EAD UNISANTA

ENGENHARIA DE SOFTWARE

Me. Joseffe Barroso de Oliveira

GUIA DA
DISCIPLINA

1. OBJETIVO E CONCEITO DA ENGENHARIA DE SOFTWARE

Objetivo

O objetivo deste capítulo é apresentar os objetivos e conceitos elementares da engenharia de software.

Introdução

No mundo moderno, tudo é software. Hoje em dia, por exemplo, empresas de qualquer tamanho dependem dos mais diversos sistemas de informação para automatizar seus processos. Governos também interagem com os cidadãos por meio de sistemas computacionais, por exemplo, para coletar impostos ou realizar eleições. Empresas vendem, por meio de sistemas de comércio eletrônico, uma gama imensa de produtos, diretamente para os consumidores. Software está também embarcado em diferentes dispositivos e produtos de engenharia, incluindo automóveis, aviões, satélites, robôs etc. Por fim, o software está contribuindo para renovar indústrias e serviços tradicionais, como telecomunicações, transporte em grandes centros urbanos, hospedagem, lazer e publicidade.

1.1. Objetivo e conceito

O Instituto de engenheiros eletricistas e eletrônicos (IEEE) elaborou a seguinte definição para engenharia de software: Engenharia de software: A aplicação de uma abordagem sistemática, disciplinada e quantificável no desenvolvimento, na operação e na manutenção de software; isto é, a aplicação de engenharia ao software. Entretanto, uma abordagem “sistemática, disciplinada e quantificável" aplicada por uma equipe de desenvolvimento de software pode ser pesada para outra. Precisamos de disciplina, mas também precisamos de adaptabilidade e agilidade. A engenharia de software é uma tecnologia em camadas. Como observamos na figura a seguir, qualquer abordagem de engenharia (inclusive engenharia de software) deve estar fundamentada em um comprometimento organizacional com a qualidade. A gestão da qualidade total, ou Seis Sigma, e filosofias similares promovem uma cultura de aperfeiçoamento contínuo de processos. É essa cultura que, no final das contas, leva a abordagens cada vez mais eficazes na engenharia de software. A pedra fundamental que sustenta a engenharia de software é o foco na qualidade.

A base da engenharia de software é a camada de processos. O processo de engenharia de software é a liga que mantém as camadas de tecnologia coesas e possibilita o desenvolvimento de software de forma racional e dentro do prazo. O processo define uma metodologia que deve ser estabelecida para a entrega efetiva de tecnologia de engenharia de software. O processo de software constitui a base para o controle do gerenciamento de projetos de software e estabelece o contexto no qual são aplicados métodos técnicos, são produzidos artefatos (modelos, documentos, dados, relatórios, formulários etc.), são estabelecidos marcos, a qualidade é garantida e as mudanças são geridas de forma apropriada. Os métodos da engenharia de software fornecem as informações técnicas para desenvolver o software. Os métodos envolvem uma ampla variedade de tarefas, que incluem comunicação, análise de requisitos, modelagem de projeto, construção de programa, testes e suporte. Os métodos da engenharia de software se baseiam em um conjunto de princípios básicos que governam cada área da tecnologia e incluem atividades de modelagem e outras técnicas descritivas. As ferramentas da engenharia de software fornecem suporte automatizado ou semiautomatizado para o processo e para os métodos. Quando as ferramentas são integradas, de modo que as informações criadas por uma ferramenta possam ser utilizadas por outra, é estabelecido um sistema para o suporte ao desenvolvimento de software, denominado engenharia de software com o auxílio do computador.

1.2. A natureza do software

Hoje, o software tem um duplo papel. Ele é um produto e o veículo para distribuir um produto. Como produto, fornece o potencial computacional representado pelo hardware ou, de forma mais abrangente, por uma rede de computadores que podem ser acessados por hardware local. Seja localizado em um dispositivo móvel, em um computador de mesa, na nuvem ou em um mainframe ou máquina autônoma, o software é um transformador de informações – produzindo, gerenciando, adquirindo, modificando, exibindo ou transmitindo

informações que podem ser tão simples quanto um único bit ou tão complexas quanto uma representação de realidade aumentada derivada de dados obtidos de dezenas de fontes independentes e, então, sobreposta ao mundo real. Como veículo de distribuição do produto, o software atua como a base para o controle do computador (sistemas operacionais), a comunicação de informações (redes) e a criação e o controle de outros programas (ferramentas de software e ambientes).

O software distribui o produto mais importante de nossa era – a informação. Ele transforma dados pessoais (p. ex., transações financeiras de um indivíduo) de modo que possam ser mais úteis em determinado contexto; gerencia informações comerciais para aumentar a competitividade; fornece um portal para redes mundiais de informação (p. ex., Internet); e proporciona os meios para obter informações sob todas as suas formas. Ele também propicia um veículo que pode ameaçar a privacidade pessoal e um portal que permite a pessoas mal-intencionadas cometer crimes.

O papel do software passou por uma mudança significativa no decorrer dos últimos 60 anos. Aperfeiçoamentos consideráveis no desempenho do hardware, mudanças profundas nas arquiteturas computacionais, um vasto aumento na capacidade de memória e armazenamento e uma ampla variedade de opções exóticas de entrada e saída – tudo isso resultou em sistemas computacionais mais sofisticados e complexos. Sosticação e complexidade podem produzir resultados impressionantes quando um sistema é bem-sucedido; porém, também podem trazer enormes problemas para aqueles que precisam desenvolver e projetar sistemas robustos.

Atualmente, uma enorme indústria de software tornou-se fator dominante nas economias do mundo industrializado. Equipes de especialistas em software, cada equipe concentrando-se numa parte da tecnologia necessária para distribuir uma aplicação complexa, substituíram o programador solitário de antigamente. Ainda assim, as questões levantadas por esse programador solitário continuam as mesmas hoje, quando os modernos sistemas computacionais são desenvolvidos:

- Por que a conclusão de um software leva tanto tempo?
- Por que os custos de desenvolvimento são tão altos?
- Por que não conseguimos encontrar todos os erros antes de entregarmos o software aos clientes?

- Por que gastamos tanto tempo e esforço realizando a manutenção de programas existentes?
- Por que ainda temos dificuldades de medir o progresso de desenvolvimento e a manutenção de um software?

Essas e muitas outras questões demonstram a preocupação com o software e com a maneira como ele é desenvolvido – uma preocupação que tem levado à adoção da prática da engenharia de software.

2. O PROCESSO DE SOFTWARE

Objetivo

O objetivo deste capítulo é apresentar o objetivo do processo de software. Passando por suas principais etapas e detalhando suas atividades.

Introdução

Um processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum artefato. Uma atividade se esforça para atingir um objetivo amplo (p. ex., comunicar-se com os envolvidos) e é utilizada independentemente do domínio de aplicação, do tamanho do projeto, da complexidade dos esforços ou do grau de rigor com que a engenharia de software será aplicada. Uma ação (p. ex., projeto de arquitetura) envolve um conjunto de tarefas que resultam em um artefato de software fundamental (p. ex., um modelo arquitetural). Uma tarefa se concentra em um objetivo pequeno, porém bem definido (p. ex., realizar um teste de unidades), e produz um resultado tangível. No contexto da engenharia de software, um processo não é uma prescrição rígida de como desenvolver um software. Ao contrário, é uma abordagem adaptável que possibilita às pessoas (a equipe de software) realizar o trabalho de selecionar e escolher o conjunto apropriado de ações e tarefas. A intenção é a de sempre entregar software dentro do prazo e com qualidade suficiente para satisfazer àqueles que patrocinaram sua criação e àqueles que vão utilizá-lo.

2.1. A metodologia do processo

Uma metodologia (framework) de processo estabelece o alicerce para um processo de engenharia de software completo por meio da identificação de um pequeno número de atividades metodológicas aplicáveis a todos os projetos de software, independentemente de tamanho ou complexidade. Além disso, a metodologia de processo engloba um conjunto de atividades de apoio (umbrella activities) aplicáveis a todo o processo de software. Uma metodologia de processo genérica para engenharia de software compreende cinco atividades:

- **Comunicação:** Antes que qualquer trabalho técnico possa começar, é de fundamental importância se comunicar e colaborar com o cliente (e outros

envolvidos). A intenção é entender os objetivos dos envolvidos para o projeto e reunir requisitos que ajudem a definir os recursos e as funções do software.

- **Planejamento:** Qualquer jornada complicada pode ser simplificada com o auxílio de um mapa. Um projeto de software é uma jornada complicada, e a atividade de planejamento cria um “mapa” que ajuda a guiar a equipe na sua jornada. O mapa – denominado plano de projeto de software – define o trabalho de engenharia de software, descrevendo as tarefas técnicas a serem conduzidas, os riscos prováveis, os recursos que serão necessários, os artefatos a serem produzidos e um cronograma de trabalho.

- **Modelagem:** Seja um paisagista, um construtor de pontes, um engenheiro aeronáutico, um carpinteiro ou um arquiteto, que trabalha com modelos todos os dias. Cria-se um “esboço” para que se possa ter uma ideia do todo – qual será o seu aspecto em termos de arquitetura, como as partes constituintes se encaixarão e várias outras características. Se necessário, refina-se o esboço com mais detalhes, numa tentativa de compreender melhor o problema e como resolvê-lo. Um engenheiro de software faz a mesma coisa, ele cria modelos para entender melhor as necessidades do software e o projeto que vai atender a essas necessidades.

- **Construção:** O que se projeta deve ser construído. Essa atividade combina geração de código (manual ou automatizada) e testes necessários para revelar erros na codificação.

- **Entrega:** O software (como uma entidade completa ou como um incremento parcialmente concluído) é entregue ao cliente, que avalia o produto e fornece feedback, baseado na avaliação.

Essas cinco atividades metodológicas genéricas podem ser utilizadas para o desenvolvimento de programas pequenos e simples, para a criação de aplicações para a Internet, e para a engenharia de grandes e complexos sistemas baseados em computador. Os detalhes do processo de software serão bem diferentes em cada caso, mas as atividades metodológicas permanecerão as mesmas.

Para muitos projetos de software, as atividades metodológicas são aplicadas interativamente conforme o projeto se desenvolve. Ou seja, comunicação, planejamento, modelagem, construção e entrega são aplicados repetidamente, sejam quantas forem as iterações do projeto. Cada iteração produzirá um incremento de software que disponibilizará uma parte dos recursos e das funcionalidades do software. A cada incremento, o software se torna cada vez mais completo.

2.2. Atividades de apoio

As atividades metodológicas do processo de engenharia de software são complementadas por diversas atividades de apoio. De modo geral, as atividades de apoio são aplicadas por todo um projeto de software e ajudam uma equipe de software a gerenciar e a controlar o andamento, a qualidade, as alterações e os riscos. As atividades de apoio típicas são:

- **Controle e acompanhamento do projeto:** Possibilita que a equipe avalie o progresso em relação ao plano do projeto e tome as medidas necessárias para cumprir o cronograma.

- **Administração de riscos:** Avalia riscos que possam afetar o resultado ou a qualidade do produto/projeto.

- **Garantia da qualidade de software:** Define e conduz as atividades que garantem a qualidade do software.

- **Revisões técnicas:** Avaliam artefatos da engenharia de software, tentando identificar e eliminar erros antes que eles se propaguem para a atividade seguinte.

- **Medição:** Define e coleta medidas (do processo, do projeto e do produto). Auxilia na entrega do software de acordo com os requisitos dos envolvidos; pode ser usada com as demais atividades (metodológicas e de apoio).

- **Gerenciamento da configuração de software:** Gerencia os efeitos das mudanças ao longo do processo.

- **Gerenciamento da capacidade de reutilização:** Dene critérios para a reutilização de artefatos (inclusive componentes de software) e estabelece mecanismos para a obtenção de componentes reutilizáveis.

- **Preparo e produção de artefatos de software:** Engloba as atividades necessárias para criar artefatos, como modelos, documentos, logs, formulários e listas.

3. CICLOS DE VIDA

Objetivo

O objetivo deste capítulo é apresentar os principais e tradicionais ciclos de vida de software e suas características.

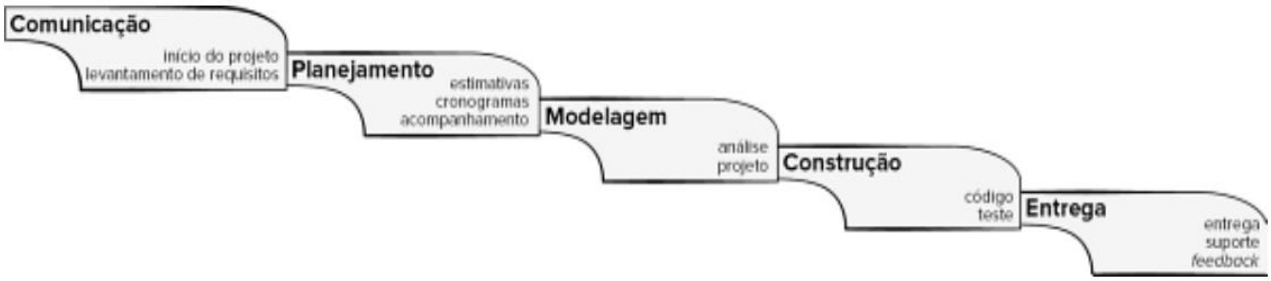
Introdução

Os ciclos de vida definem um conjunto prescrito de elementos de processo e um fluxo de trabalho de processo previsível. Um ciclo de vida concentra-se em estruturar e ordenar o desenvolvimento de software. As atividades e tarefas ocorrem sequencialmente, com diretrizes de progresso definidas. Mas os modelos prescritivos são adequados para um mundo do software que se alimenta de mudanças? Se rejeitarmos os modelos de processo tradicionais (e a ordem implícita) e os substituirmos por algo menos estruturado, tornaremos impossível atingir a coordenação e a coerência no trabalho de software? Não há respostas fáceis para essas questões, mas existem alternativas disponíveis para os engenheiros de software. A seguir será apresentado uma visão geral da abordagem dos processos prescritivos, nos quais a ordem e a consistência do projeto são questões predominantes. Chamamos esses processos de “prescritivos” porque prescrevem um conjunto de elementos de processo – atividades metodológicas, ações de engenharia de software, tarefas, artefatos, garantia da qualidade e mecanismos de controle de mudanças para cada projeto. Cada modelo de processo também prescreve um fluxo de processo (também denominado fluxo de trabalho) – ou seja, a forma pela qual os elementos do processo estão relacionados. Todos os modelos de processo de software podem acomodar as atividades metodológicas genéricas, porém cada um deles dá uma ênfase diferente a essas atividades e define um fluxo de processos que invoca cada atividade metodológica (bem como tarefas e ações de engenharia de software) de forma diversa.

3.1 O modelo cascata

Há casos em que os requisitos de um problema são bem compreendidos – quando o trabalho flui da comunicação à entrega de modo relativamente linear. Essa situação ocorre algumas vezes quando adaptações ou aperfeiçoamentos bem definidos precisam ser feitos em um sistema existente (p. ex., uma adaptação em software contábil exigida devido a mudanças nas normas governamentais). Pode ocorrer também em um número limitado de novos esforços de desenvolvimento, mas apenas quando os requisitos estão

bem definidos e são razoavelmente estáveis. O modelo cascata, algumas vezes chamado de modelo sequencial linear, sugere uma abordagem sequencial e sistemática para o desenvolvimento de software, começando com a especificação dos requisitos do cliente, avançando pelas fases de planejamento, modelagem, construção e entrega, e culminando no suporte contínuo do software concluído.



O modelo cascata é o paradigma mais antigo da engenharia de software. Entretanto, ao longo das últimas cinco décadas, as críticas a esse modelo de processo zeram até mesmo seus mais árduos defensores questionarem sua eficácia. Entre os problemas às vezes encontrados quando se aplica o modelo cascata, temos:

1. Projetos reais raramente seguem o fluxo sequencial proposto pelo modelo.

2. Com frequência, é difícil para o cliente estabelecer explicitamente todas as necessidades no início da maioria dos projetos.

3. O cliente deve ter paciência. Uma versão operacional do(s) programa(s) não estará disponível antes de estarmos próximos ao final do projeto.

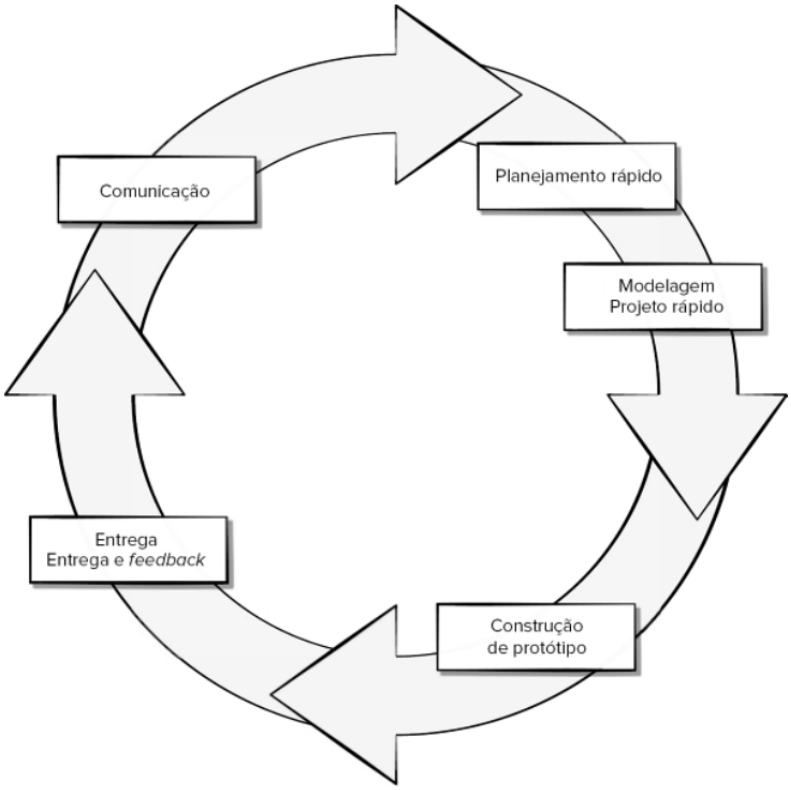
4. Erros graves podem não ser detectados até o programa operacional ser revisto.

