Unyleya EDUCACIONAL	**Material Complementar**

Disciplina:	Metodologias Ágeis
Autor:	André Luís Belmiro Moreira Ramos

Conceitos e aplicações das principais metodologias ágeis

Introdução

Durante muito tempo, a construção e manutenção de softwares foi baseada em metodologias tradicionais de desenvolvimento. A utilização de tais metodologias se mostrou como um avanço, se pensarmos que até meados de 1970 os sistemas eram produzidos sem levar em consideração qualquer tipo de processo.

A não utilização de processos bem definidos levou ao que foi chamado de crise do software e, consequentemente, ao surgimento da engenharia de software. Os problemas na época estavam relacionados ao não cumprimento de prazo dos projetos, além da baixa qualidade do que era produzido. Como consequência, não era incomum ter produtos que não satisfaziam os clientes, como também projetos ingerenciáveis em decorrência da dificuldade de manutenção do código.

Nesse contexto, um marco foi a realização da Conferência da OTAN sobre Engenharia de *Software,* que marcou o surgimento desta disciplina e teve como objetivo a discussão acerca de melhores práticas relacionadas ao desenvolvimento de software.

Figura 1. Conferência da OTAN sobre Engenharia de *Software*.

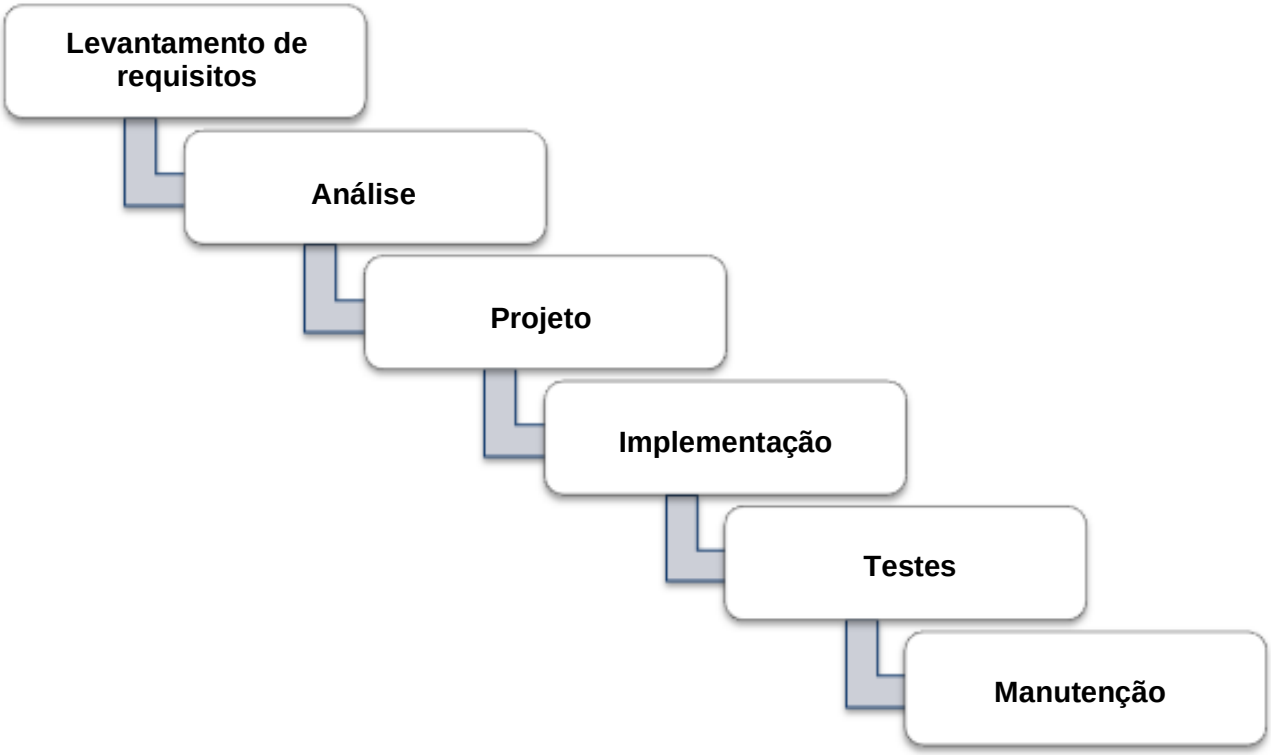
Fonte: Ramos, 2017.

A partir desta conferência, diversas iniciativas foram adotadas para que o desenvolvimento de sistemas apresentasse melhoras significativas em relação aos problemas citados anteriormente. Uma das iniciativas foi o encorajamento em relação ao uso de processo bem definidos de desenvolvimento.

As primeiras metodologias estabelecidas tiveram como inspiração o movimento da manufatura, já consolidada na época. Desta forma, as primeiras propostas tiveram como base o desenvolvimento de produtos tangíveis, fazendo com que a ideia principal fosse a divisão do processo em etapas, como planejamento, execução, testes e encerramento.

O primeiro modelo estabelecido foi o método cascata, também conhecido como ciclo de vida clássico, cuja principal característica é o sequenciamento de atividades, onde necessariamente uma atividade subsequente só tem início quando a anterior é finalizada. A figura abaixo representa o fluxo das atividades descritas no modelo em questão.

Figura 2. Fases do modelo cascata.



Fonte: O autor.

Mesmo possuindo vantagens em relação ao desenvolvimento sem a utilização de um processo bem definido, o modelo em cascata apresenta um grande problema: Dificilmente, na prática, conseguimos encerrar totalmente uma etapa para iniciar a próxima. Por exemplo, o fato de termos que estabelecer todos os requisitos ainda na fase inicial do projeto faz com que mudanças sejam desencorajadas, o que pode acarretar no desenvolvimento de um produto que não tenha utilidade para o cliente. De acordo com Ramos (2017), podemos citar ainda outros problemas, tais quais:

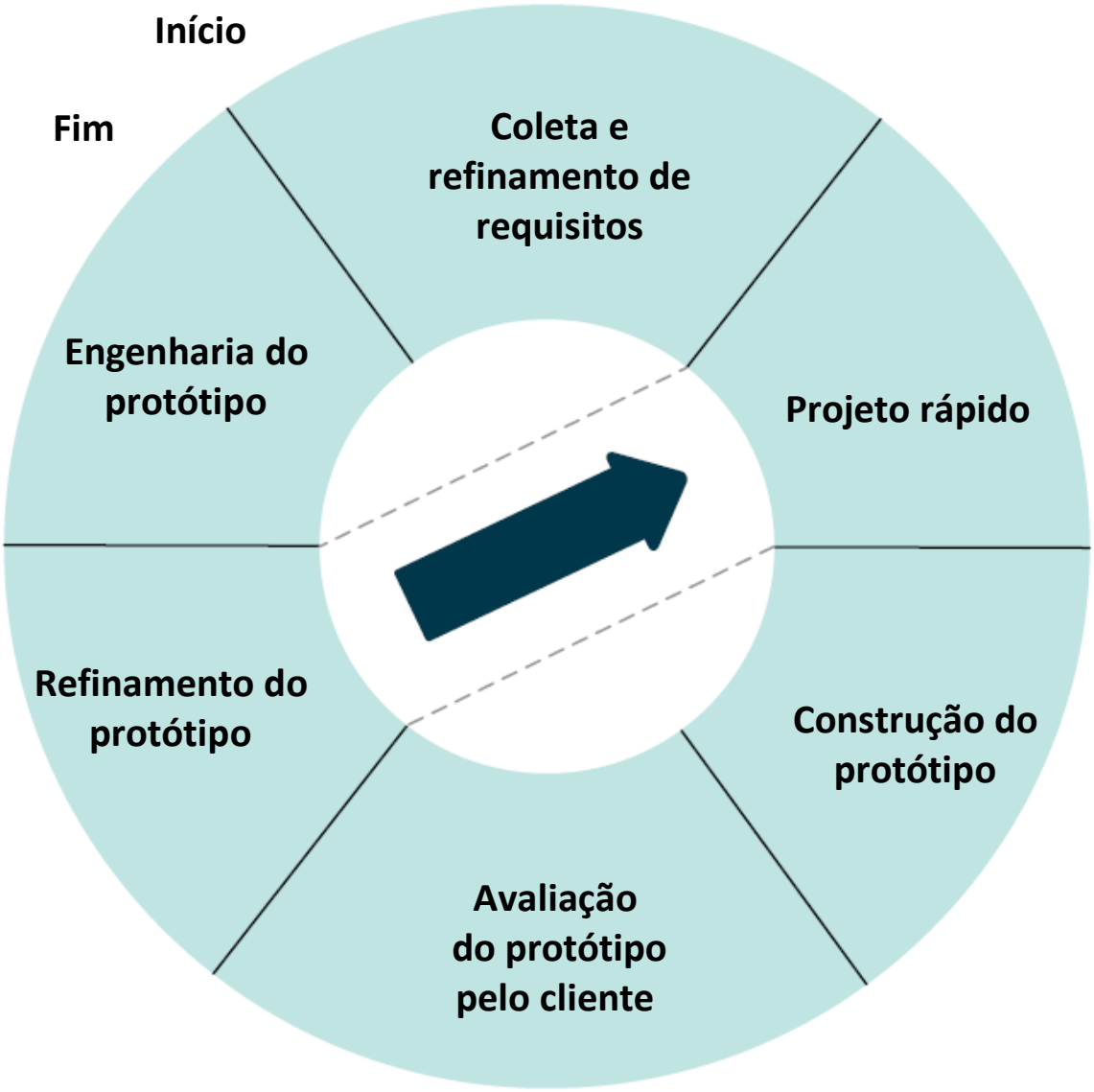
- O modelo não prevê prototipação;
- Erros graves poderão ser detectados apenas num momento tardio do desenvolvimento;
- O cliente só conhece o produto na fase final do processo;
- Projetos reais raramente seguem o fluxo sequencial que o modelo propõe.

Apesar dos problemas apontados, segundo Sommerville (2011), o modelo em cascata pode ser utilizado nos contextos em que os requisitos são facilmente estabelecidos ainda no início do desenvolvimento, o que diminui a probabilidade de ocorrer mudanças.

