FullCycle

Full Cycle

Wesley Willians

Esse livro está à venda em http://leanpub.com/fullcycle

Essa versão foi publicada em 2021-12-24

Esse é um livro Leanpub. A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. Publicação Lean é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

Conteúdo

Introdução

A mudança de perspectiva

Foi em 2015 quando comecei ouvir com frequência o termo microsserviços. A cada palestra e apresentação de cases de grandes empresas, ficava mais evidente de que essa tendência iria perdurar por algum tempo.

Grandes empresas e unicórnios precisavam crescer rapidamente, gerar mais valor a cada dia, contratar mais pessoal e ter mais independência entre seus projetos. Por outro lado, quanto mais ouvia falar sobre microsserviços, também mais era evidente que utilizar tal arquitetura não era trivial. Muitas peças ainda precisavam se encaixar para que esse modelo pudesse se tornar algo mais *natural* nas organizações.

A complexidade de arquitetar, desenvolver, testar, realizar o deploy e monitorar uma única aplicação estava sendo multiplicada pelo número de microsserviços que cada empresa possuia. Apesar de muitos projetos serem pequenos, ainda assim, todos precisavam passar por essas etapas.

Com o número de sistemas crescendo exponencialmente, a área de operações dessas empresas também começou a colapsar. Profissionais que estavam acos-

tumados a receber demandas de devs para realizarem quatro deploys diários, tiveram suas rotinas alteradas para realizar quarenta ou quatrocentos.

O número de aplicações a serem monitoradas também foi se multiplicando, e conflitos entre pessoas desenvolvedoras e sysadmins se intensificaram. Estava muito claro que já estávamos em uma nova era. Uma era que não tinha mais volta.

Full Cycle Developers @Netflix

Em 17 de maio de 2018, alguns profissionais da Netflix que já possuíam anos de casa, também compartilharam suas dores e tentativas que vinham realizando desde 2012.

Naquela época, não muito diferente de outras organizações, eles possuíam papéis extremamente bem definidos para o ciclo de desenvolvimento de um software.

O número de aplicações a serem monitoradas também foi se multiplicando, e conflitos entre pessoas desenvolvedoras e sysadmins se intensificaram. Estava muito claro que já estavamos em uma nova era. Uma era que não tinha mais volta.

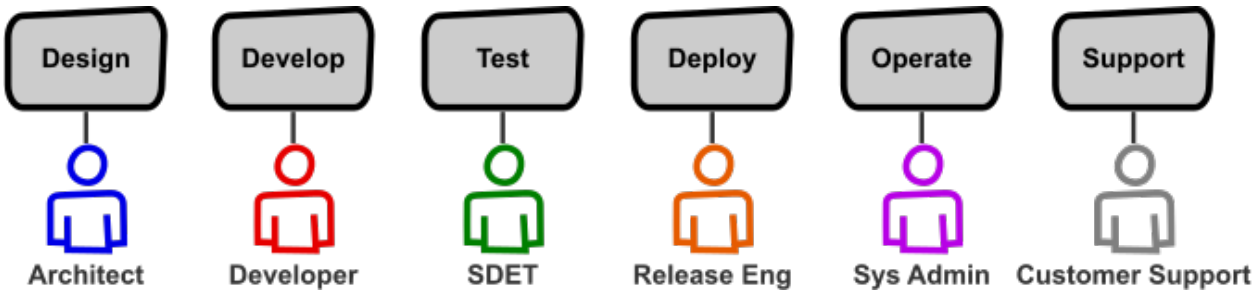

Imagem retirada do Tech Blog da Netflix.

A Netflix deixou de ser apenas um arquivo "war" e também foi dividida em microsserviços.

Depois de muitos erros e acertos, perceberam que um dos principais pontos que sem dúvidas mudaria o jogo, seria transferir totalmente a responsabilidade de cada projeto para seus times de desenvolvimento, ou seja: agora os desenvolvedores fariam parte de todo o ciclo de desenvolvimento de suas aplicações. Da arquitetura ao deploy e monitoramento.

O grande lema se tornou: *"Operate what you build"*, ou opere o que você mesmo constrói. O raciocínio foi remover intermediários de todo processo e fazer com que a equipe de dev fique 100% responsável por seu microsserviço sendo capaz de trabalhar com *feedbacks* curtos de todo processo e aprender rapidamente com isso.