Hoje, o trabalho com software tem um ritmo acelerado e está sujeito a uma cadeia de mudanças intermináveis (em características, funções e conteúdo de informações). O modelo cascata é frequentemente inadequado para esse trabalho.

3.2 Modelo de prototipação

Frequentemente, o cliente define uma série de objetivos gerais para o software, mas não identifica, com detalhes, os requisitos para funções e recursos. Em outros casos, o

desenvolvedor se encontra inseguro quanto à eficiência de um algoritmo, quanto à adaptabilidade de um sistema operacional ou quanto à forma em que deve ocorrer a interação homem-máquina. Em situações como essas, e em muitas outras, o paradigma da prototipação pode ser a melhor abordagem. Embora a prototipação possa ser utilizada como um modelo de processo isolado (stand-alone process), ela é mais comumente utilizada como uma técnica a ser implementada no contexto de qualquer um dos modelos de processo. Independentemente da forma como é aplicado, quando os requisitos estão obscuros, o paradigma da prototipação auxilia os envolvidos a compreender melhor o que está para ser construído. Por exemplo, um aplicativo de fitness desenvolvido usando protótipos incrementais poderia oferecer as telas básicas da interface do usuário necessárias para sincronizar um telefone ao dispositivo de fitness e mostrar a data atual; a capacidade de definir metas e armazenar os dados do dispositivo na nuvem seriam incluídas no segundo protótipo, que criaria e modificaria as telas da interface do usuário com base no feedback dos clientes; e o terceiro protótipo incluiria integração com mídias sociais para permitir que os usuários definissem metas e compartilhassem o seu progresso com os amigos. O paradigma da prototipação começa com a comunicação. Faz-se uma reunião com os envolvidos para definir os objetivos gerais do software, identificar os requisitos já conhecidos e esquematizar quais áreas necessitam, obrigatoriamente, de uma definição mais ampla. Uma iteração de prototipação é planejada rapidamente e ocorre a modelagem (na forma de um “projeto rápido”). Um projeto rápido se concentra em uma representação dos aspectos do software que serão visíveis para os usuários (p. ex., o layout da interface com o usuário ou os formatos de exibição na tela). O projeto rápido leva à construção de um protótipo. O protótipo é entregue e avaliado pelos envolvidos, os quais fornecem feedback que é usado para refinar ainda mais os requisitos. A iteração ocorre conforme se ajusta o protótipo às necessidades de vários envolvidos e, ao mesmo tempo, possibilita a melhor compreensão das necessidades que devem ser atendidas.



Na sua forma ideal, o protótipo atua como um mecanismo para identificar os requisitos do software. Caso seja necessário desenvolver um protótipo operacional, pode-se utilizar partes de programas existentes ou aplicar ferramentas que possibilitem gerar rapidamente tais programas operacionais. Tanto os envolvidos quanto os engenheiros de software gostam do paradigma da prototipação. Os usuários podem ter uma ideia prévia do sistema final, ao passo que os desenvolvedores passam a desenvolver algo imediatamente. Entretanto, a prototipação pode ser problemática pelas seguintes razões:

1. Os envolvidos enxergam o que parece ser uma versão operacional do software. Eles podem desconhecer que a arquitetura do protótipo (a estrutura do programa) também está evoluindo. Isso significa que os desenvolvedores podem não ter considerado a qualidade global do software, nem sua manutenção em longo prazo.

2. O engenheiro de software pode ficar tentado a fazer concessões na implementação para conseguir que o protótipo entre em operação rapidamente. Se não tomar cuidado, essas escolhas longe de ideais acabam se tornando uma parte fundamental do sistema.

Embora possam ocorrer problemas, a prototipação pode ser um paradigma eficiente para a engenharia de software. O segredo é definir as regras do jogo logo no início; ou seja, todos os envolvidos devem concordar que o protótipo é construído para servir como um mecanismo para definição de requisitos. Muitas vezes, é recomendável projetar um protótipo de modo que este possa evoluir e se transformar no produto final. A realidade é que os desenvolvedores podem precisar descartar (pelo menos em parte) um protótipo para melhor atender às novas necessidades do cliente.

3.3 Modelo de processo evolucionário/espiral

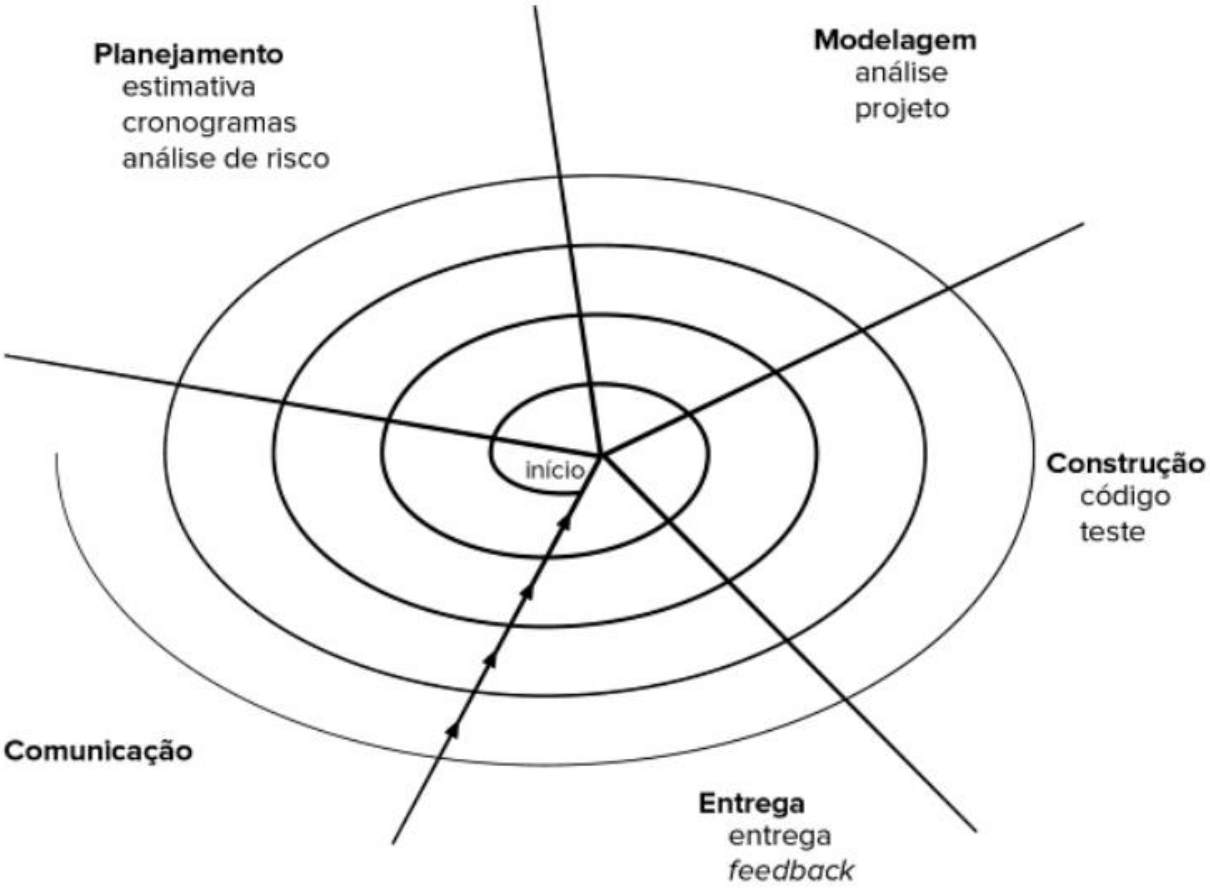
Como todos os sistemas complexos, o software evolui ao longo do tempo. Conforme o desenvolvimento do projeto avança, os requisitos do negócio e do produto frequentemente mudam, tornando inadequado seguir um planejamento em linha reta de um produto final. Prazos apertados tornam impossível concluir um produto de software abrangente. Pode ser possível criar uma versão limitada do produto para atender às pressões comerciais ou da concorrência e lançar uma versão refinada após um melhor entendimento de todas as características do sistema. Em situações como essa, faz-se necessário um modelo de processo que tenha sido projetado especificamente para desenvolver um produto que cresce e muda.

Originalmente proposto por Barry Boehm, o modelo espiral é um modelo de processo de software evolucionário que une a natureza iterativa da prototipação aos aspectos sistemáticos e controlados do modelo cascata. Tem potencial para o rápido desenvolvimento de versões cada vez mais completas do software.

Com o modelo espiral, o software será desenvolvido em uma série de versões evolucionárias. Nas primeiras iterações, a versão pode consistir em um modelo ou em um protótipo. Já nas iterações posteriores, são produzidas versões cada vez mais completas do sistema que passa pelo processo de engenharia.

O modelo espiral é dividido em um conjunto de atividades metodológicas definidas pela equipe de engenharia de software. A título de ilustração, utilizam-se as atividades metodológicas genéricas discutidas anteriormente. Cada uma dessas atividades representa um segmento do caminho espiral ilustrado na figura abaixo. Assim que esse processo evolucionário começa, a equipe de software realiza atividades indicadas por um circuito em

torno da espiral, no sentido horário, começando pelo seu centro. Os riscos são levados em conta à medida que cada evolução é realizada. Marcos de pontos-âncora – uma combinação de artefatos e condições satisfeitas ao longo do trajeto da espiral – são indicados para cada passagem evolucionária.



O primeiro circuito em volta da espiral (iniciando na linha de fluxo interna mais próxima ao centro) pode resultar no desenvolvimento de uma especificação de produto; passagens subsequentes em torno da espiral podem ser usadas para desenvolver um protótipo e, então, progressivamente, versões cada vez mais sosticadas do software. Cada passagem pela região de planejamento resulta em ajustes no planejamento do projeto. Custo e cronograma são ajustados de acordo com o feedback (a realimentação) obtido do cliente após a entrega. Além disso, o gerente de projeto faz um ajuste no número de iterações planejadas para concluir o software.

Diferentemente de outros modelos de processo, que terminam quando o software é entregue, o modelo espiral pode ser adaptado para ser aplicado ao longo da vida do software. O modelo espiral é uma abordagem realista para o desenvolvimento de sistemas

e de software em larga escala. Ele usa a prototipação como mecanismo de redução de riscos. O modelo espiral exige consideração direta dos riscos técnicos em todos os estágios do projeto e, se aplicado apropriadamente, reduz os riscos antes destes se tornarem problemáticos.

Como outros paradigmas, esse modelo não é uma panaceia. Pode ser difícil convencer os clientes (particularmente em situações contratuais) de que a abordagem evolucionária é controlável. Ela exige considerável especialização na avaliação de riscos e depende dessa especialização para seu sucesso. Se um risco muito importante não for descoberto e administrado, sem dúvida ocorrerão problemas.

Conforme já mencionado, o software moderno é caracterizado por contínuas modificações, prazos muito apertados e por uma ênfase na satisfação do cliente-usuário. Em muitos casos, o tempo de colocação de um produto no mercado é o requisito mais importante a ser gerenciado. Se o momento oportuno de entrada no mercado for perdido, o projeto de software pode perder o sentido.

O objetivo dos modelos evolucionários é desenvolver software de alta qualidade de modo iterativo ou incremental. Entretanto, é possível usar um processo evolucionário para enfatizar a exigibilidade, a extensibilidade e a velocidade de desenvolvimento. O desafio para as equipes de software e seus gerentes será estabelecer um equilíbrio apropriado entre esses parâmetros críticos de projeto e produto e a satisfação dos clientes (o árbitro nal da qualidade de um software).

3.4 Modelo de processo unificado

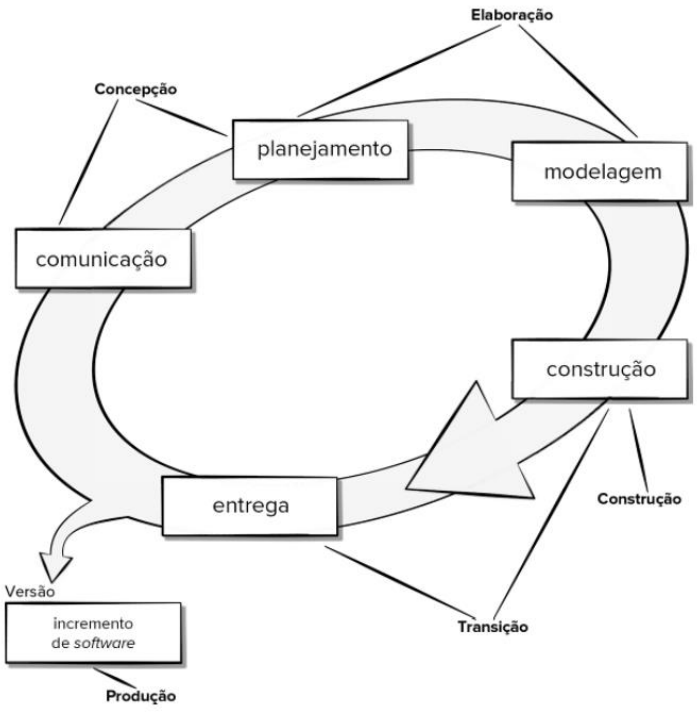
Sob certos aspectos, o Processo Unificado é uma tentativa de aproveitar os melhores recursos e características dos modelos tradicionais de processo de software, mas caracterizando-os de modo a implementar muitos dos melhores princípios do desenvolvimento ágil de software. O Processo Unificado reconhece a importância da comunicação com o cliente e de métodos racionalizados para descrever a visão do cliente sobre um sistema (os casos de uso). Ele enfatiza o importante papel da arquitetura de software e “ajuda o arquiteto a manter o foco nas metas corretas, como compreensibilidade, confiança em mudanças futuras e reutilização”. Ele sugere um fluxo de processo iterativo e

incremental, proporcionando a sensação evolucionária que é essencial no desenvolvimento de software moderno.

A UML (unified modeling language), a linguagem de modelagem unificada, foi desenvolvida para apoiar o seu trabalho, contém uma notação robusta para a modelagem e o desenvolvimento de sistemas orientados a objetos e tornou-se um padrão da indústria para o desenvolvimento de software de todos os tipos. A UML é usada para representar tanto modelos de projeto quanto de requisitos.

Na fase de concepção do PU ocorre a comunicação com o cliente e o planejamento. Requisitos de negócio fundamentais são descritos em um conjunto de casos de uso preliminares, descrevendo quais recursos e funções são esperados na arquitetura de software por cada categoria principal de usuário. O planejamento identifica recursos, avalia os principais riscos e define um cronograma preliminar para os incrementos de software.

A fase de elaboração inclui as atividades de planejamento e modelagem do modelo de processo genérico. A elaboração refina e expande os casos de uso preliminares e cria uma base de arquitetura, incluindo cinco diferentes visões do software: modelo de caso de uso, modelo de análise, modelo de projeto, modelo de implementação e modelo de entrega. 8 Normalmente, as modificações no planejamento são feitas nesta fase.



A fase de construção do PU é idêntica à atividade de construção definida para o processo de software genérico. Implementam-se, então, no código-fonte, todos os recursos e funções necessários e exigidos para o incremento de software (i.e., para a versão). À medida que os componentes são implementados, desenvolvem-se e executam-se testes de unidade 9 para cada um deles. Além disso, realizam-se atividades de integração (montagem de componentes e testes de integração). Os casos de uso são utilizados para se obter um pacote de testes de aceitação, executados antes do início da fase seguinte do PU.

A fase de transição do PU abrange os últimos estágios da atividade de construção genérica e a primeira parte da atividade de emprego genérico: entrega e feedback. Entrega-se o software aos usuários para testes beta, e o feedback dos usuários relata defeitos e mudanças necessárias. Na conclusão da fase de transição, o incremento torna-se uma versão utilizável do software.

A fase de produção do PU coincide com a atividade de entrega do processo genérico. Durante esta fase, monitora-se o uso contínuo do software, disponibiliza-se suporte para o ambiente (infraestrutura) operacional, realizam-se e avaliam-se relatórios de defeitos e solicitações de mudanças.

É provável que, ao mesmo tempo em que as fases de construção, transição e produção estejam sendo conduzidas, já se tenha iniciado o incremento de software seguinte. Isso significa que as cinco fases do PU não ocorrem em sequência, mas de forma concomitante e escalonada.

Deve-se observar que nem toda tarefa identificada para um uxo de trabalho do PU é conduzida em todos os projetos de software. A equipe adapta o processo (ações, tarefas, subtarefas e artefatos) para ficar de acordo com suas necessidades.

