ENGENHARIA DE SOFTWARE I

CAPÍTULO 1 - COMO PRODUZIR UM *SOFTWARE* DE QUALIDADE ELEVADA?

Igor Fernandes Menezes

INICIAR

Introdução

Neste capítulo você vai estudar os princípios da Engenharia de *Software*, os modelos de processo de desenvolvimento adotados nas fábricas, principais métodos ágeis de desenvolvimento e metodologia para levantamento de requisitos. Vamos estudar como é feita a produção de artefatos de levantamento de requisitos para o desenvolvimento de *softwares*, compreendendo como trabalha uma equipe de desenvolvimento de sistemas.

Para nossos estudos, vamos considerar que você deseja atuar como analista e desenvolvedor de sistemas e recebe uma oportunidade de desenvolvimento de um novo *software* ou de um aplicativo. Assim, a primeira questão que aparece é: como você irá fazer o levantamento das informações para desenvolver este *software*? Existem metodologias que você pode utilizar para facilitar suas atividades, como o levantamento de requisitos?

Vamos começar a estudar nesta área de grandes oportunidades.

Bons estudos!

1.1 Introdução à Engenharia de *Software*

Para começar o estudo sobre Engenharia de *Software*, precisamos entender a diferença entre ciência e engenharia. Você sabe qual é? Vamos lá! Os dois estão relacionados, mas existe uma grande diferença nos conceitos e no objetivo dos profissionais de cada uma destas áreas. Na ciência, se busca estudar os detalhes de um determinado escopo, observando suas características, seu funcionamento e aplicando experimentos para aproximar esta pesquisa do mundo prático. Por exemplo: um cientista tem que encontrar os melhores materiais para o asfalto que é usado em pontes superiores a 500 metros de extensão, como foco de pesquisa.

Por outro lado, a engenharia busca responder demandas específicas de trabalho, utilizando recursos adequados, desenvolvidos por cientistas como solução para problemas práticos. Isso se resume no que a pesquisadora Shari Pfleeger (2004, p. 2) afirma: “como engenheiros de *software*, utilizamos nosso conhecimento sobre computadores e computação para ajudar a resolver problemas”.

Vamos entender, então, como o profissional de *software* se desenvolve para atender a essas demandas.

1.1.1. Desenvolvimento do profissional de software

Com a necessidade do desenvolvimento de *softwares* complexos e com grande processamento de informações, se tornou necessário aprimorar as técnicas e metodologias aplicadas no processo de produção. Esse aprimoramento é feito pelos cientistas da computação, que trabalham para melhorar ou criar novos procedimentos, auditando e certificando sua eficiência e eficácia. Eles realizam estudos minuciosos, que resultam em um *framework* com as melhores práticas relacionadas ao desenvolvimento de sistemas e, com ele os profissionais e engenheiros de *software* buscam atender suas necessidades.

O profissional de *software* deve ser capaz de entender e aplicar as tecnologias emergentes no mercado para produção de sistemas que respondam às demandas dos clientes e de seus negócios.

VOCÊ QUER VER?

Steve Jobs tinha uma percepção aguçada do que as pessoas desejavam em um produto tecnológico. Era comum ele adiar o lançamento de um produto porque não estava satisfeito com o *design*. No filme *Steve Jobs* (SORKIN, 2015), o roteiro mostra como essa percepção motivava sua equipe a sempre pensar além da parte técnica das máquinas.

Para começar o desenvolvimento de um sistema é necessário criar metodologias para entender o que o cliente precisa e transformá-las em funcionalidades dentro de um produto de *software*. Para isso, é necessário aprender técnicas de levantamento, definição e modelagem de requisitos e de banco de dados, para passar para a linha de produção, utilizando linguagens de desenvolvimento de telas (*Front End*) e de aplicação de lógicas sistêmicas (*Back End*) e, se necessário, comunicando e desenvolvendo uma bases de dados. Em seguida, é o momento da aplicação de testes funcionais, para verificar se a aplicação desenvolvida está conforme as exigências do cliente, podendo assim, ser disponibilizada para utilização.

Observe que são envolvidos diversos profissionais para o desenvolvimento de sistemas, com responsabilidades diferentes dentro de um projeto. Ao entender o que cada um faz e as habilidades específicas necessárias, você pode se aprofundar em suas áreas de interesse, alinhadas com seu perfil.

O analista de sistema faz a mediação da comunicação entre o cliente/usuário com a equipe de produção. Para que essa comunicação seja feita de forma eficiente, ele deve utilizar técnicas e metodologias definidas pela Engenharia de *Software*, de forma que as necessidades sejam transformadas em soluções de sistemas. Em uma equipe de produção de sistema é necessário também ter um profissional responsável pela administração e gerenciamento do banco de dados, um gestor de projetos, programadores de sistemas, *designers* de interface e analistas de testes.

1.1.2 Ética na Engenharia de Software

Quando alocados em um projeto em andamento, é comum que os profissionais iniciantes façam críticas a diagramas, artefatos e principalmente códigos produzidos por uma outra pessoa. Claro que buscamos sempre a perfeição, mas, com o passar do tempo, entendemos que deve haver um equilíbrio no desenvolvimento e essa crítica pode nos ajudar a melhorar nossos próprios processos. Ao analisar um código produzido, você deve pensar em fatores que podem ter influenciado nesta produção. Qual tempo destinado para a produção? Teve mudanças contínuas de escopo? Equipe e local adequado para o desenvolvimento? Teve comprometimento da equipe? Foram aplicadas métricas e metodologias da Engenharia de *Software*? Muitos são os fatores que podem impactar diretamente na qualidade da produção de um *software*.

1.2 Modelos de desenvolvimento de *software*

Vamos entender um pouco sobre os modelos utilizado em fábricas de *software* para o desenvolvimento de sistemas de informação. Uma fábrica define o modelo a ser aplicado para o desenvolvimento de um *software*, de acordo com as características delineadas e com o perfil dos profissionais que possui, a dimensão da equipe e tipos de produtos de desenvolve. É importante entender os mais variados modelos de produção para que seja fácil sua adaptação ao incorporar ou migrar de empresa ou equipe de trabalho.

Vamos entender isso melhor nos tópicos a seguir.

1.2.1 Processo Unificado de software

No desenvolvimento de sistemas é necessário ter uma melhor organização nas fases de desenvolvimento do produto, proporcionando uma continua melhoria nos processos de desenvolvimento. A criação do Processo Unificado veio com o objetivo de proporcionar uma melhor divisão entre as etapas de criação do *software*.

O engenheiro Roger Pressman (2016, p. 56), define o PU (Processo Unificado) como “uma tentativa de aproveitar os melhores recursos e características dos modelos tradicionais de processo de *software*”.

VOCÊ O CONHECE?

Roger S. Pressman é dos mais famosos escritores e engenheiro de *software*, uma autoridade reconhecida em todo o mundo em relação a melhoria de processos de desenvolvimento de *software*. Trabalhou como engenheiro de *software*, professor e escritor por décadas e, após receber o título de Ph.D em Engenharia, dedicou-se à vida acadêmica.

Assim, o Processo Unificado, representado na figura a seguir, começa com a fase de **concepção**, passando para as fases de **elaboração** e **construção**, e fechando a interação com a fase de **transição**.

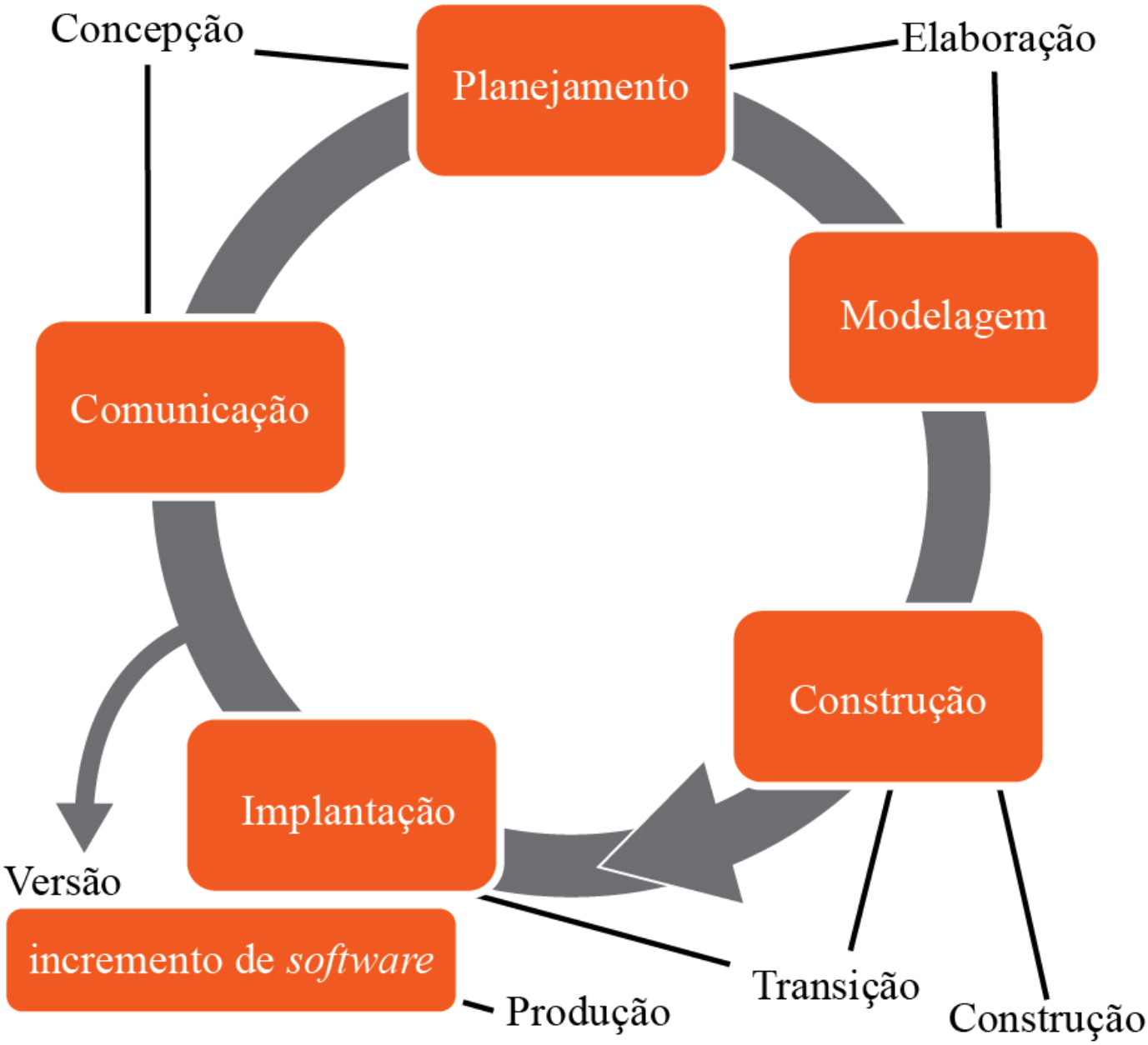


Figura 1 - O Processo Unificado é definido por quatro fases e cada uma tem disciplinas vinculadas. Fonte: PRESSMAN, 2016, p. 57.