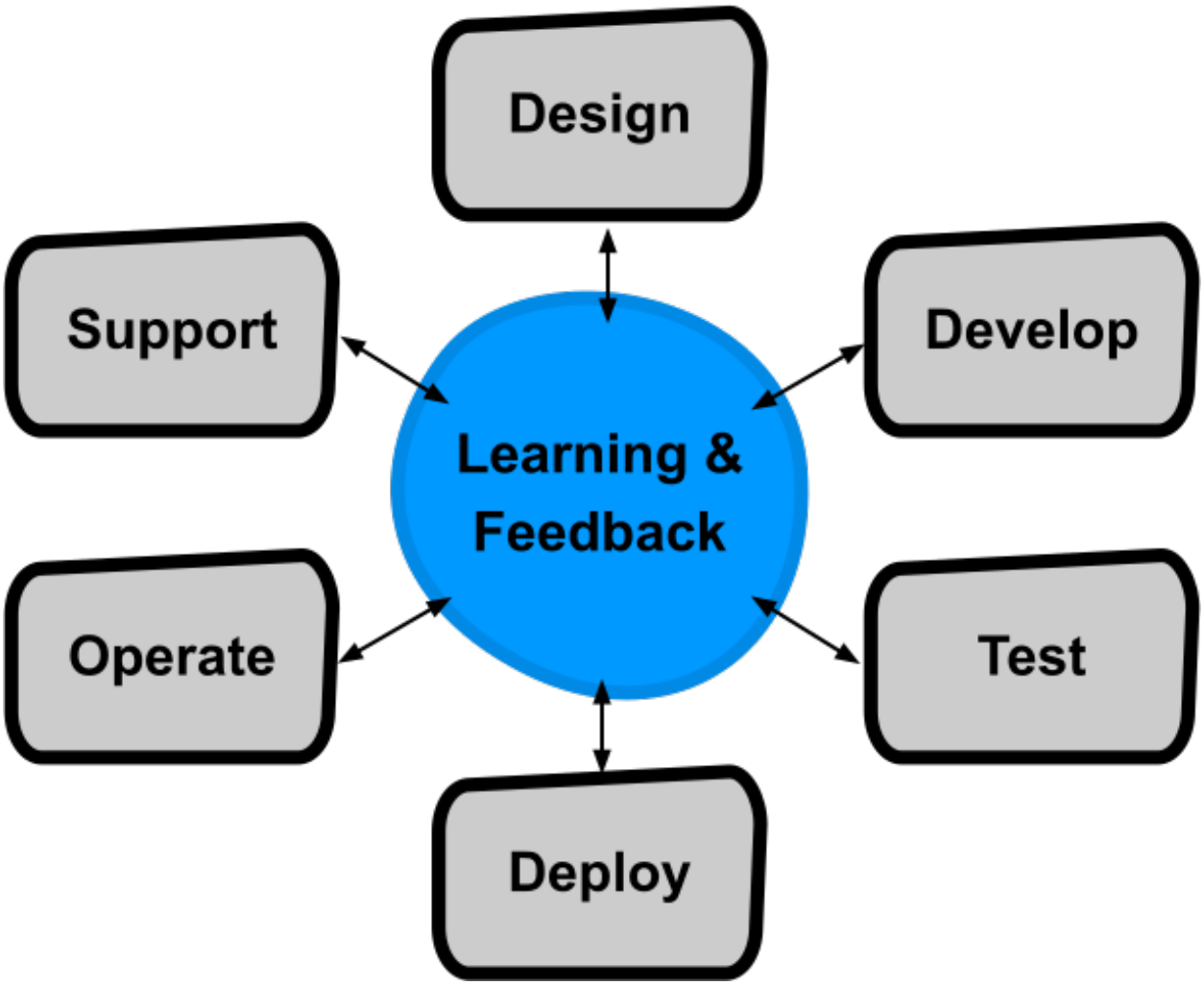

Imagem retirada do Tech Blog da Netflix.

Devs com muitas responsabilidades

Se você é uma pessoa desenvolvedora, com certeza já sabe o número imenso de responsabilidades que possui no dia a dia para entregar software com alta qualidade.

A grande questão é: colocar mais responsabilidades nas “costas” da pessoa

desenvolvedora é realmente a solução?

Foi exatamente essa pergunta que fiz para mim mesmo assim que estava na metade do artigo. Porém, ao continuar com a leitura percebi que os autores deram uma solução para minimizar esse fardo e ao mesmo empoderar devs cansados e estressados com tantos conflitos que resultavam em idas e vindas junto a área de operações para solucionar problemas em produção e para colocar suas aplicações no ar.

A solução criada estava na utilização de ferramentas desenvolvidas especificamente para que a pessoa desenvolvedora tivesse total autonomia para que realizar um deploy e entender em tempo real o comportamento de uma aplicação, de forma simples, rápida e que fundamentalmente fizesse parte do fluxo natural de trabalho.

Times de plataforma

Se você atualmente trabalha em alguma empresa que minimamente possui processos de desenvolvimento bem definidos, bem como possui o mínimo de uma cultura DevOps implementada, acredito que tudo que você leu até o momento representa a sua realidade de trabalho atual.

Grande parte das empresas já possui times de *plataforma*, que têm o objetivo de dar suporte e autonomia para as pessoas desenvolvedoras no dia a dia. Esses times criam ferramentas, padronizam pipelines exatamente para que o processo

de desenvolvimento no dia a dia seja fluido e produtivo, sem tirar o foco na entrega de valor pelos devs.

Você é Full Cycle

Se você participa do fluxo completo de uma aplicação, você é um Full Cycle Developer, todavia, como qualquer profissional de tecnologia, a evolução precisa ser constante; e esse é o objetivo desse livro: abranger os principais aspectos do mundo Full Cycle, e te tornar capaz de desenvolver aplicações de grande porte utilizando as tecnologias mais modernas do mercado onde quer que você esteja.

A partir de agora, vamos partir para uma jornada completa do *Software Development Life Cycle*. Vamos partir do mundo da arquitetura de software até o deploy e monitoramento de aplicações de grande porte.

Introdução a Arquitetura de Software

Um dos pontos fundamentais que sem dúvidas diferencia um desenvolvedor no mercado é o quão preparado ele está para desenvolver softwares sustentáveis, ou seja, aquele software que consegue ao longo do tempo gerar valor para uma organização. O software sustentável é flexível e permite grandes mudanças sem muita necessidade de reescrita.