3.5 Comparação entre os ciclos de vida

A tabela abaixo resume alguns dos pontos fortes e fracos dos modelos de processo. A realidade é que nenhum processo é perfeito para todos os projetos. Em geral, a equipe de software adapta um ou mais dos modelos de processo ou os modelos de processo ágil para atender às necessidades do projeto do momento.

Prós do modelo cascata	É fácil de entender e planejar. Funciona para projetos pequenos e bem compreendidos. A análise e o teste são simples e diretos.
Contras do modelo cascata	Não se adapta bem a mudanças. O teste ocorre nas fases finais do processo. A aprovação do cliente vem no nal.
Prós da prototipação	O impacto das alterações aos requisitos é reduzido. O cliente se envolve bastante e desde o início. Funciona bem para projetos pequenos. A probabilidade de rejeição do produto é reduzida.
Contras da prototipação	O envolvimento do cliente pode causar atrasos. Pode haver a tentação de “embalar” o protótipo. Desperdiça-se trabalho em um protótipo descartável. É difícil de planejar e gerenciar.
Prós do modelo espiral	Há envolvimento contínuo dos clientes. Os riscos de desenvolvimento são gerenciados. É apropriado para modelos grandes e complexos. Funciona bem para artefatos extensíveis.
Contras do modelo espiral	Falhas de análise de risco podem fadar o projeto ao fracasso. O projeto pode ser difícil de gerenciar. Exige uma equipe de desenvolvimento especializada.
Prós do Processo Unificado	A documentação de alta qualidade é enfatizada. Há envolvimento contínuo dos clientes. Adapta-se a alterações aos requisitos. Funciona bem para projetos de manutenção.
Contras do Processo Unificado	Os casos de uso nem sempre são precisos. A integração de incrementos de software é complicada. A sobreposição das fases pode causar problemas. Exige uma equipe de desenvolvimento especializada.

4. AGILIDADE E A EVOLUÇÃO DOS PROCESSOS

Objetivo

O objetivo deste capítulo é apresentar os conceitos e objetivos da agilidade, passando pela evolução dos processos.

Introdução

Em 2001, um grupo composto por desenvolvedores de software, autores e consultores de renome assinou o “Manifesto para o desenvolvimento ágil de software” (“Manifesto for Agile software Development”), no qual defendiam “indivíduos e interações acima de processos e ferramentas, software operacional acima de documentação completa, colaboração dos clientes acima de negociação contratual e respostas a mudanças acima de seguir um plano”.

As ideias fundamentais que norteiam o desenvolvimento ágil levaram ao desenvolvimento dos métodos ágeis, projetados para sanar fraquezas, supostas e reais, da engenharia de software convencional. O desenvolvimento ágil oferece benefícios importantes; no entanto, não é indicado para todos os projetos, produtos, pessoas e situações. Também não é a antítese da prática de engenharia de software confiável e pode ser aplicado como uma filosofia geral para todos os trabalhos de software.

Na economia moderna, frequentemente é difícil ou impossível prever como um sistema computacional (p. ex., um aplicativo móvel) vai evoluir com o tempo. As condições de mercado mudam rapidamente, as necessidades dos usuários se alteram, e novas ameaças competitivas surgem sem aviso. Em muitas situações, não se conseguirá definir os requisitos completamente antes que se inicie o projeto. É preciso ser ágil o suficiente para dar uma resposta a um ambiente de negócios fluido.

Fluidez implica mudança, e mudança é cara – particularmente se for sem controle e mal gerenciada. Uma das características mais convincentes da metodologia ágil é sua habilidade de reduzir os custos da mudança no processo de software.

Em um texto instigante sobre desenvolvimento de software ágil, Alistair Cockburn argumenta que o modelo de processo prescritivo, apresentado, tem uma falha essencial: esquece-se das fraquezas das pessoas que desenvolvem o software. Os engenheiros de

software não são robôs. Eles apresentam grande variação nos estilos de trabalho e diferenças significativas no nível de habilidade, criatividade, organização, consistência e espontaneidade. Alguns se comunicam bem na forma escrita, outros não. Para que funcionem, os modelos de processos devem fornecer um mecanismo realista que estimule a disciplina necessária ou, então, devem ter características que apresentem “tolerância” com as pessoas que realizam trabalhos de engenharia de software.

4.1 O que é agilidade?

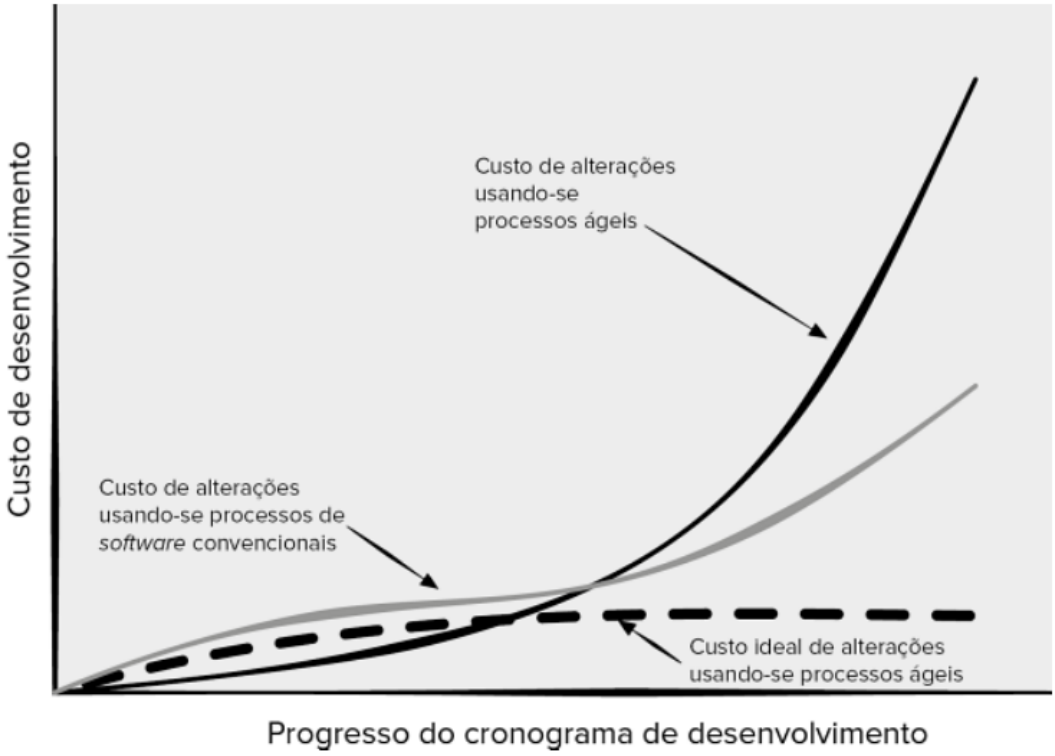
Afinal, o que é agilidade no contexto da engenharia de software? Ivar Jacobson argumenta que a difusão da mudança é o principal condutor para a agilidade. Os engenheiros de software devem ser rápidos caso queiram assimilar as rápidas mudanças que Jacobson descreve.

Entretanto, agilidade é mais do que uma resposta à mudança. Ela abrange também a filosofia proposta no manifesto. Ela incentiva a estruturação e as atitudes em equipe que tornam a comunicação mais fácil (entre membros da equipe, entre o pessoal ligado à tecnologia e o pessoal da área comercial, entre os engenheiros de software e seus gerentes). Ela enfatiza a entrega rápida do software operacional e diminui a importância dos artefatos intermediários (nem sempre um bom negócio); aceita o cliente como parte da equipe de desenvolvimento e trabalha para eliminar a atitude de “nós e eles” que continua a impregnar muitos projetos de software; reconhece que o planejamento em um mundo incerto tem seus limites, e que o plano (roteiro) de projeto deve ser flexível.

A agilidade pode ser aplicada a qualquer processo de software. No entanto, para alcançá-la, é essencial que o processo seja projetado de modo que a equipe possa adaptar e alinhar (racionalizar) tarefas; possa conduzir o planejamento, compreendendo a fluidez de uma metodologia de desenvolvimento ágil; possa eliminar tudo, exceto os artefatos essenciais, conservando-os enxutos; e possa enfatizar a estratégia de entrega incremental, conseguindo entregar ao cliente, o mais rapidamente possível, o software operacional para o tipo de produto e ambiente operacional. Não cometa o erro de supor que a agilidade lhe dará licença para abreviar soluções. Processo é um requisito, e disciplina é essencial.

4.2 Agilidade e o custo das mudanças

O senso comum no desenvolvimento de software (baseado em décadas de experiência) é que os custos de mudanças aumentam de forma não linear conforme o projeto avança (curva em preto contínua). É relativamente fácil acomodar uma mudança quando a equipe de software está reunindo requisitos (no início de um projeto). Talvez seja necessário alterar um detalhamento do uso, ampliar uma lista de funções ou editar uma especificação por escrito. Os custos desse trabalho são mínimos, e o tempo demandado não afetará negativamente o resultado do projeto. Mas, se adiantarmos alguns meses, o que aconteceria? A equipe está em meio aos testes de validação (que ocorrem relativamente no final do projeto), e um importante envolvido está solicitando uma mudança funcional grande. A mudança exige uma alteração no projeto da arquitetura do software, projeto e desenvolvimento de três novos componentes, modificações em outros cinco componentes, projeto de novos testes e assim por diante. Os custos crescem rapidamente, e o tempo e os esforços necessários para assegurar que a mudança seja feita sem efeitos colaterais inesperados não serão insignificantes.



Os proponentes da agilidade argumentam que um processo ágil bem elaborado “achata” o custo da curva de mudança, permitindo que uma equipe de software assimile as alterações, realizadas posteriormente em um projeto de software, sem um impacto

significativo nos custos ou no tempo. Já foi mencionado que o processo ágil envolve entregas incrementais. O custo das mudanças é atenuado quando a entrega incremental é associada a outras práticas ágeis, como testes contínuos de unidades e programação em pares. Há evidências que sugerem ser possível alcançar redução significativa nos custos de alterações, embora haja um debate contínuo sobre qual o nível em que a curva de custos se torna “achatada”.

4.3 O que é processo ágil?

Qualquer processo ágil de software é caracterizado de uma forma que trate de uma série de preceitos-chave acerca da maioria dos projetos de software:

1. É difícil prever quais requisitos de software vão persistir e quais sofrerão alterações. É igualmente difícil prever de que maneira as prioridades do cliente sofrerão alterações conforme o projeto avança.

2. Para muitos tipos de software, o projeto e a construção são intercalados. Ou seja, ambas as atividades devem ser realizadas em conjunto para que os modelos de projeto sejam provados conforme são criados. É difícil prever quanto de trabalho de projeto será necessário antes que a sua construção (desenvolvimento) seja implementada para avaliar o projeto.

3. Análise, projeto, construção (desenvolvimento) e testes não são tão previsíveis (do ponto de vista de planejamento) quanto gostaríamos que fossem.

Dados esses três preceitos, surge uma importante questão: como criar um processo capaz de administrar a imprevisibilidade? A resposta, conforme já observado, está na adaptabilidade do processo (alterar rapidamente o projeto e as condições técnicas). Portanto, um processo ágil deve ser adaptável.

Adaptação contínua sem progressos, entretanto, de pouco adianta. Um processo ágil de software deve adaptar de modo incremental. Para conseguir uma adaptação incremental, a equipe ágil precisa de feedback do cliente (de modo que as adaptações apropriadas possam ser feitas). Um catalisador eficaz para o feedback do cliente é um protótipo operacional ou parte de um sistema operacional. Dessa forma, deve-se instituir

uma estratégia de desenvolvimento incremental. Os incrementos de software (protótipos executáveis ou partes de um sistema operacional) devem ser entregues em curtos períodos, de modo que as adaptações acompanhem o mesmo ritmo das mudanças (imprevisibilidade). Essa abordagem iterativa capacita o cliente a avaliar o incremento de software regularmente, fornecer o feedback necessário para a equipe de software e influenciar as adaptações feitas no processo para incluir o feedback adequadamente.

4.3.1. Princípios de agilidade

A Agile Alliance estabelece 12 princípios para alcançar a agilidade, os quais são sintetizados nos parágrafos a seguir.

A satisfação do cliente é alcançada com a oferta de valor por meio do software entregue ao cliente o mais rapidamente possível. Para tanto, os desenvolvedores ágeis reconhecem que os requisitos irão se alterar. Eles entregam os incrementos de software com frequência e trabalham ao lado de todos os envolvidos para que o seu feedback sobre as entregas seja rápido e significativo.

Uma equipe ágil é composta por indivíduos motivados, que se comunicam presencialmente e trabalham em um ambiente propício ao desenvolvimento de software de alta qualidade. A equipe segue um processo que incentiva a excelência técnica e o bom projeto, enfatizando a simplicidade – “a arte de maximizar o volume de trabalho não realizado”.

Um software operacional que atenda às necessidades do cliente é o objetivo principal, e o ritmo e a direção do trabalho da equipe devem ser “sustentáveis”, permitindo-a trabalhar de forma eficaz por longos períodos. Uma equipe ágil é uma “equipe auto-organizada” – capaz de desenvolver arquiteturas bem estruturadas, que levam a projetos sólidos e à satisfação do cliente. Parte da cultura da equipe é considerar o seu trabalho introspectivamente, sempre com a intenção de melhorar a forma como busca o seu objetivo principal.

Nem todo modelo de processo ágil atribui pesos iguais às características descritas nesta seção, e alguns modelos preferem ignorar (ou, pelo menos, minimizar) a importância de um ou mais desses princípios. Entretanto, os princípios definem um espírito ágil mantido em cada um dos modelos de processo.

4.3.2 A política do desenvolvimento ágil

Há debates consideráveis (algumas vezes acirrados) sobre os benefícios e a aplicabilidade do desenvolvimento de software ágil, em contraposição aos processos de engenharia de software mais convencionais. Jim Highsmith estabelece extremos ao caracterizar o sentimento do grupo pró-agilidade (“os agilistas”): “Os metodologistas tradicionais são um bando de ‘pés na lama’ que preferem produzir documentação sem falhas em vez de um sistema que funcione e atenda às necessidades do negócio”. Em um contraponto, ele apresenta (mais uma vez em tom jocoso) a posição do grupo da engenharia de software tradicional: “Os metodologistas de pouco peso, quer dizer, os metodologistas ‘ágeis’ são um bando de hackers pretensiosos que vão acabar tendo uma grande surpresa ao tentarem transformar seus brinquedinhos em software de porte empresarial”.

Como todo argumento sobre tecnologia de software, o debate sobre metodologia corre o risco de descambar para uma guerra santa. Se for deflagrada uma guerra, a racionalidade desaparecerá, e crenças, em vez de fatos, orientarão a tomada de decisão.

Ninguém é contra a agilidade. A verdadeira questão é: qual a melhor maneira de atingi-la? Lembre-se que software ativo é importante, mas não se deve esquecer que ele também deve apresentar uma série de atributos de qualidade, incluindo confiabilidade, usabilidade e facilidade de manutenção. Como desenvolver software que atenda às

necessidades atuais dos clientes e que apresente características de qualidade que o permitam ser estendido e ampliado para responder às necessidades dos clientes em longo prazo?

Não há respostas absolutas para nenhuma dessas perguntas. Mesmo na própria escola ágil, existem vários modelos de processos propostos, cada um com uma abordagem sutilmente diferente a respeito do problema da agilidade. Em cada modelo existe um conjunto de “ideias” (os agilistas relutam em chamá-las de “tarefas de trabalho”) que representam um afastamento significativo da engenharia de software tradicional. E, ainda assim, muitos conceitos ágeis são apenas adaptações de bons conceitos da engenharia de software. Conclusão: pode-se ganhar muito considerando o que há de melhor nas duas escolas e praticamente nada denegrindo uma ou outra abordagem.

5. SCRUM

Objetivo

O objetivo deste capítulo é apresentar o framework mais famoso do mundo ágil, o Scrum. Veremos também todos os atores envolvidos nesse framework, além de todas as suas cerimônias.

