GERÊNCIA DE CONFIGURAÇÃO E EVOLUÇÃO

AULA 6

Profª Adriana Bastos da Costa

CONVERSA INICIAL

Temos discutido ao longo do nosso estudo que a gerência de configuração é uma disciplina fundamental quando falamos em projetos de desenvolvimento de *software*, tanto para organizar a construção de um novo *software* quanto para garantir uma manutenção segura e adequada de um *software* já em uso.

Discutimos também que mudanças e evoluções são inerentes a qualquer *software*, pois este sempre é criado para agregar valor a uma empresa, melhorando a produtividade, automatizando algum processo manual ou gerando resultados mais rápidos e precisos. Dessa forma, como a tecnologia e as necessidades do mercado e dos consumidores evoluem, é natural que o *software* também evolua e incorpore novos requisitos ao seu escopo.

Já ficou claro como a gerência de configuração é importante para manter todas as mudanças e os componentes criados para satisfazê-las organizados e gerenciados. Pois bem, a gerência de configuração continua sendo a maior aliada da gestão correta e completa dos componentes de *software*, enquanto eles forem criados e evoluídos.

Sabendo que a evolução de um *software* é algo inevitável, é preciso analisar alguns conceitos importantes relacionados com as implicações para ele, trazidas pelas modificações ocorridas ao longo do tempo. Por isso, nesta etapa vamos discutir alguns temas que envolvem os riscos e as necessidades de garantir que o *software* se mantenha íntegro, mesmo quando passa por constantes alterações.

Esta etapa está dividida em 5 temas principais:

- Processo de evolução;
- Sistemas legados;
- Riscos de substituição do *software*;
- Reengenharia de *software*;
- Refatoração.

Vamos explorar os temas acima e discutir sobre a importância de garantir o funcionamento adequado ao *software* que está sendo evoluído, como também o funcionamento adequado dos *softwares* que de alguma forma estão relacionados entre si e juntos mantêm o funcionamento da empresa, trazendo organização e melhores resultados para os negócios.

TEMA 1 – PROCESSO DE EVOLUÇÃO

Já discutimos anteriormente que os *softwares* passam, naturalmente, por modificações e evoluções. Essas alterações são naturais e fazem parte de manter um *software* útil para seus usuários. Isso acontece porque os sistemas refletem situações do mundo real, logo há a necessidade de que o *software* mude acompanhando as mudanças de requisitos impostos pelo ambiente em que está inserido. Se o sistema não sofre alterações necessárias para o negócio, ele pode ficar obsoleto e cair em desuso. É por isso mesmo que a manutenção em um *software* é algo que precisa ser tratado como um processo, que é planejado e gerenciado.

De forma resumida, o processo de manutenção envolve as modificações que são realizadas no *software* ao longo de todo seu ciclo de vida, podendo ocorrer durante o desenvolvimento ou após a sua entrega, quando já está em utilização pelos usuários finais. Como já exploramos anteriormente, as modificações podem ser de várias formas e voltadas para atingir objetivos distintos, como modificações evolutivas ou corretivas, por exemplo. De forma geral, as alterações servem para correção de erros, atualização do sistema, aperfeiçoamento do *software* ou para sua adaptação a uma nova necessidade do mercado ou do cliente, incorporando ou retirando requisitos do seu escopo inicial.

Apesar de termos discutido bastante sobre a importância da manutenção do *software* para mantê-lo sempre útil, até pouco tempo atrás ela era considerada uma etapa secundária no ciclo de vida do *software*, uma etapa de pouco valor. Era considerada por muitos como uma fonte de gastos que comprometiam a criação de *software*.

Conforme afirma o site DevMedia (O papel..., 2010), era muito comum que,

> durante o desenvolvimento de um *software*, não houvesse a preocupação de que um dia ele precisaria sofrer alguma alteração para que requisitos fossem criados ou alterados para se adequarem às novas necessidades do usuário. Quando isso ocorria, a manutenção era feita de forma precária e desorganizada, pois não existia um gerenciamento adequado para que fossem feitas as mudanças. Com isso, as mudanças costumavam gerar novos erros que aumentariam ainda mais o tempo necessário para que fossem feitas as correções desejadas pelo usuário.

Ou seja, o *software* era construído com foco nas necessidades imediatas, não pensando no seu futuro.

O envelhecimento de um *software* é um processo inevitável e pode causar a perda de desempenho devido às constantes modificações ou mesmo ao não acompanhamento da evolução da tecnologia. O *software* pode envelhecer por dois grandes e principais motivos: o sistema não é alterado para acompanhar as novas necessidades do negócio ou, quando é alterado, é feito de maneira desordenada, inserindo defeitos ou um funcionamento não adequado ao que se espera do *software*.

Apesar de sabermos que o envelhecimento é um processo inevitável quando falamos de *software*, ele pode ser atrasado ou minimizado, se forem seguidos alguns cuidados no desenvolvimento e evolução do *software* em questão. Dentre alguns cuidados que podem ser tomados para minimizar o efeito do envelhecimento está a definição de um processo de manutenção constante e organizado do *software*. Esse processo deve ser voltado para o entendimento correto e completo da alteração, a análise de impacto da alteração do *software* como um todo e nos testes que devem ser feitos para garantir que o *software* continua íntegro e estável, mesmo depois da implementação da alteração.

De acordo com Sommerville (2019), na evolução de *software* devem ser considerados diversos fatores que servirão de base para decidir se o sistema atual deve ser evoluído, ou se deve ser abandonado para que um novo sistema seja construído com base nos novos requisitos do *software* e na tecnologia mais atual. Essa nem sempre é uma decisão simples, mas precisa ser analisada levando em consideração vários fatores, pois nem sempre a evolução é o melhor caminho para o *software*, por conta do grau de dificuldade e no custo envolvidos com a manutenção.