Nesse capítulo gostaria essencialmente de fazer três "provocações" que ajudarão a você entender a real essência da Arquitetura de Software:

1. Qual a fórmula para criarmos softwares sustentáveis?
2. O que realmente é Arquitetura de Software?
3. Qual a relação entre a sustentabilidade de um software e a sua arquitetura?

Sustentabilidade no dia zero

Quando um executivo contrata uma empresa para desenvolver um software é porque ele possui uma "dor" que se resolvida fará com que a companhia chegue

mais próxima de seus objetivos. Por outro lado, o retorno sobre esse investimento muitas vezes pode demorar alguns meses ou eventualmente anos. Software bem feito não é barato e obviamente o mínimo que se espera é que ele se pague com o passar do tempo. O grande ponto, é que na maioria das vezes há a necessidade de fazer com que a solução acompanhe a evolução do negócio, e é exatamente nesse momento que muitas corporações acabam perdendo diversas oportunidades, pois, infelizmente, a cada dia que passa mais difícil é fazer o software evoluir.

Quando as empresas se dão conta de que determinado software chegou em seu limite da evolução, novos projetos de "modernização tecnológica" são criados exatamente para dar conta do que o software "legado" não conseguiu.

Com certeza se você já está no mercado há alguns anos, você já viu algum caso parecido com o citado acima. Agora, a pergunta que não quer calar: O software parou de evoluir realmente porque a tecnologia evoluiu MUITO ou porque a forma com que ele foi criado tornou sua evolução cada vez mais caótica?

Sem dúvidas a tecnologia evolui. Novas linguagens, frameworks, bibliotecas são criadas todos os dias, porém, em muitos casos, a forma com que um software é criado tem um peso infinitamente superior do que a tecnologia utilizada pelo mesmo.

O software precisa nascer sustentável e continuar em produção pelo maior tempo possível. Quanto mais tempo ele fica no ar, mais retorno ele gera para empresa e isso só é possível se sua base for bem feita. Só é possível se no

dia zero cada desenvolvedor, arquiteto, product owner, entre outros, de forma intencional, pensarem como fazer com que esse software fique no ar e entregue valor por diversos anos.

Obviamente o software precisa ser bem arquitetado, porém a grande pergunta é: O que realmente é arquitetura de software?

O que é Arquitetura de Software

Apesar de não haver apenas uma resposta válida sobre o que é arquitetura de software, podemos dizer que arquitetura software é a relação entre os objetivos de negócio e suas restrições com os componentes a serem criados e suas responsabilidades visando sua evolução.

> "É a organização fundamental de um sistema e seus componentes, suas relações, seu ambiente, bem como os princípios que guiam seu design e evolução." (IEEE Standard 1471)

O processo de arquitetar um software estabelece que o que está sendo desenvolvido faça parte de um conjunto maior.

Pilares da Arquitetura de Software

Agora que temos um ponto de partida sobre o que é arquitetura de software, é importante que consigamos separar alguns de seus conceitos em pilares para

melhor entendimento.

Organização

Quando falamos em arquitetura, falamos em organizar um sistema (não o software em si) que possibilite a fácil componentização, evolução, bem como um fluxo rico para que possamos atender os objetivos de negócio gerando um produto para o cliente final.

Componentização

Você não pode pensar em arquitetura de software deixando de lado seus componentes, pois é através da junção desses componentes que nós conseguimos atingir os atributos de qualidade do sistema.

Nós devemos pensar em todo o ecossistema que existe em torno do processo que nós pretendemos fazer até o resultado final, já que esses componentes serão usados em diversos momentos do nosso trabalho.

Grande parte do trabalho no mundo da arquitetura de software envolve dominar as formas de realizar a componentização dos processos para que eles operem com eficácia, evitando o retrabalho.

Relacionamento entre sistemas

Nenhum software é uma ilha.

A integração entre de softwares a cada dia é mais comum. Logo, o software e seus componentes precisam estar preparados para que eles consigam fazer parte de uma cadeia de integração entre sistemas dentro de um processo maior.

Governança

Principalmente nas grandes empresas a vasta quantidade de sistemas e integrações são cada vez maiores. Muitas pessoas veem governança como uma forma de burocratizar o processo de desenvolvimento; por outro lado, na grande maioria das vezes, o software a ser desenvolvido é apenas mais um dentre as centenas de outros que já estão em operação. Nesse ponto, ter controle, guidelines, padrão no processo de documentação, entre outros, torna-se extremamente necessário para o crescimento organizado entre todo o ecossistema da organização.

Na governança também constam os times, tecnologias, banco de dados e as integrações. São os procedimentos que são utilizados para definir toda a solução que será provida. Essa parte engloba diversas definições que os arquitetos de software e de soluções encontram em suas abordagens.

Sustentabilidade vs Arquitetura

Começamos esse capítulo falando da real necessidade de termos softwares sustentáveis ao longo do tempo para que eles possam gerar valor para a

organização tendo assim possibilidade de se pagar e ainda contribuir de forma direta ou indireta para o crescimento negócio.

Sem dúvidas, uma das formas para fazer com que isso ocorra é trabalhar expressivamente no processo arquitetural. O software que é gerado sem visão de futuro está fadado a existir apenas no curto prazo, logo, há uma grande probabilidade de que a iniciativa que deveria melhorar os resultados da organização possa acabar tendo efeito reverso.

Trabalhar com arquitetura de software de forma intencional pode definir o sucesso ou o fracasso do projeto.