Introdução

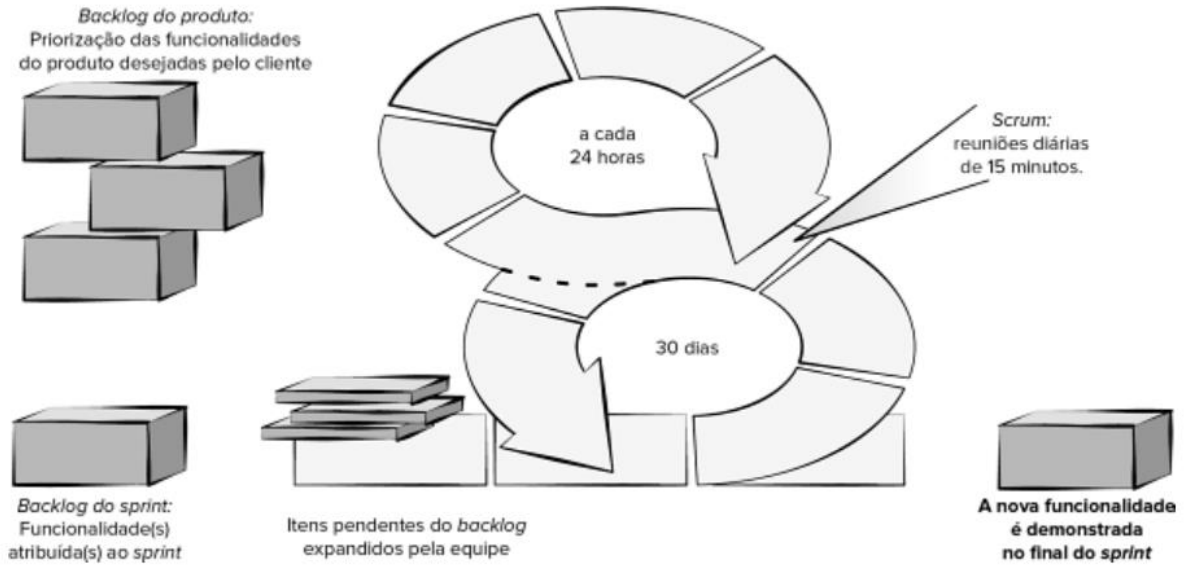
Scrum (o nome provém de uma atividade que ocorre durante a partida de rugby) é um método de desenvolvimento ágil de software bastante popular concebido por Jeff Sutherland e sua equipe de desenvolvimento no início dos anos 1990. Schwaber e Beedle realizaram desenvolvimentos adicionais nos métodos Scrum. Os princípios do Scrum são coerentes com o manifesto ágil e são usados para orientar as atividades de desenvolvimento dentro de um processo que incorpora as seguintes atividades metodológicas: requisitos, análise, projeto, evolução e entrega. Em cada atividade metodológica, ocorrem tarefas realizadas em um período (janela de tempo) chamado de sprint. O trabalho realizado dentro de um sprint (o número de sprints necessários para cada atividade metodológica varia dependendo do tamanho e da complexidade do produto) é adaptado ao problema em questão e definido, e muitas vezes modificado em tempo real, pela equipe Scrum.

5.1 Equipes e artefatos do Scrum

A equipe (ou “time”) Scrum é uma equipe interdisciplinar que se auto-organiza, e é composta por um product owner, um Scrum master e uma pequena equipe de desenvolvimento (três a seis pessoas). Os principais artefatos do Scrum são o backlog do produto, o backlog do sprint e o incremento de código. O desenvolvimento avança pela divisão do projeto em uma série de períodos de desenvolvimento incremental do protótipo, com duração de 2 a 4 semanas, chamados de sprints.

O backlog do produto é uma lista priorizada de requisitos ou características do artefato que agregam valor de negócio para o cliente. Itens podem ser adicionados ao backlog a qualquer momento com a aprovação do product owner e o consentimento da equipe de desenvolvimento. O product owner ordena os itens no backlog do produto para cumprir as metas mais importantes de todos os envolvidos. O backlog do produto nunca

está completo enquanto o produto evolui para atender às necessidades dos envolvidos. O product owner é a única pessoa que pode decidir encerrar um sprint antecipadamente ou estendê-lo caso o incremento não seja aceito.



O backlog do sprint é um subconjunto de itens do backlog do produto selecionado pela equipe do produto para ser completado na forma do incremento de código durante o sprint ativo atual. O incremento representa a união de todos os itens do backlog do produto completados nos sprints anteriores e todos os itens do backlog a serem completados nos sprints atuais. A equipe de desenvolvimento cria um plano para entregar um incremento de software que contém as características selecionadas, com a intenção de cumprir uma meta importante, como negociado com o product owner no sprint atual. A maioria dos sprints tem uma limitação de tempo (ou seja, são time-boxed), e devem ser completados em 2 a 4 semanas. O modo como a equipe de desenvolvimento completa o incremento é deixado para que a própria equipe decida. A equipe de desenvolvimento também decide quando o incremento está pronto e pode ser demonstrado para o product owner. Nenhuma nova característica pode ser adicionada ao backlog do sprint a menos que o sprint seja cancelado ou reiniciado.

O Scrum master atua como facilitador para todos os membros da equipe Scrum. Ele comanda a reunião diária do Scrum e é responsável por remover os obstáculos identificados pelos membros da equipe durante a reunião. Ele orienta os membros da equipe de desenvolvimento para que consigam ajudar uns aos outros a completar as tarefas do sprint quando têm disponibilidade de tempo. Ele ajuda o product owner a identificar

técnicas para gerenciar os itens do backlog do produto e a garantir que os itens do backlog estejam enunciados de forma clara e concisa.

5.2 Reunião de planejamento do sprint (Planning)

Antes de iniciar, toda a equipe de desenvolvimento trabalha com o product owner e com todos os outros envolvidos para desenvolver os itens no backlog do produto. O product owner e a equipe de desenvolvimento ordenam os itens no backlog do produto pela importância das necessidades de negócio do primeiro e a complexidade das tarefas de engenharia de software (programação e teste) necessárias para completá-los. Às vezes, o resultado é a identificação de características ausentes que serão necessárias para produzir a funcionalidade exigida para os usuários.

Antes do início de cada sprint, o product owner enuncia a meta de desenvolvimento para o incremento a ser completado no próximo sprint. O Scrum master e a equipe de desenvolvimento selecionam os itens a serem transferidos para o backlog do sprint. A equipe de desenvolvimento determina o que pode ser entregue no incremento dentro dos limites da janela de tempo alocada ao sprint e, junto ao Scrum master, qual será o trabalho necessário para entregar o incremento. A equipe de desenvolvimento decide quais papéis são necessários e como precisarão ser atendidos.

5.3 Reunião diária do Scrum (Daily)

A reunião diária do Scrum é um evento de 15 minutos programado no início de cada dia de trabalho para permitir que os membros da equipe sincronizem as suas atividades e se planejem para as próximas 24 horas. O Scrum master e a equipe de desenvolvimento sempre participam da reunião diária. Algumas equipes permitem que o product owner participe ocasionalmente.

São feitas três perguntas-chave que são respondidas por todos os membros da equipe:

- O que você realizou desde a última reunião de equipe?
- Quais obstáculos está encontrando?
- O que planeja realizar até a próxima reunião da equipe?

O Scrum master conduz a reunião e avalia as respostas de cada integrante. A reunião Scrum, realizada diariamente, ajuda a equipe a revelar problemas em potencial o mais cedo possível. É responsabilidade do Scrum master eliminar os obstáculos apresentados antes da próxima reunião diária, se possível. Não são reuniões de solução de problemas, pois estas ocorrem posteriormente e envolvem apenas as partes afetadas. A reunião diária também leva à “socialização do conhecimento” e, portanto, promove uma estrutura de equipe auto-organizada.

Algumas equipes usam essas reuniões para declarar quais itens do backlog do sprint estão completos ou prontos. Quando considerar que todos os itens do backlog do sprint estão completos, a equipe pode decidir marcar uma demonstração ou revisão do incremento completo junto com o product owner.

5.4 Reunião de revisão do sprint (Review)

A revisão do sprint ocorre no final do sprint, quando a equipe de desenvolvimento decidiu que o incremento está completo. Em geral, a revisão do sprint é limitada a uma reunião de 4 horas para um sprint de 4 semanas. O Scrum master, a equipe de desenvolvimento, o product owner e alguns envolvidos participam.

A atividade principal é uma demo (ou demonstração) do incremento de software completado durante o sprint. É importante notar que a demo pode não ter toda a funcionalidade planejada, mas sim funções que seriam entregues na janela de tempo estipulada para o sprint.

O product owner pode aceitar o incremento como completo ou não. Se ele não for aceito, o product owner e os envolvidos fornecem feedback para possibilitar a realização de uma nova rodada de planejamento do sprint. É neste momento que novas características podem ser adicionadas ou eliminadas do backlog do produto. As novas características podem afetar a natureza do incremento desenvolvido no próximo sprint.

5.5 Retrospectiva do sprint (Retro)

Em uma situação ideal, antes de começar outra reunião de planejamento do sprint, o Scrum master marca uma reunião de 3 horas (para um sprint de 4 semanas) com a equipe

de desenvolvimento, chamada de retrospectiva do sprint. Durante essa reunião, a equipe debate:

- O que deu certo no sprint.
- O que poderia melhorar.
- Com o que a equipe se compromete em melhorar no próximo sprint.

O Scrum master lidera a reunião e encoraja a equipe a melhorar as suas práticas de desenvolvimento para se tornar mais eficaz para o próximo sprint. A equipe planeja formas de melhorar a qualidade do produto com a adaptação da sua definição de “pronto”. Ao final da reunião, a equipe deve ter uma boa ideia sobre as melhorias necessárias no próximo sprint e sobre como estar preparada para planejar o incremento na próxima reunião de planejamento do sprint.

6. OUTROS FRAMEWORKS ÁGEIS

Objetivo

O objetivo deste capítulo é apresentar outros frameworks ágeis além do Scrum. Veremos também todas as suas etapas.

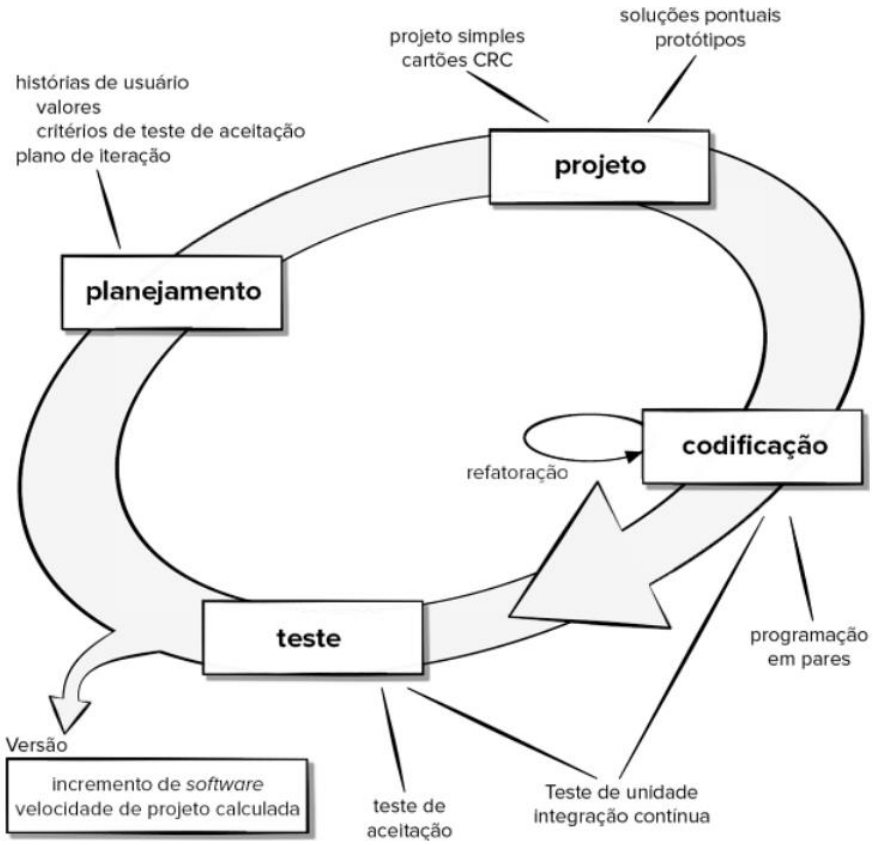
Introdução

A história da engenharia de software é recheada de metodologias e descrições de processos, métodos e notações de modelagem, ferramentas e tecnologias obsoletas. Todas atingiram certa notoriedade e foram ofuscadas por algo novo e (supostamente) melhor. Com a introdução de uma ampla variedade de frameworks de processos ágeis – todos disputando aceitação pela comunidade de desenvolvimento de software –, o movimento ágil seguiu o mesmo caminho histórico.

Conforme citado na última seção, o framework ágil mais utilizado de todos é o Scrum. Porém, muitos outros têm sido propostos e encontram-se em uso no setor. Nesta seção, apresentamos um breve panorama de três métodos ágeis populares: Extreme Programming (XP), Kanban e DevOps.

6.1 O framework XP

A Extreme Programming (Programação Extrema) envolve um conjunto de regras e práticas constantes no contexto de quatro atividades metodológicas: planejamento, projeto, codificação e testes. A figura abaixo ilustra o processo XP e destaca alguns conceitos e tarefas-chave associados a cada uma das atividades metodológicas. As atividades-chave da XP são sintetizadas nos parágrafos a seguir.



Planejamento. A atividade de planejamento (também chamada de o jogo do planejamento) se inicia com ouvir. A atividade de ouvir conduz à criação de um conjunto de "histórias" (também denominadas histórias de usuário) que descreve o resultado, as características e a funcionalidade solicitados para o software a ser construído. Cada história de usuário é escrita pelo cliente e colocada em uma ficha. O cliente atribui um valor (i.e., uma prioridade) à história baseando-se no valor de negócio global do recurso ou da função. Os membros da equipe XP avaliam, então, cada história e atribuem um custo – medido em semanas de desenvolvimento – a ela. É importante notar que podem ser escritas novas histórias a qualquer momento.

Clientes e desenvolvedores trabalham juntos para decidir como agrupar histórias para a versão seguinte (o próximo incremento de software) a ser desenvolvida pela equipe XP. Conseguindo chegar a um compromisso básico (concordância sobre quais histórias serão incluídas, data de entrega e outras questões de projeto) para uma versão, a equipe XP ordena as histórias que serão desenvolvidas em uma de três formas: (1) todas as histórias serão implementadas imediatamente (em um prazo de poucas semanas); (2) as histórias de maior valor serão deslocadas para cima no cronograma e implementadas

primeiro; ou (3) as histórias de maior risco serão deslocadas para cima no cronograma e implementadas primeiro.

Depois de a primeira versão do projeto (também denominada incremento de software) ter sido entregue, a equipe XP calcula a velocidade do projeto. De forma simples, a velocidade do projeto é o número de histórias de clientes implementadas durante a primeira versão. Assim, a velocidade do projeto pode ser utilizada para ajudar a estimar as datas de entrega e o cronograma para versões subsequentes. A equipe XP modifica seus planos para se adaptar a ela.

Projeto. O projeto XP segue rigorosamente o princípio KISS (keep it simple, stupid!, ou seja, não complique!). O projeto de funcionalidade extra (pelo fato de o desenvolvedor supor que ele será necessário no futuro) é desestimulado.

A XP estimula o uso de cartões CRC (classe-responsabilidade-colaborador) como um mecanismo eficaz para pensar o software em um contexto orientado a objetos. Os cartões CRC identificam e organizam as classes orientadas a objetos relevantes para o incremento de software corrente. Os cartões CRC são o único artefato de projeto produzido como parte do processo XP.

Se for encontrado um problema de projeto difícil, como parte do projeto de uma história, a XP recomenda a criação imediata de um protótipo operacional dessa parte do projeto. Um aspecto central na XP é o de que a elaboração do projeto ocorre tanto antes quanto depois de se ter iniciado a codificação. Refatoração (modificar/otimizar o código sem alterar o comportamento externo do software) significa que o “projetar” é realizado continuamente enquanto o sistema estiver em elaboração. Na realidade, a própria atividade de desenvolvimento guiará a equipe XP quanto ao aprimoramento do projeto.

Codificação. Depois de desenvolvidas as histórias e de o trabalho preliminar de elaboração do projeto ter sido feito, a equipe não passa para a codificação, mas desenvolve uma série de testes de unidades que exercitarão cada uma das histórias a ser incluída na versão corrente (incremento de software). Uma vez criado o teste de unidades, o desenvolvedor poderá se concentrar melhor no que deve ser implementado para ser aprovado no teste. Estando o código completo, ele pode ser testado em unidade

imediatamente e, dessa forma, fornecer feedback para os desenvolvedores instantaneamente.

Um conceito-chave na atividade de codificação (e um dos mais discutidos aspectos da XP) é a programação em pares. A XP recomenda que duas pessoas trabalhem juntas em uma mesma estação de trabalho para criar código para uma história. Isso fornece um mecanismo para solução de problemas em tempo real (duas cabeças normalmente funcionam melhor do que uma) e garantia da qualidade em tempo real (o código é revisto à medida que é criado).

Conforme a dupla de programadores conclui o trabalho, o código que desenvolveram é integrado ao trabalho de outros. Essa estratégia de "integração contínua" ajuda a evitar problemas de compatibilidade e de interface precocemente.

Testes. Os testes de unidades criados devem ser implementados usando-se uma metodologia que os capacite a ser automatizados (assim, poderão ser executados fácil e repetidamente). Isso estimula uma estratégia de testes de regressão toda vez que o código for modificado (o que é frequente, dada a filosofia de refatoração da XP). Os testes de aceitação da XP, também denominados testes de cliente, são especificados pelo cliente e mantêm o foco nas características e na funcionalidade do sistema total que são visíveis e que podem ser revistas pelo cliente. Os testes de aceitação são obtidos de histórias de usuário implementadas como parte de uma versão do software.

6.2 Kanban

O método Kanban é uma metodologia enxuta que descreve métodos para melhorar qualquer processo ou fluxo de trabalho. O Kanban enfoca a gestão de alterações e a entrega de serviços. A gestão de alterações define o processo por meio do qual uma alteração solicitada é integrada a um sistema baseado em software. A entrega de serviços é incentivada com o foco no entendimento das necessidades e expectativas do cliente. Os membros de equipe gerenciam o trabalho e recebem liberdade para se organizarem de modo a completá-lo. As políticas evoluem quando necessário para melhorar os resultados.