Baseado nesse conceito da análise e tomada de decisão quanto a evoluir ou substituir um *software*, devem ser levados em conta vários fatores, tais como: o custo envolvido no processo, a integridade e confiabilidade do *software* após a manutenção, a capacidade de o *software* continuar evoluindo e se adaptando a mudanças futuras, o desempenho geral, as limitações de suas atuais funções por conta de tecnologia, e ainda outras alternativas e opções disponíveis no mercado que atendam à necessidade do negócio de maneira mais completa, rápida e barata. Mas, de uma maneira bem simplificada, se o *software* está funcionando, é muito comum que não seja substituído por um novo sistema. É

mais natural que ele passe pelos ajustes desejados por quem o utiliza, incorporando assim novos requisitos ou regras de negócio. É preciso analisar o custo e benefício da decisão a ser tomada. O que vale mais a pena: manter o *software* que já existe ou construir um novo?

Quando pensamos em um *software* livre, o conceito muda um pouco. Você sabe o que é um *software* livre? De acordo com o *site* da Free Soft Foundation (S.d.),

> A Free Software Foundation (FSF, *Fundação para o Software Livre*) é uma organização sem fins lucrativos, fundada em 4 de outubro de 1985 por Richard Stallman e que se dedica a eliminação de restrições sobre a cópia, redistribuição, estudo e modificação de programas de computadores – bandeiras do movimento do *software* livre, em essência. Faz isso promovendo o desenvolvimento e o uso de *software* livre em todas as áreas da TI mas, particularmente, ajudando a desenvolver o sistema operacional GNU e suas ferramentas.

Em outras palavras, a expressão *software livre* refere-se a todo programa de computador que pode ser executado, copiado, modificado e redistribuído sem que haja a necessidade da autorização do seu proprietário para isso. Esse tipo de *software* disponibiliza para seus usuários e desenvolvedores o livre acesso ao código-fonte para que possam realizar alterações da maneira que desejarem.

Dessa forma, podemos concluir que um projeto de *software* livre não costuma ter um final definitivo, pois como é um projeto colaborativo, pode passar por necessidades diversas de seus autores. Ou seja, mesmo quando o *software* alcança um patamar de desenvolvimento considerado satisfatório para alguns autores, este patamar pode não ser o adequado para os demais usuários ou autores do projeto, gerando assim um ciclo de melhorias constantes.

A equipe de desenvolvimento de um *software* livre costuma ser algo em constante modificação, pois mesmo que os idealizadores originais deixem o projeto em algum momento, outros desenvolvedores podem continuar o desenvolvimento do *software* em uma versão paralela ou até mesmo podem passar a assumir a gerência da versão original do projeto, continuam assim a sua evolução. Esse fato nos permite dizer que um projeto de *software* livre precisa ser controlado e gerenciado constantemente e de maneira adequada para garantir a qualidade e a eficiência do seu funcionamento.

Dessa forma, independentemente de ser um *software* livre ou de propriedade de uma empresa, todo *software* passa por manutenções, que, quando realizadas, mantêm o *software* vivo e útil para o seu objetivo final. E o

foco da gerência de configuração se mostra cada vez mais importante para garantir a excelência e o funcionamento adequado de todo *software*.

Mas é preciso compreender que a manutenção de um *software* não é algo trivial. Além de compreender o requisito que será incluído ou alterado no *software*, é preciso minimizar o risco de mantê-lo em uso por vários usuários e que pode também impactar outros *softwares* que estão em uso e que fazem interface com o *software* que está em manutenção. Ou seja, é uma organização que funciona como uma engrenagem e que precisa ser analisada e gerenciada como um todo para garantir que o negócio continue estável e em funcionamento adequado, gerando os resultados esperados.

Vamos agora explorar o que são sistema legados e como eles se relacionam com este conteúdo que estamos discutindo nesta etapa.

TEMA 2 – SISTEMA LEGADO

Quando falamos sobre uma empresa consolidada no mercado e que investe em tecnologia e *softwares* para melhorar seus resultados, você consegue imaginar quantos deles já foram desenvolvidos para manter os negócios? Pois é, quanto mais tempo a empresa tem de funcionamento, mais *softwares* ela terá apoiando o seu dia a dia. Todos esses *softwares* criados ao longo do tempo, atualmente já são antigos, porém ainda em uso. Eles são chamados de *sistema legado*.

Hoje, um grande problema para as organizações é gerenciar a manutenção em seus sistemas legados. Apesar de os sistemas legados serem sistemas fundamentais para os negócios de uma empresa, normalmente eles possuem documentação incompleta ou até mesmo inexistente. É muito comum também que tenham passado, ao longo dos anos, por manutenções realizadas por diversos profissionais, que nem sempre seguiram as boas práticas de engenharia de *software*. Esse fato pode ter gerado um código mal escrito e de difícil manutenção, muitas vezes até com defeitos ainda não identificados pelos usuários.

Ou seja, os *softwares* em funcionamento em uma empresa são chamados de *software legado*, que também constituem um desafio na rotina das empresas, pois ele é imprescindível para que os processos funcionem, mas, por outro lado, os sistemas legados podem estar passando por um processo natural de envelhecimento. O problema vai muito além da área de TI e envolve todos os

colaboradores que utilizam ferramentas digitais ao longo do dia. Recursos lentos, *bugs*, ausência de funcionalidades, travamento e perda de dados são algumas das dificuldades enfrentadas e que afetam diretamente o resultado do trabalho dos colaboradores que utilizam sistemas legados.

Um sistema legado, como vimos anteriormente, pode ter sido construído sem o adequado projeto de solução técnica, pensando nas evoluções futuras, e pode ter passado por várias modificações sem o planejamento completo ou vir a passar por elas por dificuldades para ser mantido por falta de desenvolvedores que conheçam a tecnologia e a forma como ele foi construído. Tudo isso pode gerar alguns problemas comuns, que precisam ser analisados e, para isso, alternativas de solução precisam ser definidas.