Como evolução do modelo em cascata, foi desenvolvido o modelo de prototipação. Nesta nova abordagem, o objetivo principal era fazer com que os requisitos fossem melhor entendidos, antes do início do desenvolvimento do produto. Desta forma, os desenvolvedores buscavam minimizar problemas relacionados à aceitação do produto final por parte dos clientes, a partir do alinhamento prévio das expectativas com relação à

interface apresentada. Na imagem abaixo, podemos ver os passos do modelo em questão.

Figura 3. Fases do modelo de prototipação



Fonte: O autor.

Neste modelo, as atividades têm início com a coleta e refinamento de requisitos. Diferente do modelo anterior, não é preciso descrever todas as funcionalidades e característica do produto na sua totalidade ainda no início, já que o modelo é evolutivo. Após a coleta inicial de requisitos, é realizado um projeto rápido, necessário para a construção do protótipo.

Com o protótipo construído, o cliente pode avaliá-lo. Neste momento, os requisitos inicialmente coletados podem ser confirmados ou alterados. Por ser uma ferramenta visual, é também muito comum que novos requisitos sejam descobertos neste momento, sendo necessário o refinamento do protótipo. O ciclo de construção e validação de protótipo acontece até que o cliente valide uma versão do produto, fazendo com que o

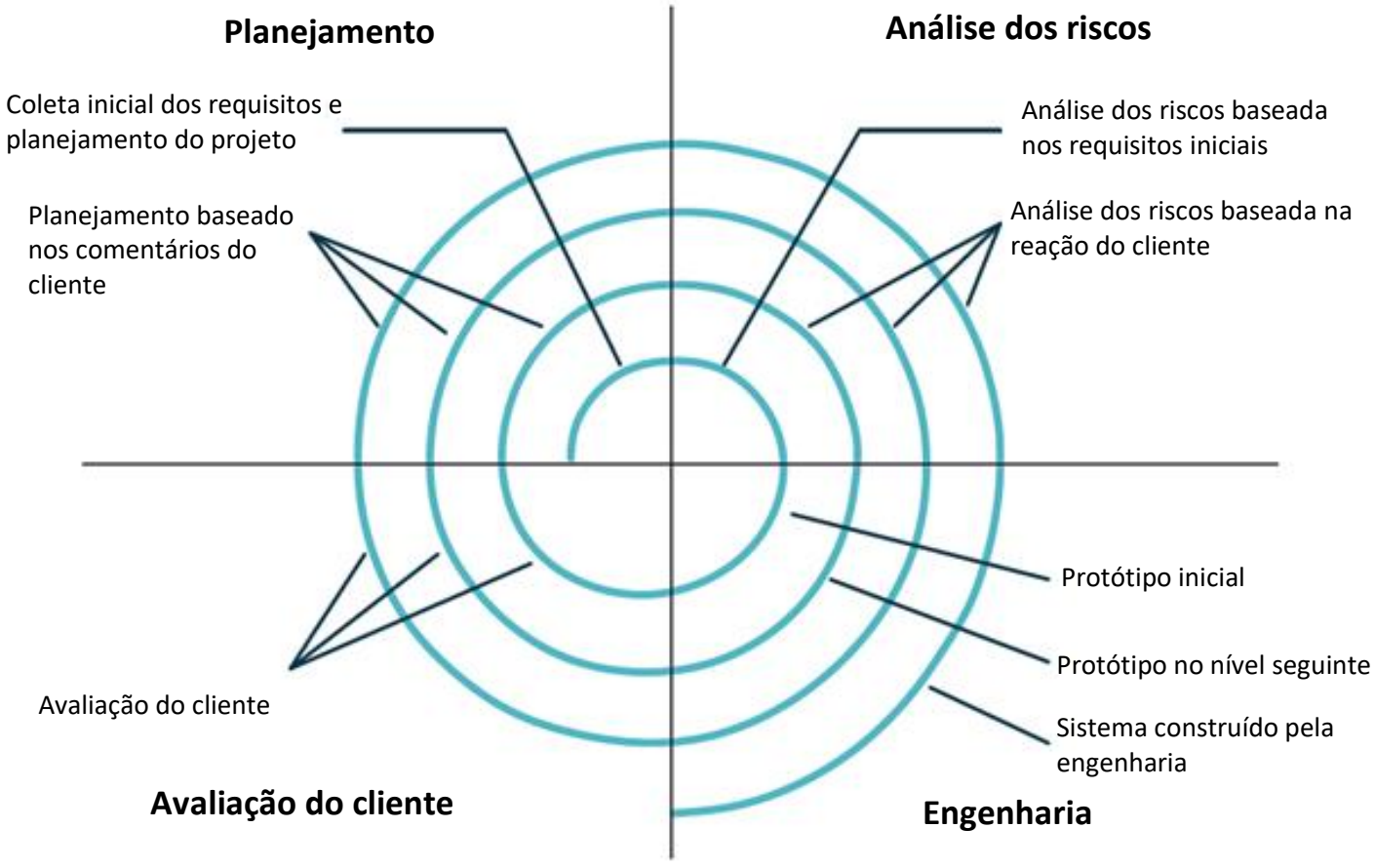
desenvolvimento possa ser iniciado. Mesmo apresentando vantagens em relação ao modelo em cascata, a prototipação não resolve todos os problemas em relação ao modelo de desenvolvimento. Além disto, apresenta novos problemas.

Por ser uma técnica com o foco na interface do produto, a prototipação pode encorajar a análise superficial. A equipe de desenvolvimento e o cliente podem se ater a parte visual do produto, podendo não dedicar tempo suficiente às regras de negócio e outros diferentes aspectos técnicos do sistema. Outro problema é que, ao participar da validação do protótipo, o cliente pode achar que a interface apresentada já é o próprio sistema e que o produto já está pronto, o que não é verdade. Neste sentido, pode se tornar difícil negociar um prazo necessário para que de fato o sistema fique pronto por completo.

Por fim, outro problema é que o desenvolvedor pode fazer concessões para que o protótipo seja apresentado de forma mais rápida, o que pode acarretar em prejuízo com relação a aspectos importantes de negócio e/ou implementação.

Na sequência, temos o modelo em espiral. Também evolutivo, o modelo em questão utiliza as etapas do modelo em cascata, organizadas em ciclos, fazendo com que possamos revisitar as atividades. Desta forma, podemos dizer que cada ciclo representa uma fase do modelo, como mostrado na figura a seguir.

Figura 4. Fases do modelo espiral



Fonte: O autor.

O modelo é apresentado em quatro fases: planejamento, análise dos riscos, engenharia e avaliação do cliente. Inicialmente, o time de desenvolvimento deve se concentrar na coleta inicial dos requisitos e no planejamento inicial do projeto. Percebam que neste momento não se tem a obrigação de entender todos os detalhes do projeto, mas sim os aspectos macro. Com a evolução do projeto, os requisitos e planejamento serão revisitados e refinados.

Na sequência, deve-se realizar a análise dos riscos. Inicialmente os riscos serão relacionados aos requisitos iniciais, porém esta definição tende a se intensificar ao logo do projeto. Novos riscos surgem ao longo do desenvolvimento. Vale salientar que o modelo espiral é o primeiro a definir uma etapa que tem como base a análise dos riscos envolvidos no projeto.

Após a análise dos riscos, temos o desenvolvimento do produto. Nesta etapa, teremos inicialmente a definição de protótipos para validação dos requisitos iniciais. Com o passar do tempo de projeto, esta etapa representará a implementação do produto. Por fim, a última etapa do modelo consiste na avaliação do cliente.

> "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."
>
> SOMMERVILLE, 2011, página 32.

Com relação aos modelos anteriores, o modelo em espiral apresenta vantagens como: a adição da análise dos risos do projeto, fazendo que com projetos complexos sejam mais seguros de serem desenvolvidos; o fato de ser incremental faz com que novas funcionalidades sejam adicionadas a cada fase; o cliente se apresenta como membro do time, estando presente constantemente na validação do produto, fazendo com que o resultado final atenda a expectativa de todos os envolvidos no projeto.

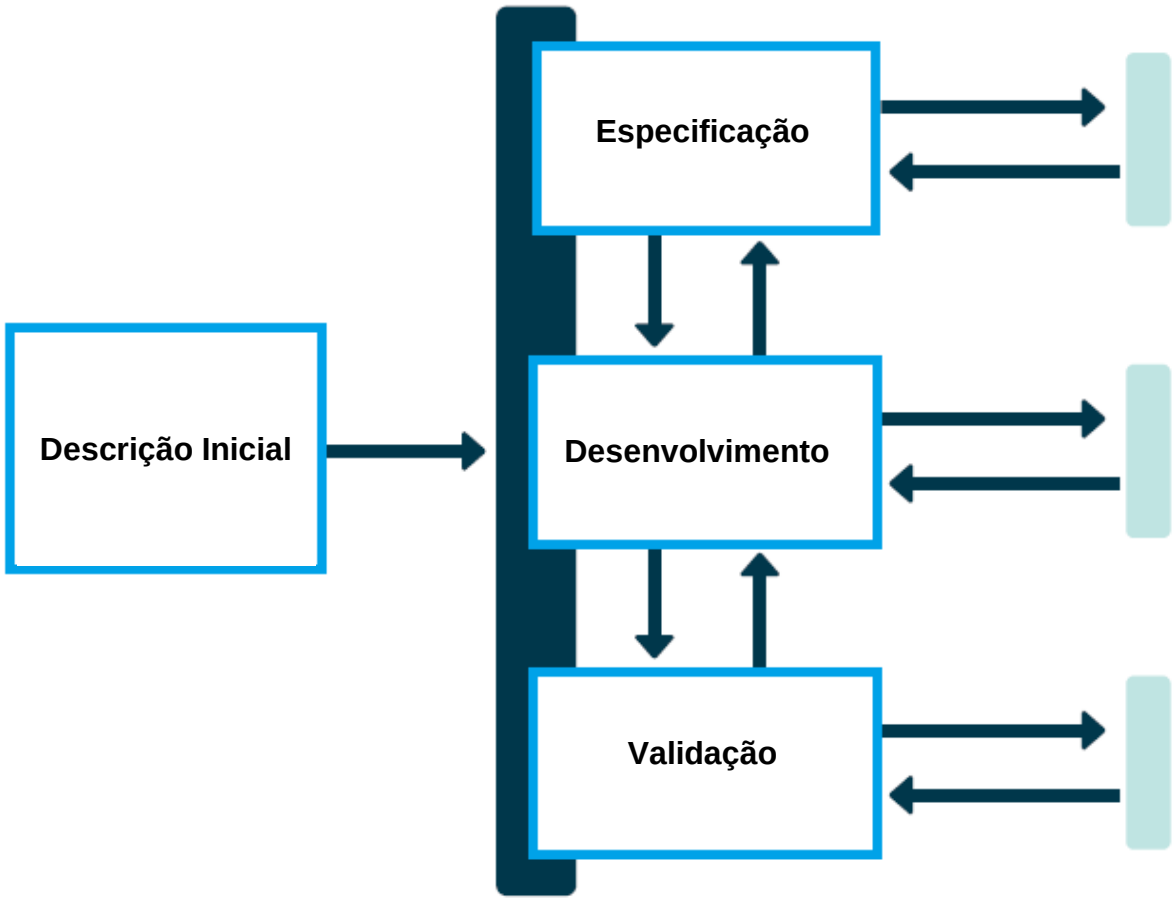
Apesar das vantagens, o modelo também apresenta problemas, como: necessidade de maturidade para definição dos riscos e dificuldade de controle com relação à abordagem evolutiva do produto. Desta forma, podemos concluir que o modelo espiral é indicado para projetos complexos, que necessitem de uma maior participação dos clientes/usuários e que possua um time de desenvolvimento maduro e experiente.