A figura a seguir, detalha as fases de um ciclo de vida de um *software*, iniciando com a necessidade do cliente na construção de um sistema para melhorar ou sistematizar um determinado procedimento.

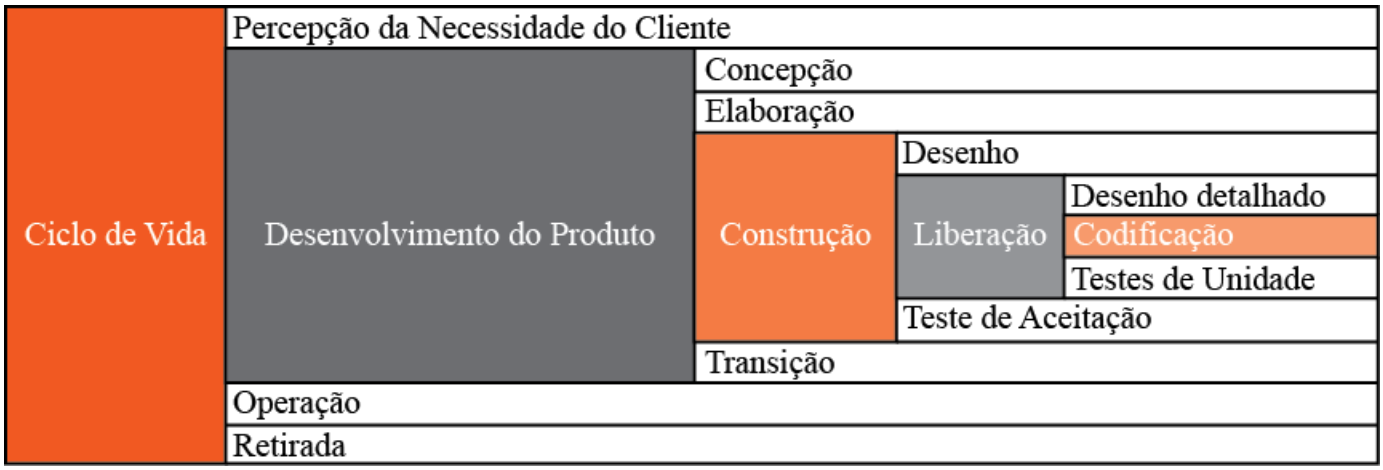


Figura 2 - Detalhamento das etapas de um ciclo de vida de um software, desde a identificação da necessidade até a retirada do produto. Fonte: Elaborada pelo autor, 2018.

Observe na figura, que o produto é desenvolvido e disponibilizado para operação, com base na necessidade, sendo utilizado por um determinado tempo, até que seja necessário um aprimoramento ou substituição por outro, retirando-o de operação.

Então, vamos detalhar um pouco mais a etapa de desenvolvimento do produto, passando por todas as fases.

Concepção

Fase de identificação por parte do cliente da necessidade de desenvolvimento de uma funcionalidade ou produto de *software,* a fim de resolver um determinado problema de processo ou sistematização. Com essa demanda, o analista de sistemas faz um *brainstorming* com o cliente e sua equipe, a fim de buscar o máximo de informações possíveis para compor o artefato de levantamento de requisitos. Em resumo é a fase de comunicação com o cliente e planejamento do que será desenvolvido (PRESSMAN, 2016).

Elaboração

Nesta fase, o analista de sistema deve montar o artefato de requisitos de *software*, ou seja, desenvolver requisitos funcionais, não funcionais e regras de negócios, acrescidos, se necessário, de outras informações e diagramas da engenharia de *software*. Este é o momento de definição dos requisitos mais importantes do *software*, definindo-os em prioridades e negociando com o cliente. “A negociação

é, sem dúvida, o ponto nevrálgico de toda e qualquer gestão empresarial. O precioso tempo que se investe em uma reunião ou o aval definitivo em um processo decisório determinam o sucesso ou o fracasso de uma organização" (WANDERLEY, 1998, p. 1).

Sem dúvida, este é um momento de grande importância no processo de desenvolvimento de *software*. Cabe ao analista desenvolver um bom planejamento (elaboração) de requisitos de *software*, para acertar com o cliente valores e prazos para conclusão e entrega dos requisitos. É aconselhável, neste momento, que o analista esteja munido de estratégias de negociação e acompanhado por um gerente de projetos. Para Pressman (2016, p. 57), "a fase de elaboração inclui as atividades de comunicação e de modelagem do modelo de processo genérico".

Existem técnicas de métricas de *software* que podem ser utilizadas neste momento do processo de desenvolvimento do sistema. Com a métrica de *software*, é possível mensurar o tamanho do sistema, quantidade de recursos, complexidade, custo, valor e prazo de desenvolvimento.

Construção

Esta é a fase de construção das telas, da parte lógica, do armazenamento em banco de dados e uso de todos os recursos necessários para o funcionamento do requisito, aplicando suas regras de negócio.

Para Pressman (2016, p. 57), "a fase de construção do PU é idêntica à atividade de construção definida para o processo genérico de *software*. Usando modelo arquitetural como entrada, a fase de construção desenvolve ou adquire componentes de *software* que vão tornar cada caso de uso operacional para os usuários finais".

Geralmente nas fábricas de *software*, o analista de sistemas repassa as responsabilidades de execução e o documento feito na fase anterior (Especificação de Requisitos de *Software*) para o Gerente de Projetos. Cabe ao Gerente de Projetos repassar as tarefas para sua equipe e monitorar principalmente custos e prazos acordados com o cliente.

Transição

Nesta fase, o produto é colocado à disposição dos usuários para o teste de aceitação. Após o deferimento da funcionalidade, é feita a transição das funcionalidades do servidor de homologação para o servidor de produção. “A fase de transição do PU abrange os últimos estágios da atividade genérica de construção e primeira parte da atividade genérica de implantação” (PRESSMAN, 2016, p. 57).

VOCÊ SABIA?

As empresas de desenvolvimento de sistemas utilizam diferentes processos. Por isso, é importante conhecer os processos utilizados pelas principais empresas do setor, em sua região ou estado. Faça um levantamento dessas empresas e de quais os principais produtos ou serviços que elas oferecem, para entender como o mercado está atuando.

Se for necessário e, caso tenha sido acordado na fase de elaboração, cabe, neste momento, a aplicação de um treinamento para os usuários do sistema. O treinamento é um ponto importante a ser negociado com o cliente, para evitar conflitos de relacionamento com ele. Normalmente, o contrato de prestação de serviço das fábricas de *software* define que o treinamento seja repassado para dois ou três usuários, sendo eles, responsáveis pela disseminação do conhecimento para todos da empresa.

1.2.2 Ciclo de vida do processo de desenvolvimento de software

Cada empresa define o modelo de ciclo de vida do desenvolvimento de *software* a ser utilizado, de acordo com as características discutidas anteriormente. Entretanto, é importante estudar as características de cada tipo, para entendimento do processo de desenvolvimento de *software* aplicado para cada organização.

A garantia do processo deve assegurar que o ciclo de vida do *software* utilizado e as práticas de engenharia de *software* estejam conforme o contrato e de acordo com o planejamento feito e, mesmo no caso de subcontratação, deve haver uma supervisão, para que os requisitos sejam conhecidos por todos os envolvidos e atendidos plenamente (BOENTE; MORÉ; COSENZA, 2009, p. 6).

Com uma visão empreendedora, além de estudar o propósito de cada tipo de processo de desenvolvimento de *software*, é possível fazer uma gestão de conflitos e interesses na relação com o cliente, para garantir sua satisfação com o produto que ele está recebendo.

Segundo Boente e Moré (2009, p. 1): “ter um cliente satisfeito é objeto de desejo de qualquer organização, independentemente se o cliente é interno ou externo, independentemente se estamos falando de organizações públicas ou privadas”.

Mais adiante, vamos entender melhor os conceitos de cada ciclo de desenvolvimento de *software* para fazer uma comparação entre eles e estimular visões e estratégias empreendedoras.

Algumas empresas recentes no mercado, e mesmo programadores que estão começando a aprender a codificar e criar sistemas, tem o hábito de não adotar um modelo de desenvolvimento de sistemas. Isso dificulta o desenvolvimento de uma análise dos requisitos de *software,* correndo o risco de não seguir as práticas definidas e consolidadas pela engenharia. Assim, por não pular essa fase, o programador começa a desenvolver sem nenhum planejamento, sendo necessário, em vários momentos, voltar em funcionalidades já criadas, para readaptação de algum procedimento. Nisso, ele perde produtividade, pois toda vez que é necessário voltar em uma funcionalidade já criada, o programador deve ler e interpretar todo o raciocínio lógico dela. A conformidade com os requisitos é o que vai atestar a qualidade do produto (KOSCIANSKI, 2007).

VOCÊ SABIA?

A maioria das empresas e profissionais que trabalha com desenvolvimento de sistemas não adota um modelo de ciclo de desenvolvimento de sistemas de qualidade, ou bem estruturado, gerando retrabalho e entregas de produtos incoerentes com o que foi solicitado.

Esse formato em codifica-remenda, sem planejamento, acarreta um alto risco para o projeto; não proporciona qualidade de *software*, conforme definição da engenharia, e gera dificuldades para uma futura manutenção ou aprimoramento.

Também pode gerar insatisfação do cliente, e a retirada e substituição do aplicativo, o que reduz o tempo de vida de utilização.

“Um *software* ou sistema de informação tem qualidade quando está adequado à empresa, ao cliente e/ou usuário e atende a padrões de qualidade predefinidos” (REZENDE, 1999, *apud* BOENTE; MORÉ; COSENZA, 2009, p. 2).

Como o formato codifica-remenda não possui um documento de especificação de requisitos, pela engenharia de *software*, se torna difícil mensurar a qualidade do produto ou do processo de desenvolvimento.

Vamos ver, a seguir, alguns dos modelos de desenvolvimento mais utilizados.

1.2.3 Cascata

O modelo em cascata foi o primeiro criado pela Engenharia de *Software*, em 1970, pelo cientista computacional Winston Royce. Este ciclo é caracterizado pela divisão em várias fases bem definidas, permitindo uma auditoria, ou pontos de controles, para avaliação de cada fase, garantindo os requisitos necessários para a próxima fase. É importante ter modelos para essa avaliação, pois isso garante a qualidade dos produtos de *software* (BOENTE; MORÉ; COSENZA, 2009).

VOCÊ O CONHECE?

Winston W. Royce (1929-1995) foi o cientista computacional que desenvolveu o modelo de ciclo de vida em cascata. Ele era PhD em Engenharia Aeronáutica e também foi diretor na *Lockheed Software Technology Center*, em Austin, Estados Unidos, durante os anos 1980.

Sobre os pontos de controle, para que tudo o que foi definido no projeto de *software* seja realizado, é possível criar um grupo de SQA (*Software Quality Assurance*), que vai ter a tarefa de ajudar a equipe de *software* a desenvolver um produto final de alta qualidade, com base em um conjunto de atividades (BOENTE; OLIVEIRA; ALVES, 2009).

Para Sommerville (2011, p. 33) o ciclo de cascata “considera as atividades fundamentais do processo de especificação, desenvolvimento, validação e evolução, e representa cada uma delas como fases distintas, como: especificação de requisitos, projeto de *software*, implementação, teste e assim por diante”.