Por conta de tudo isso, um sistema legado é um forte candidato a ser substituído por um *software* novo, mais moderno e com escopo mais adequado à realidade do mercado e dos clientes. A decisão de substituir ou não um *software* pode avaliar os seguintes itens:

- O *software* está gravemente desatualizado?
- Falta de mobilidade na forma como o *software* foi construído?
- Falta de escalabilidade na forma como o *software* foi construído?
- Falta de suporte capacitado para evoluir o *software*?
- Existe incompatibilidade com sistemas modernos e necessários para o negócio?
- O *software* está hospedado em servidores físicos e é dependente dele?

Mas é claro que nem todos os *softwares* que são chamados de *sistema legados* são *softwares* antigos. Alguns, relativamente novos, já podem estar precisando de alterações por não terem sido muito bem planejados. Vamos detalhar as características afetam todo *software* e que o classificam como sendo um sistema legado:

- ***Software* desatualizado** – um *software* desatualizado pode apresentar defeitos em relação a requisitos obsoletos e pode também apresentar ameaças quanto à segurança, pois ele não evoluiu para incorporar mecanismos mais seguros que são atualmente utilizados na construção dos *softwares* modernos;
- **Falta de mobilidade** – antigamente não era comum pensar em *softwares* para serem utilizados em dispositivos móveis, por isso é muito comum

que os sistemas legados não tenham sido construídos para serem utilizados em *smartphones* ou *tablets*. O problema é que, com a necessidade atual de utilização de *softwares* em qualquer lugar e em qualquer horário, não ter uma versão *mobile* é um atraso para o negócio e para as pessoas que utilizam o *software*. Com o crescimento do uso de aplicativos móveis, passa a ser essencial ter uma versão mobile dos *softwares*;

- **Falta de escalabilidade** – a escalabilidade é uma característica que permite o *software* crescer e a ter mais usuários com o passar do tempo. É muito provável que os sistemas legados não tenham sido projetados pensando na perspectiva de possível crescimento, o que se torna um desafio para mantê-lo estável e disponível para uso quando passa da capacidade prevista originalmente no projeto;
- **Falta de suporte** – no caso de sistemas legados mais antigos e desatualizados, pode ocorrer a dificuldade de encontrar profissionais com a competência adequada para dar o suporte e implementar as modificações necessárias. Dessa forma, o desafio passa a ser encontrar profissionais com um custo razoável para executar as alterações necessárias. Um sistema que recentemente deixou de ter suporte foi o Windows XP, por exemplo. A falta de suporte pode ser um grande problema para os usuários, pois a dependência de um *software* sem suporte pode afetar outros softwares relacionados;
- **Incompatibilidade com sistemas modernos** – quando um *software* é criado, é preciso pensar que é muito provável que ele tenha interface com outros sistemas, para garantir uma gestão única e eficiente da empresa. O problema é que, nos sistemas legados, é muito comum que essa preocupação não tenha sido levada em conta no momento de criar o projeto técnico da solução. Isso dificulta a integração e compromete o uso do *software* pela falta;
- ***Software* hospedado em servidores físicos** – alguns sistemas legados podem ter sido construídos em uma época com nem se cogitava o armazenamento em nuvem, ou este era muito caro. Por isso, a maioria dos *softwares* legados têm grande dependência dos dispositivos de *hardware*, o que impede que a gestão da TI pense em diminuição de custos com a contratação de serviços de nuvens ou de outras alternativas

disponíveis atualmente no mercado. A realidade atual mostra que a hospedagem em servidores físicos tem um valor muito mais alto que o custo de manter os *softwares* na nuvem.

Analisando todas essas características acima e conhecendo a realidade atual da TI, sabemos que atualizar um sistema legado e para deixá-lo compatível com as novas tecnologias e necessidades dos usuários, diz respeito diretamente à saúde competitiva de uma empresa e à produtividade dos profissionais que trabalham nela. Se os sistemas não estão mais agregando o valor esperado para o negócio, não vale a pena mantê-los. Mas é claro que sempre é importante fazer uma análise de custo *versus* benefício quanto à decisão a ser tomada.

Vamos explorar algumas dicas importantes para garantir um processo de manutenção controlado para a evolução dos sistemas legados, tais como:

- Atualizar um sistema legado envolve, também, um processo de migração de dados para o novo sistema. Logo, a empresa deve tomar alguns cuidados essenciais para que a mudança seja feita de forma organizada e segura;
- Adotar técnicas de refatoração, que tem como objetivo melhorar internamente o código de um sistema sem comprometer a experiência do usuário enquanto está utilizando o software. Vamos falar mais em detalhes sobre essa técnica ainda nessa etapa;
- Reescrever o código do *software*, pode ser mais complexo, mas permite que novas metodologias de desenvolvimento possam ser aplicadas para estruturar o sistema, permite eliminar vulnerabilidades quanto à segurança e permite até mesmo integrar tecnologias mais atualizadas ao software. Portanto, reescrever ou substituir um *software* pode gerar ganhos com manutenções futuras;
- Todo *software* refatorado ou reescrito precisa passar por testes completos, manuais ou automatizados. Os testes são essenciais para garantir a qualidade e o bom funcionamento dos *softwares*. É preciso garantir que as partes do *software* que foram alteradas e também as possíveis partes relacionadas sejam testadas, evitando inserir novos defeitos por conta de modificações no *software*.

Todos os *softwares*, mais cedo ou mais tarde, serão chamados de *sistemas legados*. Por isso, é preciso definir um processo de desenvolvimento

de *software* completo, que tenha visão de longo prazo para o uso do *software*, e um processo de manutenção robusto, que atrase a necessidade de substituição do *software*. Dessa forma, é possível gerenciar de maneira organizada as necessidades do mercado e do usuário, e adequar o *software* para que esteja entregando o resultado esperado pela empresa.

No próximo tópico, vamos discutir sobre a necessidade de identificar e gerenciar os riscos envolvidos em aceitar um *software* legado por mais tempo na empresa e identificar e gerenciar os riscos envolvidos em substituir um *software* legado.

TEMA 3 – RISCOS DE SUBSTITUIÇÃO DO *SOFTWARE*

Apesar das dificuldades de gerenciar a manutenção de um *software*, é fundamental mantê-los atualizados. E muitas vezes, gestores e proprietários de empresas não sabem da importância dessa atualização, tanto daqueles utilizados no cotidiano dos processos corporativos internos quanto os utilizados no atendimento ao cliente.

