-de-pareto-produtividade-e-outras-teorias-gamonar?trk=mp-author-card>.

REFERÊNCIAS BIBLIOGRÁFICAS

PRESSMAN, R. S. **Engenharia de *Software***. 6ª Edição. Editora McGraw Hill, 2011.

SABBAGH, R. Scrum: **Gestão ágil para projetos de sucesso**, 2016

SOMMERVILLE, I. **Engenharia de *Software***. 9ª Edição. Editora Pearson, 2011.

GABARITO

Capítulo 1

01. Um processo de desenvolvimento de sistemas é um conjunto de regras que permite organizar um projeto em particular ao estabelecer o sequenciamento de atividades a serem realizadas, devendo estabelecer quem realiza a atividade, de que forma, como e em que momento. Por sua vez, um modelo de processo de software é uma representação simplificada de um processo de *software*, não sendo responsável por estabelecer os papéis das pessoas envolvidas.

02. O modelo cascata é baseado na engenharia tradicional, onde podemos perceber uma abordagem sistemática ao estabelecer uma sequência entre as atividades envolvidas. Pode ser utilizado em um contexto onde os requisitos sejam bem definidos e que mudanças não sejam frequentes.

03. D. 04. D. 05. E.

Capítulo 2

01. O principal objetivo desta fase é produzir o documento de especificação de requisitos, onde devem estar contempladas todas as funcionalidades, características e restrições que o sistema deve conter, do ponto de vista dos usuários e clientes. Para que este documento seja produzido, é necessário inicialmente verificar a viabilidade do projeto, para na sequência elicitar, documentar e validar os requisitos do produto.

02. A fase de análise é a etapa de modelagem do problema, ou seja, contém as atividades necessárias para o entendimento do domínio do problema. Por sua vez, a fase de projeto é a etapa de modelagem da solução, ou seja, contém as atividades necessárias para resolver o problema. Também podemos comprar estas fases em relação à participação do usuário. Neste sentido, a fase de análise é o conjunto de atividades realizadas com a ciência do cliente, ou seja, os usuários devem discutir e validar a informação produzida. Já a fase de projeto representa o conjunto das atividades que produzem informação destinada aos programadores.

03. A padronização do código ajuda na integração do trabalho realizado pela equipe. O objetivo é alinhar a forma de escrita para que o resultado seja uniforme e, consequentemente, mais fácil de ser mantido.

04. Verificação e validação de *software* é o processo utilizado para verificar se o produto foi construído com base no que foi pedido na especificação e validar o produto do ponto de vista do usuário.

05. Um maior custo atrelado à manutenção pode ser explicado pelo fato da equipe de manutenção nem sempre ser a mesma equipe que desenvolveu o produto, fazendo com que o esforço envolvido no entendimento do produto seja maior. Além disso, os desenvolvedores de um sistema podem não ter responsabilidade contratual pela manutenção, a equipe de manutenção é geralmente menos experiente do que a equipe de desenvolvimento, além de

possuir um conhecimento limitado do domínio do problema; e com o passar do tempo, os programas tem a sua estrutura degradada, tornando-os mais difíceis de serem entendidos e alterados.

Capítulo 3

01. As principais características da ferramenta RUP é ser guiado pelos casos de uso, centrado na arquitetura, e iterativo e incremental.

02. A 03. A 04. B

05. Sim. O RUP pode ser utilizado na construção de sistemas *Web*, independente da complexidade e estabilidade dos requisitos. Como a ferramenta pode ser customizada, a depender da complexidade, o processo pode ser composto por mais ou menos atividades e artefatos.

Capítulo 4

01. B

02. A partir da utilização de princípios baseados na manufatura que tem como objetivo reduzir custos dos projetos, tais como: eliminar de perdas, amplificar o aprendizado, tomar decisões o mais tarde possível, fazer entregas o mais rápido possível, tornar a equipe responsável, construir integridade e visualizar o todo.

03. E 04. D 05. D

Capítulo 5

01. E 03. E 05. E

02. D 04. A

ANOTAÇÕES

ANOTAÇÕES

ANOTAÇÕES