Na figura a seguir, podemos visualizar as etapas do ciclo de vida cascata.

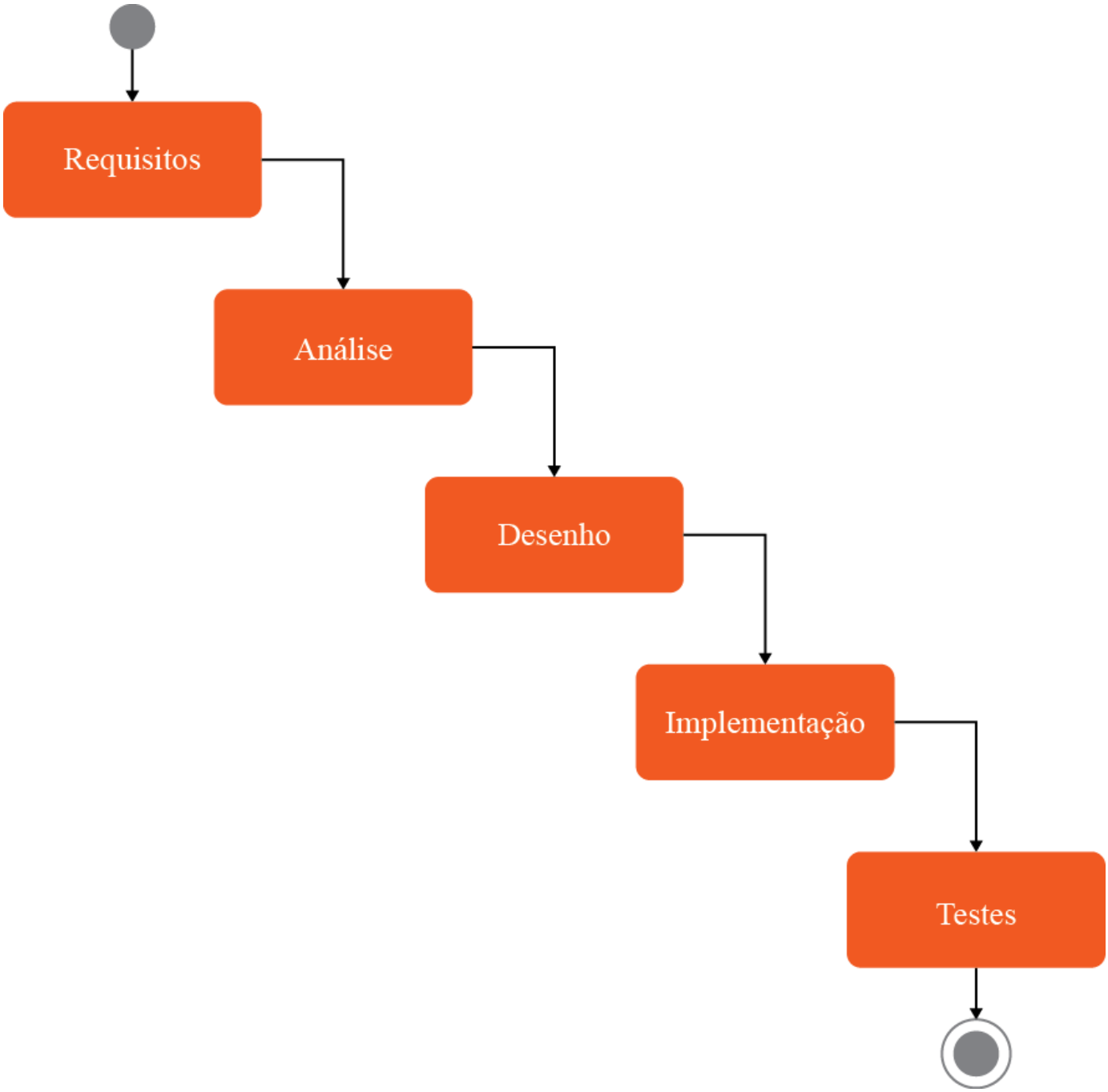


Figura 3 - Representação do Ciclo de Vida Cascata e sua estrutura de fases definida. Fonte: RODRIGUES, 2005, p. 16.

Na figura acima, vemos que o ciclo de vida em cascata define suas etapas para a construção de *software*. As etapas de requisitos e análise são de definição das necessidades do cliente e das funcionalidades, passando assim, para a terceira e quarta etapa, na qual se inicia o desenho do funcionamento das telas e sua codificação. “Em princípio, o resultado de cada fase consiste de um ou mais documentos aprovados (‘assinados’). A fase seguinte não deve começar antes que a fase anterior tenha terminado” (SOMMERVILLE, 2011, p. 21).

Atualmente, há gestores de TI com esse tipo de visão, em que repassam as responsabilidades do desenvolvimento do *software* para o cliente, gerando um documento de aprovação após a conclusão de cada fase. Esse procedimento garante minimizar esforço de retrabalho, não permitindo após a aprovação, voltar às fases anteriores. Entretanto, isso pode provocar insatisfação do cliente, já que no momento de assinatura do documento de aprovação, o cliente ainda não tem a abstração necessária sobre um determinado assunto que gerou o conflito entre as partes.

Pressman (2016) define o modelo cascata em cinco etapas: Comunicação, Planejamento, Modelagem, Construção e Implantação, conforme vemos na figura a seguir.

Figura 4 - Representação moderna do Ciclo de Vida Cascata e sua estrutura de fases definida, conforme visão de Pressman. Fonte: PRESSMAN, 2016, p. 42.

Podemos obter outras informações importantes da análise desse procedimento. Por exemplo, a entrega de algo funcional para o cliente só se dá após a passagem de todas as fases, acarretando uma baixa visibilidade para ele. O diferencial da abordagem da figura acima, que representa o Ciclo de Vida Cascata e sua estrutura de fases definida conforme a visão de Pressman (2016), em relação à figura anterior, que mostra sua estrutura de fases definida, é a utilização clara de etapas de gestão e planejamento do projeto, aplicando métricas de estimativas, cronogramas e pontos de controle, conforme estipulado na fase de planejamento.

1.2.4 Cascata com realimentação

Percebendo a insatisfação por parte do cliente ao não permitir voltar às fases anteriores, no modelo cascata com realimentação é permitido voltar para alguma correção. Neste modelo, a grande polêmica é conseguir mensurar o nível de retrabalho em fases anteriores, podendo até inviabilizar o projeto. Neste modelo é permitido voltar a apenas uma fase anterior, ou seja, não é permitido voltar duas fases. Isso geraria uma grande possibilidade de retrabalho de forma, podendo até inviabilizar o projeto caindo no formato codifica-remenda. Veja na figura a seguir, a sequência de estrutura cascata com realimentação e a indicação dos retornos.

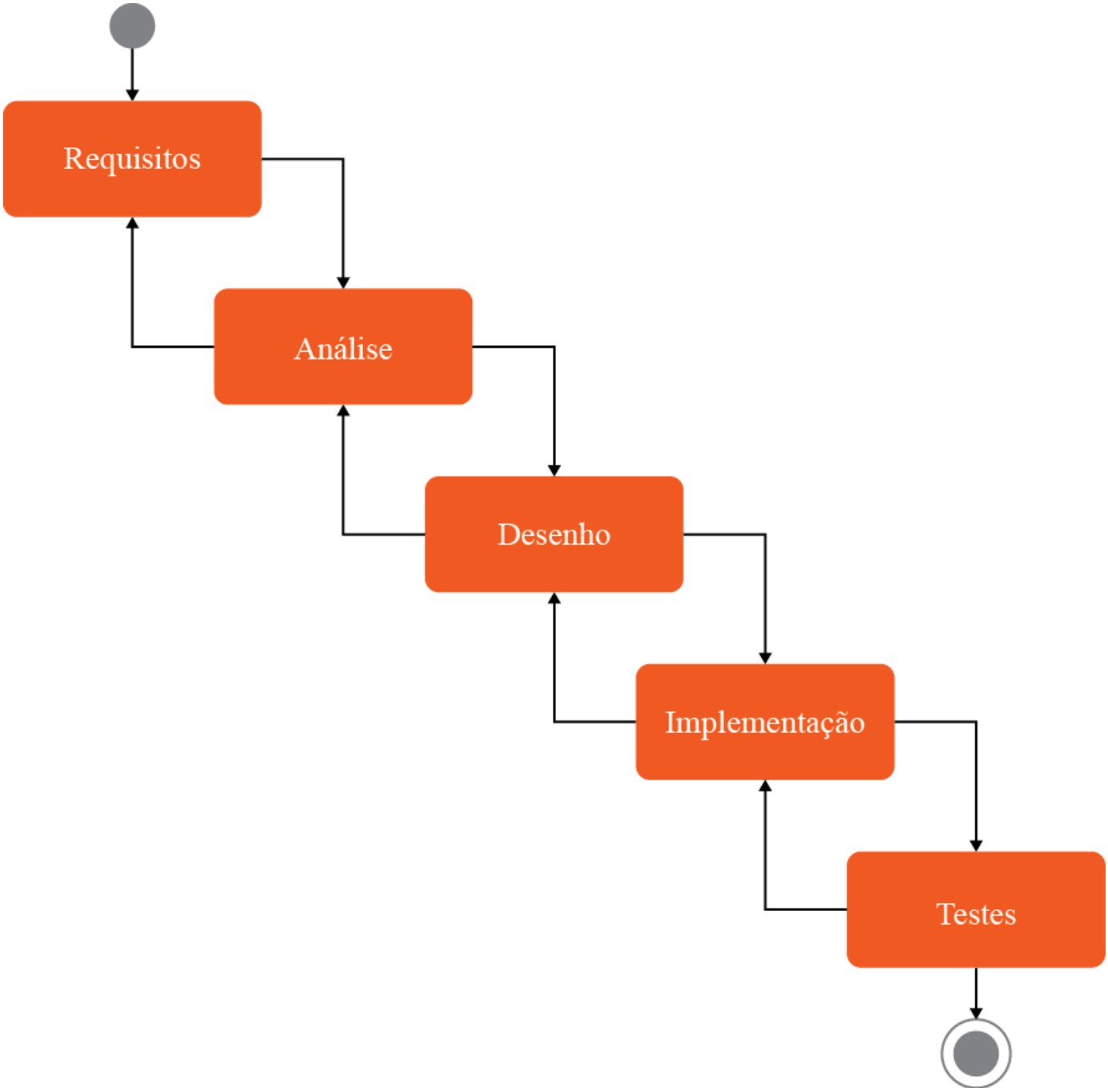


Figura 5 - Representação do ciclo de vida cascata com realimentação e sua estrutura, com indicação de retorno nas fases anteriores. Fonte: RODRIGUES, 2005, p. 17.

Resumindo, este modelo tem como características:

- pontos de controle bem definidos;
- baixa visibilidade para o cliente;
- entrega do produto como um todo (finalização de todas as etapas) e retrabalho em fases.

A seguir, vamos conhecer mais um modelo de desenvolvimento.