Um *software* desatualizado pode trazer riscos para o bom andamento dos negócios, podendo causar prejuízos gravíssimos que poderiam ser evitados se a manutenção adequada dos sistemas fosse executada e gerenciada de maneira padronizada pelos responsáveis.

Sabemos que a TI evolui em uma velocidade que nem sempre é possível acompanhar facilmente, o que afeta mais ainda os sistemas legados. A modernização costuma ser cara ou até mesmo inviável para a empresa por conta do tamanho e da dificuldade que as evoluções envolvem. Muitas vezes, estamos falando de sistemas grandes e complexos, que rodam também em ambientes antigos, o que torna ainda mais cara sua evolução. Além dos requisitos estarem desatualizados, e infraestrutura onde o *software* está rodando também pode estar obsoleta. Juntando tudo isso, suportar esses *softwares* e a infraestrutura como um todo pode representar um risco à segurança da organização e um custo bastante elevado.

Os *softwares* legados podem também ser afetados pela evolução dos sistemas operacionais e da infraestrutura. A solução técnica projetada para o sistema legado pode então tornar-se obsoleta, apresentando problemas no funcionamento, como compatibilidade, estabilidade e até mesmo segurança. A empresa passa a ficar vulnerável e dependente de tecnologias obsoletas,

afetando o resultado do negócio. Mas, apesar de tudo isso, construir um *software* novo é um investimento grande tanto de tempo quanto financeiro. Por isso, análise da melhor solução para um problema de envelhecimento de um *software* não é simples ou trivial. Envolve várias variáveis a serem analisadas.

Precisamos compreender que a questão de segurança dos dados é algo crítico e que precisa ser bem controlada, pois se a empresa fizer a opção de incluir ou atualizar as tecnologias existentes na organização, a segurança dos dados também pode ficar comprometida. Isso ocorre porque durante o processo de manutenção do *software*, as informações correm o risco de serem perdidas, terem sua integridade afetada ou até mesmo serem roubadas. É por isso que muitos autores defendem a importância de definir um processo completo e robusto de construção e manutenção de *software*, de forma a gerenciar os riscos envolvidos e inserir controles que diminuíam os possíveis impactos nos negócios das evoluções em um *software*.

Já discutimos que é bastante normal que as empresas que possuem processos automatizados por meio de *software* tenham necessidade de migrar dados de sistemas legados para sistemas mais modernos. A necessidade de migração de dados ocorre, muitas vezes, para diminuir custos e despesas, para otimizar e melhorar processos ou ainda para atender a leis regulatórias ou a normas de mercado.

Apesar de a manutenção constante e organizada do *software* trazer maior competitividade, o que é importante em um mercado cada vez mais disputado e competitivo, o processo de evolução exige cuidados para que não aconteçam problemas, como a instabilidade ou a parada de uma operação, por exemplo. Por isso, nem sempre o melhor é partir para a manutenção sem analisar os riscos para o negócio. As mudanças podem ser grandes e podem até impactar em atividades essenciais ao negócio, gerando perdas financeiras ou de produtividade. Dessa forma, é preciso agir conforme o planejado, priorizando através de regras claras e de forma incremental, o que agrega mais valor para o negócio. Priorizar as mudanças baseadas no valor agregado para o negócio é uma boa forma de organizar a implantação das melhorias, de forma controlada e organizada.

Mas é preciso pensar de forma geral, pois é comum pensar que apenas transferir os dados do sistema legado para o novo sistema resolve o problema de atender às novas definições, o que na verdade pode não acontecer, pois a

migração de dados históricos é um processo complexo e exige atenção da gestão do projeto, que, apesar de estar focada nas funcionalidades do novo sistema, precisa olhar o *software* como um todo, cuidando dos dados, das interfaces e da infraestrutura. Todos esses fatores podem influenciar no funcionamento do *software*. Ou seja, é preciso pensar no novo, mas sem se descuidar do antigo e na integração entre os dois. Pode ser necessário que o novo e o antigo funcionem, por um tempo, de maneira paralela, possibilitando a análise dos resultados e mantendo uma contingência para os negócios, caso dê algum problema inesperado.

Essas dificuldades – todo o período necessário para o planejamento adequado e a exigência de um processo sistemático, dividido em etapas – leva certo tempo e esforço da equipe envolvida, o que pode desencorajar as empresas. Afinal, qual organização quer colocar seus dados em risco e ainda esperar por meses pela conclusão de um processo de modernização e evolução do *software*? Mas isso não quer dizer que a empresa deve abandonar o projeto e abrir mão de se tornar mais competitiva.

A solução é organizar o processo de manutenção de *software* e organizar o projeto em partes. Quebrar a manutenção em etapas menores facilita o gerenciamento do todo e ajuda a mostrar, de forma mais rápida, alguma melhoria para a operação. A finalização de uma parte do processo de evolução do *software* pode ser considerada pequena em relação ao todo, mas já é capaz de trazer resultados visíveis e de valor para a empresa. Apesar de os resultados já poderem ser vistos mesmo em pequenas melhorias implementadas, é preciso trabalhar para que todo o projeto seja concluído de forma priorizada. Ter acesso a melhorias antes do término de toda a manutenção planejada pode ser perigoso se a empresa se contentar apenas com as pequenas partes. É preciso manter o planejamento e estimular a equipe e os gestores a seguir com o plano de manutenção, porque mesmo já obtendo resultados, é preciso ter todas as mudanças planejadas concluídas para que todo o ganho esperado para o negócio seja alcançado.

Como já discutimos, um grande desafio é cuidar da segurança das informações durante todo o processo de manutenção. Gerenciar riscos é muito importante para evitar o surgimento de problemas nos sistemas. Dessa forma, independentemente de a empresa escolher por manter os sistemas legados por mais tempo ou por seguir com a modernização destes sistemas, é preciso

investir na segurança e proteção dos dados. Em qualquer situação, é preciso definir e divulgar a política e um processo de gestão de riscos, que garanta a segurança das informações da empresa.