O Kanban nasceu na Toyota, onde era uma série de práticas de engenharia industrial, e foi adaptado ao desenvolvimento de software por David Anderson. O Kanban em si depende de seis práticas fundamentais:

1. Visualizar o fluxo de trabalho utilizando um quadro Kanban. O quadro Kanban é dividido em colunas que representam o estágio de desenvolvimento de cada elemento de funcionalidade do software. Os cartões no quadro podem conter histórias de usuário isoladas ou defeitos recém-descobertos em notas adesivas, que a equipe leva da coluna “por fazer” para a “fazendo” e então para a “feito” à medida que o projeto avança.

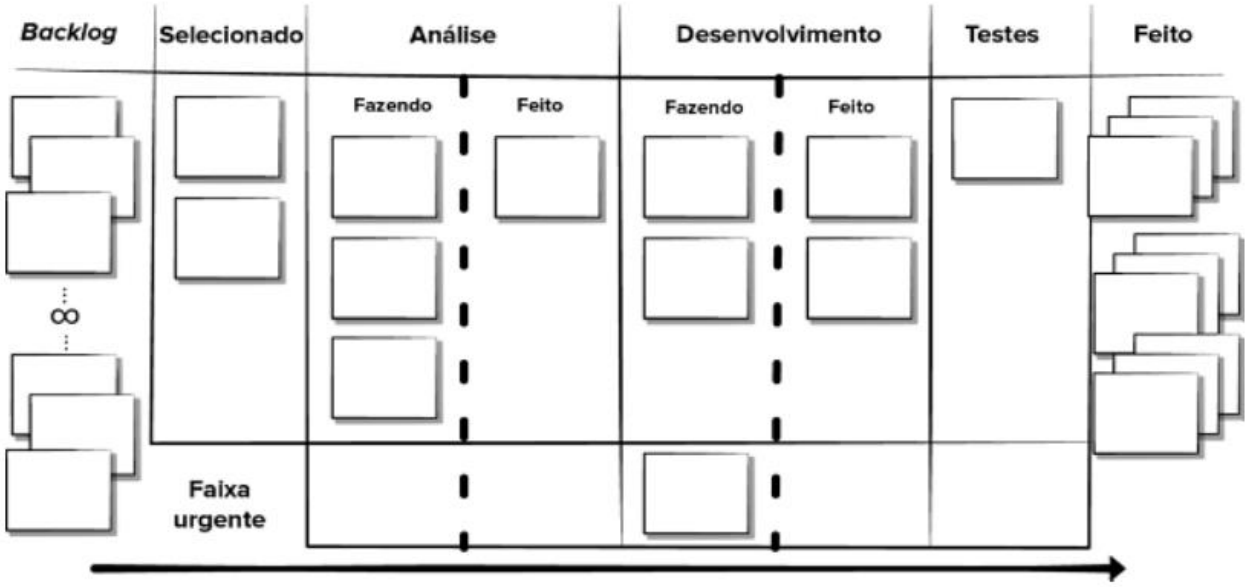
2. Limitar a quantidade de estoque em processo (WIP, do inglês work in progress) em um dado momento. Os desenvolvedores são incentivados a completar a sua tarefa atual antes de iniciarem outra. Isso reduz o tempo de ciclo, melhora a qualidade do trabalho e aumenta a capacidade da equipe de fornecer funcionalidades de software com maior frequência para os envolvidos.

3. Gerenciar o fluxo de trabalho reduz os desperdícios por meio do entendimento do fluxo de valor atual, pela análise dos pontos em que sofre interrupções, pela definição de mudanças e pela implementação subsequente dessas mudanças.

4. Explicitar as políticas de processo (p. ex., anotar os motivos pelos quais um determinado item foi selecionado para ser trabalhado e os critérios usados para se definir “feito”).

5. Enfocar a melhoria contínua com a criação de ciclos de feedback, nos quais as alterações são introduzidas com base em dados de processo, e os efeitos da mudança sobre o processo são medidos após as alterações serem realizadas.

6. Alterar o processo colaborativamente e engajar todos os membros de equipe e outros envolvidos quando necessário.



As reuniões de equipe para o Kanban são semelhantes àquelas realizadas na metodologia Scrum. Se o Kanban está sendo introduzido em um projeto existente, nem todos os itens iniciam na coluna do backlog. Para colocar os seus cartões na coluna do processo da equipe, os desenvolvedores precisam se perguntar: Onde estão agora? De onde vieram? Aonde vão?

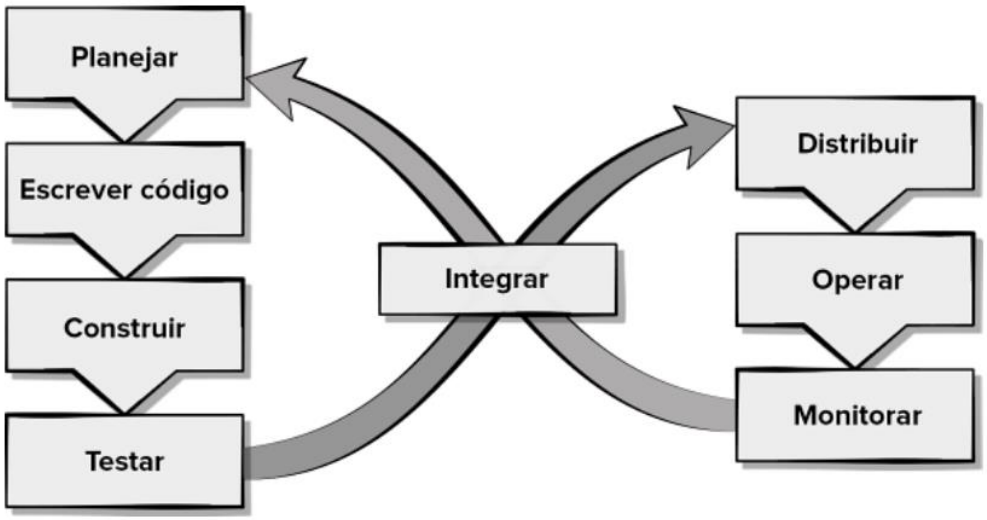
A base para a reunião diária em pé do Kanban (também chamada de standup meeting) é uma tarefa conhecida como “caminhar pelo quadro”. A liderança dessa reunião muda diariamente. Os membros da equipe identificam os itens ausentes do quadro que estão sendo trabalhados e os adicionam ao quadro. A equipe tenta fazer com que todos os itens possíveis avancem para a coluna de “feito”. O objetivo é avançar primeiro os itens com alto valor de negócio. A equipe observa o fluxo e tenta identificar obstáculos à finalização com uma análise da carga de trabalho e dos riscos.

Durante a reunião de retrospectiva semanal, são examinadas medidas de processo. A equipe considera onde podem ser necessárias melhorias de processo e propõe mudanças a serem implementadas. O Kanban pode ser combinado facilmente com outras práticas de desenvolvimento ágil para adicionar um pouco mais de disciplina ao processo.

6.3 DevOps

O DevOps foi criado por Patrick DeBois para combinar Desenvolvimento (Development) e Operações (Operations). O DevOps tenta aplicar os princípios do

desenvolvimento ágil a toda a cadeia logística de software. A figura abaixo apresenta um panorama do fluxo de trabalho do DevOps. A abordagem envolve diversas etapas que formam ciclos contínuos até que o produto desejado exista de fato.



Desenvolvimento contínuo. Os produtos de software são divididos e desenvolvidos em múltiplos sprints, com os incrementos entregues para os membros da equipe de testes e da equipe de desenvolvimento para serem testados.

Teste contínuo. Ferramentas de teste automatizadas são utilizadas para ajudar os membros da equipe a testar múltiplos incrementos de código ao mesmo tempo para garantir que não há defeitos antes da integração.

Integração contínua. Os elementos de código com a nova funcionalidade são adicionados ao código existente e ao ambiente de execução (run-time) e, então, examinados para garantir que não há erros após a entrega.

Entrega contínua. Nesta etapa, o código integrado é entregue (instalado) ao ambiente de produção, que pode incluir múltiplos locais em nível global, que, por sua vez, precisam ser preparados para receber a nova funcionalidade.

Monitoramento contínuo. Os membros da equipe de operações que pertencem à equipe de desenvolvimento ajudam a melhorar a qualidade do software, monitorando o seu desempenho no ambiente de produção e buscando proativamente possíveis problemas antes que os usuários os identifiquem.

O DevOps melhora as experiências dos clientes, pois acelera a reação às mudanças nas suas necessidades ou desejos. Isso pode aumentar a fidelidade de marca e a participação de mercado. Abordagens enxutas como o DevOps podem dar às organizações uma maior capacidade de inovação, pois reduzem o retrabalho e permitem a transição para atividades com maior valor de negócio. Os produtos não geram renda até os consumidores terem acesso a eles, e o DevOps pode reduzir o tempo de entrega para as plataformas de produção.

7. PROCESSO DE DESENVOLVIMENTO DE SOFTWARE E SUAS MELHORES PRÁTICAS

Objetivo

O objetivo deste capítulo é apresentar e detalhar as etapas de desenvolvimento de software com suas melhores práticas.

Introdução

Cada projeto é diferente e cada equipe é diferente. Nenhuma metodologia de engenharia de software é apropriada para todo artefato de software possível. Neste capítulo abordaremos o uso de um processo adaptável que pode ser ajustado para atender às necessidades de desenvolvedores de software que trabalham em produtos dos mais diversos tipos.

7.1 Definição dos requisitos

Todo projeto de software começa com a equipe tentando entender o problema a ser resolvido e determinando quais resultados são importantes para os envolvidos. Isso inclui entender as necessidades de negócios que motivam o projeto e as questões técnicas que o limitam. O processo é chamado de engenharia de requisitos. As equipes que não dedicam uma quantidade de tempo razoável a essa tarefa logo descobrem que o seu projeto contém retrabalho caro, orçamentos estourados, artefatos de baixa qualidade, entregas atrasadas, clientes insatisfeitos e equipes desmotivadas. A engenharia de requisitos não pode ser ignorada e não pode ser deixada em ciclos intermináveis antes que a construção do artefato possa começar.

Seria razoável questionar quais melhores práticas devem ser seguidas para se produzir uma engenharia de requisitos ágil e completa. Scott Ambler sugere diversas melhores práticas para a definição de requisitos ágeis:

1. Corresponda à disponibilidade dos envolvidos e valorize as suas contribuições para encorajar a sua participação ativa.

2. Use modelos simples (p. ex., notas adesivas, rascunhos, histórias de usuário) para reduzir as barreiras à participação.

3. Reserve algum tempo para explicar as suas técnicas de representação de requisitos antes de utilizá-las.

4. Adote a terminologia dos envolvidos e evite o jargão técnico sempre que possível.

5. Use uma abordagem de visão panorâmica (“enxergar a oresta”) para obter uma visão geral do projeto antes de se prender aos detalhes.

6. Permita que a equipe de desenvolvimento refine (com a contribuição dos envolvidos) os detalhes dos requisitos no modelo just-in-time à medida que a implementação das histórias de usuário se aproxima.

7. Trate a lista de características a serem implementadas como uma lista priorizada e implemente as histórias de usuário mais importantes primeiro.

8. Colabore de perto com os seus envolvidos e documente os requisitos apenas no nível que será útil para todos durante a criação do próximo protótipo.

9. Questione a necessidade de manter modelos e documentos que não serão consultados no futuro.

10. Garanta que você tenha o apoio da gerência para garantir a disponibilidade de recursos e dos envolvidos durante a definição dos requisitos.

É importante reconhecer duas realidades: (1) é impossível que os envolvidos descrevam um sistema completo antes de observarem o software operacional; e (2) é difícil que os envolvidos descrevam os requisitos de qualidade necessários para o software antes de vê-lo em ação. Os desenvolvedores precisam reconhecer que os requisitos serão adicionados e refinados à medida que os incrementos de software são criados. Capturar as descrições dos envolvidos sobre o que o sistema precisa fazer em suas próprias palavras, em uma história de usuário, é um bom ponto de partida.

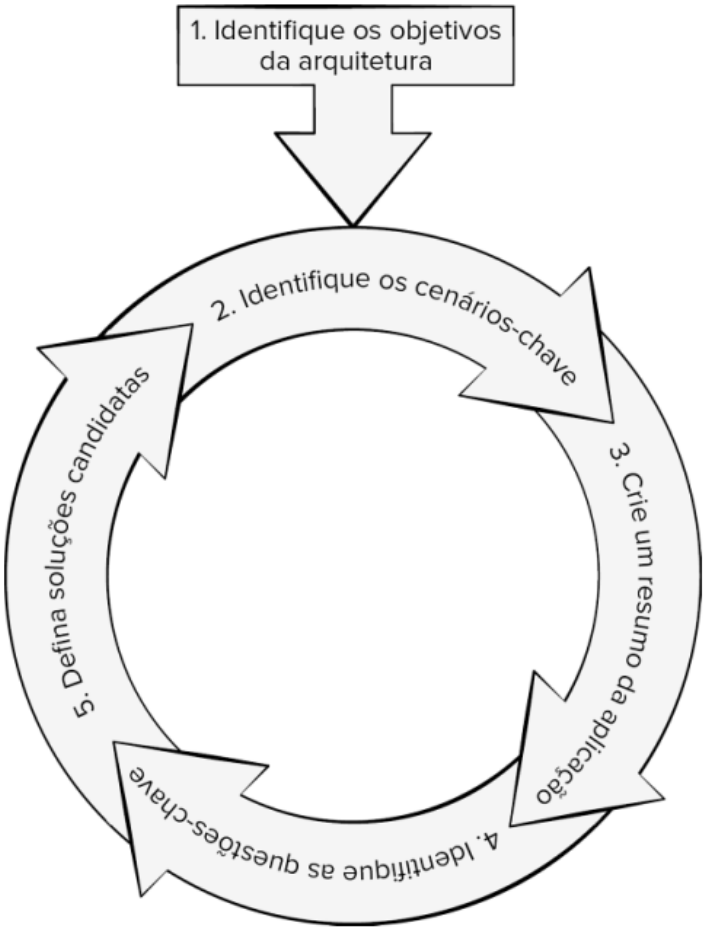
Se conseguir fazer os envolvidos definirem os critérios de aceitação para cada história de usuário, sua equipe começou muito bem. É provável que os envolvidos precisem ver uma história de usuário codificada e em operação para saber se ela foi implementada

corretamente ou não. Assim, a definição de requisitos precisa ser realizada interativamente e incluir o desenvolvimento de protótipos para a revisão por parte dos envolvidos.

Os protótipos são concretizações dos planos do projeto que podem ser consultados facilmente pelos envolvidos quando estes tentam descrever as mudanças desejadas. Os envolvidos são motivados a discutir as mudanças de requisitos em termos mais concretos, o que melhora a comunicação. É importante reconhecer que os protótipos permitem que os desenvolvedores se concentrem nos objetivos de curto prazo ao enfocarem os comportamentos visíveis dos usuários. Será importante revisar os protótipos com foco em qualidade. Os desenvolvedores precisam estar cientes que usar protótipos pode aumentar a volatilidade dos requisitos caso os envolvidos não estejam concentrados em acertar o trabalho na primeira vez. Também há o risco de criar protótipos antes de desenvolver um bom entendimento sobre os requisitos de arquitetura do software, o que pode resultar em protótipos que precisam ser descartados, gerando um desperdício de tempo e de recursos.

7.2 Projeto de arquitetura preliminar

As decisões de projeto preliminares muitas vezes precisam ocorrer quando os requisitos são definidos. Como mostrado na figura abaixo, em algum momento, as decisões sobre arquitetura precisarão ser alocadas a incrementos de produtos. De acordo com Bellomo e seus colegas, um entendimento inicial sobre as escolhas de arquitetura e requisitos é essencial para gerenciar o desenvolvimento de artefatos de software grandes e complexos.



Os requisitos podem ser utilizados para informar o projeto de arquitetura. Explorar a arquitetura à medida que o protótipo é desenvolvido facilita o processo de detalhamento dos requisitos. É melhor conduzir essas atividades simultaneamente para se atingir o equilíbrio apropriado. O projeto de arquitetura ágil possui quatro elementos fundamentais:

1. Enfoque os atributos fundamentais de qualidade e incorpore-os aos protótipos à conforme estes são construídos.

2. Durante o planejamento dos protótipos, mantenha em mente que artefatos de software bem-sucedidos combinam recursos visíveis para os clientes com a infraestrutura que os possibilita.

3. Reconheça que uma arquitetura ágil permite a capacidade do código de ser mantido e de evoluir caso se preste atenção suficiente às decisões sobre arquitetura e questões de qualidade relacionadas.

4. A gestão e a sincronização contínuas de dependências entre os requisitos funcionais e de arquitetura são necessárias para garantir que o alicerce arquitetural em evolução estará pronto a tempo para incrementos futuros.

A tomada de decisões sobre arquitetura de software é essencial para o sucesso do sistema de software. A arquitetura de um sistema de software determina as suas qualidades e impacta o sistema durante todo o seu ciclo de vida. Segundo Dasanayake e colaboradores, os arquitetos de software tendem a cometer erros quando suas decisões são tomadas sob níveis de incerteza. Os arquitetos tomam menos más decisões se utilizam melhor gestão do conhecimento de arquitetura para reduzir essa incerteza. Apesar do fato de as abordagens ágeis desestimularem o excesso de documentação, não registrar decisões de projeto e as suas justificativas desde o início do processo de projeto dificulta o trabalho de revisitá-las durante a criação de protótipos futuros.