1.2.5 Espiral

Nos modelos anteriores, o grande desafio era garantir a integridade dos requisitos de sistemas durante as fases de desenvolvimento, já que o cliente tem acesso ao produto somente no final do processo de desenvolvimento de *software*. O ciclo de vida espiral tem como característica básica, o formato de divisões de um determinado sistema em módulos funcionais. Dessa forma, o cliente tem a possibilidade de utilizar os módulos (partes) já liberados e, com isso, a fábrica de *software* tem maior facilidade de fidelização do cliente. Na figura a seguir vemos o modelo espiral e sua dinâmica de iteração.

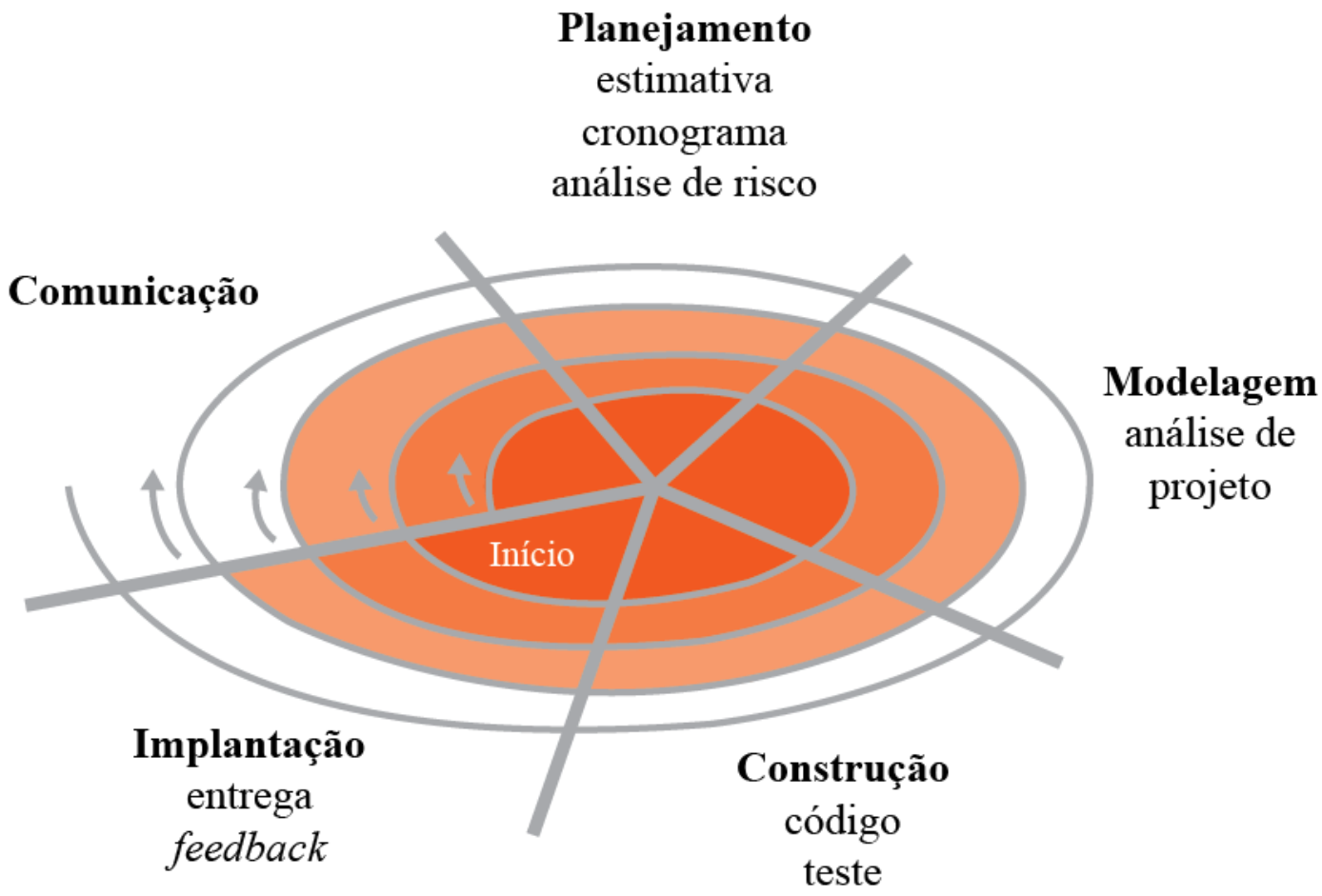


Figura 6 - Representação do ciclo de vida espiral, iniciando na fase de comunicação e finalizando na implantação. Fonte: PRESSMAN, 2016, p. 48.

Este modelo é dividido em cinco fases, sendo:

- **comunicação** - primeiro contato com o cliente e entendimento das necessidades;
- **planejamento** - abstração das necessidades, momento de aplicação do *brainstorming* e ativação de uma nova unidade de gerenciamento ou módulo;
- **modelagem** - consolidação e especificação dos requisitos;
- **construção** - construção das telas, codificação e testes, e;
- **implantação** - transição dos artefatos funcionais do servidor de homologação para o de produção.

Ao visualizar o processo de *software* em uma espiral, Sommerville (2011, p. 46) destaca que não se trata de uma sequência de atividades com alguns retornos de uma para outra, pois “cada volta na espiral representa uma fase do processo de *software*”.

Assim, é possível observar que o modelo espiral possui **iterações**, ou seja, metas parciais entregues para o cliente e cada volta no espiral representa um novo ciclo.

Uma das qualidades deste modelo é a gestão de riscos, já que um “grande problema” é dividido em partes menores, tornando o gerenciamento mais eficaz. Essa é a principal diferença entre este modelo e os outros modelos de processo de *software* (SOMMERVILLE, 2011, p. 47). Este modelo possui a possibilidade de abstrações de novos requisitos, tanto para o analista quanto para o cliente, permitindo que o cliente passe uma informação mais refinada para os próximos módulos (iterações).

1.2.6 Prototipagem evolutiva

A prototipagem evolutiva é o modelo em que se desenvolvem versões “provisórias” (protótipos) para a aprovação dos requisitos. Em sistemas baseados em tecnologias para internet, a criação das telas se torna simples, com grandes possibilidades do protótipo das telas serem as mesmas telas da versão, que entrará em produção. Com a criação dos protótipos de tela, mesmo que ainda não funcionando, o cliente tem a oportunidade de abstrair melhor os requisitos do *software*, permitindo a mudança dos requisitos antes da codificação das telas, já que a codificação exige um esforço relativamente maior que a simples criação dos protótipos. Com a estratégia de protótipos se evita o retrabalho e se garante excelente visibilidade sobre as funcionalidades para os clientes. Para Rodrigues (2005, p. 18), “os protótipos cobrem cada vez mais requisitos, até atingir o produto desejado”. Isso permite também uma gestão de risco mensurável, já que o *software* é dividido por iterações e proporciona maior qualidade, além de facilitar o levantamento das regras de negócios. A figura a seguir representa o modelo de prototipagem evolutiva.

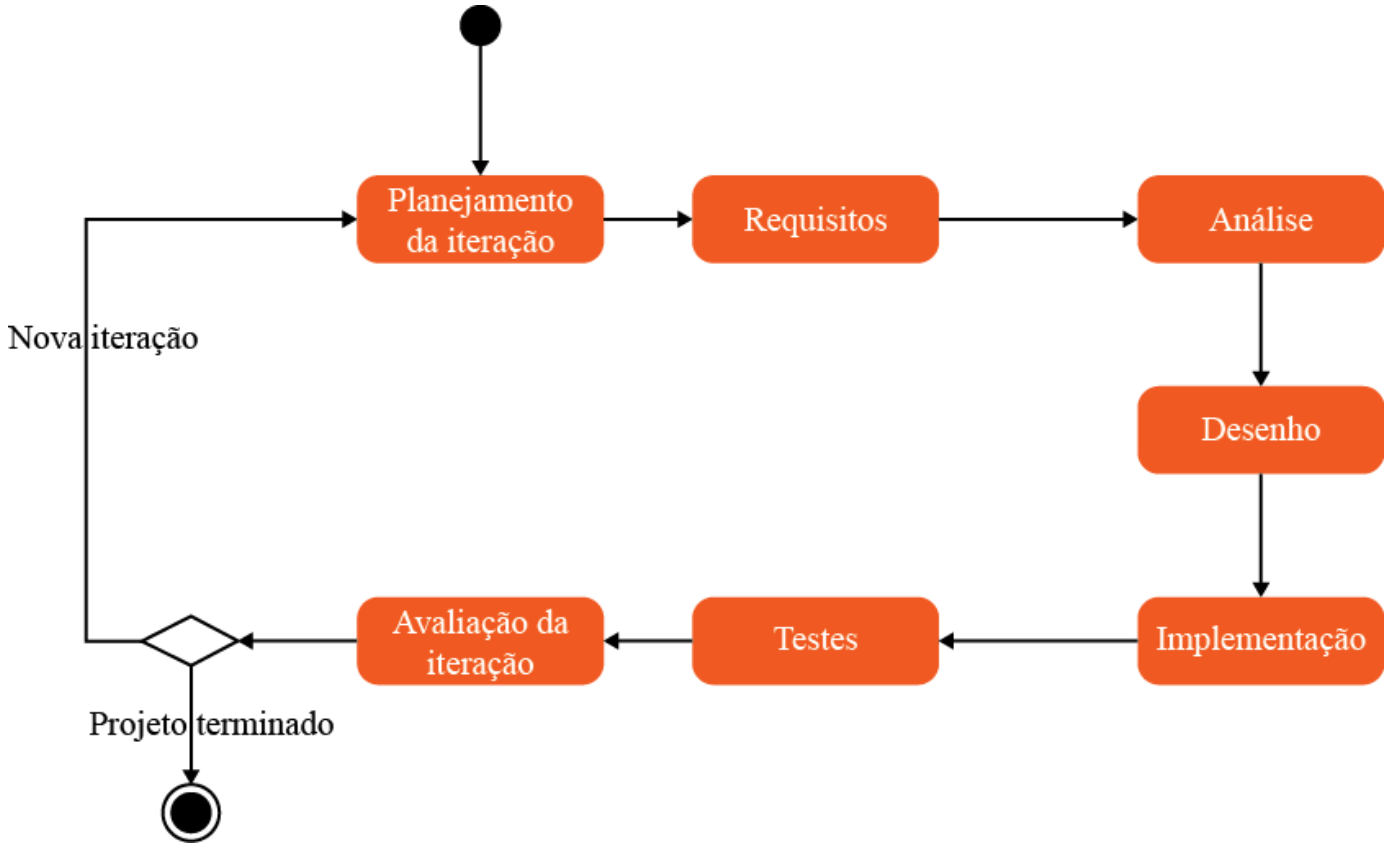


Figura 7 - Representação do ciclo de vida prototipagem evolutiva e sua definição estrutural em um formato espiral. Fonte: RODRIGUES, 2005, p. 18.

A grande adaptação deste modelo é a inclusão da fase de desenho, na qual é realizada a prototipagem do sistema e suas funcionalidades. Para Sommerville (2011, p. 44), “um protótipo é uma versão inicial de um sistema de *software*, usado para demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções”.

Diversos recursos são úteis para uma consolidação dos requisitos, principalmente o protótipo e diagramas de casos de uso, com ênfase no primeiro. O cliente consegue visualizar mais facilmente o que será o produto final, quando tem contato com algo menos abstrato. Dessa maneira, ele já consegue, inclusive, dar sugestões e verificar se os requisitos serão atendidos.