No processo de manutenção de *software*, um dos primeiros passos deve ser analisar as alterações e calcular os riscos envolvidos com o impacto de cada alteração no *software* como um todo. O problema é que, muitas vezes, por ser um sistema legado que já faz parte da empresa há anos, os gestores não se atentam para sua importância e por isso esses sistemas não despertam na empresa o senso de que precisam ser mantidos e continuar seguros. Por isso, acabam não recebendo atenção e, inclusive, ficam fora do escopo de auditorias e do orçamento de manutenção.

O resultado disso pode ser um descuido que gera problemas de segurança e que abre espaço para fraudes. São frequentes os casos em que os invasores aproveitam a vulnerabilidade dos sistemas, principalmente os sistemas que possuem tecnologias obsoletas. Como consequência, a empresa pode ter perdas não apenas financeiras, mas também de informações estratégicas, que são vitais para o negócio.

É fundamental também ter uma política eficiente de *backup* e de recuperação de dados, que possa ser usada em caso de emergência. A contingência de ter uma política bem estruturada de *backup* pode ser cara e necessitar de investimentos, mas garante a tranquilidade de se manter os negócios funcionando por mais tempo, mostrando para o mercado, os clientes e os concorrentes que a estabilidade é um valor estimado pela empresa.

TEMA 4 – REENGENHARIA DE *SOFTWARE*

O conceito de reengenharia não é exclusivo de *software*; na verdade, ele surgiu na década de 90, como um conceito de gestão. O objetivo era procurar uma forma de responder ao turbulento mundo onde as empresas estavam inseridas no início dos anos 90. Essa movimentação constante, motivada pela evolução da internet, exigia ações rápidas e constantes para garantir a competitividade das empresas, logo era preciso pensar diferente e agir diferente. Nada melhor para atender à velocidade esperada, como implantar reengenharia nas empresas.

A reengenharia então significa abandonar antigos processos e conceitos e analisar em detalhes todo o trabalho necessário para produzir os produtos ou

serviços, simplificando o que é possível e eliminando o que for desnecessário, proporcionando assim maior valor para os negócios. Podemos entender que a reengenharia está totalmente relacionada à reestruturação dos processos da empresa para alcançar melhorias nos resultados obtidos, principalmente nos indicadores de desempenho como indicadores de custos, de qualidade, de atendimento e na velocidade dos serviços prestados. O foco está em melhorar a competitividade da empresa e os resultados obtidos por meio da análise e da simplificação dos processos definidos e executados.

Quando falamos em melhorar a competitividade ou os resultados, no fundo estamos falando de ações que gerem redução de custos, aumento do grau de satisfação do cliente e aumento da produtividade. Os processos que sejam complicados para serem remodelados podem até mesmo ser eliminados e substituídos por outros mais simples e leves. Dessa forma, a reengenharia é vista como uma técnica para proporcionar às empresas os meios necessários para se manter competitiva mediante às novas demandas e exigências, por meio de adaptação necessária à nova realidade imposta pelo mercado.

Seguindo a mesma linha de raciocínio da reengenharia como técnica de gestão, vemos que a reengenharia de *software* é o processo de reconstrução de um *software* já existente e em uso. Muitos dos passos e processos da reengenharia de *software* são os mesmos que os de um processo de desenvolvimento de *software*, tendo a mesma preocupação com o planejamento e os controles para garantir a qualidade do produto de *software*.

Engenharia reversa é o processo de descobrir os princípios tecnológicos e o funcionamento de um *software*, por meio da análise de sua estrutura, funcionalidades e operação. Mas em relação aos *softwares*, reengenharia e engenharia reversa possuem os mesmos objetivos e conceitos e podem ser tratadas como sinônimos.

Quando falamos em *hardware*, um conceito que costuma ser confundido, mas que tem outro objetivo, é o da *engenharia reversa*, que consiste em, por exemplo, desmontar uma máquina para descobrir como ela funciona. Percebem a diferença? A engenharia reversa desmonta algo para conhecer seu funcionamento, já a reengenharia analisa algo para conhecer seu funcionamento. Mas em se tratando de *software*, os conceitos realmente são muito próximos.

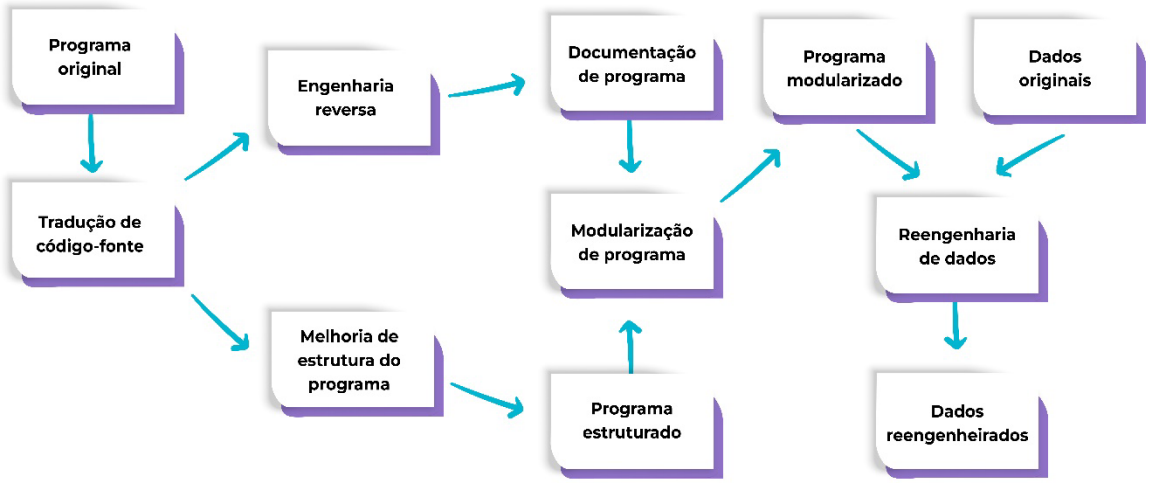
Com a rápida evolução da tecnologia disponibilizada para o mercado, as empresas acabam ficando com poucas alternativas possíveis para se manter competitivas e lucrativas, sendo elas: manter os *softwares* legados afetando a produtividade dos colaboradores e com custos cada vez maiores, reconstruir os *softwares* novamente ou realizar a reengenharia para melhorar e facilitar a manutenção dos *softwares*, quanto para buscar uma solução técnica mais simples e atual, podendo ou não envolver a mudança da linguagem. Não é uma decisão fácil de ser tomada, pois envolve analisar o custo *versus* o benefício esperado com cada uma das alternativas.