Lembre-se: o sucesso de um projeto de software não está na primeira entrega, mas sim nas evoluções subsequentes com o passar dos anos.

Arquitetura vs Design de Software

É muito comum ver profissionais substituírem a palavra arquitetura por design de software. A grande pergunta é: Arquitetura e design de software representam a mesma coisa?

A resposta é não! A arquitetura de software trata de conseguirmos alinhar os objetivos do negócio ao projeto a ser entregue. Decisões estritamente arquiteturais podem impactar desde a escolha entre Cloud Providers até mesmo a divisão de times e a quantidade de sistemas que uma organização precisará.

Decisões que impactam diretamente em quais componentes e *vendors* que um projeto utilizará são decisões arquiteturais.

Exemplo: Para que os logs de um sistema possam ser centralizados e facilmente recuperados, a Elastic Stack será utilizada.

Podemos perceber que tal decisão afetará a aplicação como um todo, além da possível contratação de mais infraestrutura para que a Elastic Stack seja instalada, ou mesmo, uma eventual contratação de um serviço gerenciado na Elastic Cloud.

Decisões como essa podem afetar todos os sistemas de uma ogranização, seu orçamento, o tempo em que um possível bug pode ser corrigido, a forma com que cada time trabalhará no dia a dia com observabilidade, além do conhecimento basico na stack que será requirida por cada desenvolvedor e eventuais treinamentos que os mesmos deverão receber para operar tais ferramentas.

Por outro lado, quando tomamos decisões de quais patterns GoF (Gang of Four) o projeto utilizará; SOLID, DRY, Clean Code, a quantidade de camadas de uma aplicação, estamos nos referindo fundamentalmente ao design do software.

O design de software possui seu escopo voltado as boas práticas no processo de desenvolvimento. Ele define diretamente padrões que normalmente são refletidos em linha de código, garantindo sua fácil manutenção, extensibilidade e evolução.

Muitas vezes decisões de design impactam diretamente em decisões arquite-

turais, como no exemplo citado sobre a Elastic Stack. Se uma aplicação foi desenhada com o intuito de centralizar seus logs e utilizará a Elastic Stack, tal decisão afetará o negócio, uma vez que tal ferramenta auxiliará no processo de recuperação de erros do software afetando assim sua disponibilidade.

O papel do Arquiteto(a) de software

Atualmente com a grande tendência de que pessoas desenvolvedoras sejam full cycle e de que as equipes sejam cada vez menores e com responsabilidades claras no processo de criação e manutenção de seus microsserviços, muitas organizações acabam questionando a necessidade da existência de um arquiteto ou arquiteta de software.

Apesar de ser cada vez mais natural de que as equipes tenham autonomia para tomar decisões de design e arquitetura, é muito evidente de a grande pressão do dia a dia faça que com o software tome caminhos arquiteturais diferentes do que se havia planejado para garantir sua sustentabilidade, bem como na garantia do atendimento dos atributos de negócio; por conta disso, um arquiteto ou arquiteta de software deve se fazer presente nos projetos.

Perfil profissional

Um arquiteto ou arquiteta normalmente possui uma grande bagagem como pessoa desenvolvedora e comumente é considerada *sênior*. Apesar de ter um

grande conhecimento técnico, esse tipo de profissional tem facilidade em “furar a bolha da techniquês” para entender melhor o negócio e o papel da solução tecnológica.

Entender a estrutura da organização, suas prioridades, disponibilidade financeira e do conhecimento do perfil profissional das equipes envolvidas nos projetos, o arquiteto(a) precisa ter a habilidade de balancear o que é melhor para a companhia naquele exato contexto.

Muitas vezes esse tipo de profissional pode ser extremamente criticado(a) pelas decisões tomadas, principalmente pela opção da não utilização das tecnologias mais modernas em determinados momentos, ou mesmo por decisões impopulares que se fazem necessárias.

É cada vez mais usual vermos arquiteto(a)s tendo a função de intermediar o processo de comunicação e conhecimento dos projetos entre diversos times e *stakeholders* diminuindo assim o ruído entre as partes para então viabilizar a implementação de decisões arquiteturais.

Um arquiteto(a) por time

Em muitos times e organizações não vemos formalmente o cargo ou título de um arquiteto(a). Em contrapartida, por conta de experiências anteriores, pessoas desenvolvedoras atribuídas em cargos de liderança técnica muitas vezes acabam tomando decisões arquiteturais fazendo assim, informalmente, o papel

de arquiteto(a).

Área de arquitetura nas organizações

Algumas empresas, principalmente as que possuem muitas equipes e projetos em andamento, preferem criar um departamento especifico de arquitetura de software. Os profissionais dessa área ficam responsáveis por revisar, aprovar e auxiliar as equipes e projetos a tomarem as melhores decisões alinhadas ao contexto atual da organização.

Além disso, a área de arquitetura monitora, gerencia os processos de mudança e garante a governança reforçando assim um padrão de qualidade em todos os projetos.

Apesar de muitos entenderem que áreas como essa podem burocratizar o processo de desenvolvimento e retirar a liberdade dos devs, é evidente que quando há uma grande quantidade de equipes, sistemas e tecnologias envolvidas é imperativo que haja controle para garantir a sustentação de todo ecossistema a médio e longo prazo.