Como evolução dos modelos anteriores, surge o modelo iterativo e incremental. Este modelo se baseia no modelo espiral, porém apresenta divisões nas entregas denominadas de incrementos. Neste contexto, cada entrega representa um pedaço do produto, contendo funcionalidades. Fazendo uma comparação é como se o produto final fosse um quebra cabeça completo e cada peça simbolizasse parte da funcionalidade. Esta característica propicia que a especificação e o desenvolvimento caminhem juntos durante a construção do sistema.

> “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.”
>
> PFLEEGER, S.L., , 2004., página 44.

A imagem a seguir apresenta a ideia central do modelo em questão. O objetivo desta forma de trabalho é permitir que, a partir de uma descrição inicial, o time possa realizar refinamentos sucessivos com o objetivo de, ao final do processo, ter o produto final implementado. As atividades de especificação, desenvolvimento e validação devem ser realizadas diversas vezes, cada conjunto de atividades sendo focada em partes incrementais do produto.

Figura 6. Fases do modelo iterativo e incremental



Fonte: O autor.

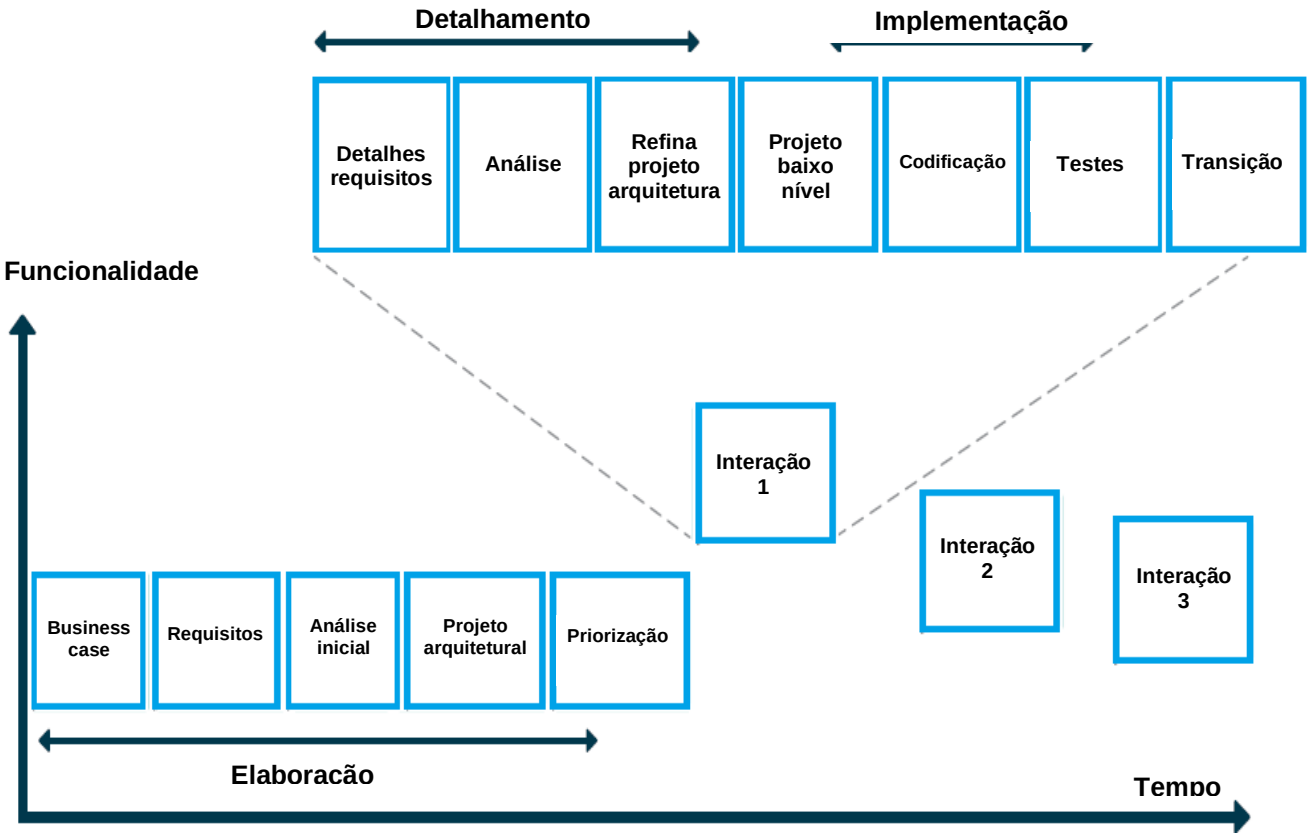
O desenvolvimento interativo entrega um sistema completo desde o começo e então muda a funcionalidade de cada subsistema a cada nova versão. (PFLEEGER,2004).

A partir da imagem acima, percebemos que não existe uma sequencialidade nas atividades propostas pelo modelo, como temos no método cascata. Nesse caso, como podemos observar na imagem 6, em cada iteração temos as atividades de refinamento de requisitos, refinamento do modelo conceitual (etapa de análise), refinamento do projeto arquitetural, projeto de baixo nível (etapa de projeto), codificação e testes (etapa de implementação), além de documentação, instalação e outras (etapa de transição).

Sempre que um novo modelo surge, ele traz vantagens em relação aos propostos anteriormente. No caso do modelo iterativo e incremental, podemos destacar a 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 seja realizada a tempo.

Além disto, a abordagem iterativa e incremental permite uma melhor resposta a mudanças, já que os requisitos são mantidos atualizados a partir da aproximação do cliente e das frequentes validações, viabilizadas pelas entregas mais curtas, o que ajuda no alinhamento das expectativas.

Figura 7. Detalhamento das iterações no modelo interativo e incremental.



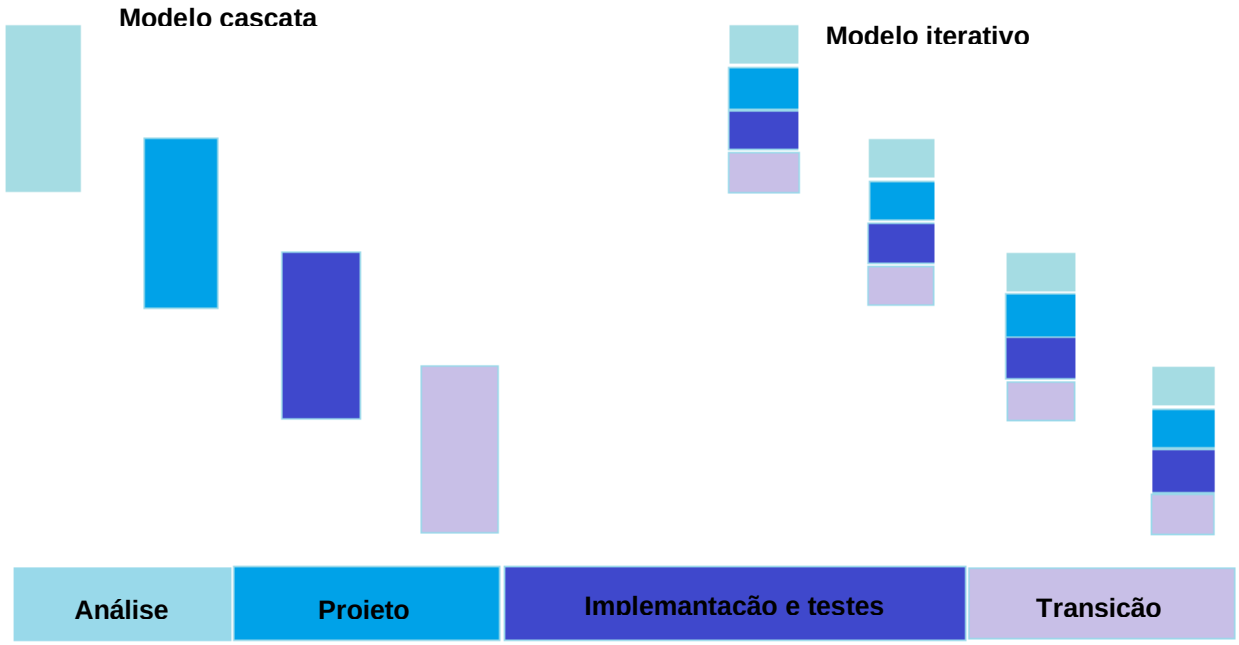
Fonte: O autor.

O fato de ser incremental também está ligado ao produto ser implantado de forma mais rápida, já que, a cada incremento, o time disponibiliza parte das funcionalidades. Essa característica acarreta em um aumento da motivação do time, decorrente da materialização do trabalho de forma mais concreta, resultando na 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 abaixo.

Figura 8. Diferenças entre os modelos iterativo, incremental e cascata.



Fonte: O autor.

Agora que vimos e entendemos os principais modelos de desenvolvimento, vamos nos concentrar nas principais metodologias tradicionais e ágeis, baseadas nos modelos estudados. Após a verificação das diferenças entre os tipos de metodologia, vamos detalhar as metodologias ágeis.

