Princípios e Práticas em Arquitetura de Software

Paulo Nascimento

2020

Princípios e Práticas em Arquitetura de Software

Paulo Nascimento

Sumário

Capítulo 1. Introdução e Organização do curso

O curso de Princípios e Práticas em Arquitetura de Software aborda todos os aspectos inerentes ao trabalho de um arquiteto de software, na concepção mais moderna desta atividade. Este curso permite entender a atuação de um arquiteto de software, suas responsabilidades, seus desafios e, tão importante quanto, permite entender que a atuação do arquiteto de software é muito além de um programador que se destaca por sua qualidade técnica. Esta característica talvez seja a que mais assuste os profissionais que assumem cargos de arquiteto de software, pensando que irão dar continuidade a suas atividades de desenvolvimento de software, porém com um nível mais aprofundado de conhecimento.

Pensando neste sentido, propositalmente este curso está organizado em 9 capítulos, sendo que os 6 primeiros capítulos são destinados a aspectos de gestão que devem estar presentes em qualquer profissional que deseja se tornar um arquiteto de software. Os capítulos deste curso são:

1. Introdução aos Princípios e Práticas em Arquitetura de Software;
2. Arquitetura de Software como Estratégia Corporativa;
3. Liderança Arquitetural;
4. Decisões e Riscos;
5. Métodos de Gestão e Desenvolvimento de Software;
6. Estratégias de Gestão de Configuração e Versionamento;
7. Requisitos Arquiteturais;
8. Modelagem Arquitetural;
9. Estilos e Padrões Arquiteturais.

Ao analisar a lista de capítulos apresentada, talvez você esteja sentindo falta de aspectos importantes de arquitetura de software moderna como DevOps, SOA,

Arquitetura MVC, dentre outras. Como citado anteriormente, estas tecnologias realmente são importantes no desenvolvimento moderno de software, mas não é o objetivo deste curso ensiná-las. O que se deseja formar com esse curso são líderes técnicos, que geralmente precisam desenvolver mais os aspectos de liderança do que técnicos. Além disso, tecnologias mudam. O expoente de tecnologia de hoje certamente estará obsoleto em poucos anos. Assim, mais importante que aprender uma tecnologia específica é entender o porquê ela é importante no contexto de desenvolvimento de software, permitindo assim compreender mais facilmente não somente a tecnologia da moda de hoje, mas as que virão no futuro.

É importante salientar que é esperado do público alvo do curso alguns pré-requisitos de conhecimento, importantes para a compreensão do que será abordado. Como são pré-requisitos, os mesmos não serão aprofundados em nenhum capítulo, apenas citados como parte da compreensão de outros assuntos. Os pré-requisitos para completa compreensão deste curso são:

Noções de Governança de TI (ITIL): as principais empresas de tecnologia do país já possuem, em um momento ou outro de sua atuação, alguma disciplina ITIL empregada. O ITIL hoje é a principal ferramenta para direcionar uma boa gestão em Tecnologia da Informação. Em alguns momentos do curso, serão citadas algumas dessas disciplinas como Gestão de Capacidade (Capacity Management), Gestão de Disponibilidade (Availability Management), Gestão de Nível de Serviço (Service Level Management), entre outras. É importante que você tenha noção do que são estes termos e sua importância dentro da governança corporativa. Não é necessário, apesar de desejável para um bom gestor de TI, que se conheça todas as disciplinas ITIL e os artefatos e controles por elas produzidos.

Noções de Gestão: a não ser que você tenha trabalhado por conta própria em toda sua carreira, seja como freelancer ou empreendedor, em algum momento dela você foi gerido por alguém ou geriu alguém ou uma equipe. Entender as nuances referentes a esse trabalho de gestão de pessoal, e os desafios por trás disso, é necessário para entender as ferramentas de gestão que serão apresentadas no decorrer deste curso.