Sistemas Monolíticos

Sistemas “tradicionais”

Nada melhor de que exemplos concretos para facilitar entendimento quando falamos em desenvolvimento de sistemas, logo, para começarmos esse capítulo, imagine um sistema de gerenciamento de produtos onde tais produtos podem ser categorizados e disponibilizados para venda. Também tal sistema permite que o usuário(a) possua uma área de busca, bem como o recurso de checkout para realizar a compra.

Com isso em mente, já é possível termos ideia de outras áreas que poderiam existir dentro desse sistema, como um catálogo para exibição dos produtos, evoluindo assim para uma loja virtual.

Podemos utilizar plataformas para desenvolver tal loja como um *Magento* por exemplo.

Essa é uma abordagem super comum principalmente devido a evolução de ferramentas, CMSs e frameworks. Todos os recursos fazem parte de uma mesma aplicação.

Esse é o clássico exemplo de um sistema monolítico, onde todas as operações acontecem em sua própria estrutura utilizando normalmente uma única linguagem de programação. Nele, são incorporadas todas as entidades, regras de negócio, disponibilização de APIs e integrações. Obviamente tudo está fortemente acoplado em um único sistema que normalmente possui mais de uma responsabilidade.

Restrições

Pela própria natureza dos sistemas monolíticos, normalmente eles utilizam uma única linguagem de programação. Por exemplo, se nós temos um sistema monolítico feito em Ruby, é improvável tenhamos alguma coisa escrita em PHP ou Python.

Em muitos casos essa possível "limitação" de utilizar uma única tecnologia dentro de um sistema pode fazer com que a organização não tire proveito de tecnologias, que para determinados casos de uso, sejam mais eficientes trazendo assim mais valor para o negócio como um todo.

Além disso, temos que concordar que ao colocarmos, cem, duzentas, ou mesmo mil pessoas para trabalhar na mesma base de código pode ser em determinadas situações algo caótico.

Monolitos não são ruins

Apesar das restrições citadas acima, trabalhar com sistemas monolíticos não é nenhum demérito ou atestado de obsolescência.

Sistemas monolíticos na maioria das vezes sem dúvidas é a melhor opção para grande parte das empresas. Afinal de contas, nem toda empresa possui 6000 devs como Mercado Livre.

Trabalhar com sistemas monolíticos reduz a complexidade e aumenta a eficiência principalmente em situações em que as próprias regras de negócio da empresa estão em constante mudança. Startups, empresas fazendo validações de modelos de negócio, bem como corporações que não fazem o uso "pesado" de tecnologia como principal ponto de sustentação, são grandes candidatas a trabalharem com sistemas monolíticos.

Via de regra, "todo" sistema deve começar com um monolito.

Martin Fowler em seu artigo MonolithFirst[1] faz duas grandes observações logo no início:

1. "Quase todas as histórias de sucesso de sistemas utilizando microsserviços começaram com um monolito que ficou grande e depois foi quebrado em partes."

[1] https://martinfowler.com/bliki/MonolithFirst.html

2. “Todos os casos que ouvi de sistemas que começaram diretamente utilizando microsserviços tiveram grandes problemas”

Para que uma companhia inicie o processo de desenvolvimento de seus sistemas utilizando uma abordagem diferente da monolítica, sem dúvidas os profissionais ali inseridos devem ter muitas cicatrizes com experiências de sucesso anteriores, caso contrário, há uma grande probabilidade do projeto fracassar.

Deploy

Sistemas monolíticos possuem uma característica singular quando precisam ir ao ar. Como tudo encontra-se dentro da mesma estrutura, por mais simples que seja determinada mudança, o deploy de 100% da aplicação precisa ser realizado.

É evidente que quando todas as áreas de uma empresa estão concentradas em um único sistema, o risco de interrupção de todas as áreas aumenta a cada deploy realizado. Em contrapartida, complexidades de comunicação entre diversos sistemas, a necessidade de criação de uma grande quantidade de pipelines de entrega, bem como todos os aspectos comuns em entregar e gerir projetos em produção são reduzidos drasticamente.

Necessidade de escala

Quando trabalhamos com grandes sistemas, invariavelmente teremos a necessidade de escala.

Segundo a definição da Gartner[2]:

"Escalabilidade é a medida da capacidade de um sistema de aumentar ou diminuir o desempenho e o custo em resposta às mudanças nas demandas de seus aplicativos e processamento."

Nesse ponto, é evidente de que quando temos a necessidade de escalar um sistema monolítico, não há a possibilidade de escalarmos apenas as funcionalidades que naquele momento estão exigindo mais recursos computacionais, ou seja, todo sistema precisa escalar.

Por exemplo, se um ecommerce receber uma grande quantidade de acessos em seu catálogo de produtos e por consequência precisar de mais recursos computacionais para segurar a carga, todas as outras funcionalidades, que muitas vezes não precisariam ser escaladas, terão de escalar, pois tudo faz parte de um único conjunto.

Em determinados casos essa necessidade pontual de escala por um recurso pode fazer com que os custos com infraestrutura sejam elevados, pois claramente há uma ineficiência técnica embutida no processo.

[2] https://www.gartner.com/en/information-technology/glossary/scalability

Em contrapartida, em determinadas situações, não é porque um sistema precisa ser escalado por inteiro que seus custos serão mais elevados quando trabalhamos com outro tipo de arquitetura, como a de microsserviços por exemplo. Temos que ter em mente que quando trabalhamos com arquiteturas distribuídas, há também outros custos inerentes a essa modalidade.