Conceitos e aplicações das principais metodologias ágeis

Na introdução, podemos conversar sobre os diferentes modelos que dão origem aos processos de desenvolvimento que utilizamos na área. Quero que vocês entendam primeiramente a diferença entre estes dois conceitos: modelo e metodologia de desenvolvimento.

Um modelo consiste em uma representação da realidade. É algo abstrato. Basicamente, podemos dizer que utilizamos modelos como fonte de inspiração: A partir dele podemos criar a realidade com os detalhes específicos que nos convém. Por sua vez, uma metodologia de desenvolvimento é formada a partir de um determinado modelo: O modelo se torna o modo de representação necessário para que a metodologia seja apresentada.

Visto isso, nesta primeira unidade iremos apresentar as principais diferenças entre as metodologias tradicionais e as metodologias ágeis. Além disto, iremos verificar os princípios ágeis que norteiam as metodologias que iremos estudar na sequência, nas demais unidades.

Diferenças entre metodologias tradicionais e ágeis

O objetivo principal deste capítulo é apresentar as diferenças entre metodologias tradicionais e ágeis. Para que seja aberta a discussão, inicialmente iremos apresentar brevemente uma das principais metodologias tradicionais bem conhecidas da área: O método RUP.

1.1 Metodologia RUP

O RUP consiste em um arcabouço de processo baseado no Modelo Iterativo e Incremental, visto na introdução desta unidade. A sua utilização teve como expectativa a resolução dos problemas encontrados no Modelo Cascata. Desta forma, o maior objetivo da metodologia é permitir que as tarefas envolvidas sejam realizadas de forma a possibilitar a visitação de ações já realizadas, propiciando a adaptação de possíveis mudanças que ocorram durante a execução do projeto.

A metodologia RUP possui as características relacionadas ao fato de ser customizável, iterativa e incremental, o que representa o oposto da sequencialidade abordada no modelo cascata. Este fato ajuda na 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.

A figura abaixo apresenta a metodologia RUP.

Como podemos perceber pela imagem, o RUP possui duas divisões claras: as fases e suas disciplinas. Essas divisões propiciam justamente a iteratividade do modelo: se prestarmos atenção, as disciplinas (que aqui representam as atividades necessárias para execução do projeto) podem ser realizadas ao longo de das diversas fases, sendo que elas possuem momentos onde serão executadas com mais intensidade.

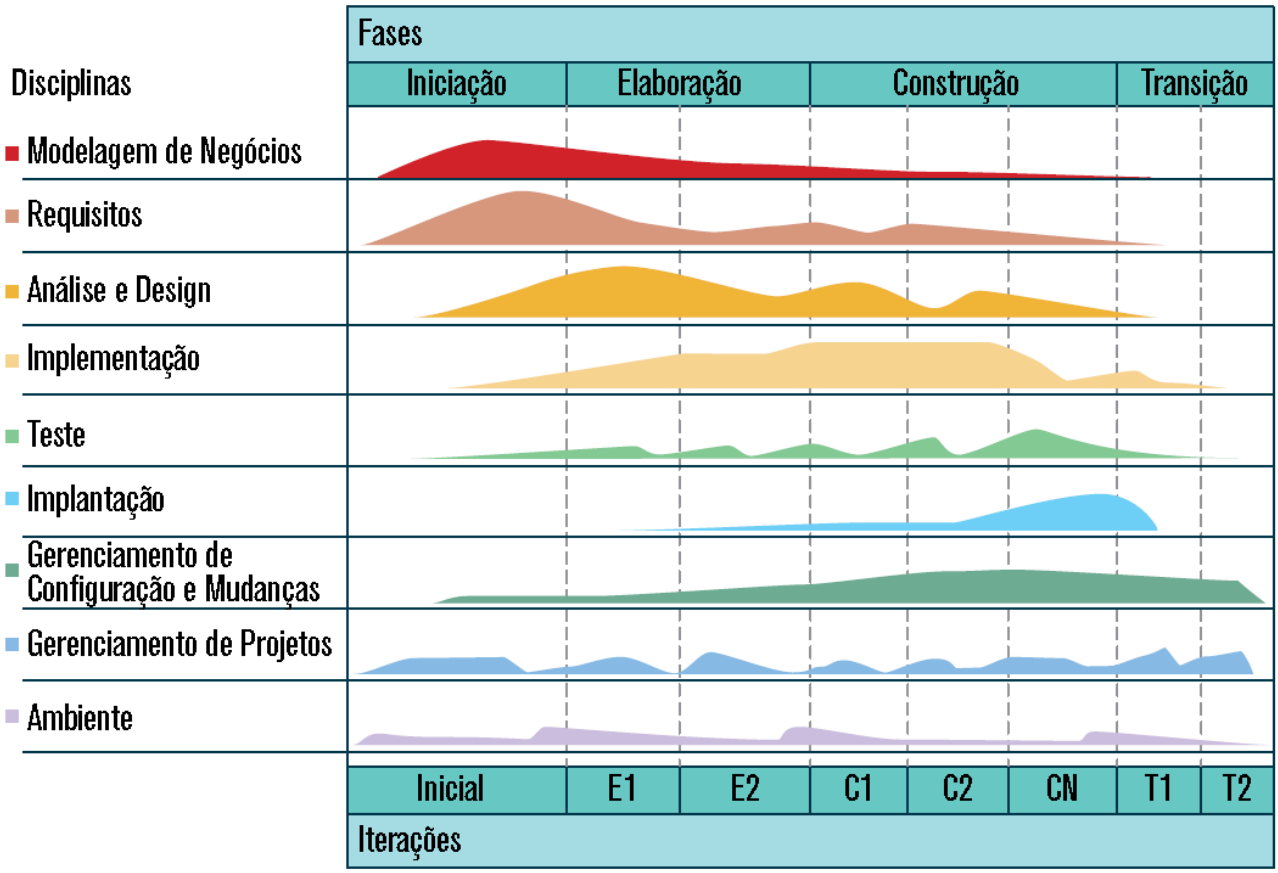
Por exemplo: No início do projeto, na fase de iniciação, teremos mais atividades relacionadas às disciplinas de modelagem de negócios e requisitos. Este fato acontece em razão de ser neste momento em que o entendimento do negócio relacionado ao software a ser construído deve ser estabelecido. É também neste momento em que deve ser realizada a análise da viabilidade do projeto. Porém, isso não implica que neste momento não possamos também antecipar tarefas de análise e design, implementação, testes, dentre outras disciplinas apontadas na imagem.

Ao caminharmos mais no tempo do projeto, na fase de elaboração, o foco muda um pouco: Ainda realizamos atividades de modelagem de negócio e requisitos, até porque sabemos que é muito difícil fechar todo o entendimento do projeto na fase inicial. Porém, o objetivo desta fase é arquitetar o sistema e pensar nos riscos associados. Desta forma,

as atividades relacionadas à análise e design, além da implementação, estarão mais presentes. É neste momento onde serão construídos os modelos UML e os protótipos.

A UML (*Unified Modeling Language*), que significa Linguagem Unificada de Modelagem é uma linguagem padrão para modelagem orientada a objetos. Esta linguagem de modelagem não proprietária de terceira geração, não é um método de desenvolvimento. Tem como papel auxiliar a visualizar o desenho e a comunicação entre objetos. Ela permite que desenvolvedores visualizem os produtos de seu trabalho em diagramas padronizados, e é muito usada para criar modelos de sistemas de software (Marina, 2020).

Figura 9. Disciplinas e Fases da metodologia RUP.



Fonte: SOMMERVILLE, I. 2011

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.

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

Tabela 1. Resumo das disciplinas do RUP

Disciplina	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.
Análise e Design	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.
Gerenciamento de Projetos	Esse workflow de apoio gerencia o desenvolvimento do sistema
Ambiente	Esse workflow está relacionado com a disponibilização de ferramentas apropriadas para a equipe de desenvolvimento de *software*.

Fonte: O autor.

> “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.”
>
> SOMMERVILLE, I. 2011, página 35.

1.1.1 Elementos do RUP

Por ser uma metodologia robusta, o RUP possui diversos elementos bem definidos para compor os processos gerados a partir desta ferramenta. O objetivo central do método é deixar claro, a todo tempo, quem faz o que ao longo da implementação do produto. Desta forma, são necessárias a definições de elementos como: papéis, atividades, artefatos, fluxos de trabalho e disciplinas.

Um papel tem a finalidade de definir o comportamento dos envolvidos no processo. Esse comportamento é determinado a partir das atividades estabelecidas para cada grupo de indivíduos. Desta forma, cada papel possui também responsabilidades ao longo do processo de desenvolvimento. São exemplos de papéis: engenheiro de processos, programador, testador, arquiteto de software, administrador de sistema, dentre outros.