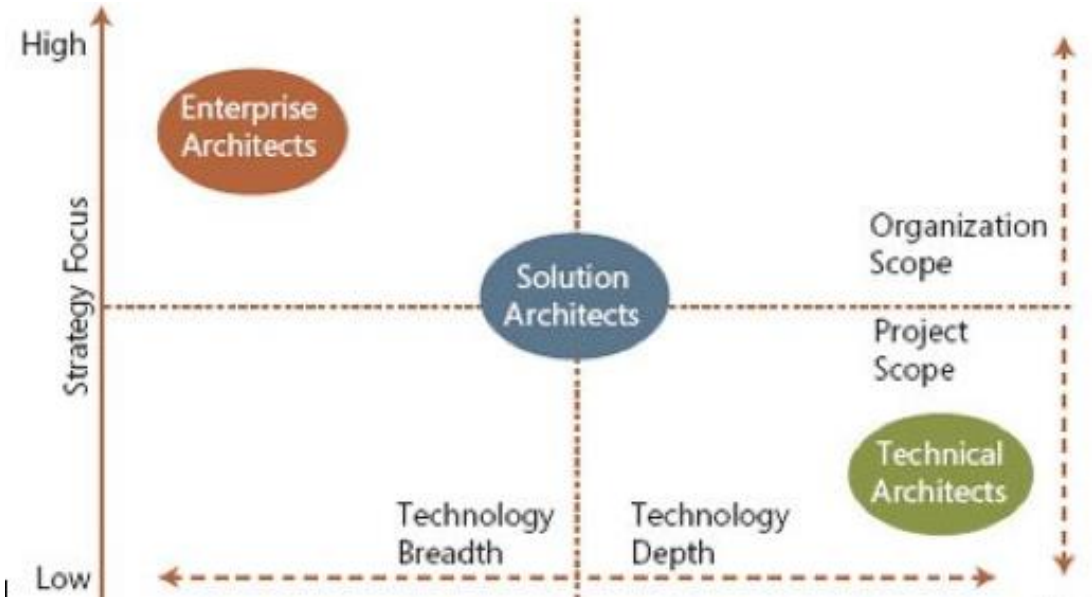
Análise Estruturada / UML: em determinados momentos deste curso serão abordadas questões relativas à Unified Modeling Language – UML. Por se tratar de uma ferramenta bastante consolidada e, relativamente antiga se tratando de TI, espera-se que esse conhecimento já seja inerente de quem está cursando a disciplina. Não é necessário um conhecimento profundo em todas as documentações geradas pela UML, e sim o entendimento do conceito por trás da UML e quais os problemas ela se propõe a resolver.

Inglês: não custa salientar que, em se tratando de um curso de TI, boa parte do material disponível, principalmente se tratando de novidades, está em inglês. Durante o curso foi feito o esforço de traduzir o que é possível para o português, porém, alguns termos simplesmente perdem o sentido quando traduzidos, sendo globalmente entendidos na sua escrita original.

Vamos analisar então o que será visto em cada um dos capítulos a seguir.

Visão do Capítulo 2 – Arquitetura de Software como Estratégia Corporativa

O capítulo 2 é inteiramente focado na Arquitetura Corporativa. Neste modelo são apresentados os três níveis de arquiteto geralmente presentes na área de desenvolvimento: arquiteto técnico, arquiteto de soluções e o arquiteto corporativo.



Neste contexto, a Arquitetura Corporativa é apresentada como uma grande aliada aos processos de desenvolvimento de software, principalmente quando conjugada com boas práticas provenientes do ITIL, por exemplo. Com a adoção da Arquitetura Corporativa, atinge-se esse objetivo através da aproximação da área de desenvolvimento de software, com a visão estratégica da empresa.

Neste capítulo também será abordada a importância de um arquiteto, seja ele um arquiteto técnico, um arquiteto de soluções ou um arquiteto corporativo como líder da equipe, e dependendo do nível de atuação, como direcionador de estratégias envolvendo questões tecnológicas dentro da organização. O principal efeito desta forma de agir e pensar é o tão desejado alinhamento da Tecnologia da Informação com os objetivos estratégicos da organização.

Da mesma forma que serão abordados os tópicos relativos à Arquitetura Corporativa, bem como os aspectos inerentes e necessários para os diferentes níveis de arquitetos de desenvolvimento de software, não serão abordados nesse capítulo assuntos tecnológicos referentes à frameworks ou ferramentas de tecnologia.