1.2.7 Entrega evolutiva

A prototipagem evolutiva é o modelo que combina os modelos cascata e prototipagem evolutiva. Este ciclo tem como características: etapa de abstração do problema, que enfatiza aspectos visuais do sistema, entrega evolutiva; permite melhor gerência dos projetos, gerenciamento de riscos e permite uma melhor visão do sistema por parte do cliente. Para Rodrigues (2005, p. 19), “entrega

evolutiva enfatiza as funcionalidades centrais do sistema que são improváveis de serem alteradas pelo *feedback* do cliente". O objetivo deste ciclo é separar as etapas de análise de sistemas (requisitos, análise e desenho arquitetônico) das etapas relacionadas à produção em iterações funcionais (desenho detalhado, implementação, testes e avaliação da iteração). Na figura a seguir, vemos uma representação do ciclo de vida de entrega evolutiva.

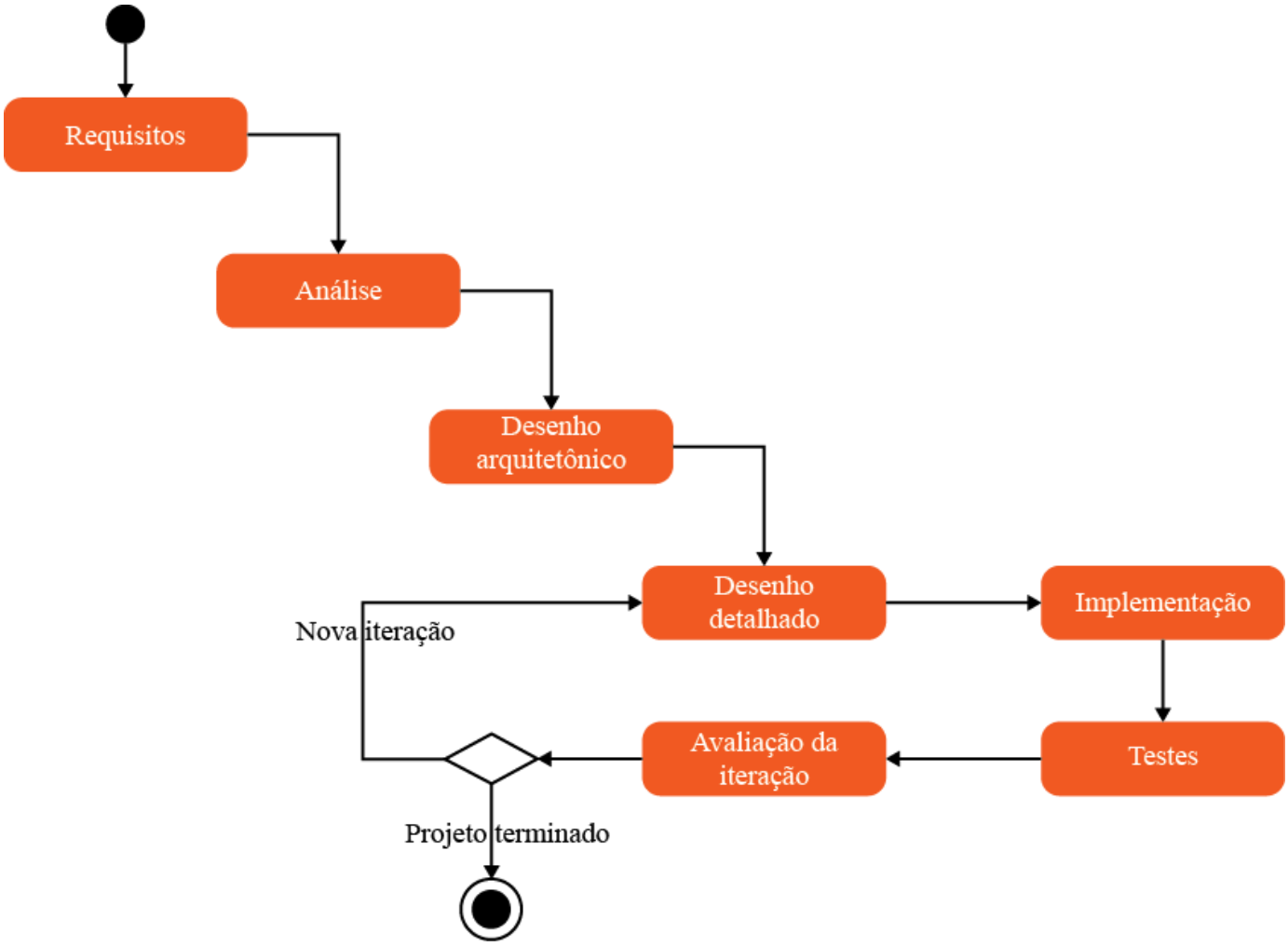


Figura 8 - Representação do ciclo de vida entrega evolutiva, separando a fase de análise de sistemas e gestão de projetos. Fonte: RODRIGUES, 2005, p. 19.

Este modelo também contempla a utilização de protótipo em dois formatos:

a) desenho arquitetônico, apresentando um protótipo geral e posicionamento dos objetos e funcionamento da interface e;

b) desenho detalhado, no qual é definido, por meio de protótipo, o formato e operação da funcionalidade que será desenvolvida na iteração.

O modelo entrega evolutiva tem como papel principal evitar o desperdício de mão de obra no desenvolvimento de *software*, principalmente dos arquitetos de *software*. Neste modelo, o arquiteto de *software* faz a análise de requisitos e desenho arquitetônico do projeto, como um todo. Após finalizar a análise do *software*, o projeto é repassado para um gerente de projeto, que deve cuidar da construção do sistema, seguindo as etapas de: desenho detalhado, implementação, testes e avaliação da iteração.

1.2.8 RUP

O RUP (*Rational Unified Process* ou Processo Unificado da *Rational*) é um processo proprietário da empresa IBM (International Business Machines), que fornece técnicas e artefatos a serem seguidos pelos colaboradores de uma equipe de desenvolvimento de *software*, com o objetivo de melhorar os processos e aumentar a produtividade.

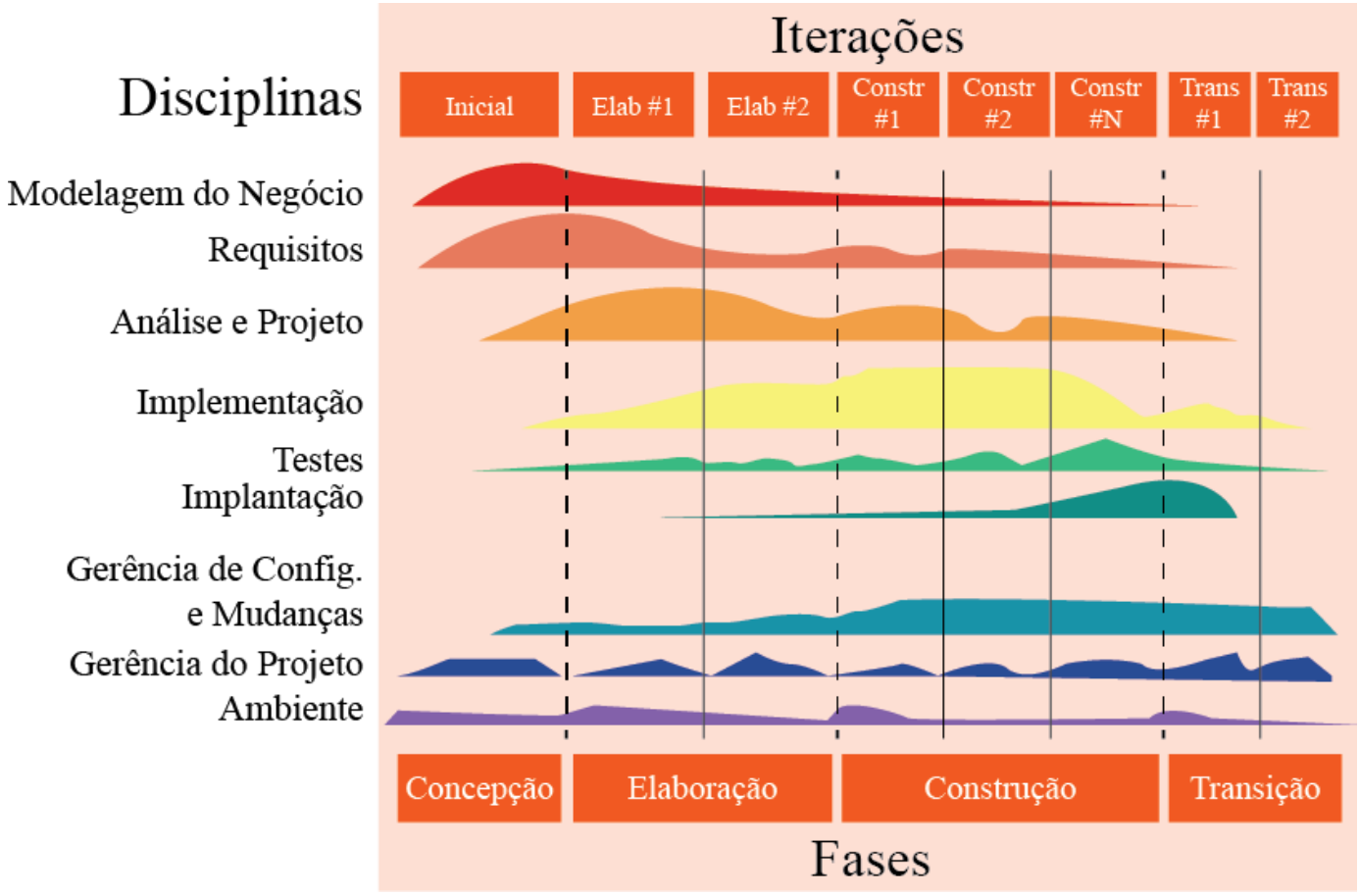


Figura 9 - Representação do ciclo de vida RUP: disciplinas x fases. Fonte: IBM BRASIL, 2008, p. 3.

O processo RUP promove uma aproximação disciplinada para atribuir tarefas e responsabilidades em uma organização de desenvolvimento, com a meta de garantir que a produção do *software* tenha alta qualidade e esteja de acordo com as necessidades do usuário final. Isso com um orçamento bem estabelecido e previsível (RATIONAL, 1998, tradução nossa).

As características são: ciclo de iterações, fases, prazo e esforço entre as etapas bem definidas, disciplinas e artefatos bem delineados, qualidade de desenvolvimento de *software*, baixo risco de desenvolvimento e possibilidade de desenvolvimento incremental.

Uma vantagem do RUP destacada por Rodrigues (2005, p. 20) é que “as fases podem ser divididas em iterações, que entregam parte da funcionalidade antecipadamente para o cliente. Isso reduz o risco, pois o cliente pode visualizar se o que foi desenvolvido é o que ele deseja”.

Na representação da figura abaixo, vemos a mensuração dos pontos de controle (marcos) do processo de desenvolvimento de *software* do RUP.