Débitos técnicos

É evidente que qualquer tipo de aplicação, independente da arquitetura a ser utilizada em algum momento terá débitos técnicos, ou seja, pequenas melhorias, refatorações, implementações que deveriam ser realizadas e que foram "deixadas para depois".

A medida em que os débitos técnicos se acumulam ao longo do tempo, a instabilidade do sistema como um todo acaba aumentando, gerando assim uma queda considerável na produtividade da manutenção e implementação de novos recursos.

Quando trabalhamos com sistemas monolíticos, isto acaba sendo potencializado, uma vez que há um alto acoplamento em todos os componentes da aplicação.

Por outro lado, isso não significa que pequenos sistemas não possuem ou não possuirão débitos técnicos. Todavia, como a base de código é limitada, tais débitos não terão tanta influência na solução como um todo.

Domain Driven Design

Introdução

O design orientado ao domínio, também conhecido como DDD (Domain Driven Design) é uma abordagem que trabalha com práticas de design e desenvolvimento, oferecendo ferramentas de modelagem tática e estratégica para entregar um software de alta qualidade, acelerando o seu desenvolvimento e garantindo sua sustentabilidade.

Alguns desenvolvedores acreditam que o DDD se resume a apenas uma série de design patterns como agregados e repositórios, criando uma pasta de infra para separar as camadas do nosso projeto, entre outros. Outros já veem o DDD com mais abrangência, mas, no fim das contas, também não conseguem explicar claramente como isso funciona.

Isso tudo é muito estranho porque de fato o DDD parece complexo quando nós consultamos as principais literaturas a respeito dele. E mesmo quando pesquisamos, isso ainda nos deixa dúvidas sobre como nós podemos colocar isso em prática no nosso dia a dia.

Nesse capítulo não vamos focar apenas nos aspectos práticos porque o DDD vai

muito além disso. O seu foco é conhecer não apenas o ambiente, mas também os contextos e as pessoas que trabalham em um projeto. E ainda, baseado nisso, permitir uma separação que faça sentido para a organização em si.

Não basta começar diversos projetos e dizer que estamos aplicando o DDD em tudo, sendo que no final das contas isso resulta em diversas pastas repetidas em diversos projetos.

O intuito deste capítulo é fazer com que essa filosofia de trabalho mude a sua forma de pensar em software, principalmente no seu trabalho com projetos de grande porte. Normalmente não aplicamos isso em pequenos projetos porque o DDD é fundamentalmente utilizado quando nós não temos clareza total do projeto e suas áreas.

Uma palavra que resume bem o DDD é clareza. A clareza em um projeto minimiza seus riscos em diversas perspectivas, principalmente quando o mesmo deve perdurar por anos. Nós não podemos fazer um software que 6 meses se tornará o famoso “legado” e que nenhuma pessoa desenvolvedora irá querer mantê-lo.

O DDD é sem dúvidas um recurso que pode nos ajudar com esse objetivo e, com a sua devida aplicação, desenvolver software se tornará mais divertido e com menos riscos.

Ponto de partida no DDD

Agora iremos explorar a filosofia e os conceitos teóricos que estão por trás do DDD, considerando que ao termos mais entendimento desses pontos, facilitará o processo de aplicar DDD na prática.

Como o próprio nome sugere, o Domain Driven Design se refere a como podemos desenhar o software guiado ao domínio, que é o coração da aplicação.

Não pense apenas sobre os design patterns, pastas dentro do seu projeto, entre outros, porque o DDD foca muito mais em como modelar o software do que desenvolvê-lo em si.

O DDD é uma forma de desenvolver o software focando no coração da aplicação, o que nós chamamos de domínio. Seu objetivo é entender os contextos e regras do projeto, seus os procedimentos e complexidades, separando-as de outros pontos complexos que são adicionados durante o processo de desenvolvimento.

O DDD surgiu de um autor chamado Eric Evans, de seu livro publicado em meados de 2003.

É importante termos isso em mente porque o DDD é um assunto que oscila muito durante os anos. Ora é muito falado, ora não. Atualmente, com a importância dos microsserviços, o DDD também tem destaque porque o grande desafio de trabalhar com microsserviços é modelar o software e os seus contextos.

Quando observamos o livro de Eric Evans, é notável que existe um lado filosófico em torno dele, que chega ser até mais importante do que os padrões de projeto que utilizamos em nosso dia a dia.

Essa filosofia parte de uma visão madura para que o desenvolvedor trabalhe em seu projeto com orientação de trazer soluções para problemas complexos. Não podemos ser inocentes ao ponto de ver um projeto e pensar somente no banco de dados, cadastros, CRUDs, entre outros.

Trabalhar em torno dessa filosofia esclarece o quão é importante é entender e modelar um software baseado em suas complexidades de negócio.

Foram através desses conceitos iniciais de Evans que surgiram entusiastas que tiveram mais clareza nas falhas que existiam em seus grandes projetos.

Depois do lançamento do livro de “capa azul” (como assim é conhecido o livro de Evans), houve uma série de lançamentos de outros livros que foram imprescindíveis para que pudéssemos nos aprofundar na filosofia em torno do DDD.

Entre os mais destacados está o livro de “capa vermelha” de um autor chamado Vaughn Vernon. Este segundo livro é um pouco mais prático em relação ao livro de Eric Evans.