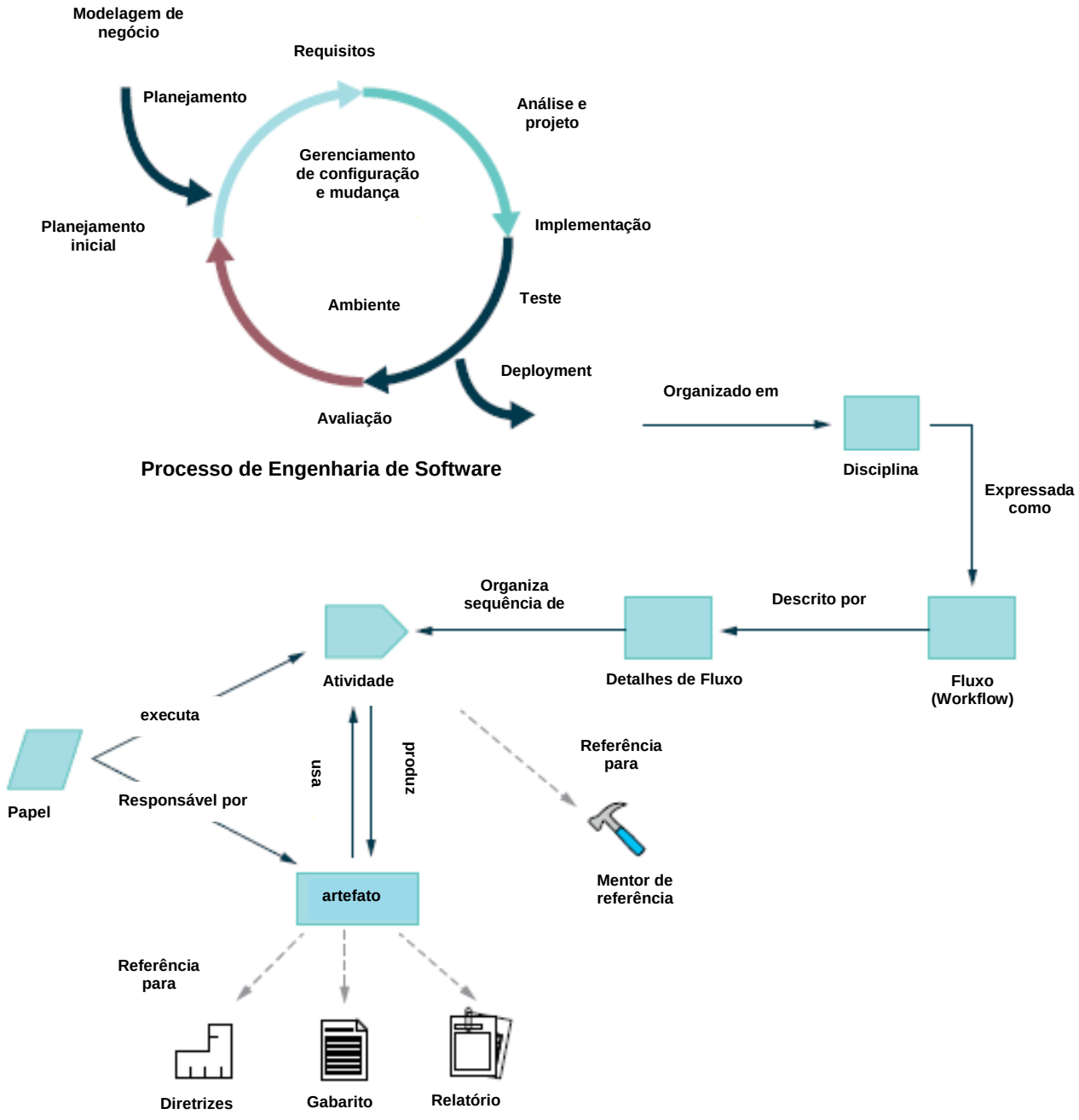
Por sua vez, as atividades representam o sequenciamento dos passos necessários para a produção do resultado esperado no processo. Esse resultado pode ser parcial ou final, normalmente sendo materializado na forma de um artefato. São exemplos de atividades no RUP: Definir abordagem de teste, estabelecer processos de controle de mudanças, planejar integração de sistemas, implementar conjunto de testes, dentre outras.

Como já foi dito, um artefato é a consolidação do resultado da execução de uma determinada atividade atribuída a um papel, podendo ser um documento, um modelo, um código-fonte, um programa-executável, dentre outros. São exemplos de artefatos no RUP: plano de gerenciamento de requisitos, visão, glossário, documento de arquitetura, dentre outros.

Para que um processo seja montado e executado, é necessário definir o sequenciamento das atividades. Esse sequenciamento é chamado de fluxo de trabalho, que representa a ordem correta de execução dos passos pré-estabelecidos.

Na imagem abaixo, temos a representação gráfica dos elementos da metodologia RUP.

Figura 10. Elementos do RUP.



Fonte: SOMMERVILLE, I. 2011

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.

Por fim, temos as disciplinas como o conjunto de atividades que possuem relação entre si. Como já discutido anteriormente, o RUP possui nove disciplinas, classificadas 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.

1.1.2 Ciclo de vida do RUP

O ciclo de vida do RUP é bem robusto. Não vamos entrar nos detalhes, mas é necessário entendermos bem para que possamos compreender as diferenças em relação às metodologias ágeis.

Como já discutido, o ciclo de vida do RUP é dividido em quatro fases e nove disciplinas. Cada fase é composta de diversos objetivos que devem ser cumpridos para se alcançar o sucesso do projeto.

Na fase de concepção (iniciação), a equipe de desenvolvimento deve definir o escopo do produto. Esta definição deve ser feita a partir da elaboração de casos de uso, que irão representar as necessidades funcionais e não funcionais do ponto de vista do cliente.

> Um caso de uso descreve uma sequência de ações “com o objetivo de demonstrar o comportamento do sistema (ou parte dele), através de interações com atores” (Melo, 2002).
>
> Cada caso de uso é o detalhamento de um ou mais requisitos funcionais do sistema e sua descrição deve ser de alto nível. Embora os clientes precisem validar o fluxo de eventos dos casos de uso, outras pessoas também farão uso dos mesmos. Dentre elas, estão gestores, arquitetos de software, analistas, especificadores, implementadores e testadores. Dessa forma, este artefato pode ir sendo refinado até atingir um nível de detalhamento que atenda às necessidades das pessoas envolvidas com a construção do produto.
>
> Um caso de uso não deve detalhar aspectos de implementação. Seu objetivo é mostrar o que o sistema faz, e não como ele faz. Portanto, o caso de uso deve descrever a interação entre o ator e o sistema sob uma ótica externa, como se o especificador assistisse à utilização do caso de uso pelo ator.
>
> Detalhes de interface não devem ser descritos no caso de uso, pois torna-o passível de alteração sempre que houver qualquer mudança em campos da interface, além de prejudicar o entendimento do caso de uso devido ao grande volume de informações que detalham a tela.

Além da definição dos casos de uso, é neste momento em que os custos e o tempo do projeto devem ser estimados. É também na fase de concepção onde os riscos do projeto são inicialmente estimados, a arquitetura inicial é proposta, o documento de visão é elaborado, casos de negócio são disponibilizados, o plano de desenvolvimento do software é implementado, o modelo de caso de uso é definido e o glossário é iniciado.

Na fase de elaboração, as pessoas envolvidas no projeto irão focar na definição de uma arquitetura executável e estável, tratar os riscos identificados do ponto de vista de projeto, selecionar componentes visando o reuso, criar planos de iterações para a fase de construção, construir protótipos, criar o documento de arquitetura de software, criar os diagramas de modelo de projeto e criar o modelo de dados.

Na fase de construção, o time deve realizar a implementação do produto, visando a minimização dos custos de desenvolvimento, a otimização dos recursos e evitando o retrabalho. O time deve: Disponibilizar versões executáveis do programa, concluir a análise, o projeto, o desenvolvimento e o teste de todas as funcionalidades e verificar e decidir se o software está pronto para implantação. Desta forma, deve-se produzir os seguintes artefatos: código executável, plano de Implantação e testes.

Por fim, na fase de transição, o time deve disponibilizar o software ao usuário final, obter feedback do usuário e treinar os usuários e a manutenção. Para que isto ocorra, devem ser produzidos os seguintes artefatos: notas de Release, artefatos de Instalação e material de treinamento.

1.2 Comparação do RUP com metodologias ágeis

Como podemos perceber, a metodologia RUP é bastante robusta, sendo necessário a criação de diversos artefatos ao logo do processo de desenvolvimento. Esta característica de metodologias tradicionais faze com que mudanças não sejam facilmente aceitas. Imaginem um contexto onde o cliente não sabe exatamente o que quer? Quaisquer mudanças que aconteçam ao logo do projeto fará com que os artefatos produzidos sejam atualizados. Na prática, raramente os times conseguem manter os artefatos atualizados, fazendo com que a documentação não ajude muito. Outro problema é que muitas vezes a equipe abre muitas concessões para que a mudança não seja implementada, o que pode resultar em um produto que não atenda às expectativas do cliente.

Desta forma, podemos dizer que, a forma como as metodologias tradicionais são desenhadas, não favorece a flexibilidade do projeto com relação às mudanças de requisitos. Por serem fortemente prescritivas, estas metodologias forçam uma previsibilidade a partir da criação de planos detalhados definidos ainda nas fases iniciais do projeto. Da construção de um cronograma detalhado e da manutenção de uma documentação extensa.

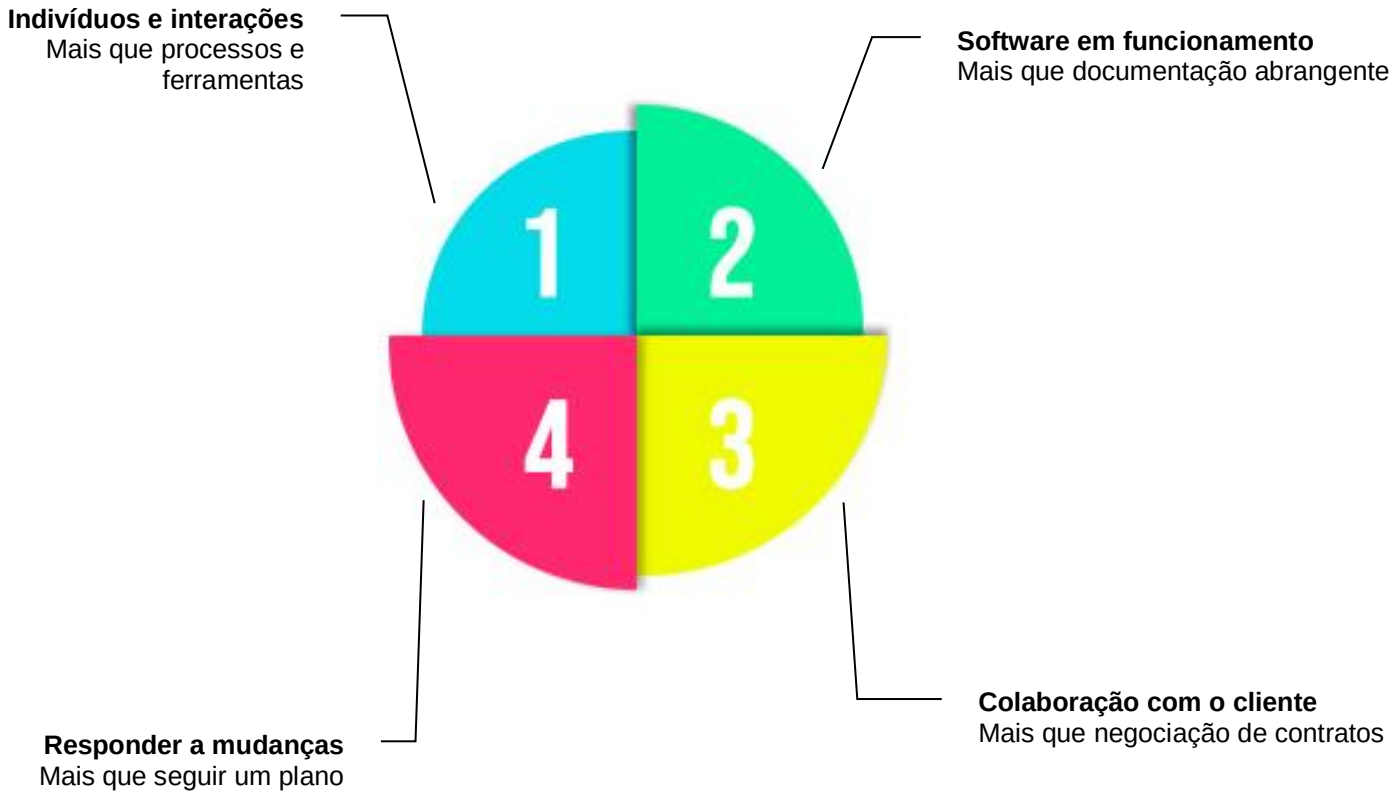
O problema destas características tradicionais é que os requisitos não são completamente compreendidos antes do início do projeto, os usuários só sabem exatamente o que querem após ver uma versão inicial do produto, os requisitos mudam frequentemente durante o processo de desenvolvimento e novas ferramentas e tecnologias tornam as estratégias de desenvolvimento imprevisíveis.