Conforme já explicado, os frameworks, padrões, ferramentas tecnológicas, linguagens de programação, etc., mudam constantemente. A nossa área vive um ciclo interessante em relação a esses assuntos. As linguagens mais esquecidas, como C ou R, estão voltando com tudo para o desenvolvimento de complexos sistemas de

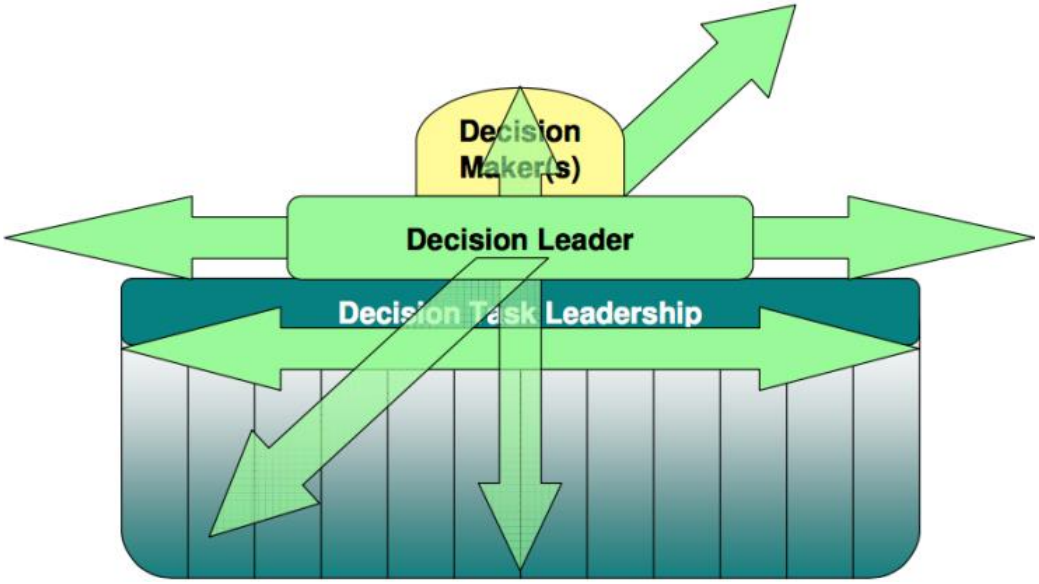
Machine Learning. Da mesma forma, tecnologias da moda, vistas muitas vezes como direcionadoras de novas arquiteturas duram poucos meses. Assim, não faz sentido aprofundar nessas tecnologias mutáveis. Mais importante é entender o contexto destas, possibilitando assim compreender a importância de outras tecnologias que surgirem.

Espera-se que o aluno, ao final desse capítulo, seja capaz de:

- Entender a evolução da carreira do arquiteto de softwares e suas principais responsabilidades.
- Entender o arquiteto corporativo com atribuições distintas de apenas um bom desenvolvedor técnico.
- Entender como a Arquitetura de Software pode contribuir para que a organização atinja seus objetivos estratégicos com maior eficiência.

Visão do Capítulo 3 – Liderança Arquitetural

O papel do arquiteto de software, como já falado, vai muito além do conhecimento puramente técnico de desenvolvimento de software. Espera-se deste arquiteto um papel de liderança de equipe, que muitas vezes é negligenciado. O capítulo 3 irá tratar destes aspectos de liderança.