Vamos analisar cada uma das alternativas em detalhes. De acordo com o artigo publicado no *site* da *DevMedia* (Reengenharia..., 2011),

> 1. No caso de manter um *software* legado, apenas efetuando-se as manutenções para que o mesmo continue operando, muitos problemas podem ocorrer, tais como a alocação de pessoal para essa tarefa que pode ter uma porcentagem bastante significativa do esforço de uma organização, além da falta de sua documentação, comum nesses casos e que torna ainda mais crítica a situação;
> 2. A opção de reconstrução do um software legado também tem problemas associados, pois o fato do software ter regras de negócios embutidas, regras estas que podem não estar documentadas e a possibilidade das pessoas que as dominam não estarem mais na empresa, faz com que a sua completa reconstrução não seja tão confiável ou fácil;
> 3. A reengenharia é a forma que muitas organizações estão buscando para manter/refazer seus *softwares*, livrando-se das manutenções difíceis e da degeneração de suas estruturas. Por esse motivo, é importante que o resultado desse processo seja definido, organizado e confiável.

Segundo Sommerville (2007), a principal característica da reengenharia de *software* é o risco reduzido, ou seja, a chance de ocorrerem problemas ou erros durante o redesenvolvimento de um *software* é maior do que o da reengenharia. Ele afirma também que o custo da reengenharia é em torno de quatro vezes menor do que o utilizado para desenvolver um *software* novo. A diferença entre a engenharia convencional e a reengenharia é o ponto de partida de ambas, ou seja, a primeira parte do zero, enquanto a segunda parte de algo já construído e em uso. Enquanto a engenharia começa com uma especificação escrita, a reengenharia se inicia com o próprio sistema antigo, que servirá como especificação para o novo sistema, conforme podemos analisar na Figura 1:

Figura 1 – Processo de reengenharia



Fonte: Sommerville, 2007.

Pelo processo proposto por Sommerville (2007), é possível acompanhar as fases previstas de maneira organizada para garantir o entendimento tanto do código fonte quanto dos dados, que serão a base para a construção do novo *software*.

TEMA 5 – REFATORAÇÃO

Outro conceito importante que precisamos explorar é o conceito de refatoração de *software*. Esse conceito surgiu na década de 80 e está pautado em melhorar do código de um *software* sem, necessariamente, envolver a criação de novas funcionalidades. O objetivo é realizar uma melhoria interna, para transformar um código que foi mal construído ou construído de maneira desordenada em código limpo e com *design* simples. O que se busca alcançar com a refatoração é a melhoria da lógica empregada para construir o código, a facilidade de leitura e manutenção desse *software* e até mesmo o aumento do desempenho e da eficiência. Ou seja, o foco não é melhoria na apresentação externa do *software*, mas sim melhorias internas, na lógica de construção do código.

Pois bem, a refatoração envolve conceitos simples, mas que na prática apresentam dificuldades para serem implementados, principalmente se for em um projeto grande ou complexo. A melhor forma de lidar e organizar a refotaração é com planejamento e documentação, registrando o que está sendo

refatorado, sempre focando em agregar valor para o *software*. Refatorar sem um objetivo direto e específico é perda de tempo e desperdício de recurso.

O conceito de refatoração ganhou notoriedade com o XP (*Extreme Programming* ou *programação extrema*). O XP é uma metodologia ágil focada na parte técnica do desenvolvimento de *software*, por isso possui práticas voltadas para a engenharia de *software*, pensando na análise, projeto técnico e construção de um deles.

Baseado nos conceitos do XP, a refatoração é sinônimo de código limpo. Ou seja, tornar o código limpo também é um processo de refatoração, ou pelo menos pode ser, dependendo se esse cuidado foi tomado durante o desenvolvimento do *software*.

Mas, afinal, o que é o código limpo? Um código limpo pode ser entendido como um código simples, elegante e direto. Exatamente o conceito das metodologias ágeis, que pregam um processo leve de desenvolvimento de *software*. A vantagem do código limpo é ser fácil de ser entendido por qualquer desenvolvedor que trabalhe no código com o propósito de realizar evoluções e manutenções corretivas, o que faz todo sentido quando falamos que um *software* sofre, inevitavelmente, constantes manutenções ao longo da sua vida.

De acordo com Cavalcante (2020), algumas práticas conhecidas de código limpo são:

- Utilizar nomes significativos para variáveis, métodos e classes;
- Criar funções curtas e simples;
- Utilizar bons comentários no código, de forma a informar o objetivo do código;
- Padronizar a formatação de código, facilitando o entendimento;
- Fazer o tratamento de erros;
- Remover todo código duplicado, deixando o código mais leve;
- Remover classes desnecessárias, e fazer modificações que as deixem com responsabilidades bem definidas;
- Fazer o código de tal forma que todos os testes passem, evitando desperdício de código.

É possível, então, entender que refatorar evita a deterioração durante o ciclo de vida de um código, o que não é incomum. Outro motivo é a melhora no entendimento do código, facilitando a manutenção e os famosos defeitos. Além

disso, é de fundamental importância que todo o código tenha cobertura de testes automatizada para garantir que o comportamento externo não tenha mudado.

Para começar a refatoração do código, devemos ter sempre em mente que: é preciso realizar uma série de alterações atômicas, se possível, com um escopo pequeno. Isso vai deixar o código existente cada vez melhor.