Dessa forma, se para o cliente o mais importante é ter certeza do escopo a ser implementado antes do início do desenvolvimento do software, ele deverá optar pela utilização de uma metodologia tradicional, como o RUP, por exemplo. Se o cliente não tem certeza sobre o escopo ou ainda está em definição, ele deverá optar por uma estratégia que utiliza métodos ágeis.

No universo ágil, as características são bem diferentes das apresentadas até aqui. No desenvolvimento ágil de software, valoriza-se os 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. Essa definição é o que forma o manifesto ágil.

O Manifesto Ágil é uma declaração de valores e princípios essenciais para o desenvolvimento de software. Ele foi criado em fevereiro de 2001, onde se reuniram 17 profissionais que já praticavam métodos ágeis como XP, DSDM, SCRUM, FDD e etc. Durante a reunião, foram observados os pontos em comum de projetos que tiveram sucesso em suas metodologias e com base nesses pontos foi criado o Manifesto para Desenvolvimento Ágil de Software, no qual chamamos de Manifesto Ágil. O Manifesto Ágil aborda valores que todos os profissionais ali reunidos acordaram em seguir e disseminar. (Roberto, 2020).

Figura 11. Resumo do Manifesto Ágil

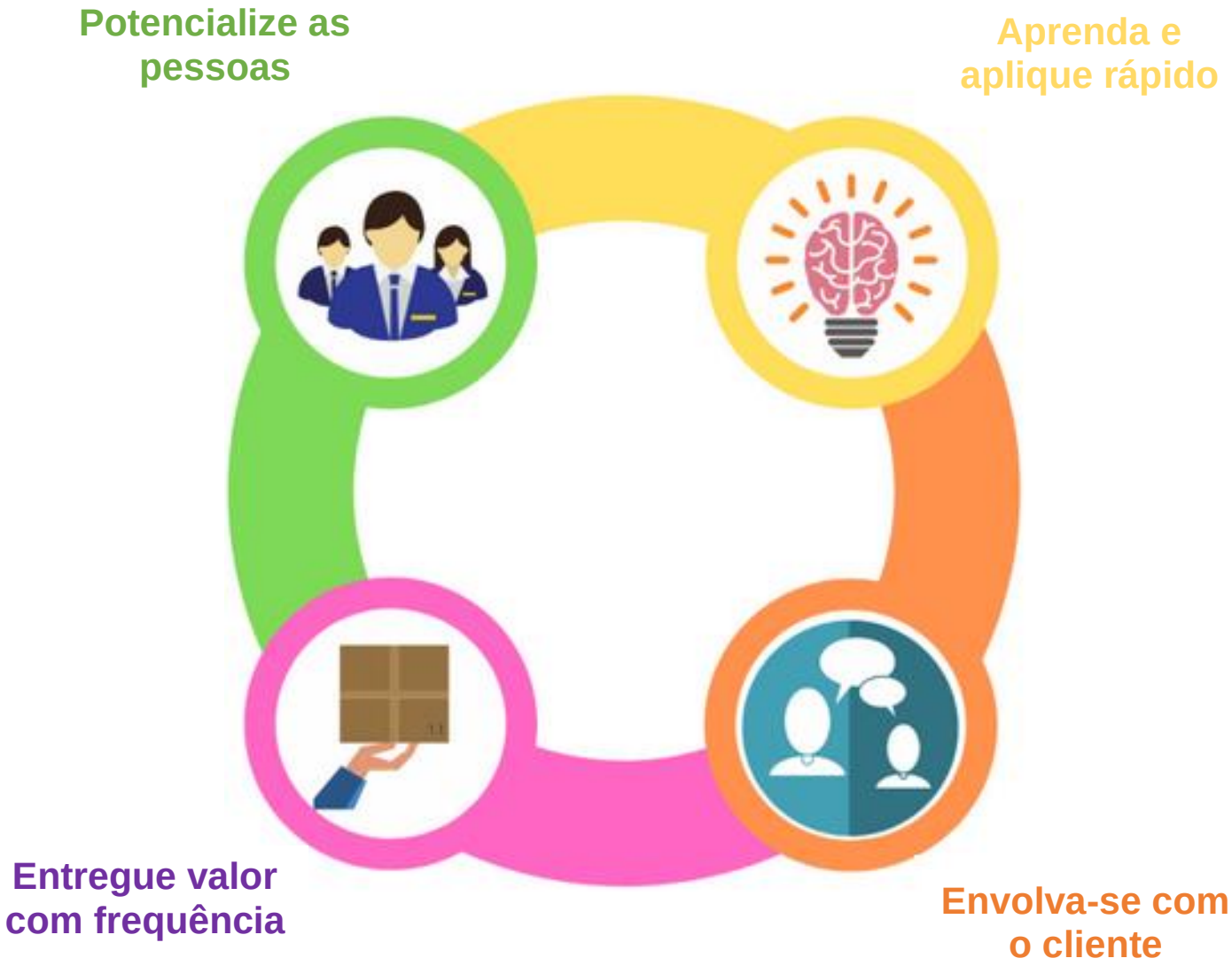


Fonte: O autor.

O objetivo do manifesto ágil é colocar a atenção nas pessoas envolvidas no desenvolvimento e não no processo em si. Isso não significa que um conjunto de atividades estruturadas não seja importante (assim como a documentação associada), mas que a geração de valor será maior quando estamos trabalho em meio a um contexto colaborativo. O resultado será sempre a entrega de um produto que esteja mais adaptado ao que o cliente espera.

A imagem abaixo mostra os princípios decorrentes do manifesto ágil.

Figura 12. Princípios do Manifesto Ágil



Fonte: QuestionPro, 2020.

> “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.”
>
> AGILE MANIFESTO, 2001.

Agora que entendemos as diferenças entre metodologias tradicionais e ágeis, vamos detalhar os princípios ágeis. Na sequência, iremos nos aprofundar nas principais metodologias ágeis de desenvolvimento.

Princípios Ágeis

A partir do manifesto ágil, podemos discutir sobre alguns princípios que norteiam este tipo e desenvolvimento. A imagem abaixo resume os doze princípios a serem discutidos.

Figura 13. Resumo do Manifesto Ágil

Fonte: FabioBmed, 2020.

O primeiro princípio apresenta como prioridade a satisfação do cliente por meio da entrega cedo e frequente de software com valor. O objetivo deste princípio é garantir que o produto entregue seja útil ao cliente, atendendo a sua expectativa. A ideia é ter entregas curtas e frequentes para que o cliente participe do processo de implementação, inclusive propondo novos requisitos e alterando requisitos existentes.

O segundo princípio faz alusão ao aceite das mudanças. Mudança é normal! Elas são bem-vindas, mesmo em fases tardias do desenvolvimento. Ao logo do processo de desenvolvimento de qualquer produto vamos ter alterações nos requisitos. O time tem que entender que é melhor aceitar as mudanças e entregar um produto útil do que não ser

flexível e acabar tendo como resultado um produto que já é entregue obsoleto ou sem atender as expectativas do cliente.

O terceiro princípio menciona o poder das entregas frequentes. Como já dito, entregas curtas faz com que o retorno do cliente seja sempre atualizado. A ideia é entregar software em funcionamento com frequência, desde a cada duas semanas até a cada dois meses, com uma preferência por prazos mais curtos.

O quarto princípio traz a ideia de trabalho colaborativo. As pessoas do negócio e os desenvolvedores devem trabalhar em conjunto diariamente ao longo do projeto. Isso implica em facilitar o trabalho por meio da comunicação, que deve ser cara a cara.

O quinto princípio comenta sobre a confiança e apoio dentro do time. Neste sentido, os projetos devem ser construídos em torno de indivíduos motivados, dentro de ambientes que ofereçam o suporte necessário. Além disto, o time deve ter a confiança necessária parar a realização do trabalho.

O sexto princípio foca na comunicação cara a cara, sendo mostrado como o método mais eficiente e efetivo de se transmitir informação. Porém, hoje em dia temos tecnologias que amenizam o problema do trabalho a distância, como as videoconferências. Porém, a conversa face a face continua sendo o melhor caminho para alinhar o entendimento entre membros do projeto.

O sétimo princípio menciona a importância de se ter software funcionado como principal medida de progresso. Quando mais entregas funcionais forem realizadas, mais perto a equipe se encontra de alcançar o objetivo de finalizar o projeto com sucesso. É mais fácil garantir o sucesso do projeto fazendo entregas de partes funcionais do produto do que simplesmente deixar para entregar tudo no final.

O oitavo princípio traz a ideia de desenvolvimento sustentável. A ideia é manter um ritmo constante de desenvolvimento, evitando a utilização de horas extras. De acordo com a filosofia ágil, um ritmo muito acelerado de trabalho faz com que não haja qualidade na entrega, além de aumentar a chance de a equipe ter retrabalho, fazendo com que as horas extras acarretem uma produtividade ainda menor.