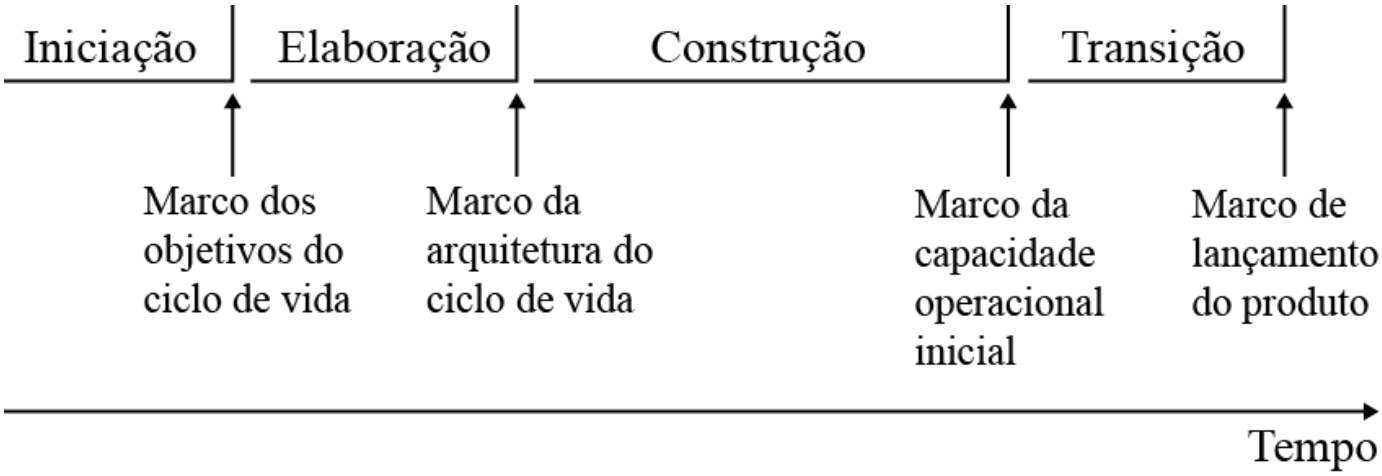


Figura 10 - Marcação dos pontos de controle do ciclo de vida do RUP. Fonte: BOENTE, OLIVEIRA, ALVES, 2009, p. 12.

A grande vantagem do desenvolvimento do *software*, utilizando a metodologia RUP é a possibilidade de o gerente de projeto saber o nível de mão de obra, que cada profissional da sua equipe vai ter, e quando vai precisar desta mão de obra. Isso é possível acompanhar no gráfico da figura “Representação do ciclo de vida RUP: disciplinas x fases”, vista anteriormente. Este gráfico é popularizado no

mercado como “o gráfico das baleias”, pela semelhança visual: quanto maior é a “barriga da baleia”, em uma determinada fase, maior o esforço do profissional, com base nas disciplinas.

VOCÊ QUER LER?

A *Rational Software* é uma família de *software* da IBM para desenvolvimento, suporte, gerência, construção e desenvolvimento de projetos de desenvolvimento de *software*. Você pode ler mais sobre ela no endereço: <https://www-01.ibm.com/ (https://www.ibm.com/developerworks/br/rational/?cm_mc_uid=00254170293015283098742&cm_mc_sid_50200000=66114401528309874236&cm_mc_sid_52640000=37609411528309874240)*software (https://www.ibm.com/developerworks/br/rational/?cm_mc_uid=00254170293015283098742&cm_mc_sid_50200000=66114401528309874236&cm_mc_sid_52640000=37609411528309874240)*/br/rational/ (https://www.ibm.com/developerworks/br/rational/?cm_mc_uid=00254170293015283098742&cm_mc_sid_50200000=66114401528309874236&cm_mc_sid_52640000=37609411528309874240)>.

A figura a seguir, ilustra a proporção dos riscos nas fases do processo de desenvolvimento, comparando o RUP em relação ao modelo de cascata.

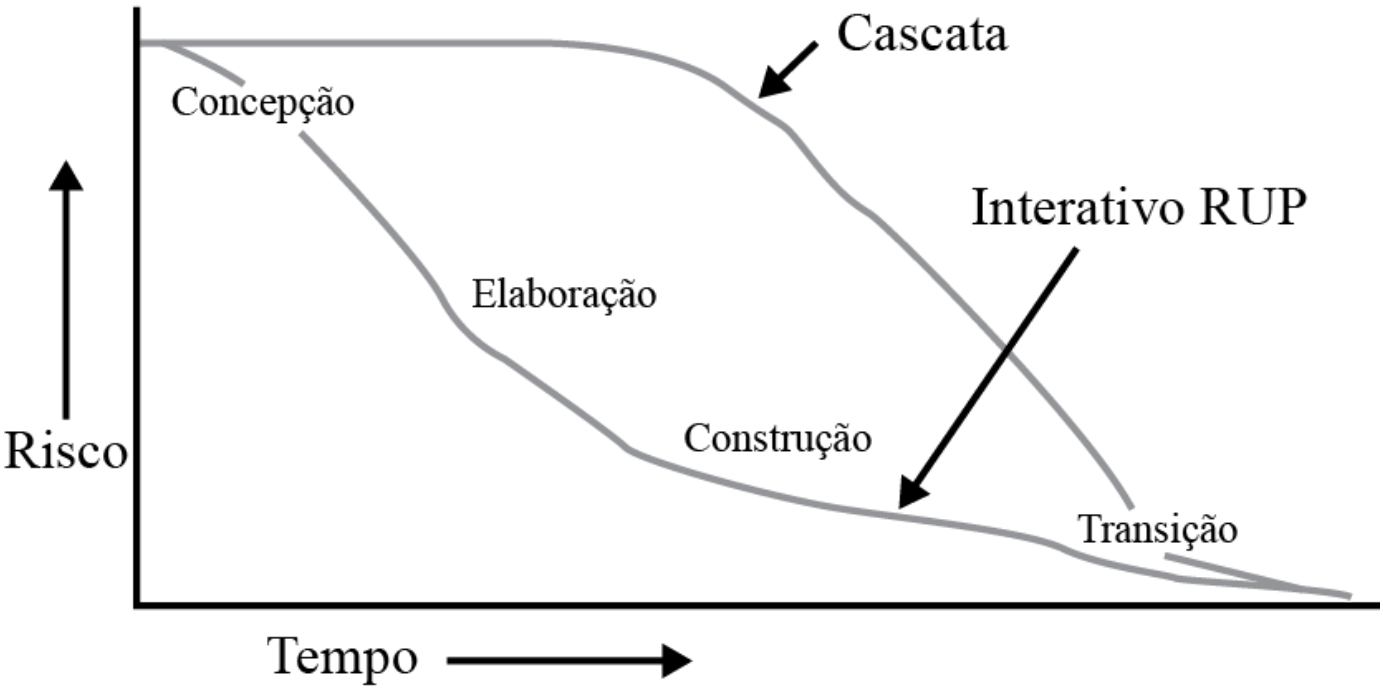


Figura 11 - Comparação do risco nos ciclos de vida do RUP e cascata, avaliando o risco em uma linha do tempo. Fonte: RODRIGUES, 2005, p. 22.

Como metodologia de desenvolvimento de *software*, o RUP gera qualidade, "produtividade no desenvolvimento, operação e manutenção de *software*, controle sobre o desenvolvimento dentro de custos, prazos e níveis de qualidade desejados, sem deixar de levar em conta a estimativa de prazos e custo com maior precisão" (BOENTE; OLIVEIRA; ALVES, 2009, p. 12).

Os modelos de ciclo de desenvolvimento de *software* que vimos aqui, são normalmente aplicados dentro de um fábrica de *software* tradicional, com equipes médias ou grandes. Existe também no mercado as empresas de pequeno porte ou que possui uma parte de sua equipe trabalhando *outsourcing*, ou seja, nas instalações do cliente. É uma estratégia das fábricas de *software* optar pelo *outsourcing* quando há clientes com requisitos vagos ou em constantes mudanças. Para casos assim, é recomendada a utilização de metodologias ágeis, como as que veremos a seguir.

1.3 Métodos ágeis de desenvolvimento de *software*

Diversas metodologias foram criadas pela Engenharia de *Software* para sistematizar e padronizar o desenvolvimento do *software*. As metodologias tradicionais que vimos no tópico anterior, são incorporadas nas fábricas de *software* e os métodos ágeis para aplicação *outsourcing*, sendo necessárias à consideração de um novo paradigma de desenvolvimento. As metodologias ágeis prometem melhorias na produção de *software* e em sua qualidade.

Para se escolher o uso de uma metodologia ágil é preciso avaliar sua viabilidade junto às necessidades do sistema a ser desenvolvido. Alguns conceitos não se aplicam mais na época atual, mas em contrapartida, podemos incorporar vários outros para ajudar no desenvolvimento de aplicações para o cliente. Alguns questionamentos sobre essas metodologias são aplicáveis, como listamos abaixo.

- Qual a melhor metodologia ser seguida?
- Posso criar minha própria metodologia de trabalho, conforme o ambiente da organização?
- As metodologias ágeis funcionam em fabricas de *softwares*?

O *outsourcing* designa a ação que existe por parte de uma organização em obter mão-de-obra terceirizada, de fora da empresa. A fábrica reconhece a necessidade de se desenvolver o *software* nas instalações do cliente, ou seja, *outsourcing*, quando os requisitos do *software* são extremamente vagos, ou de grande complexidade para especificação.

1.3.1 Extreme Programming *(XP)*

A Programação eXtrema (*eXtreme Programming*), ou simplesmente XP, é uma metodologia aplicável em pequenas e médias equipes. Normalmente, a XP é adotada em um ambiente de requisitos vagos, de difícil definição e constante mudanças. Neste ambiente, a utilização de metodologias ágeis ganha força, permitindo ajustes ao longo do desenvolvimento e possibilitando pequenas versões que serão imediatamente incorporadas e disponibilizadas para os usuários.

Os requisitos, neste tipo de programação, são expressos como cenários (que recebem o nome de histórias do usuário), que são implementados diretamente como uma série de tarefas, uma linha de produção (SOMMERVILLE, 2011, p. 58). Os programadores podem trabalhar em pares e desenvolver testes para cada tarefa, antes da escrita do código. Os testes são implementados, logo que um novo recurso é integrado à versão final disponível para os usuários. Sommerville (2011, p. 59), destaca como valores e princípios da XP:

- equipe de desenvolvimento e cliente devem estabelecer uma boa comunicação;
- desenvolvimento por iteração, focando a simplicidade;
- tendo requisitos tão vagos, é fundamental uma cultura de *feedbacks* constantes em relação ao desenvolvimento e satisfação do cliente;
- desenvolvimentos incrementais, baseados nas necessidades do dia-a-dia;
- receptividade a mudanças, sabendo que é sinônimo da prestação de serviço.

A metodologia XP se baseia em conceitos que devem ser observados em sua adoção pela equipe de desenvolvimento. Na lista a seguir, de acordo com Teles (2004), vemos alguns deles.