Outro livro de Vernon para se aprofundar é o Domain Driven Design Distilled. Ele é uma obra mais recente em relação as outras contendo um resumo sobre o DDD. É um livro para se ler com cautela pois ele deixa algumas partes

importantes de fora.

A vantagem de ler esse livro são os tópicos que vão direto ao ponto, que desmistificam grande parte do DDD de uma forma menos densa.

As complexidades de um software

Normalmente consideramos aplicar o DDD em projetos de software mais complexos. Não faz sentido usar essa abordagem em um sistema típico de estabelecimentos de pequeno porte, como uma “padaria”, que só vai vender um único produto e receber o troco.

A complexidade de um software como a do exemplo anterior é tão pequena que não há quase o que modelar. É um nível de complexidade tão simples que boa parte dos softwares de empresas desse tipo são softwares de prateleira.

São softwares que podem ser adaptados a qualquer tipo de negócio sem customização.

O DDD é normalmente utilizado quando temos problemas maiores que nos impedem de termos a clareza de como as áreas e pessoas de um projeto se relacionam e se comunicam.

Quando trabalhamos com pessoas de diferentes departamentos, por exemplo, nós percebemos que elas usam termos completamente diferentes entre as suas respectivas áreas.

O DDD nos deixa claro de que em grandes projetos há muitas áreas, regras de negócio e pessoas com diferentes visões da organização que estão situadas em diferentes contextos.

Vamos usar como exemplo uma empresa que seu “core business” é fazer cobranças de contas em aberto em nome de diversas corporações. Essa operação envolve atendentes de telemarketing que usam um software de discagem automática. Se pensarmos bem, com certeza existe um diferencial na automatização desses processos de cobrança em relação ao de uma empresa “tradicional”, que liga para cobrar boletos bancários em aberto de seus clientes.

Isso acontece porque o coração do negócio da empresa de telemarketing é a cobrança. Cobrar é a razão da empresa existir, logo, a modelagem de um problema de cobrança nesse caso torna-se muito mais complexo pois provavelmente envolve seu diferencial competitivo no mercado.

Também, normalmente quando tratamos de empresas e seus diversos departamentos, esses departamentos possuem a sua própria forma de se “expressar” e falar do negócio utilizando certos jargões no dia a dia.

Os bancários, por exemplo, podem usar o termo ‘francesinha’, que é o nome que eles dão para um tipo de relatório de quem realizou pagamentos. Mas quando esse termo é mencionado para funcionários de outro departamento, isso pode não fazer sentido nenhum.

Quando percebemos isso, é possível ter mais clareza para entender que o

software não é apenas uma simples unidade. Ele é feito de contextos, regras, implementações que possuem objetivos diferentes.

Muitas vezes os softwares são independentes, sejam eles microsserviços ou sistemas monolíticos. No fim das contas o software é vivo porque ele é movido a pessoas que atuam em diferentes contextos e entender isso é crucial durante a criação de uma modelagem para cada tipo de contexto.

Não é possível deixar de utilizar técnicas avançadas em projetos de alta complexidade porque não podemos tratar de um software grande, que tem diversos departamentos, complexidades e regras, de uma forma simplória. O software precisa se adaptar a organização e não a organização ao software.

Em torno de uma solução há política, pessoal e cultura. Tudo isso deve ser levado em consideração. Se não levarmos isso em conta, sem dúvidas o projeto já fracassou em seu primeiro dia de desenvolvimento.

Seja qual for a sua experiência trabalhando em grandes projetos, você já deve ter visto um projeto falhar devido a alguns pontos que foram citados até aqui.

É notável que grande parte da complexidade desse tipo de software não vem da tecnologia; mas da comunicação e separação de contextos que envolvem o negócio por diversos ângulos.

Nós perguntamos como as coisas devem ser feitas, seguindo as instruções que nos foram dadas. E de repente o responsável pelo produto diz que quer um resultado diferente.

Frequentemente alguns desenvolvedores são inocentes e recebem o feedback do cliente dizendo que está tudo em ordem com o software, mas em outras reuniões eles levam o que foi solicitado e descobrem que outro membro da equipe não concorda com o que foi feito.

Como o DDD pode te ajudar

De forma geral o Domain Driven Design vai te ajudar a ter uma visão ampla do problema a ser resolvido e a quebra-lo em problemas menores. Também ele te dará técnicas de como minimizar ruídos de comunicação entre todos os envolvidos, bem como trabalharmos com patterns que visam deixar nossas aplicações cada vez mais desacopladas preservando ao máximo suas regras de negócio.

Para que o DDD te ajude a entender os principais desafios de como desenvolver software e suas complexidades geradas em torno de uma organização, é importante que entendamos os pontos abaixo. Lembrando que todos eles serão aprofundados durante nosso capítulo sobre DDD.

Domínio

O domínio, ou “domain”, é a razão pelo qual o software existe. Em outras palavras, é o coração do software. E esse conceito por si já define que nós temos um desafio para resolver.

Se não entendermos o real objetivo pelo qual o software será desenvolvido, a batalha já está perdida. Normalmente quando recebemos o "problema" do cliente, tal problema é apenas a ponta do iceberg. Precisamos mergulhar a fundo, entender mais sobre a organização, suas áreas, e as intenções reais por de trás da solução a ser desenvolvida.