Documentar as coisas certas pode ajudar as atividades de melhoria de processo. Documentar as suas lições aprendidas é um dos motivos para se conduzir retrospectivas após avaliar o protótipo entregue e antes de iniciar o próximo incremento do programa. A reutilização de soluções para problemas de arquitetura que foram bem-sucedidas no passado também é útil.

7.3 Estimativa de recursos

Um dos aspectos mais controversos do uso da prototipação ágil ou espiral é estimar o tempo necessário para completar um projeto quando este não pode ser definido completamente. É importante entender, antes de começar e de aceitar o projeto, se há ou não uma probabilidade razoável de entregar artefatos de software a tempo e com custos aceitáveis. As primeiras estimativas correm o risco de estarem incorretas porque o escopo do projeto não está bem definido e tende a mudar após o início do desenvolvimento. As estimativas produzidas quando o projeto está quase terminado não servem para orientar o gerenciamento do projeto. O truque é estimar o tempo de desenvolvimento de software no início, com base no que é conhecido naquele momento, e revisar suas estimativas regularmente à medida que requisitos são adicionados ou após incrementos de software serem entregues.

Vamos analisar como um experiente gerente de projetos de software produziria uma estimativa para um projeto usando o modelo espiral ágil que propusemos. As estimativas produzidas por esse método precisariam ser ajustadas de acordo com o número de desenvolvedores e o número de histórias de usuário que podem ser completadas simultaneamente.

1. Use dados históricos e trabalhe em equipe para desenvolver uma estimativa de quantos dias serão necessários para completar cada uma das histórias de usuário conhecidas no início do projeto.

2. Organize as histórias de usuário informalmente em conjuntos, cada um dos quais comporá um sprint 1 planejado para completar um protótipo.

3. Some o número de dias para completar cada sprint de modo a gerar uma estimativa da duração total do projeto.

4. Revise a estimativa à medida que os requisitos são adicionados ao projeto ou os protótipos são entregues e aceitos pelos envolvidos.

Mantenha em mente que dobrar o número de desenvolvedores quase nunca reduz o tempo de desenvolvimento pela metade.

Rosa e Wallshein descobriram que conhecer os requisitos de software iniciais no começo do projeto gera uma estimativa adequada, mas nem sempre precisa, do tempo de conclusão do projeto. Para obter estimativas mais precisas, também é importante conhecer o tipo de projeto e a experiência da equipe.

7.4 Construção do primeiro protótipo

Os desenvolvedores podem usar o primeiro protótipo para provar que o seu projeto de arquitetura inicial é uma abordagem viável para a entrega da funcionalidade exigida ao mesmo tempo que atende às restrições de desempenho do cliente. Criar um protótipo operacional sugere que a engenharia de requisitos, o projeto de software e a construção procedem todos em paralelo. O processo é mostrado na figura a seguir. Esta seção descreve os passos que serão usados para criar os primeiros protótipos.

Sua primeira tarefa é identificar as características e funções mais importantes para os envolvidos. Estas ajudarão a definir os objetivos para o primeiro protótipo. Se os envolvidos e os desenvolvedores já criaram uma lista priorizada de histórias de usuário, deve ser fácil confirmar quais são as mais importantes.

A seguir, decida quanto tempo deve ser alocado à criação do primeiro protótipo. Algumas equipes escolhem um período fixo – por exemplo, um sprint de 4 semanas – para entregar cada protótipo. Nesse caso, os desenvolvedores analisam a sua estimativa de tempo e recursos e determinam quais histórias de usuário de alta prioridade podem ser concluídas em 4 semanas. A equipe então confirma com os envolvidos que as histórias de usuário selecionadas são as melhores para serem incluídas no primeiro protótipo. Uma abordagem alternativa seria pedir para que os envolvidos e desenvolvedores escolhessem juntos uma pequena quantidade de histórias de usuário de alta prioridade para incluir no primeiro protótipo e usar suas estimativas de tempo e recursos para desenvolver o cronograma até a conclusão do primeiro protótipo.

Os engenheiros que trabalham na National Instruments publicaram um artigo que descreve o seu processo para a criação de um primeiro protótipo funcional. As seguintes etapas podem ser aplicadas a diversos projetos de software:

1. Faça a transição do protótipo de papel para o projeto de software.
2. Crie um protótipo da interface do usuário.
3. Crie um protótipo virtual.
4. Adicione entradas e saídas ao seu protótipo.
5. Desenvolva os seus algoritmos.
6. Teste o seu protótipo.
7. Mantenha a entrega em mente enquanto cria o protótipo.

Consultando esses sete passos, criar um protótipo de papel para um sistema é um processo barato e pode ser feito logo no início do processo de desenvolvimento. Os clientes e os envolvidos não costumam ser desenvolvedores experientes. Usuários não técnicos muitas vezes sabem reconhecer rapidamente o que gostam ou não gostam em uma interface do usuário depois que a veem desenhada. A comunicação entre as pessoas é repleta de mal-entendidos. Elas esquecem de dizer umas às outras o que precisam mesmo saber ou supõem que todos estão em sintonia. Criar um protótipo de papel e revisá-lo com

o cliente antes de começar a programação pode ajudar a poupar o tempo que seria gasto construindo o protótipo errado.

Criar um protótipo da interface do usuário como parte do primeiro protótipo funcional é uma boa ideia. Muitos sistemas são implementados na Web ou na forma de aplicativos móveis e dependem muito de interfaces do usuário de toque. Os jogos de computador e os aplicativos de realidade virtual precisam de bastante comunicação com os usuários para que possam operar corretamente. Se tiverem facilidade para aprender e usar um artefato de software, os clientes terão maior probabilidade de utilizá-lo de fato.

Muitos mal-entendidos entre desenvolvedores e envolvidos podem ser atenuados se começarmos com um protótipo de papel da interface do usuário. Às vezes, os envolvidos precisam enxergar os elementos básicos da interface do usuário em ação para conseguirem explicar o que gostaram e desgostaram de fato. É mais fácil descartar um projeto inicial de interface do usuário do que terminar o protótipo e tentar implementar uma nova interface do usuário por cima dele.

Adicionar entradas e saídas ao seu protótipo da interface do usuário é uma maneira fácil de começar a testar o protótipo em evolução. O teste das interfaces de componentes de software deve ser realizado antes do teste do código que compõe os algoritmos do componente. Para testar os algoritmos em si, os desenvolvedores muitas vezes utilizam um “frame de teste” para garantir que os algoritmos implementados estão funcionando como desejado. Criar um frame de teste separado e descartá-lo quase sempre é um desperdício de recursos. Se bem projetada, a interface do usuário pode servir de frame de teste para os algoritmos do componente, o que elimina o esforço necessário para a construção de frames de teste independentes.

Desenvolver os seus algoritmos se refere ao processo de transformar as suas ideias e os seus rascunhos em código escrito em linguagem de programação. É preciso considerar os requisitos funcionais enunciados na história de usuário e as limitações de desempenho (explícitas e implícitas) ao se projetar os algoritmos necessários. É nesse ponto que a funcionalidade de suporte adicional tende a ser identificada e adicionada ao escopo do projeto, caso já não exista em uma biblioteca de código.

Testar o seu protótipo demonstra a funcionalidade exigida e idêntica de defeitos ainda não descobertos antes da demonstração para o cliente. Às vezes, pode ser uma boa ideia envolver o cliente no processo de teste antes de o protótipo ser analisado para evitar o desenvolvimento da funcionalidade errada. O melhor momento para criar casos de teste é durante a coleta de requisitos ou quando os casos de uso foram selecionados para a implementação.

Manter a entrega em mente enquanto cria o protótipo é muito importante, pois o ajuda a evitar atalhos que levam à criação de software que será difícil de manter no futuro. Isso não significa que cada linha de código será parte do artefato de software nal. Assim como muitas tarefas criativas, o desenvolvimento de um protótipo é iterativo. Rascunhos e revisões fazem parte do processo.

À medida que o desenvolvimento do protótipo ocorre, você deve considerar com cuidado as escolhas que faz sobre arquitetura de software. Mudar algumas linhas de código é relativamente barato quando os erros são identificados antes da entrega. Alterar a arquitetura de um aplicativo de software após o seu lançamento para usuários pode ser muito caro.

7.5 Avaliação do protótipo

O teste é conduzido pelos desenvolvedores enquanto o protótipo é construído e se torna uma parte importante da sua avaliação. O teste demonstra que os componentes do protótipo são operacionais, mas é improvável que os casos de teste identifiquem todos os defeitos. No modelo espiral, os resultados da avaliação permitem que envolvidos e desenvolvedores determinem se continuar o desenvolvimento e criar o próximo protótipo é mesmo desejável. Parte dessa decisão se baseia na satisfação dos usuários e dos envolvidos, e parte é derivada de uma avaliação dos riscos de excesso de custos e de não entregar um artefato operacional no final do projeto. Dam e Siang sugerem diversas dicas de melhores práticas para coletar feedback sobre o seu protótipo.

1. Forneça scaffolding quando solicitar feedback sobre o protótipo.
2. Teste o seu protótipo com as pessoas certas.
3. Faça as perguntas certas.
4. Seja neutro quando apresentar alternativas aos usuários.

5. Adapte durante o teste.

6. Permita que os usuários contribuam com ideias.

Fornecer scaffolding (literalmente, “andaime” ou “estrutura temporária”) é um mecanismo que permite que o usuário dê feedback de uma maneira que não gera confrontos. Muitas vezes, os usuários relutam em dizer aos desenvolvedores que odiaram o produto que estão usando. Para evitar esse problema, costuma ser mais fácil pedir ao usuário que forneça seu feedback usando estruturas do tipo “eu gosto”, eu “gostaria”, “e se” como forma de melhorar a franqueza e honestidade do feedback. Os enunciados do tipo eu gosto incentivam os usuários a oferecer feedback positivo sobre o protótipo. Os enunciados do tipo eu gostaria levam os usuários a compartilhar ideias sobre como melhorar o protótipo e podem gerar feedback negativo e críticas construtivas. Os enunciados e se incentivam os usuários a sugerir ideias para a equipe explorar durante a criação de protótipos em iterações futuras.

Recrutar as pessoas certas para avaliar o protótipo é essencial para reduzir o risco de desenvolver o artefato errado. Pedir que os membros da equipe de desenvolvimento realizem todos os testes é uma má ideia, pois eles provavelmente não serão representativos da população de usuários pretendida. É importante obter a combinação certa de usuários (p. ex., novatos, típicos e avançados) para que estes forneçam feedback sobre o protótipo.

Fazer as perguntas certas sugere que todos os envolvidos estão de acordo em relação aos objetivos do protótipo. Enquanto desenvolvedor, é importante manter a mente aberta e fazer o possível para convencer os usuários que o feedback deles é valioso. O feedback é o motor do processo de prototipação enquanto você planeja atividades futuras de desenvolvimento de produtos. Além do feedback geral, tente fazer perguntas específicas sobre os novos recursos incluídos no protótipo.

Ser neutro quando apresentar alternativas permite que a equipe de software evite o problema de causar a impressão de estar tentando convencer os usuários a fazer alguma coisa de uma determinada maneira. Se quiser feedback honesto, informe aos usuários que você não decidiu que só existe uma maneira de fazer as coisas. A programação sem ego (egoless programming) é uma filosofia de desenvolvimento focada na produção do melhor artefato que a equipe consegue criar para os usuários pretendidos. Criar protótipos

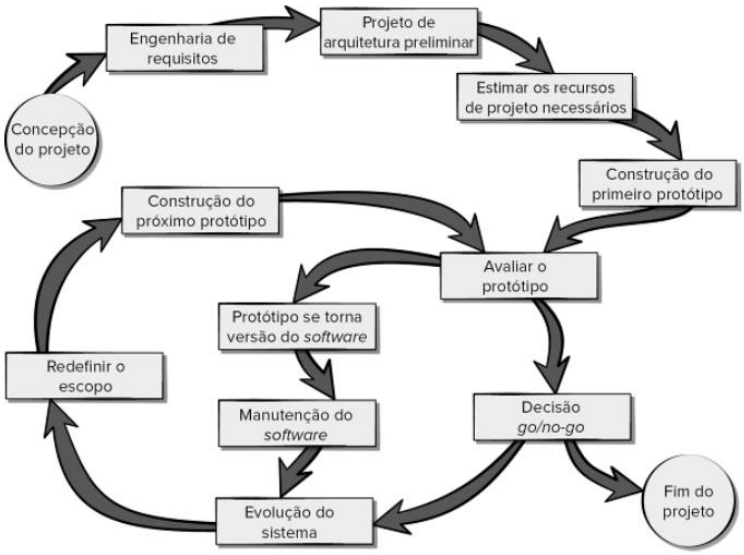
descartáveis não é algo desejável, mas a programação sem ego sugere que aquilo que não está funcionando precisa ser consertado ou descartado. Assim, tente não se prender demais às suas ideias enquanto cria os primeiros protótipos.

Adaptar durante o teste significa que é preciso ter uma mentalidade flexível enquanto os usuários trabalham com o protótipo. Isso pode significar alterar o seu plano de teste ou fazer mudanças rápidas no protótipo e então reiniciar o teste. O objetivo é obter o melhor feedback possível dos usuários, incluindo observá-los diretamente enquanto interagem com o protótipo. O importante é que você obtenha o feedback de que precisa para ajudá-los a decidir se construirão o próximo protótipo ou não.

Permitir que os usuários contribuam com ideias significa exatamente o que diz. Garanta que tem uma maneira de registrar as suas sugestões e perguntas (eletronicamente ou não).

7.6 Decisão go/no-go

Após o protótipo ser avaliado, os envolvidos do projeto decidem se devem continuar ou não o desenvolvimento do artefato de software. Observando a figura abaixo, vemos que uma decisão ligeiramente diferente baseada na avaliação do protótipo seria disponibilizá-lo para os usuários e começar o processo de manutenção. A primeira volta em torno da espiral poderia ser usada para solidificar os requisitos do projeto. Mas, na verdade, precisamos de mais. No método que estamos propondo aqui, cada volta em torno da espiral desenvolve um incremento significativo do artefato de software final. Você pode trabalhar na história de usuário do projeto ou backlog de recursos para identificar um subconjunto importante do artefato final a ser incluído no primeiro protótipo e repetir esse ciclo para cada protótipo subsequente.



Após o processo de avaliação, temos uma passagem pela região de planejamento. São propostas estimativas de custo revisadas e alterações ao cronograma com base no que foi descoberto durante a avaliação do protótipo de software atual. Isso pode envolver a adição de novas histórias de usuário ou características ao backlog do projeto à medida que o protótipo é avaliado. O risco de exceder o orçamento e não atingir o prazo de entrega do projeto é avaliado por meio da comparação das novas estimativas de custo e tempo com as antigas. O risco de não atender às expectativas do usuário também é considerado e discutido com os envolvidos, e até com a alta gerência, em alguns casos, antes de tomarmos a decisão de criar outro protótipo.

O objetivo do processo de avaliação de riscos é obter o compromisso de todos os envolvidos e da gerência da empresa de fornecer os recursos necessários para criar o próximo protótipo. Se o compromisso não se materializar porque o risco de falha do projeto é grande demais, o projeto pode ser encerrado. Na metodologia Scrum, a decisão go/no-go (“prosseguir/não prosseguir”) poderia ser tomada durante a reunião de retrospectiva do Scrum, realizada entre a demonstração do protótipo e a reunião de planejamento do novo sprint. Em todos os casos, a equipe de desenvolvimento apresenta o seu caso para os product owners e deixa-os decidir se devem ou não continuar com o desenvolvimento do artefato.

7.7 Evolução do protótipo

Após o protótipo ter sido desenvolvido e revisado pela equipe de desenvolvimento e por outros envolvidos, é o momento de considerar o desenvolvimento do próximo protótipo. O primeiro passo é coletar todo o feedback e os dados da avaliação do protótipo atual. Então, os desenvolvedores e envolvidos começam as negociações para planejar a criação de mais um protótipo. Após chegarem a um acordo sobre as características do novo protótipo, consideram-se as restrições temporais e orçamentárias conhecidas, além da viabilidade técnica de se implementar o protótipo. Se os riscos de desenvolvimento são considerados aceitáveis, o trabalho continua.

O modelo de processo evolucionário de prototipação é usado para integrar as alterações que sempre ocorrem durante o desenvolvimento do software. Cada protótipo deve ser projetado de modo a permitir alterações futuras e evitar que ele precise ser jogado fora, com o próximo protótipo começando da estaca zero. Isso sugere favorecer características importantes e compreendidas quando se define as metas para cada protótipo. Como sempre, as necessidades do cliente devem receber a devida importância nesse processo.

7.7.1 Escopo do novo protótipo

O processo de determinar o escopo de um novo protótipo é semelhante ao processo de determinar o escopo do protótipo inicial. Os desenvolvedores: (1) selecionam as características a serem desenvolvidas dentro do tempo alocado ao sprint ou (2) alocam tempo suficiente para implementar as características necessárias para cumprir os objetivos estabelecidos pelos desenvolvedores com a colaboração dos envolvidos. Ambas as abordagens exigem que os desenvolvedores mantenham uma lista priorizada de características ou histórias de usuário. As prioridades usadas para ordenar a lista devem ser determinadas pelos objetivos definidos para o protótipo pelos envolvidos e pelos desenvolvedores.