- **Planejamento:** desenvolvimento feito por semana, chamado de iterações. No início da semana (reunião de planejamento) são identificadas as prioridades das funcionalidades, junto ao cliente. Neste momento também são feitas as estimativas, pelos desenvolvedores, do tempo gasto para cada tarefa. O cliente tem a liberdade de alterar a ordem de prioridade das tarefas a qualquer momento. Ao final da tarefa, o cliente recebe a funcionalidade para aplicar o teste de aceitação e validação, se está conforme o requisito levantado.
- **Pequenas versões:** sempre que é feito um conjunto de funcionalidades é liberada uma nova versão do sistema. A tendência da metodologia ágil é a liberação de versões de forma mais rápida, assim que um conjunto mínimo de tarefas foram integradas.
- **Metáfora:** entender a realidade do cliente e utilizar os termos de comunicação do negócio é importante para a sintonia entre as partes, evitando que a equipe técnica utilize conceitos e palavras da área.
- **Projeto simples:** o XP deve ser simples. O projeto deve ser elaborado para durar apenas uma semana. Deve fazer justamente o que foi solicitado. Nos *sprints* seguintes, as funcionalidades serão melhoradas e novos recursos serão incluídos.
- **Time coeso:** a equipe de desenvolvimento e o cliente devem ter uma comunicação frequente.
- **Testes de aceitação:** teste feito pelo cliente avaliando se está conforme solicitação.
- **Ritmo sustentável:** o ritmo de trabalho normal em um projeto bem planejado, costuma ser de 40 horas/semana, 8 horas/dia, evitando horas extras. Se o projeto necessitar de horas extras por mais de duas semanas, é porque tem um problema mais grave que precisa ser corrigido.
- **Reuniões em pé:** reuniões literalmente em pé, rápidas e com duração de, no máximo, 10 minutos.
- **Posse coletiva:** o código pode ser alterado por qualquer outro desenvolvedor, mesmo não sendo o criador.

- **Programação em pares:** dois desenvolvedores trabalham em um único computador.
- **Padrões de codificação:** o padrão de codificação deve ser único na equipe.
- **Desenvolvimento orientado a testes:** o primeiro passo é a elaboração do teste unitário. Depois é criado o código compatível para que o teste funcione, abordagem contrária do que diz a Engenharia de *Software* tradicional.
- **Refatoração:** processo de melhoria contínua do código da funcionalidade.
- **Integração contínua:** sempre que uma funcionalidade é criada ou alterada, uma nova versão é imediatamente incorporada e disponibilizada para o usuário.
- **Cliente no desenvolvimento (*on-site*):** cliente precisa estar presente e disponível em tempo integral para que os desenvolvedores possam solucionar suas dúvidas, evitando, assim, problemas no entendimento e desempenho na produção.
- **Padrões de código:** padrões existem e são seguidos pelos desenvolvedores, facilitando a manutenção do código por qualquer membro da equipe.
- **Ambiente de trabalho aberto:** sala ampla, ao invés de cubículos.
- **Apenas regras:** time tem regras a serem seguidas, mas elas podem ser alteradas.

A metodologia XP é uma das mais importantes dentro do mundo ágil e, por isso, é importante conhecê-la. Mas há outras, então vamos estudar novas abordagens ágeis a seguir.

1.3.2 Scrum

A metodologia *Scrum* foi criada para a gestão de projetos ágeis e empresas de fabricação de automóveis e produtos de consumo. O método *Scrum* pode ser aplicado em equipes pequenas e médias, que têm tarefas dinâmicas, com mudanças frequentes.

Sommerville (2011, p. 64) define o método *Scrum* como uma abordagem de iterações com um período limitado, de 30 dias. Nesse espaço de tempo, são feitas reuniões diárias breves, de pé, que levantam três questões especiais, respondidas pelos membros da equipe.

O *Scrum* é utilizado principalmente no gerenciamento de projetos de *software*, mas pode ser utilizada em qualquer segmento de negócio. Normalmente é utilizado juntamente com outras metodologias ágeis no mercado, dentre elas a *Extreme Programming*.

VOCÊ QUER LER?

O quadro de MoSCoW é uma importante técnica para priorização de tarefas, criado por Dai Clegg, que classifica a ordem das tarefas em “tem que fazer”, “deve fazer”, “poderia fazer” e “interessante fazer”. Leia mais no artigo “A Técnica de Priorização MoSCoW” (OLIVEIRA, 2014): <https://issuu.com/ronielton/docs/artigoprince2moscow2014_mpbr (https://issuu.com/ronielton/docs/artigoprince2moscow2014_mpbr)>.

No tópico a seguir, vamos estudar a engenharia de requisitos, necessária para essas metodologias ágeis e para a Engenharia de *Software* tradicional.

1.4 Engenharia de requisitos

Os requisitos (no contexto da Engenharia de *Software*) representam o levantamento e abstrações de informações que contribuem com o processo de desenvolvimento de *software* e sua manutenção.

Para Pressman (2016, p. 167), “análise de requisitos resulta na especificação das características operacionais do *software*”.

A engenharia de requisitos é composta por seis fases distintas, sendo: requisitos, análise, projeto, implementação, teste e manutenção.

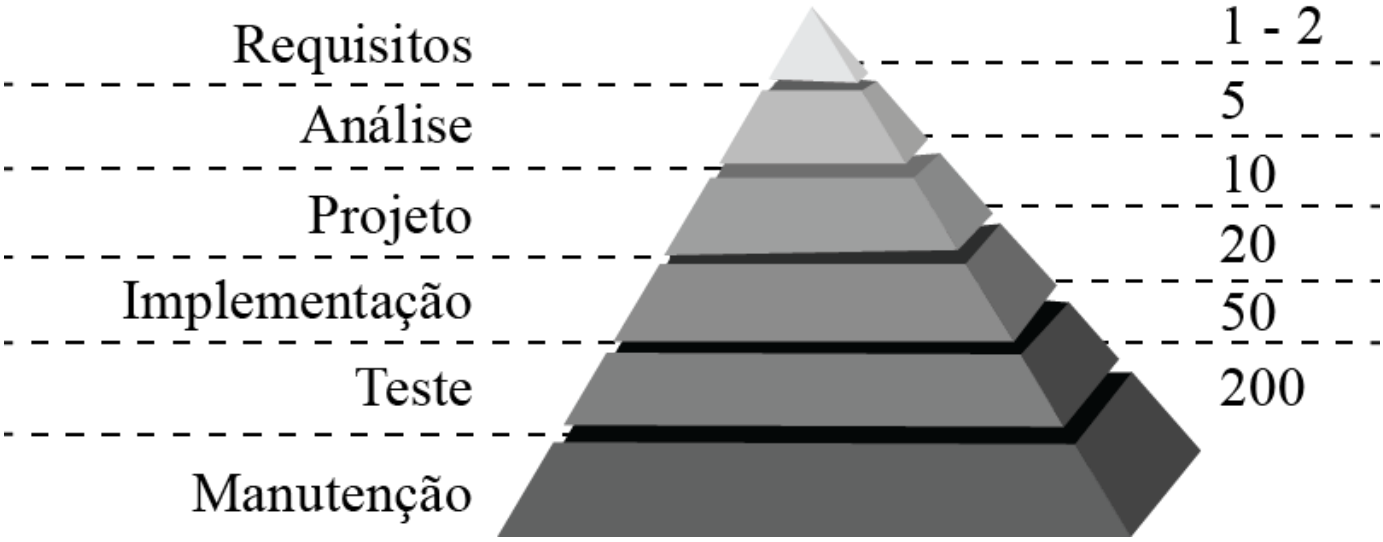


Figura 12 - Pirâmide de propagação de erro. Relação do tempo versus momento de identificação de um bug. Fonte: MAGELA, 2006, p. 13.

De acordo com Magela (2006, p. 13), podemos entender a pirâmide da seguinte forma:

> O usuário pode imaginar os números ao lado da figura [...] como valores monetários ou tempo em minutos de um requisito omitido ou com erro. Ou seja, omitir ou especificar erroneamente um requisito custa 20 vezes mais se sua descoberta ou correção for realizada na fase implementação e 200 vezes mais se for descoberta na manutenção.

Podemos compreender melhor a engenharia de requisitos a partir de três importantes conceitos: requisitos funcionais, não funcionais e regras de negócios. Vamos entender melhor cada um deles no próximo tópico.

1.4.1 Requisitos funcionais

Os requisitos funcionais descrevem as funcionalidades (telas) que o sistema de informação deve ter. Esses requisitos são “declarações de serviços que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e de como o sistema deve se comportar em determinadas situações” (SOMMERVILLE, 2011, p. 73), que também podem explicitar o que o sistema não deve fazer.

Para entender melhor isso, vamos ver dois exemplos de requisitos funcionais em um sistema bancário.

Exemplo Requisitos Funcional 1

O Sistema XB deve permitir que o atendente faça o cadastramento dos dados do cliente.

Exemplo Requisitos Funcional 2

O Sistema XB deve permitir que o gerente faça a abertura de conta para um novo cliente.

Para que o requisito tenha conformidade e evite problemas de interpretação do texto, algumas regras devem ser respeitadas para a construção da frase. Veja no requisito de exemplo, as marcações do sistema, ator e funcionalidade:

o Sistema XB[1] deve permitir que o gerente[2] faça a abertura de conta[3] para um novo cliente.

Sendo que:

[1] identificação do sistema no qual deve ser construída a funcionalidade;

[2] ator que provoca estímulos à funcionalidade. Um ator pode ser do tipo pessoa (gerente, vendedor, consultor e outros), sistema legado (outro sistema que envia ou recebe informações do sistema XB), ou um periférico (leitora de códigos de barras, leitora biométrica e outros periféricos de entrada e saída);

[3] requisito funcional de *software*.

1.4.2 Requisitos não funcionais

Os requisitos não funcionais são as qualidades ou características gerais de um *software*, no que diz respeito a sua facilidade de alterações (*changeability*), usabilidade, desempenho, reusabilidade, facilidade de instalação, portabilidade, concorrência, segurança, necessidade de treinamento, complexidade de processamento e eficiência *on-line*. Para Sommerville (2011, p. 73), os requisitos

não funcionais “são restrições aos serviços ou funções oferecidas pelo sistema. Incluem restrições de *timing*, restrições no processo de desenvolvimento e restrições impostas pelas normas”.

VOCÊ QUER LER?

O levantamento de requisitos certamente é um dos principais entendimentos que o analista de sistemas deve ter para iniciar a produção de um projeto. Para se aprofundar nesse assunto, recomendamos a leitura do capítulo quatro do livro ***Engenharia de Software***, de Ian Sommerville (2011).

Agora, vamos ver dois exemplos de requisitos não funcionais de um sistema de vendas.

Exemplo Requisito Não Funcional 1

O Sistema XB deve permitir o acesso via *Internet Explorer*, versões 7 e 8.

Exemplo Requisito Não Funcional 2

O Sistema XB deve funcionar em horário de atendimento da empresa, das 8h às 18h.

Observe que, para o requisito não funcional, também é informado qual o sistema que deve respeitar a característica escrita neste requisito.

1.4.3 Regras de negócio

As regras de negócio são todas das regras especificas de uma empresa ou ramo de atividade, que podem ou não ser incorporadas em um sistema de informação.

Para Larman (2007, p. 85), “regras de negócio (também chamadas de regras de domínio) descrevem tipicamente requisitos ou políticas que transcendem um projeto de *software* – elas são necessárias no domínio ou no negócio e muitas

aplicações podem precisar respeitá-las”.

Vamos entender melhor com dois exemplos de requisitos não funcionais de um sistema bancário (exemplo 1) e um sistema acadêmico (exemplo 2).