O nono princípio chama atenção para a importância da atenção contínua à excelência técnica e a um bom projeto. Estas características aumentam a agilidade.

O décimo princípio comenta sobre a simplicidade. Aqui temos uma grande diferença em relação às metodologias tradicionais, onde diversos artefatos eram produzidos, muitas vezes, sem trazer benefícios para o projeto. No ágil, o lema é maximizar a quantidade de trabalho não feito, ou seja, não é que não se deve produzir artefatos, mas só será realizada a documentação que for necessária.

O décimo primeiro princípio fala sobre a forma do time se organizar. Neste sentido, podemos afirmar que as melhores arquiteturas, requisitos e projetos emergem de equipes que se auto organizam. Tudo o que é necessário para o desenvolvimento do projeto deve ser realizado pelos próprios membros do time.

Por fim, o décimo segundo princípio apresenta a necessidade constante dos ajustes no processo de desenvolvimento e no produto. Desta forma, em intervalos de tempo regulares, a equipe deve refletir sobre como se tornar mais efetiva e então refinar e ajustar seu comportamento de acordo com as novas ideias.

Dentre todos os princípios apresentados, podemos destacar que muitos deles nos levam a dois importantes conceitos dentro do ágil: entrega de valor e comunicação. Estes dois termos resumem bem o foco das metodologias ágeis, onde o objetivo é, por meio colaborativo, realizar a entrega de um produto que esteja alinhado com a expectativa do cliente. Na próxima unidade, iremos começar a conhecer algumas das principais metodologias ágeis, como Lean e XP.

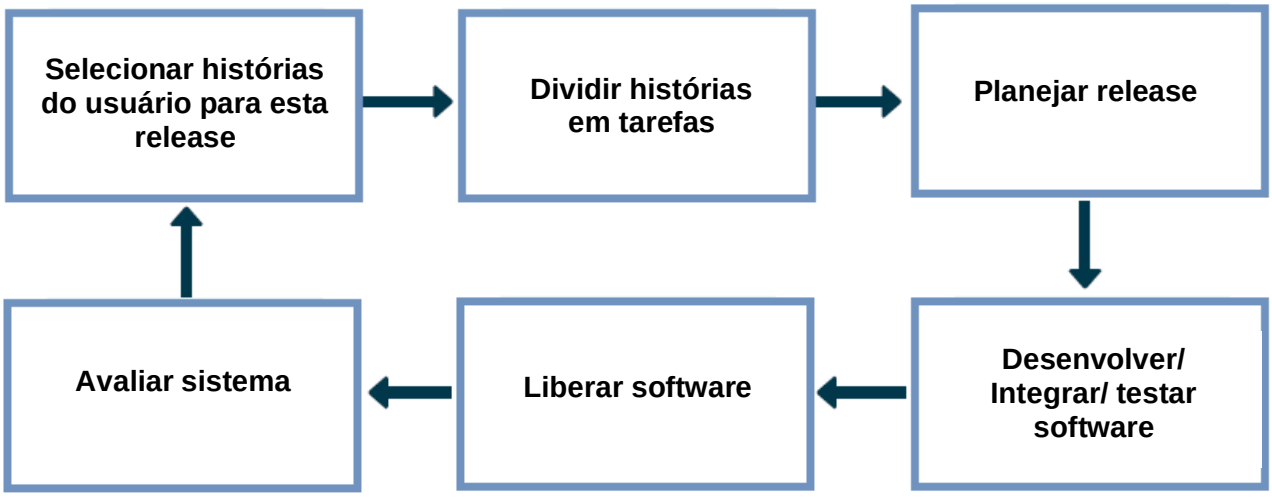
Metodologia XP (Extreme Programming)

O método XP é uma metodologia que também tem como base os princípios ágeis de desenvolvimento, sendo formada a partir de princípios que norteiam o desenvolvimento de sistemas em contextos caóticos, onde a mudança de requisitos acontece com frequência e/ou a tecnologia utilizada é desconhecida. Isso acontece pelo fato do XP ser uma forma de se trabalhar flexível e adaptável, além de possibilitar entregar rápidas a partir da utilização de ciclos curtos de desenvolvimento.

> “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.”
>
> SOMMERVILLE, I. 2011, página 44.

A imagem abaixo apresenta o ciclo de release da metodologia em questão.

Figura 14. Ciclo de release do XP



Fonte: SOMMERVILLE, I., 2011

Inicialmente, o time deve selecionar as histórias de usuário que serão tratadas na entrega.

Histórias de usuário são escritas para facilitar a liberação e planejamento da iteração, e para servir como espaços reservados para conversas detalhadas sobre as necessidades dos usuários. Enquanto os requisitos sugerem o que deve ser feito, as histórias de usuário focam nos objetivos, e isso torna a visão do produto completamente diferente. Outra diferença é que as histórias não são detalhadas no início do projeto, mas são elaboradas somente na hora em que serão necessárias, evitando assim atrasos no desenvolvimento. Elas precisam de pouca ou nenhuma manutenção e pode ser descartado com segurança depois.

Na sequência, o time deve quebrar as histórias de usuário em tarefas. Uma das técnicas que pode ser utilizada para isto é a técnica SMART. O SMART é frequentemente usado para guiar na construção de objetivos alcançáveis, mas é totalmente adaptável para o contexto das tarefas. Estes critérios são atribuídos ao conceito de gestão de objetivos de Peter Druker e vem sendo usado desde a década de 80.

- Specific (Específico): Tarefas precisam ser específicas para que todos do time entendam como elas se conectam e como juntas elas contribuem para atingir os critérios de aceitação da história do usuário.

- Measurable (Mensurável): uma tarefa é mensurável quando pode ser marcada como “concluída”, seguindo os critérios técnicos do time. Por exemplo, com os testes escritos e o código refatorado.

- Achievable (Realizável): as tarefas devem ser alcançáveis pelos seus responsáveis. Aqui entra a capacidade do time de identificar pontos fortes, fracos e oportunidades de melhoria dos membros. Neste critério, por exemplo, o time pode optar pela programação em pares, para desenvolver habilidades e alcançar seu objetivo.

- Relevant (Relevante): todas as tarefas devem ser relevantes. As histórias são quebradas em tarefas para auxiliar o desenvolvimento, mas o dono do produto ainda espera que todas elas sejam explicáveis e justificáveis.

- Time-Boxed (Duração-fixa): todas as tarefas precisam ter um tempo definido para serem concluídas. Não é necessário fazer uma estimativa formal em horas ou dias, mas deve haver uma expectativa de conclusão ou de quando será necessário pedir ajuda ao time. Quando uma tarefa se torna maior do que esperado, o time precisa saber o momento certo para tomar uma ação para que seja concluída.

Depois das tarefas estabelecidas, o próximo passo é o planejamento da entrega (release). Na sequência, as tarefas são estimadas e distribuídas entre as duplas da equipe para que

sejam desenvolvidas, integradas e testadas. Por fim, uma nova versão do sistema é liberada e avaliada pelos usuários.

Existem muitas diferenças entre os casos de uso, utilizados em metodologias tradicionais (como o RUP) e as histórias de usuário, utilizadas nas metodologias ágeis. Uma das principais diferenças das histórias de usuário está na comunicação verbal. A linguagem escrita é muitas vezes imprecisa, e não há nenhuma garantia de que um cliente e/ou desenvolvedor irá interpretar uma declaração da mesma forma. Neste sentido, a história enfatizada a necessidade de conversa com o cliente em contraste com as palavras escritas.

Outras diferenças entre as histórias de usuário e casos de uso é o seu alcance. Ambos são dimensionados para agregar valor ao negócio, mas as histórias são mantidas num tamanho menor, porque são definidas restrições ao seu tamanho para que possam caber em uma sprint. Um caso de uso quase sempre abrange um escopo muito maior do que uma história.

As histórias de usuário e casos de uso também diferem no nível de completude. Como as histórias de usuário são pequenas e simples e não capturam muitos detalhes, requer muito pouca manutenção, não se tornando um documento obsoleto rapidamente ou que exija muito esforço para mantê-lo atualizado.

Os casos de uso e as histórias de usuários são escritos para diferentes fins. Os casos de uso são escritos em formato aceitável para os clientes e desenvolvedores, para que cada um possa ler e concordar com o caso de uso. O propósito do caso de uso é documentar um acordo entre o cliente e a equipe de desenvolvimento. Histórias de usuário, por outro lado, são escritas para facilitar a liberação e planejamento da iteração, e para servir como espaços reservados para conversas detalhadas sobre as necessidades dos usuários.

As histórias de usuário incentivam a equipe a adiar detalhes de coleta. Uma história inicial pode ser escrita e, em seguida, substituída com histórias mais detalhadas quando se torna importante conhecer os detalhes. Isto torna possível transformar um grande problema em pequenas partes. Desta forma, as histórias possibilitam o desenvolvimento iterativo, permitindo que a equipe do projeto faça pequenas entregas evolutivas enquanto interage e colabora com o cliente.

Já os casos de uso contrapõem as falhas das histórias de usuário sendo mais indicados para projetos grandes e complexos, onde a interação com o cliente (usuário) não possa ser tão frequente.

Valores do XP (*Extreme Programming*)

A metodologia XP foca em cinco valores que devem guiar o desenvolvimento de sistemas. São eles: comunicação, feedback, coragem, simplicidade e respeito.

Com relação à comunicação, é importante ter em mente que a interação entre as pessoas envolvidas no projeto é essencial. Podemos afirmar que uma boa comunicação faz com que menos documentação seja necessária, o que evita desperdício de recursos. Porém, para que haja ganhos maiores, esta comunicação deve ser cara a cara, já que ferramentas de comunicação podem causar problemas de entendimento entre os indivíduos.

Já o feedback é essencial para que o resultado final esperado seja alcançado. Um retorno rápido ajuda a evitar desperdícios ao detectar problemas em fases iniciais do processo. Neste sentido, é de suma importância que o cliente esteja envolvido durante o projeto.

Ainda no contexto de mudanças, o valor da coragem é explicitado no momento em que o time deve assumir os riscos necessários durante o desenvolvimento. A equipe não pode ter medo de aceitar as mudanças ao longo do processo de construção do produto.