É importante atentar para algumas regras vitais para garantir a qualidade do *software* enquanto ele está passando pela refatoração do código, tais como: não há desenvolvimento de novas funcionalidades durante a refatoração e todos os testes devem ser executados depois da refatoração.

Construir um *software* é uma tarefa colaborativa, por isso o código deste quase sempre passa por muitas mãos. Por isso, alguns deslizes podem acontecer. Entre eles estão códigos duplicados, métodos localizados em classes indevidas, entre outras incorreções. O resultado disso tudo é, portanto, um código desorganizado, com maior probabilidade de gerar defeitos. E tudo o que um desenvolvedor não quer são problemas com queda de desempenho ou com a manutenção futura do *software*.

Para ajudar os desenvolvedores a estruturar o código baseado no conceito de código limpo, existem algumas técnicas que podem ser utilizadas para refatorar o código de maneira organizada e eficiente, tais como:

Método de extração:

- Problema: você tem um fragmento de código que pode ser agrupado;
- Solução: mova esse código para um novo método (ou função) separado e substitua o código antigo por uma chamada ao método.

Método em linha:

- Problema: quando um corpo de método é mais óbvio que o próprio método, use esta técnica;
- Solução: substitua as chamadas para o método pelo conteúdo do método e exclua o próprio método no processo de refatoração.

Extrair variável:

- Problema: você tem uma expressão difícil de entender;
- Solução: coloque o resultado da expressão ou de suas partes em variáveis separadas que são autoexplicativas.

Dividir variável temporária:

- Problema: você tem uma variável local usada para armazenar vários valores intermediários dentro de um método (exceto para variáveis de ciclo);
- Solução: use variáveis diferentes para valores diferentes. Cada variável deve ser responsável por apenas uma coisa em particular.

Remover atribuições a parâmetros:

- Problema: algum valor é atribuído a um parâmetro dentro do corpo do método;
- Solução: use uma variável local em vez de um parâmetro na refatoração.

Variável temporária:

- Problema: você tem uma variável temporária à qual é atribuído o resultado de uma expressão simples e nada mais;
- Solução: substitua as referências à variável pela própria expressão na refatoração.

Substituir variável temporária por consulta:

- Problema: você coloca o resultado de uma expressão em uma variável local para uso posterior no seu código;
- Solução: mova a expressão inteira para um método separado e retorne o resultado. Consulte o método em vez de usar uma variável. Incorpore o novo método em outros métodos, se necessário.

Substituir método por objeto de método:

- Problema: você tem um método longo no qual variáveis locais estão tão entrelaçadas que não é possível aplicar o *método de extração*;
- Solução: transforme o método em uma classe separada para que as variáveis locais se tornem campos da classe. Em seguida, você pode dividir o método dentro da mesma classe.

Algoritmo substituto:

- Problema: substituir um algoritmo existente por um novo;

- Solução: substitua o corpo do método que implementa o algoritmo por um novo algoritmo.

Refatorar é uma forma de pensar o desenvolvimento do *software*, é uma mudança cultural. Por ser uma mudança cultural, fica fácil entender o porquê de alguns desenvolvedores se recusarem a *refatorar* um código. Eles acreditam que seja um retrabalho que não vai contribuir com o seu dia a dia, ou ainda porque não aprenderam a importância desse hábito. Por isso, é preciso difundir essa prática que tem como foco a melhoria contínua do *software*. É preciso que todos entendam que, em um código mal escrito, prejudica a implantação uma nova funcionalidade. Isso ocorre porque é muito provável que os desenvolvedores acabem perdendo ainda mais tempo no ajuste do código, uma vez que precisam compreender primeiro como o *software* foi construído para depois conseguir implementar todas as alterações necessárias, testar e só depois disso colocar a funcionalidade nova ou alterada em produção.

Dessa forma, pode-se agir preventivamente realizando os ajustes necessários de tempos em tempos, ou ainda, quando a estrutura interna do *software* começar a perder um pouco da sua integridade. Dependendo da estratégia de melhoria do *software*, é possível realizar a refatoração sempre que houver uma atualização no código. Essa pode ser uma boa maneira de garantir a melhoria contínua do código, pois em razão das atividades de manutenção e inclusão de novas funcionalidades o código pode ir ficando desorganizado e sem padronização. A vantagem de manter a refatoração de forma contínua é garantir que pequenas partes do *software* sejam melhoradas de cada vez, minimizando o risco de erros indesejados serem inseridos de maneira equivocada pelo programador. Portanto, isso nos leva a entender que a refatoração pode ser utilizada como uma técnica disciplinada e que deve ser repetida continuamente, com o intuito de organizar o código, minimizando, assim, as chances de defeitos e mantendo a padronização ao longo do tempo.

É preciso difundir a ideia de que a refatoração não é um desperdício de tempo e de esforço, mas sim uma forma de melhorar a produtividade dos desenvolvedores, pois é um ajuste que irá repercutir no fluxo de trabalho de toda a equipe de desenvolvimento do *software*.

Refatorar é uma ação que busca melhorar a qualidade do código, tornando-o mais legível e fácil de entender. Se não houver um objetivo a ser alcançado com a refatoração, não vale a pena executá-la. O limite entre a

melhoria e o simples capricho de um programador pode ser tênue e precisa ser cuidado. O objetivo é gerar ganho de valor, se a refatoração não trouxer ganho realmente passa a ser simplesmente desperdício de esforço e de tempo.

Refatorar previne defeitos, já que, ao executar a revisão e melhorias no código, é possível encontrar falhas no desenvolvimento de pequenos trechos do programa. Refatorar otimiza o trabalho da equipe, pois a melhoria na construção do código fará a equipe de desenvolvimento programar mais rapidamente, já que fica mais fácil incluir uma nova funcionalidade ou alterar uma funcionalidade já existente. Refatorar também aumenta a produtividade, sendo que, com o código limpo, é possível que a equipe possa investir o tempo em outras tarefas igualmente importante, tais como focar em estudar novas tecnologias, frameworks e linguagens de programação. A criação de conhecimento também melhora o resultado do trabalho da equipe.