Na Extreme Programming (XP), os envolvidos e os desenvolvedores trabalham juntos para agrupar as histórias de usuário mais importantes em um protótipo que se tornará a próxima versão do software e determinam a sua data de conclusão. No Kanban, os desenvolvedores e os envolvidos utilizam um quadro que lhes permite enfocar o status de conclusão de cada história de usuário. É uma referência visual que pode ser utilizada para

auxiliar os desenvolvedores a usar qualquer modelo de processo de protótipo incremental para planejar e monitorar o progresso do desenvolvimento de software. Os envolvidos enxergam facilmente o backlog de características e ajudam os desenvolvedores a ordená-lo para identificar as histórias mais úteis necessárias para o próximo protótipo. Provavelmente é mais fácil estimar o tempo necessário para completar as histórias de usuário selecionadas do que encontrar histórias de usuário que se encaixam em um bloco de tempo xo. Contudo, o conselho de limitar o tempo de desenvolvimento do protótipo a 4 a 6 semanas deve ser seguido para garantir o engajamento e o feedback adequado por parte dos envolvidos.

7.7.2 Construção de novos protótipos

Uma história de usuário deve conter uma descrição de como o cliente planeja interagir com o sistema para atingir uma meta específica e uma descrição de qual é a definição de aceitação do cliente. A tarefa da equipe de desenvolvimento é criar componentes de software adicionais para implementar as histórias de usuário selecionadas para inclusão no novo protótipo, junto com os casos de teste necessários. Os desenvolvedores precisam continuar a comunicação com todos os envolvidos enquanto criam o novo protótipo.

O que torna esse novo protótipo mais complicado de construir é que os componentes de software criados para implementar novas características no protótipo em evolução precisam trabalhar com os componentes usados para implementar as características incluídas no protótipo anterior. O trabalho se torna ainda mais complexo se os desenvolvedores precisam remover componentes ou modificar aqueles que foram incluídos no protótipo anterior devido a alterações nos requisitos.

É importante que os desenvolvedores tomem decisões de projeto que tornarão o protótipo de software mais facilmente extensível no futuro. Os desenvolvedores precisam documentar as decisões de projeto de forma que facilitem o entendimento sobre o software durante a produção do protótipo seguinte. O objetivo é ser ágil no desenvolvimento e na documentação. Os desenvolvedores precisam resistir à tentação de exagerar no projeto de software para acomodar características que podem ou não ser incluídas no produto final. Eles também devem limitar a sua documentação àquilo que precisarão consultar durante o desenvolvimento ou quando alterações precisarem ser realizadas no futuro.

7.7.3 Teste dos novos protótipos

Se a equipe de desenvolvimento criou casos de teste durante o processo de projeto, antes da programação ser concluída, o teste do novo protótipo deve ser relativamente simples e direto. Cada história de usuário deve ter critérios de aceitação associados no momento da sua criação. Esses enunciados de aceitação devem guiar a criação dos casos de teste que ajudarão a verificar se o protótipo atende às necessidades do cliente. O protótipo também precisará ser testado em busca de defeitos e problemas de desempenho.

Uma preocupação adicional em relação a testes para protótipos evolucionários é garantir que a inclusão de novas características não estragará acidentalmente outras que funcionavam corretamente no protótipo anterior. O teste de regressão é o processo de verificar que o software desenvolvido e testado anteriormente ainda funciona da mesma maneira após ser alterado. É importante usar o seu tempo de teste com inteligência e sabedoria e utilizar casos de teste projetados para detectar defeitos nos componentes com maior probabilidade de serem afetados pelas novas características.

7.8 Disponibilização do protótipo

Quando um processo de prototipação evolucionário é aplicado, os desenvolvedores podem ter dificuldade em saber quando o artefato está acabado e pode ser disponibilizado para os clientes. Os desenvolvedores não querem lançar software cheio de bugs para os usuários, que então decidiriam que o software é de má qualidade. Um protótipo considerado candidato a lançamento deve ser submetido a testes de aceitação do usuário além dos testes funcionais e não funcionais (de desempenho) que seriam conduzidos durante a construção do protótipo.

Os testes de aceitação do usuário se baseiam nos critérios de aceitação concordados e registrados quando cada história de usuário foi criada e adicionada ao backlog do produto. Eles permitem que os representantes do usuário confirmem que o software se comporta como esperado e coletem sugestões para melhorias futuras. David Nielsen oferece diversas sugestões sobre como conduzir testes de protótipos em contextos industriais.

Durante o teste de uma versão candidata, os testes funcionais e não funcionais devem ser repetidos usando os casos de teste desenvolvidos durante as etapas de

construção dos protótipos incrementais. Testes não funcionais adicionais devem ser criados para confirmar que o desempenho do protótipo é consistente com os benchmarks escolhidos por consenso para o produto final. Os benchmarks de desempenho típicos podem envolver o tempo de resposta do sistema, a capacidade de dados ou a usabilidade. Um dos requisitos não funcionais mais importantes é verificar se a versão candidata rodará em todos os ambientes de execução planejados e em todos os dispositivos-alvo. O processo deve se concentrar em testes limitados aos critérios de aceitação estabelecidos antes de o protótipo ser criado. Os testes não têm como provar que um software não tem bug, apenas que os casos de teste foram executados corretamente.

O feedback dos usuários durante o teste de aceitação deve ser organizado por funções visíveis ao usuário, como representado pela interface do usuário. Os desenvolvedores devem examinar o dispositivo em questão e realizar mudanças à tela da interface do usuário se essas alterações não atrasarem o lançamento do protótipo. Se forem realizadas alterações, estas precisam ser confirmadas por uma segunda rodada de testes antes de avançarmos para a próxima etapa. Não planeje mais de duas iterações de teste de aceitação do usuário.

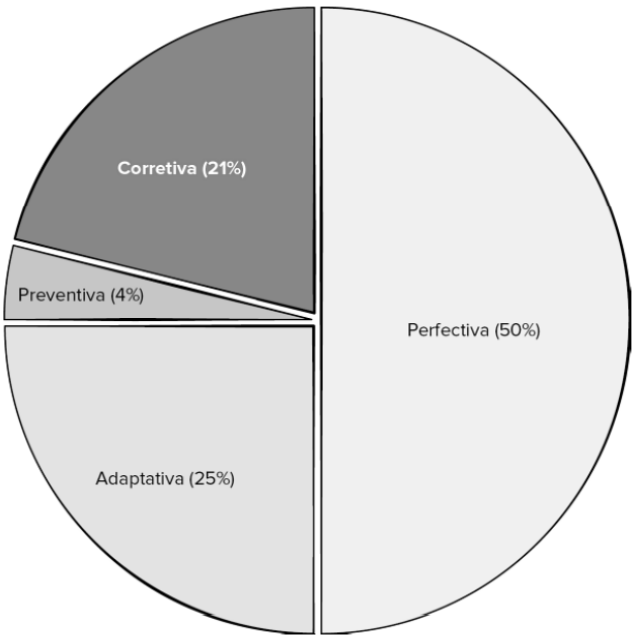
É importante, mesmo para projetos que usam modelos de processo ágil, usar um sistema de controle de falhas ou de bugs (p. ex., Bugzilla ou Jira) para capturar os resultados dos testes. Isso permite que os desenvolvedores registrem as falhas dos testes e facilita a identificação dos casos de teste que precisarão ser rodados novamente para confirmar que o reparo corrige adequadamente o problema descoberto. Em cada caso, os desenvolvedores precisam avaliar se as alterações podem ser implementadas sem estourar o orçamento ou atrasar a entrega. As consequências de não consertar um problema precisam ser documentadas e compartilhadas com o cliente e a alta gerência, que podem, por sua vez, decidir cancelar o projeto em vez de comprometer os recursos necessários para entregar o projeto final.

Os problemas e lições aprendidos com a criação do candidato a lançamento devem ser documentados e considerados pelos desenvolvedores e pelos envolvidos durante a avaliação postmortem do projeto. Essas informações devem ser consideradas antes de decidirmos investir no desenvolvimento futuro de um artefato de software após a sua disponibilização para a comunidade de usuários. As lições aprendidas com o artefato atual

podem ajudar os desenvolvedores a produzir estimativas de custo e de tempo melhores para projetos semelhantes no futuro.

7.9 Manutenção do software

A manutenção é definida como as atividades necessárias para manter o software operacional após ele ser aceito e entregue (lançado) no ambiente do usuário. A manutenção contínua durante toda a vida do artefato de software. Alguns engenheiros de software acreditam que a maior parte do dinheiro gasto em um artefato será nas atividades de manutenção. A manutenção corretiva é a modificação reativa do software para consertar problemas descobertos após o software ter sido entregue ao consumidor final. A manutenção adaptativa é a modificação reativa do software após a entrega para mantê-lo utilizável em um ambiente do usuário em mutação. A manutenção perfectiva é a modificação proativa do software após a entrega para adicionar novos recursos para o usuário, melhor estrutura do código do programa ou melhor documentação. A manutenção preventiva é a modificação proativa do software após a entrega para detectar e corrigir falhas do artefato antes que sejam descobertas pelos usuários no campo. A manutenção proativa pode ser programada e planejada. A manutenção preventiva costuma ser descrita como apagar incêndios, pois não pode ser planejada e precisa ser executada imediatamente para sistemas de software que são críticos para o sucesso das atividades dos usuários. A figura abaixo mostra que apenas 21% do tempo dos desenvolvedores costuma ser dedicado à manutenção corretiva.



Para um modelo de processo evolucionário ágil, os desenvolvedores disponibilizam soluções parciais com a criação de cada protótipo incremental. Boa parte do trabalho de engenharia realizado é manutenção preventiva ou perfectiva à medida que novas características são agregadas ao sistema de software em evolução. É tentador imaginar que a manutenção se resume a planejar outra volta em torno da espiral. Contudo, problemas de software nem sempre podem ser previstos, então pode ser necessário realizar consertos rapidamente, e os desenvolvedores podem ficar tentados a buscar atalhos quando tentam reparar o software estragado. Os desenvolvedores podem não querer dedicar tempo à avaliação de riscos ou ao planejamento. Contudo, eles não podem alterar o software sem considerar a possibilidade de que as alterações necessárias para consertar um problema causarão problemas a outras partes do programa.

É importante compreender o código do programa antes de alterá-lo. Se os desenvolvedores documentaram o código, ele se torna mais compreensível caso outras pessoas precisem realizar trabalho de manutenção. Se o software é projetado para ser estendido, a manutenção também se torna mais fácil de realizar, além do conserto emergencial de defeitos. Testar o software modificado com cuidado é essencial para garantir que as alterações produziram o efeito pretendido e não causaram estragos em outras partes do software.

A tarefa de criar artefatos de software que facilitam o suporte e a manutenção exige uma engenharia cuidadosa e consciente.

8. MODELAGEM DE SOFTWARE

Objetivo

O objetivo deste capítulo é apresentar e detalhar a UML (Unified Modeling Language) e seus principais diagramas de modelagem. Além de apresentar as principais categorias de software.

Introdução

A linguagem de modelagem unificada (UML, do inglês unified modeling language) é “uma linguagem-padrão para descrever/documentar projeto de software. A UML pode ser usada para visualizar, especificar, construir e documentar os artefatos de um sistema de software intensivo”. Em outras palavras, assim como os arquitetos criam plantas e projetos para serem usados por uma empresa de construção, os arquitetos de software criam diagramas UML para ajudar os desenvolvedores de software a construir o software. Se você entender o vocabulário da UML (os elementos visuais do diagrama e seus significados), poderá facilmente entender e especificar um sistema e explicar o projeto desse sistema para outros interessados.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desenvolveram a UML na década de 1990, com muitas opiniões da comunidade de desenvolvimento de software. A UML combinou um grupo de notações de modelagem concorrentes usadas pela indústria do software na época. Em 1997, a UML 1.0 foi apresentada ao OMG (Object Management Group), uma associação sem fins lucrativos dedicada a manter especificações para serem usadas pela indústria de computadores. A UML 1.0 foi revisada, tornando-se a UML 1.1 e adotada mais tarde naquele ano. O padrão atual é a UML 2.5.12 e agora é um padrão ISO. Como esse padrão é novo, há muitas referências mais antigas.

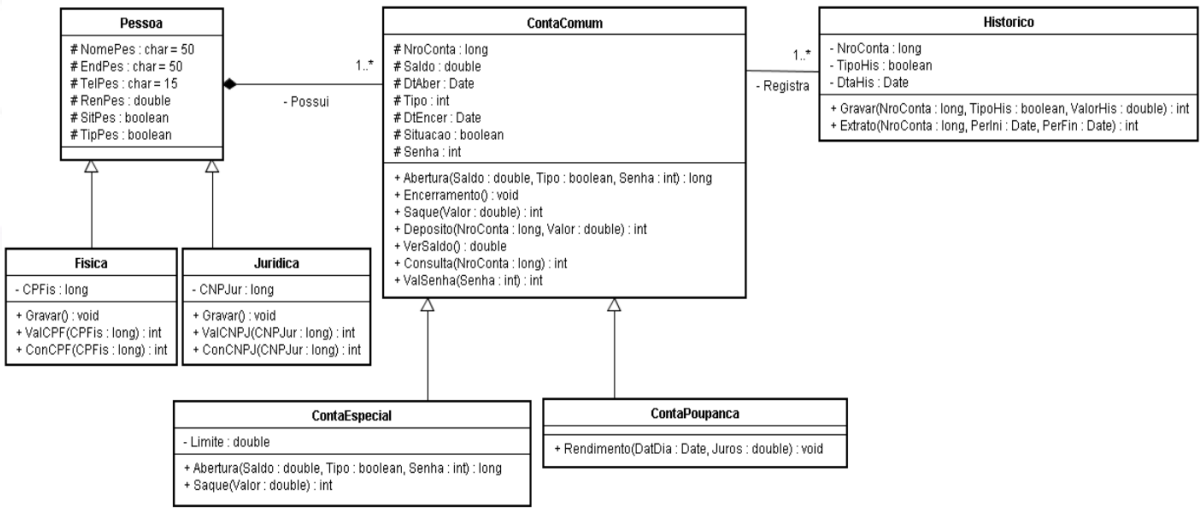
A UML 2.5.1 fornece mais de 10 diagramas diferentes para uso na modelagem de software, são eles:

Você notará que há muitas características opcionais em diagramas UML. A linguagem UML proporciona essas opções (às vezes ocultas) para que você possa expressar todos os aspectos importantes de um sistema. Ao mesmo tempo, é possível suprimir partes não relevantes ao aspecto que está sendo modelado para não congestionar o diagrama com detalhes irrelevantes. Portanto, a omissão de uma característica não significa que ela esteja ausente, mas sim que ela foi suprimida.

A seguir veremos os principais diagramas UML e suas características:

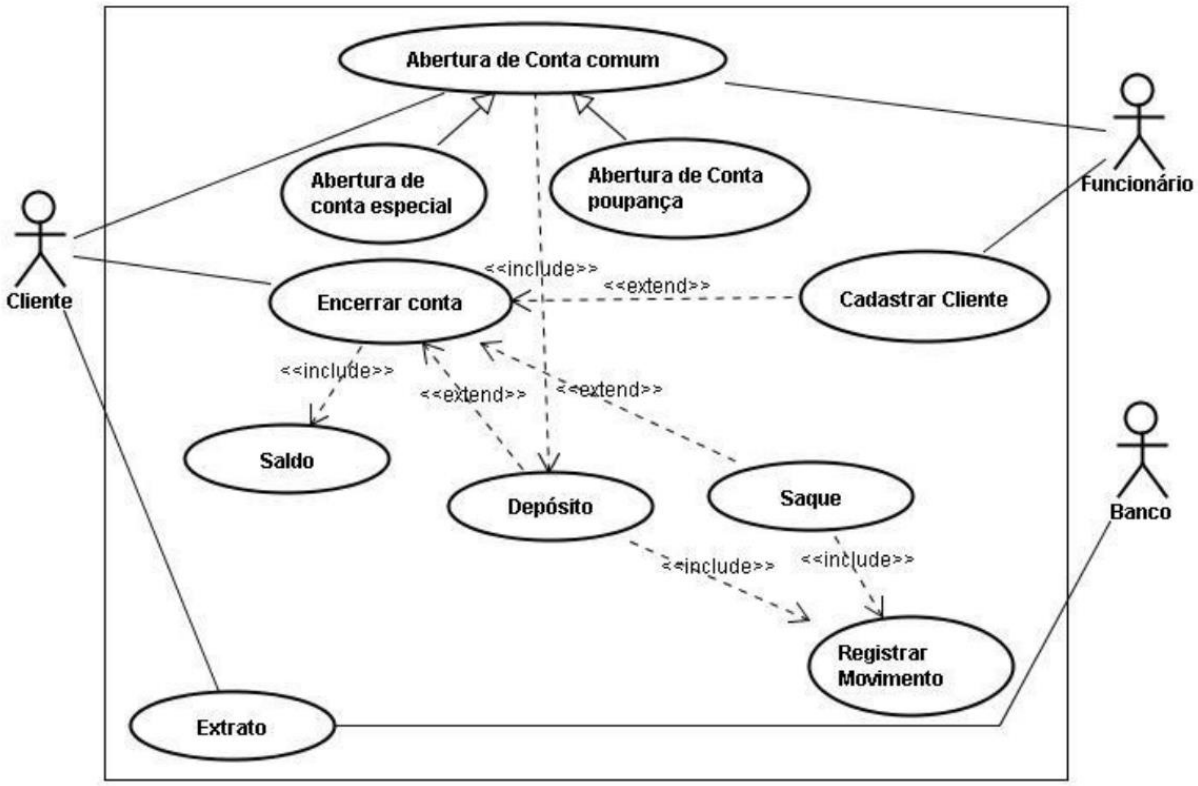
8.1. Diagrama de Classe

- Diagrama mais utilizado da UML
- Serve de apoio para a maioria dos outros diagramas
- Define a estrutura das classes do sistema, apresentando nome da classe, atributos e métodos
- Estabelece como as classes se relacionam