O quarto valor pregado na metodologia faz alusão à simplicidade. Manter um código simples faz com que o produto seja mais fácil de ser mantido e, consequentemente, mais fácil de se adaptar às mudanças propostas.

Por fim, o valor do respeito representa a forma ideal de colaboração entre os membros do time de desenvolvimento. Cada indivíduo deve ser respeitado e ter sua opinião levada em consideração ao longo do projeto.

Práticas do XP (*Extreme Programming*)

O Extreme Programming aborda diversas práticas que refletem os princípios da metodologia ágil. A tabela abaixo resume todas as práticas preconizadas pela metodologia em questão.

Tabela 3. Resumo de todas as práticas do XP.

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.
Pequenas 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.
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.

Fonte: SOMMERVILLE, I., 2011

Papéis do XP (*Extreme Programming*)

A metodologia XP estabelece diversos diferentes papéis necessários para a condução das atividades relacionadas aos projetos. A tabela abaixo resume os papéis descritos no método em questão.

Tabela 4. Resumo dos papéis do XP.

Papel	Descrição
Programador	**Responsável por produzir o código executável.**
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.**

Fonte: SOMMERVILLE, I., 2011

Princípios do XP (*Extreme Programming*)

Além dos valores já discutidos anteriormente, o Extreme Programming estabelece alguns princípios que devem ser seguidos para que se possa garantir um bom projeto de software. São eles:

Tabela 5. Resumo dos princípios do XP.

Princípio	Descrição
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.**

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.**

Fonte: SOMMERVILLE, I., 2011

Ciclo de vida do XP (*Extreme Programming*)

A metodologia XP possui ciclos de desenvolvimento curtos, divididos em seis fases: exploração, planejamento, iterações até versão, produção manutenção e morte.

Na fase de exploração o objetivo principal é a verificação da viabilidade do projeto. Para que esta atividade ocorra, é necessário realizar uma análise com base nos requisitos iniciais. Durante a análise, todos os envolvidos no projeto devem participar para que as funcionalidades e características do produto sejam detalhadas.

Com as funcionalidades detalhadas e o projeto sendo considerado viável, o próximo passo é a definição e priorização das histórias do usuário, que acontece na fase de planejamento. Também é importante que as histórias descritas sejam estimadas de acordo com a sua complexidade. Com a quantidade de pontos de histórias definidos, o time pode definir o tempo necessário para cada iteração e release.

De acordo com a metodologia, cada iteração pode ser realizada de uma a três semanas, enquanto a release deve ter de dois a quatro meses.

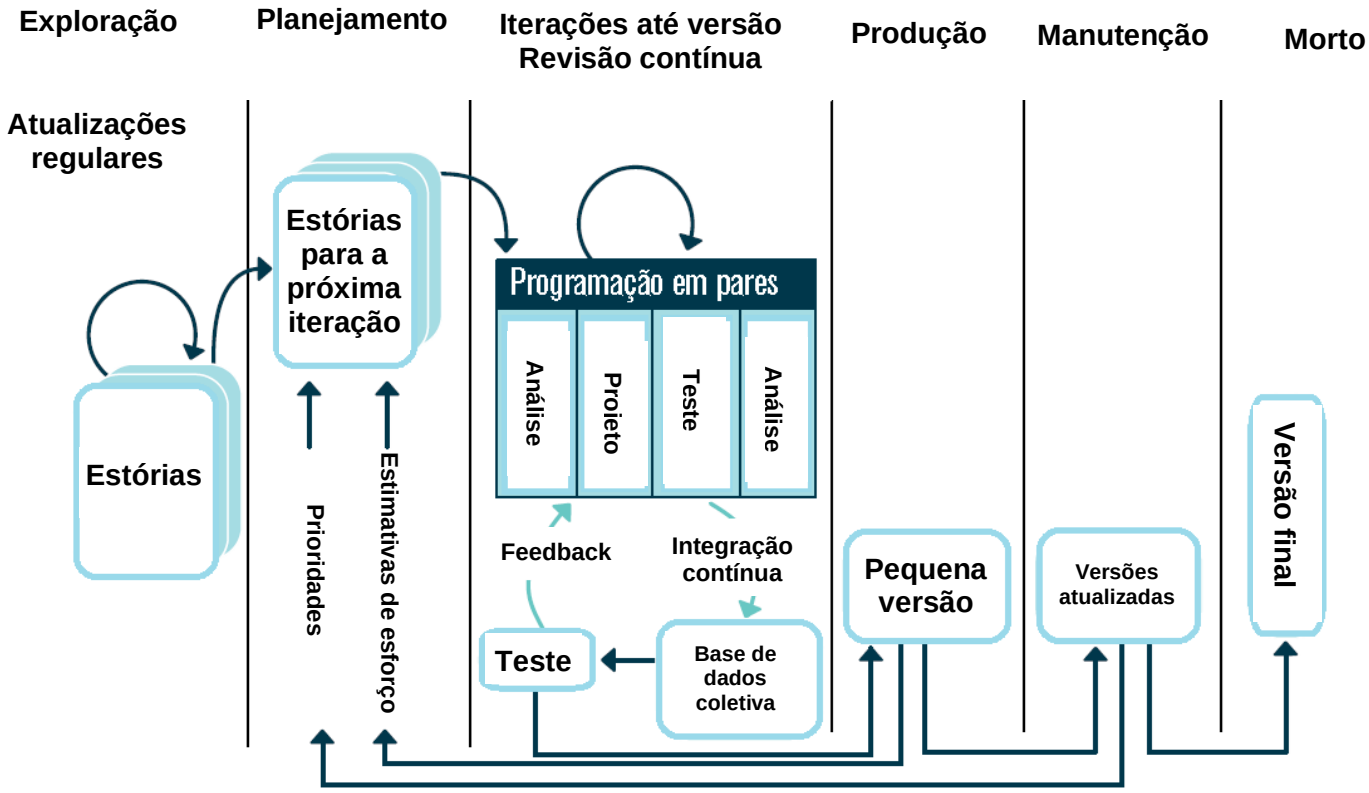
Na fase de Iteração temos as atividades da engenharia de software propriamente ditas. É neste momento em que a equipe realiza, de forma contínua e iterativa, a análise, projeto, implementação e teste do produto.

Na fase de produção, temos a integração do produto até então construído e a sua disponibilização em um ambiente de homologação. O objetivo é realizar testes em um ambiente que seja similar ao de produção. Neste momento são realizados testes como o de desempenho e aceitação.

Na fase de manutenção o foco é a adequação do produto com base em mudanças necessárias que fazem com que o produto continue sendo útil ao cliente. Nesta faze, novas funcionalidades podem ser adicionadas ao produto, como também funcionalidades existentes podem ser excluídas. Além disto, alterações podem ser realizadas com o objetivo de adequação funcional e também correção de erros.

Por fim, na fase de morte temos o encerramento do projeto. A imagem abaixo resume todo o ciclo discutido até aqui.

Figura 15. Ciclo de vida do XP



Fonte: SOMMERVILLE, I., 2011

Com o ciclo de vida do XP, finalizamos a unidade II. Na próxima unidade, iremos nos profundar em mais uma metodologia baseada nos princípios ágeis: A metodologia SCRUM.

Referências Bibliográficas

1. AGILE MANIFESTO, Manifesto for Agile Software Development, 2001. Disponível em: http://agilemanifesto.org. Acesso em: 30/01/2020

2. AMBLER, Scott e LINES,Marks. TÍTULO: Disciplined Agile Delivery - A Practitioner's Guide to Agile Software Delivery in the Enterprise

3. BRASILEIRO, Roberto. Disponível em: http://www.metodoagil.com/manifesto-agil/. Acessado em: 30/01/2020.

4. COHN, M. Agile Estimating And Planning, editora Prentice Hall 1ª Edição 304p. 2006.

5. DANDARO, Fernando; TONANI, Fabiano Rodrigo; CARVALHO, Daltro Oliveira de. Gestão de projetos como estratégia organizacional. Revista Fatec. n.19. v.06. Sertãozinho-SP, 2014. Disponível em http://www.fatecgarca.edu.br/revista/Volume6/artigos_v6/artigo19.pdf. Acesso em: 20 de jul. de 2019.

6. FABIOBMED. Disponível em: https://fabiobmed.com.br/2017/04/15/os-doze-principios-ageis/. Acessado em 30/01/2020.

7. 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.

8. KNOWLEDGE21. Disponível em: https://www.knowledge21.com.br/sobreagilidade/scrum/como-e-o-scrum/entregas/. Acessado em: 30/01/2020

9. MAIA, Jonathan. Disponível em: https://www.eunati.com.br/2017/09/scrum-planilha-e-graficos-para-acompanhar-a-evolucao-das-sprints.html. Acessado em 30/01/2020.

10. MARTINEZ, Marina. Disponível em: https://www.infoescola.com/engenharia-de-software/uml/. Acessado em: 30/01/2020.

11. MELO, Ana Cristina. Desenvolvendo Casos de Uso com UML: Do conceitual à implementação. ed. Rio de Janeiro: Brasport, 2002.

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

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

14. QUESTIONPRO. Disponível em: https://www.questionpro.com/blog/pt-br/manifesto-agil/. Acessado em: 30/01/2020.

15. QUINQUIOLO, José Manoel. Avaliação da eficácia de um sistema de gerenciamento para melhorias implantado na área de carroceria de uma linha de produção automotiva. Taubaté⁄SP: Universidade de Taubaté, 2002.

16. RAMOS, André Luís Belmiro Moreira. Metodologias de desenvolvimento de sistemas. Rio de Janeiro: SESES, 2017.

17. RIEPER, Marcos. Disponível em: https://www.guiadoexcel.com.br/grafico-burndown-scrum-excel/. Acessado em: 30/01/2020.

18. SABBAGH, R. Scrum – Gestão ágil para projetos de sucesso. Casa do código, 2013.

19. SCHWABER, K. Agile Project Management with SCRUM. Microsoft Press, 2004.

20. SCRUMDESK. Disponível em: https://www.scrumdesk.com/agile-estimation-principles/planning-poker-cards/. Acessado em: 30/01/2020

21. 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.

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

23. ZILBER, Moises Ari; FISCHMANN, Adalberto Américo; PIKIENY, Eugen Erich. Alternativas de crescimento: a alternativa de fusões e aquisições. Revista de Administração Mackenzie (Mackenzie Management Review), v. 3, n. 2, 2008.