Subdomínios

O grande desafio é que normalmente o problema a ser resolvido é muito grande, e nesse ponto, de forma inevitável, precisamos dividi-lo em partes menores. Essas partes são chamadas de subdomínios. Diferente do domínio, que acaba se tornando uma visão geral do problema, os subdomínios acabam cobrindo detalhes finos que nos ajudam a ter um pouco mais de clareza sobre o coração da aplicação.

Linguagem universal

O DDD também estabelece uma linguagem universal entre todos os que estão envolvidos no projeto. A "Ubiquitous Language", ou Linguagem Ubíqua, é um termo recorrente em qualquer livro sobre DDD.

Vamos pensar no exemplo da francesinha na área bancária. Não adianta criarmos um sistema com o menu chamado "relatório de contas pagas", pois todos naquele departamento ficarão procurando por "francesinha", considerando que ela segue um jargão popular naquele contexto.

Todo esse problema ocorre porque nós não conseguimos ter uma única linguagem universal dentro da empresa. Por mais estranho que pareça, você vai perceber que a empresa é composta por uma cultura que é modificada e adaptada aos poucos dentro de cada departamento.

E como cada área tem o seu próprio jargão, para um funcionário na área de vendas pode existir um cadastro de clientes que fecharam contratos com ele. Já no departamento de de compras também existe uma área para cadastro de clientes, porém nesse caso o cliente é a própria empresa, pois ela possui diversas filiais.

Perceba que em um departamento o cliente é chamado de uma forma, mas dentro de um setor diferente a palavra “cliente” já possui outro significado. Esses departamentos podem usar a mesma palavra e, caso se comuniquem, o cliente vai representar uma coisa completamente diferente.

Por isso é importante entender, mapear e extrair essa linguagem universal para esclarecer e minimizar os principais ruídos de comunicação.

Design estratégico e tático

Um dos objetivos do DDD é nos ajudar a criarmos o design estratégico e tático para a modelagem de nossas aplicações.

Isso significa que através da clareza da existência de um domínio e seus diversos subdomínios, podemos criar a modelagem estratégica delimitando contextos

e seus relacionamentos. Normalmente tal modelagem é chamada de “Context Map” ou mapa de contexto.

Além disso, precisamos de um design tático para mapearmos agregados, entidades, objetos de valor, entre outros, facilitando assim o processo de codificação do sistema.

Domínios, subdomínios e contextos

Agora que tivemos uma idéia geral sobre DDD, a seguir entenderemos os princípios básicos relacionados aos domínios e subdomínios como elementos fundamentais do Domain Driven Design.

Delimitação

O domínio é parte essencial do DDD, considerando que o design guiado ao domínio se refere objetivamente ao coração da aplicação. Entender o domínio é ter entendimento sobre as áreas que envolvem o negócio; e assim que reconhecemos isso nós podemos dividir tal domínio em partes menores, conhecidas como subdomínios.

Delimitar um domínio nos possibilita pensar na solução em torno de toda a complexidade do negócio. Existem, porém, diversos tipos de problemas e complexidades que são partes importantes para o software.

Na própria literatura do DDD existe um exemplo que trata a exploração do nosso domínio como se estivéssemos entrando num quarto escuro segurando apenas uma lanterna. Nós só conseguimos enxergar algumas partes do cômodo quando ligamos a lanterna. O mesmo acontece quando começamos a explorar os nossos domínios e subdomínios porque inicialmente nós não temos a visão de um todo.

E é assim que conseguimos ver, nós percebemos que existem partes que nós podemos separar e é por isso que essas partes são chamadas de subdomínios. Mas na separação dos subdomínios também percebemos que eles possuem graus diferentes de importância para o negócio.

Domínio principal ou “Core Domain”

Quando identificamos e separamos a parte mais importante desse negócio nós temos então o nosso Core Domain, ou seja, o nosso domínio principal. Caso ele não existisse não haveria sentido para todo o restante existir. Seria como a Netflix sem filmes e séries ou uma fábrica de automóveis que não tem carros.

Por outro ângulo, ainda observando através da nossa lanterna, nós também temos alguns pontos importantes para definir.

O Core Domain é o coração do negócio e também o diferencial competitivo da empresa. Normalmente quando pensamos em domínio, isso também compõe o diferencial de toda a concorrência. Se não houvesse diferencial e toda a complexidade fosse banal, nós simplesmente usaríamos softwares de prateleira.

Domínios de Suporte

Diferente do Core Domain, também existe o que chamamos de domínio de suporte. Eles apoiam o domínio principal no dia a dia e apesar de não ser o domínio principal, eles auxiliam o negócio a possuir seus diferenciais competitivos e garantir que tudo funcione plenamente.

Se tivermos um e-commerce, por exemplo, precisaremos de produtos, uma loja e a parte de checkout. Vamos ter também o centro de distribuição.

Não podemos pensar em e-commerce sem um centro de distribuição, porque, nesse contexto, isso pode ser perfeitamente um dos diferenciais da empresa, pois afeta diretamente na velocidade de entrega dos produtos vendidos. Nesse caso, poderíamos considerar o centro de distribuição como um subdomínio de suporte, pois ele viabiliza a operação do domínio principal.

Domínios Genéricos

Os subdomínios genéricos dão apoio a todo o sistema, mas geralmente não agregam tanto diferencial competitivo para a empresa. E vale mencionar que algumas empresas usam com frequência “softwares de prateleira” como parte de seus subdomínios genéricos.

Contudo, se você observar, esses “softwares de prateleira” ajudam na rotina da empresa, porém normalmente são facilmente substituíveis.