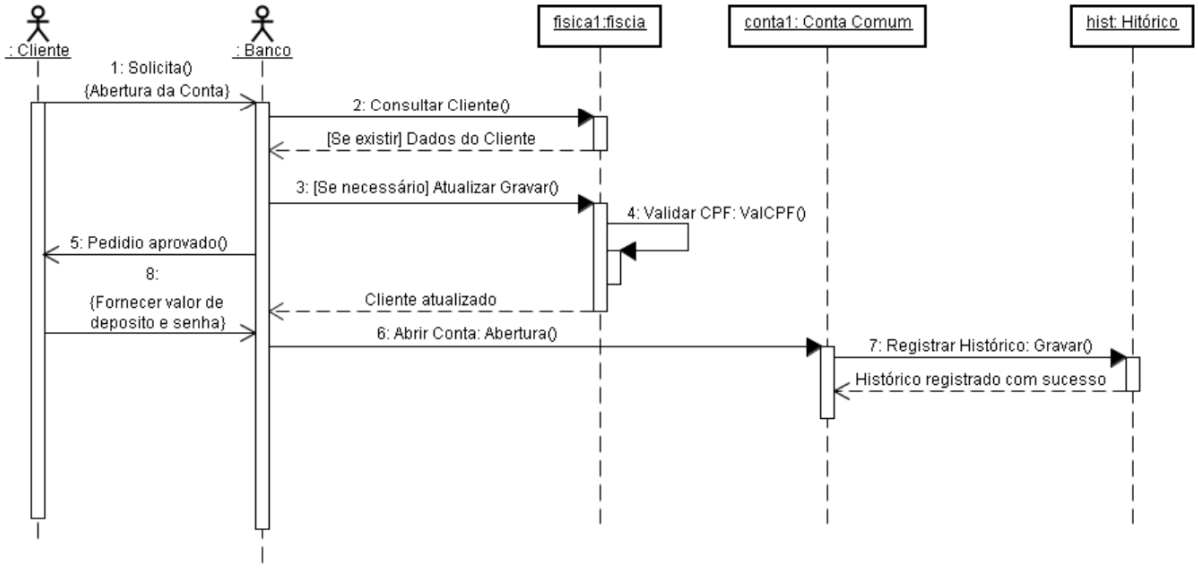
8.2. Diagrama de Caso de Uso

- Diagrama mais geral da UML
- Usado geralmente na fase de Especificação de Requisitos
- Apresenta quais usuários realizam determinadas funcionalidades do sistema
- Apresenta alguns relacionamentos entre estas funcionalidades



8.3. Diagrama de Sequência

- Preocupa-se com a ordem temporal em que as mensagens/ações são trocadas
- Pode se basear em um Caso de Uso
- Identifica ss eventos associados a funcionalidade modelada
- Identifica o ator responsável por este evento



8.4. Categorias de Software

Atualmente, sete grandes categorias de software apresentam desafios contínuos para os engenheiros de software:

Software de sistema: Conjunto de programas feito para atender a outros programas. Certos softwares de sistema (p. ex., compiladores, editores e utilitários para gerenciamento de arquivos) processam estruturas de informação complexas, mas determinadas. Outras aplicações de sistema (p. ex., componentes de sistema operacional, drivers, software de rede, processadores de telecomunicações) processam dados amplamente indeterminados.

Software de aplicação: Programas independentes que solucionam uma necessidade específica de negócio. Aplicações nessa área processam dados comerciais ou técnicos de uma forma que facilite operações comerciais ou tomadas de decisão administrativas/técnicas.

Software de engenharia/científico: Uma ampla variedade de programas de "cálculo em massa" que abrangem astronomia, vulcanologia, análise de estresse automotivo, dinâmica orbital, projeto auxiliado por computador, hábitos de consumo, análise genética e meteorologia, entre outros.

Software embarcado: Residente num produto ou sistema e utilizado para implementar e controlar características e funções para o usuário e para o próprio sistema. Executa funções limitadas e específicas (p. ex., controle do painel de um forno micro-ondas) ou fornece função significativa e capacidade de controle (p. ex., funções digitais de automóveis, como controle do nível de combustível, painéis de controle e sistemas de freio).

Software para linha de produtos: Composto por componentes reutilizáveis e projetado para prover capacidades específicas de utilização por muitos clientes diferentes. Software para linha de produtos pode se concentrar em um mercado hermético e limitado (p. ex., produtos de controle de inventário) ou lidar com consumidor de massa.

Aplicações Web/aplicativos móveis: Esta categoria de software voltada às redes abrange uma ampla variedade de aplicações, contemplando aplicativos voltados para navegadores, computação em nuvem, computação baseada em serviços e software residente em dispositivos móveis.

Software de inteligência artificial: Faz uso de heurísticas para solucionar problemas complexos que não são passíveis de computação ou de análise direta. Aplicações nessa área incluem: robótica, sistemas de tomada de decisão, reconhecimento de padrões (de imagem e de voz), aprendizado de máquina, prova de teoremas e jogos.

Milhões de engenheiros de software em todo o mundo trabalham arduamente em projetos de software em uma ou mais dessas categorias. Em alguns casos, novos sistemas estão sendo construídos, mas, em muitos outros, aplicações já existentes estão sendo corrigidas, adaptadas e aperfeiçoadas. Não é incomum um jovem engenheiro de software trabalhar em um programa mais velho do que ele! Gerações passadas de pessoal de software deixaram um legado em cada uma das categorias discutidas. Espera-se que o legado a ser deixado por esta geração facilite o trabalho dos futuros engenheiros de software.

8.4.1. Software legado

Centenas de milhares de programas de computador caem em um dos sete amplos domínios de aplicação apresentados acima. Alguns deles são software de ponta. Outros programas são mais antigos – em alguns casos, muito mais antigos. Esses programas mais antigos – frequentemente denominados software legado – têm sido foco de contínua atenção e preocupação desde os anos 1960. Dayani-Fard e seus colegas descrevem software legado da seguinte maneira:

Sistemas de software legado foram desenvolvidos décadas atrás e têm sido continuamente modificados para se adequar às mudanças dos requisitos de negócio e a plataformas computacionais. A proliferação de tais sistemas está causando dores de cabeça para grandes organizações que os consideram dispendiosos de manter e arriscados de evoluir.

Essas mudanças podem criar um efeito colateral extra muito presente em software legado – a baixa qualidade. 6 Às vezes, os sistemas legados têm projetos inextensíveis, código de difícil entendimento, documentação deciente ou inexistente, casos de teste e resultados que nunca foram documentados, um histórico de alterações mal gerenciado – a lista pode ser bastante longa. Ainda assim, esses sistemas dão suporte a “funções vitais de negócio e são indispensáveis para ele”. O que fazer? A única resposta adequada talvez seja: não faça nada, pelo menos até que o sistema legado tenha que passar por alguma modificação significativa. Se o software legado atende às necessidades de seus usuários e funciona de forma confiável, ele não está “quebrado” e não precisa ser “consertado”. Entretanto, com o passar do tempo, esses sistemas evoluem devido a uma ou mais das razões a seguir:

- O software deve ser adaptado para atender às necessidades de novos ambientes ou de novas tecnologias computacionais.
- O software deve ser aperfeiçoado para implementar novos requisitos de negócio.
- O software deve ser expandido para torná-lo capaz de funcionar com outros bancos de dados ou com sistemas mais modernos.
- O software deve ter a sua arquitetura alterada para torná-lo viável dentro de um ambiente computacional em evolução.

Quando essas modalidades de evolução ocorrem, um sistema legado deve passar por reengenharia para que permaneça viável no futuro. O objetivo da engenharia de software moderna é “elaborar metodologias baseadas na noção de evolução; isto é, na noção de que os sistemas de software modificam-se continuamente, novos sistemas são construídos a partir dos antigos e… todos devem interagir e cooperar uns com os outros”.

9. QUALIDADE DE SOFTWARE E SEUS MODELOS

Objetivo

O objetivo deste capítulo é apresentar os conceitos de qualidade de software e os seus principais pilares.

Introdução

Até mesmo os desenvolvedores de software mais experientes concordarão que software de alta qualidade é um objetivo importante. Mas como definir a qualidade de software? No sentido mais geral, a qualidade de software pode ser definida como: uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam.

A definição pode ser enfatizada em três pontos importantes:

1. Uma gestão de qualidade efetiva estabelece a infraestrutura que dá suporte a qualquer tentativa de construir um produto de software de alta qualidade. Os aspectos administrativos do processo criam mecanismos de controle e equilíbrio de poderes que ajudam a evitar o caos no projeto – um fator-chave para uma qualidade inadequada. As práticas de engenharia de software permitem ao desenvolvedor analisar o problema e elaborar uma solução consistente – aspectos críticos na construção de software de alta qualidade. Finalmente, as atividades de apoio, como o gerenciamento de mudanças e as revisões técnicas, têm muito a ver com a qualidade, assim como qualquer outra parte da prática de engenharia de software.

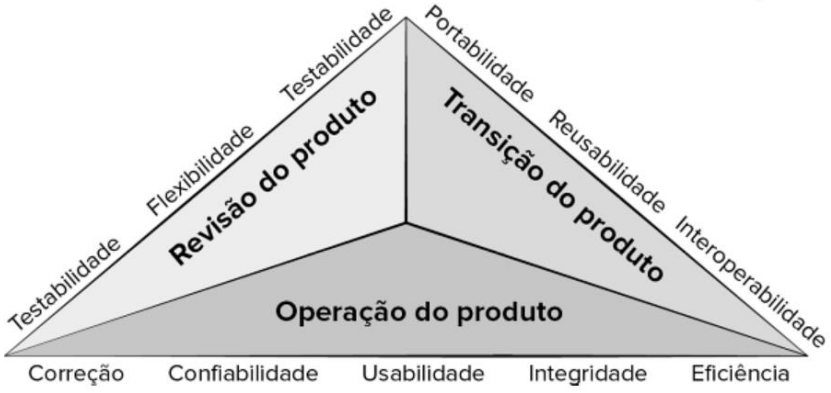
2. Um produto útil fornece o conteúdo, as funções e os recursos que o usuário deseja; além disso, e não menos importante, deve fornecer contabilidade e isenção de erros. Um produto útil sempre satisfaz às exigências definidas explicitamente pelos envolvidos. Além disso, ele satisfaz a um conjunto de requisitos implícitos (p. ex., facilidade de uso) que se espera de todo software de alta qualidade.

3. Ao agregar valor tanto para o fabricante quanto para o usuário de um produto de software, um software de alta qualidade gera benefícios para a empresa de software e para a comunidade de usuários. A empresa fabricante do software ganha valor agregado pelo fato de um software de alta qualidade exigir menos manutenção,

menos correções de erros e menos suporte ao cliente. Isso permite que os engenheiros de software despendam mais tempo criando aplicações novas e menos tempo em manutenções. A comunidade de usuários ganha um valor agregado, pois a aplicação fornece a capacidade de agilizar algum processo de negócio. O resultado final é: (1) maior receita gerada pelo produto de software; (2) maior rentabilidade quando uma aplicação suporta um processo de negócio; e/ou (3) maior disponibilidade de informações cruciais para o negócio.

9.1 Fatores de qualidade

A literatura sobre engenharia de software contém propostas de uma série de normas e modelos de qualidade de software. David Garvin escreve que a qualidade é um fenômeno multifacetado e que exige o uso de múltiplas perspectivas para ser avaliada. McCall e Walters propuseram uma forma útil de refletir sobre e organizar os fatores que afetam a qualidade de software. Seus fatores de qualidade de software se concentram em três aspectos dos produtos de software: características operacionais, capacidade de serem alterados e adaptabilidade a novos ambientes. Os fatores de qualidade de McCall servem de base para uma engenharia de software que produz altos níveis de satisfação do usuário por se concentrar na experiência do usuário geral propiciada pelo produto de software. Isso seria impossível se os desenvolvedores não garantissem que a especificação dos requisitos está correta e que os defeitos foram removidos no início do processo de desenvolvimento de software.



O modelo de qualidade ISO 25010 é a mais nova norma (criada em 2011 e revisada em 2017). Esta norma define dois modelos de qualidade. O modelo da qualidade em uso descreve cinco características apropriadas quando consideramos utilizar o produto em um determinado contexto (p. ex., o uso do produto em uma plataforma específica por um ser

humano). O modelo da qualidade de produto descreve oito características que enfocam a natureza dinâmica e a estática dos sistemas de computador.

Modelo da qualidade em uso

- **Eficácia.** Precisão e completude com as quais os usuários atingem suas metas.
- **Eficiência.** Recursos despendidos para atingir completamente os objetivos dos usuários com a precisão desejada.
- **Satisfação.** Utilidade, confiança, prazer, conforto.
- **Ausência de riscos.** Mitigação de riscos econômicos, de saúde, de segurança e ambientais.
- **Cobertura do contexto.** Completude, exigibilidade.

Qualidade de produto

- **Adequação funcional.** Completo, correto, apropriado.
- **Eficiência de desempenho.** Tempestividade, utilização de recursos, capacidade.
- **Compatibilidade.** Coexistência, interoperabilidade.
- **Usabilidade.** Adequabilidade, facilidade de aprendizagem, operabilidade, proteção contra erros, estética, acessibilidade.
- **Confiabilidade.** Maturidade, disponibilidade, tolerância a falhas, facilidade de recuperação.
- **Segurança.** Confidencialidade, integridade, responsabilidade, autenticidade.
- **Facilidade de manutenção.** Modularidade, reusabilidade, modificabilidade, testabilidade.
- **Portabilidade.** Adaptabilidade, instalabilidade, facilidade de substituição.

A adição do modelo de qualidade em uso ajuda a enfatizar a importância da satisfação do cliente na avaliação da qualidade do software. O modelo de qualidade de produto destaca a importância de avaliarmos tanto os requisitos funcionais quanto os não funcionais do produto de software.

9.2 Avaliação quantitativa da qualidade

As dimensões e os fatores de qualidade apresentados se concentram no produto de software completo e podem ser usados como uma indicação genérica da qualidade de uma aplicação. A equipe de software pode desenvolver um conjunto de características de qualidade e questões associadas que investigariam o grau em que cada fator foi satisfeito. Por exemplo, a norma ISO 25010 identifica a usabilidade como um importante fator de qualidade. Se lhe fosse solicitado para revisar uma interface do usuário e avaliar sua usabilidade, como você procederia?

Embora seja tentador criar medidas quantitativas para os fatores de qualidade citados, também podemos criar uma lista de verificação simples dos atributos que dão uma sólida indicação de que o fator está presente. Poderíamos começar com as subcaracterísticas sugeridas para a usabilidade na norma ISO 25010: adequabilidade, facilidade de aprendizagem, operabilidade, proteção contra erros, estética e acessibilidade. Você e sua equipe poderiam decidir criar um questionário para os usuários e um conjunto de tarefas estruturadas para eles realizarem. Você poderia observar os usuários enquanto eles realizam essas tarefas e pedir que completem o questionário no final.

Para fazer sua avaliação, você e sua equipe precisariam lidar com atributos específicos, mensuráveis (ou, pelo menos, reconhecíveis) da interface. Suas tarefas poderiam se concentrar em encontrar respostas para as seguintes perguntas:

- Com que velocidade os usuários determinam se o produto de software pode ou não ser usado para ajudá-los a completar a sua tarefa? (adequabilidade);
- Quanto demora para os usuários aprenderem a usar as funções do sistema necessárias para completar a sua tarefa? (facilidade de aprendizagem);
- O usuário consegue lembrar como usar as funções do sistema em sessões de teste subsequentes sem precisar reaprendê-las? (facilidade de aprendizagem);
- Quanto demora para os usuários completarem suas tarefas usando o sistema? (operabilidade);
- O sistema tenta impedir os usuários de cometer erros? (previsão de erros);
- O sistema permite que os usuários desfaçam operações que poderiam resultar em erros? (previsão de erros);

- As respostas oferecem reações favoráveis a perguntas sobre a aparência da interface do usuário? (estética);
- A interface se conforma às expectativas estabelecidas pelas regras de ouro? (acessibilidade);
- A interface do usuário se conforma aos itens da lista de verificação de acessibilidade exigidos para os usuários pretendidos? (acessibilidade);

Enquanto o projeto de interface é desenvolvido, a equipe de software revisaria o protótipo de projeto e faria as perguntas citadas. Se a resposta a essas perguntas for “sim”, é provável que a interface com o usuário apresente alta qualidade. Um conjunto de perguntas similares seria desenvolvido para cada fator de qualidade a ser avaliado. No caso da usabilidade, é sempre importante observar usuários representativos interagindo com o sistema. Para outros fatores de qualidade, pode ser importante testar o software no mundo real (ou, no mínimo, no ambiente de produção).