A refatoração também pode ser aplicada no banco de dados, além da aplicação de discutimos até o momento, feita no código do *software*.

De acordo com Ambler e Sadalage (2006), a refatoração em banco de dados pode ser dividida em seis categorias principais: *refatoração estrutural, refatoração de qualidade de dados, refatoração de integridade referencial, refatoração arquitetural, refatoração de método* e *refatoração de transformação.*

Vamos entender cada uma delas:

- **Refatoração estrutural**: consiste em alterações de um ou mais elementos, o que pode ser: dividir coluna, dividir tabela, excluir tabela, excluir visão, introduzir chave substituta por chave natural, substituir coluna, substituir campo complexo por tabela e substituir um para muitos com tabela associativa;
- **Refatoração de qualidade de dados**: baseia-se nas mudanças que aprimoram a qualidade da informação contidas no banco de dados, ou seja, buscam melhorar e/ou assegurar a coerência dos dados armazenados na base de dados. Sendo: adicionar tabela de pesquisa, aplicar códigos padrão, aplicar tipo padrão, consolidar a estratégia-chave, excluir restrição de coluna, excluir valor padrão, excluir restrição não anulável, introduzir restrição de coluna, introduzir formato comum, introduzir valor padrão, fazer coluna não nula, mover dados, substituir o código do tipo de propriedade com bandeiras;

- **Refatoração de integridade referencial**: buscam garantir a remoção adequada de um registro em uma tabela que seja referenciado dentro de outra ou linha que não seja mais necessária, o que piode se dar da seguinte forma: adicionar restrição de chave estrangeira, adicionar gatilho para coluna calculada, excluir restrição de chave estrangeira, introduzir cascata, introduzir exclusão rígida, introduzir exclusão maleável, introduzir gatilhos para histórico;
- **Refatoração arquitetural**: proporcionam melhorias de uma forma global no banco, propondo a substituição de programas ou *scripts* externos que interagem com a base de dados por programações internas ao banco. Pode ser: adicionar métodos CRUD (Criar), recuperar, atualizar, excluir, adicionar espelho de tabela, adicionar leitura de método, encapsular tabela com visão, introduzir método de cálculo, introduzir índice, apresentar tabela somente leitura, migrar método de banco de dados, método de substituição com visão, visão substituir método, utilizar fonte de dados oficiais;
- **Refatoração de método**: consistem em buscar melhorar a qualidade de *stored procedures, function ou trigger,* ou seja, de código criado dentro do próprio banco de dados. Exemplo: adicionar parâmetro;
- **Refatoração de transformação**: são mudanças que alteram a semântica do esquema do banco de dados, adicionando novos recursos a ele, por exemplo a inserção de uma nova coluna a tabela existente. Outros tipos: inserir dados, introduzir nova tabela, introduzir visão, atualizar dados.

Logo, percebemos que a refatoração é uma técnica que, quando bem aplicada, traz muitos benefícios para o *software* e para a equipe de desenvolvimento. Portanto, é fundamental incentivar a refatoração de forma a buscar códigos melhores, mais fáceis de manter e de evoluir.

Construir *software* não é algo trivial e mantê-lo*,* muito menos. São muitas variáveis que podem influenciar a qualidade de um *software*, portanto utilizar processos como o de construção e de manutenção de *software* e o de gerência de configuração é um caminho mais seguro e organizado para gerar *softwares* úteis e com qualidade.

FINALIZANDO

Nesta etapa, discutimos sobre vários assuntos relacionados com a manutenção do *software*, suas necessidades e consequências para as empresas.

Percebemos como a gerência de configuração é o pilar central no suporte ao *software*, garantindo sua organização, qualidade e evolução de maneira correta. É com a gerência de configuração que conseguimos incorporar as evoluções necessárias para manter o *software* atual, agregando valor ao cliente e minimizando o risco de inserir erros indesejáveis no *software*.

Esperamos que você tenha gostado do tema e aprendido como ele é importante no dia a dia de qualquer empresa que precise evoluir seus *softwares* para atender às demandas de mercado e às necessidades de seus clientes.

Continue estudando e se aprofundando nesse assunto, pois ele será muito utilizado ao longo de toda a sua carreira na área de desenvolvimento de manutenção de *softwares*.

Foi bom ter discutido todos esses assuntos com você. Espero que você também tenha gostado. Um abraço e nos vemos por aí, neste nosso grande e interessante mundo de TI.

REFERÊNCIAS

AMBLER, S. W.; SADALAGE, P. J. **Refactoring databases: evolutionary database design**. Boston: Addison-Wesley Professional, 2006.

CAVALCANTE, P. H. A. Entenda o que é refatoração e suas principais técnicas. **Geekhunter**, 2020. Disponível em: <https://bit.ly/3Mehkhp>. Acesso em: 5 out. 2022.

FREE SOFT FOUNDATION. Disponível em: <https://bit.ly/3rwM1F4>. Acesso em: 5 out. 2022.

O PAPEL evolutivo do *software* – Engenharia de *software* 28. **DevMedia**, 2010. Disponível em: <https://www.devmedia.com.br/o-papel-evolutivo-do-software-engenharia-de-software-28/17961>. Acesso em: 5 out. 2022.

PFLEEGER, S. L. **Engenharia de *software***: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

REENGENHARIA de *software* orientado a objetos – Engenharia de *software* 33. **DevMedia**, 2011. Disponível em: <https://www.devmedia.com.br/reengenharia-de-software-orientado-a-objetos-engenharia-de-software-33/19304>. Acesso em: 5 out. 2022.

SBROCCO, J. H. T. C. **Metodologias ágeis**: engenharia de software sob medida. São Paulo: Érica, 2012.

SOMMERVILLE, I. **Engenharia de *software***. 8. ed. São Paulo: Addison Wesley, 2007.

_____. **Engenharia de *software*.** 10. ed. São Paulo: Pearson, 2019.