Exemplo Regra de Negócio 1

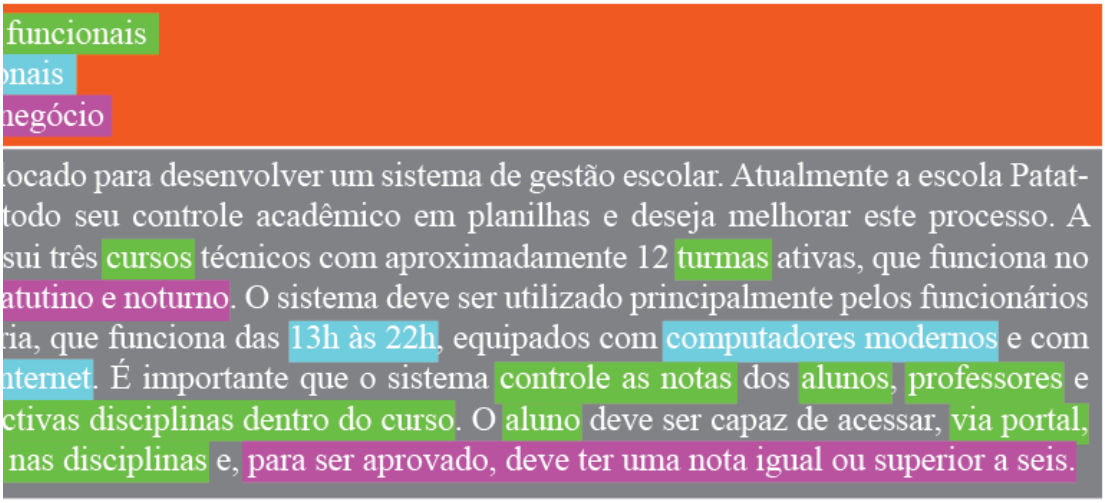
Para ativação da conta, o cliente deve depositar um valor mínimo de R$ 50,00 (cinquenta reais).

Exemplo Regra de Negócio 2

Para o aluno ser aprovado em uma disciplina, deve ter uma nota igual ou superior a 60 pontos.

SO

mpanhar uma descrição de necessidade de construção de um sistema para gestão escolar. Neste texto acados os possíveis requisitos funcionais, não funcionais e regras de negócio de forma melhorar o nto dos conceitos e aplicação na prática.


funcionais
nais
negócio

locado para desenvolver um sistema de gestão escolar. Atualmente a escola Patat- todo seu controle acadêmico em planilhas e deseja melhorar este processo. A sui três cursos técnicos com aproximadamente 12 turmas ativas, que funciona no atutino e noturno. O sistema deve ser utilizado principalmente pelos funcionários ria, que funciona das 13h às 22h, equipados com computadores modernos e com nternet. É importante que o sistema controle as notas dos alunos, professores e ctivas disciplinas dentro do curso. O aluno deve ser capaz de acessar, via portal, nas disciplinas e, para ser aprovado, deve ter uma nota igual ou superior a seis.

Ao realizar uma análise dos ciclos de vida, observamos que os ciclos Entrega Evolutiva, Prototipagem Evolutiva e o RUP, apresentam vantagens quando comparados aos de Cascata e Espiral.

Cada ciclo de desenvolvimento tem pontos de estudo que valem se estudados mais a fundo:

divisão clara de etapas e pontos de controles no modelo de cascata;

melhor gerenciamento de conflitos com clientes na questão de retrabalho no ciclo de cascata com realimentação;

divisão de módulos que proporciona melhor negociação com o cliente e qualidade no desenvolvimento de *software*;

criação de protótipos evolutivos para melhor abstração do sistema, por parte do cliente;

refinamento do processo de desenvolvimento de *software* tratado pelo ciclo de entrega evolutivo, de forma a proporcionar um melhor gerenciamento de mão de obra e com o RUP proporcionando um planejamento e gestão dos processos.

Com esses estudos, recomendamos algumas ações básicas que devem existir em uma fábrica de *software*, por exemplo:

a) dividir a fabricação do produto de *software* em etapas;

b) nos intervalos de fases, criar uma auditoria da fase que acaba de terminar, antes de passar para próxima etapa;

c) trabalhar com protótipos na fase de análise de requisitos;

d) entrega de unidades funcionais de *software* para o cliente, ao longo do desenvolvimento do *software* como um todo;

e) procurar utilizar métricas de *software*;

f) elaborar um documento padrão de análise e outras fases, de forma que atenda à realidade da fábrica;

g) escolher o ciclo de desenvolvimento, conforme qualificação dos profissionais que trabalham na fábrica, entre outras recomendações citadas neste trabalho.

Assim, podemos concluir que, dentre os modelos de ciclos de vida de desenvolvimento de sistemas que estudamos, o RUP proporciona maior quantidade de benefícios, mas para que este processo tenha sucesso é necessário uma equipe e gerentes qualificados tecnicamente para trabalhar com a ferramenta. Mas a realidade é que faltam profissionais disponíveis no mercado. Caso tenha uma equipe heterogênea, mas com um analista de sistemas com bastante experiência e conhecimentos, o modelo de Entrega Evolutiva pode ser aplicado com eficiência. Para os demais casos, o modelo de Prototipagem Evolutiva atende bem as necessidades.

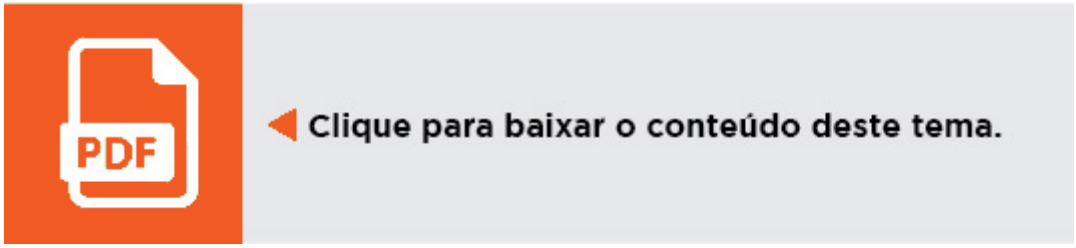
Síntese

Chegamos ao final deste capítulo. Aprendemos sobre os modelos de ciclos de vida de *software*, que possibilitam um melhor gerenciamento de conflitos com os clientes, aumento na produtividade e qualidade final do *software* e equilíbrio empresarial para as fábricas de *software* e metodologias ágeis. Vimos que essas metodologias são utilizadas com mais frequência em projetos em constantes mudanças e/ou requisitos vagos e também conhecemos conceitos relacionados à engenharia de requisitos.

Neste capítulo, você teve a oportunidade de:

- entender o que é Engenharia de *Software* e a sua importância;
- compreender que para desenvolver *softwares* distintos, podem ser necessárias técnicas diferentes de engenharia;
- conhecer algumas questões éticas e profissionais para engenheiros de *software*;
- compreender os conceitos e saber aplicar os modelos genéricos de processos de *software*;
- conhecer as atividades fundamentais do processo de engenharia de requisitos, desenvolvimento, testes e evolução de *software*;
- entender por que os processos devem ser organizados de maneira a lidar com as mudanças nos requisitos e projeto de *software*;
- compreender como o RUP integra boas práticas de Engenharia de *Software* na criação de processos de *software* adaptáveis;

- compreender os métodos ágeis de desenvolvimento de *software*;
- conhecer o manifesto ágil;
- saber diferenciar desenvolvimento ágil de desenvolvimento voltado a planos;
- conhecer as práticas de XP;
- compreender a abordagem *Scrum*;
- tomar ciência de questões de escalonamento de métodos ágeis;
- compreender os conceitos de requisitos de usuário e de sistema;
- entender por que os requisitos de usuário e de sistema devem ser escritos de formas diferentes;
- saber a diferença entre requisitos funcionais e não funcionais;
- compreender o documento de requisitos de *software*.



Bibliografia

BOENTE, A.; MORÉ J. Um modelo Fuzzy para avaliação da satisfação dos Gerentes de Projetos de *Software* de uma fundação pública estadual. **XXIX Encontro Nacional de Engenharia de Produção**, 2009.

BOENTE, A.; MORÉ, J.; COSENZA, H. Avaliação Fuzzy da qualidade de produtos de *software* numa fundação pública estadual. **VII Simpósio de Excelência em Gestão e Tecnologia**, 2009.

BOENTE, A.; OLIVEIRA, F.; ALVES, J. C. RUP como metodologia de desenvolvimento de *software* para obtenção da qualidade de *software*. **XXIX Encontro Nacional de Engenharia de Produção**, 2009.

IBM BRASIL. Rational Unified Process Best Practices for Software Development Teams. **Portal Rational Software**, 1998. Disponível em: < https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251

/1251_bestpractices_TP026B.pdf (https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf)>. Acesso em: 6/6/2018.

KOSCIANSKI, A. **Qualidade de *software***: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de *software*. 2. ed. São Paulo: Novatec, 2007.

LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientado a objetos e ao desenvolvimento iterativo. Tradução: Rosana Vaccare Braga. 3. ed. Porto Alegre: Bookman, 2007.

MAGELA, R. **Engenharia de Software aplicada – fundamentos**. Rio de Janeiro: Alta Books, 2006.

OLIVEIRA, R. R. **A Técnica de Priorização MoSCoW**. Management Plaza International, 2014. Disponível em: <https://issuu.com/ronielton/docs/artigoprince2moscow2014_mpbr (https://issuu.com/ronielton/docs/artigoprince2moscow2014_mpbr)>. Acesso em: 06/06/2018.

PFLEEGER, S. L. **Engenharia de *Software* - Teoria e Prática**. 2. ed. São Paulo: Pearson Addison Wesley, 2004.

PRESSMAN, R. **Engenharia de *Software***. 8. ed. Porto Alegre: AMGH, 2016. Disponível em: <https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset (https://Ânimabrasil.blackboard.com/webapps/blackboard/content/listContent.jsp?course_id=_198689_1&content_id=_4122211_1&mode=reset)>. Acesso em: 24/05/2018.

RATIONAL. ***Software* Corporation**. Sobre o Rational Unified Process. São Paulo, 2002. Disponível em: <ftp://public.dhe.ibm.com//software/pdf/br/RUP_DS.pdf (ftp://public.dhe.ibm.com//software/pdf/br/RUP_DS.pdf)>. Acesso em: 18/05/2018.

RODRIGUES, C. Á. **Engenharia de *Software***. Apostila da disciplina de Engenharia de *Software*, Curso de Especialização em Engenharia de *Software*, Centro Universitário UNA, 2005.

SOMMERVILLE, I. **Engenharia de *Software***. 9. ed. São Paulo: Pearson Addison Wesley. 2011.

SORKIN, A. **Steve Jobs**. Direção: Danny Boyle. Produção: Danny Boyle; Guymon Casady; Christian Colson; Mark Gordon; Scott Rudin. Estados Unidos, 2015.

TELES, V. M. **Extreme Programming**: aprenda como encantar seus usuários desenvolvendo *software* com agilidade e alta qualidade. São Paulo: Novatec, 2004.

WANDERLEY, J. A. **Negociação total**: encontrando soluções, vencendo resistências, obtendo resultados. São Paulo: Gente, 1998.