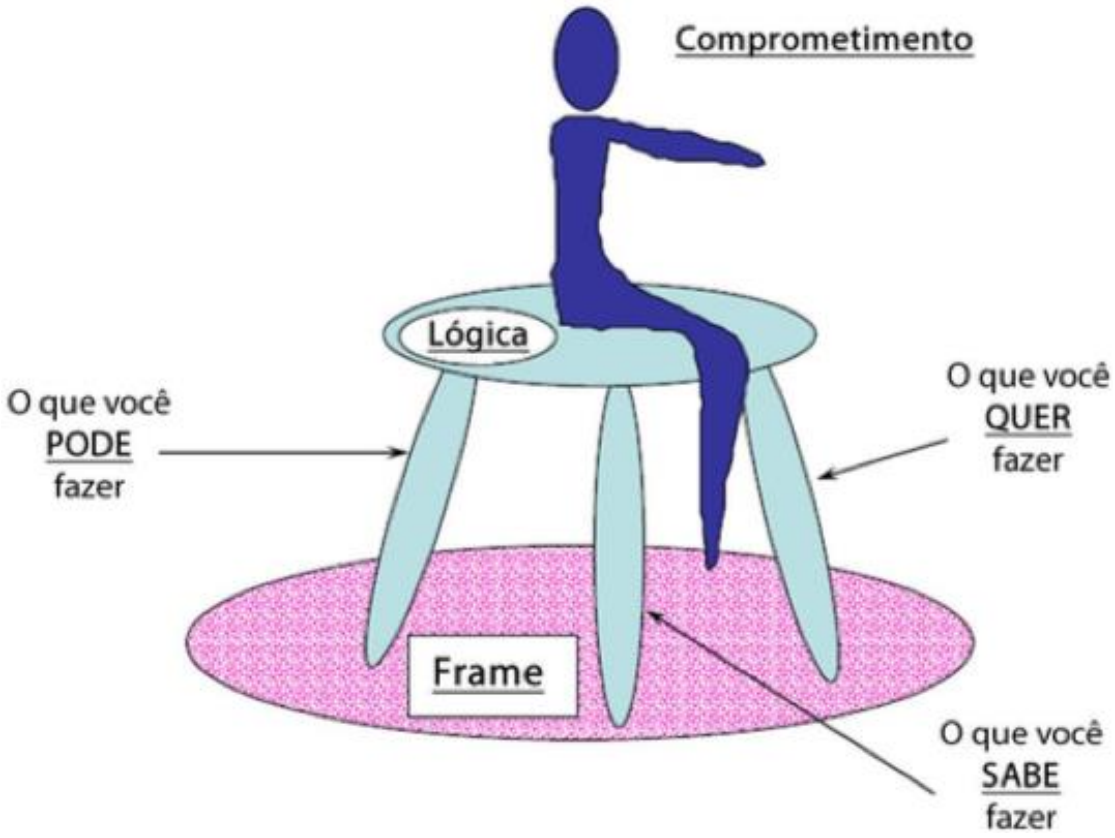
Para tanto, serão abordados aspectos relativos às habilidades e atitudes que fazem um arquiteto exercer sua posição como líder técnico; e, além, serão abordadas questões relativas à atuação do arquiteto de software como referência, não só para a equipe, mas para a diretoria da organização. Para tanto, serão apresentados os tópicos necessários para que um arquiteto se desenvolva como líder técnico e estratégico, mesmo que para isso tenha que abordar aspectos não técnicos do desenvolvimento de software.

Neste capítulo será ainda abordada a forma com o arquiteto de software pode exercer sua liderança, sendo no sentido top-down ou bottom-up, exercendo influência não somente na sua equipe, mas também em seus pares e até mesmo em gestores hierarquicamente superiores, chegando ao ponto de construir ou destruir estratégias de negócio através de requisitos técnicos. Neste capítulo também serão abordados os diferentes tipos de liderança e como elas se tornam importantes dependendo do tipo de pessoa a ser liderada, o contexto de uma organização ou um projeto específico.

O foco deste capítulo é liderança, porém não se trata de um modelo para auxiliar no plano de carreira de profissionais ou definir certificações em engenharia de software.

Visão do Capítulo 4 – Decisões e Riscos

No capítulo 3 veremos a necessidade de o arquiteto de software desempenhar funções de liderança. E é inerente ao líder em vários momentos de sua atividade profissional assumir riscos e tomar decisões. Em muitas vezes, devido à característica estratégica destas decisões, estas envolverão um alto grau de incerteza sobre seu resultado após terem sido tomadas. Assim, o capítulo 4 irá tratar das características da tomada de decisão de técnicas para que esta atividade seja feita com maior segurança.



Os mecanismos de tomada de decisão possuem uma série de estratégias atreladas, que são usadas para auxiliar esse processo. Existe uma série de mecanismos teóricos que envolvem fórmulas matemáticas e estatísticas. Estes mecanismos estatísticos não serão objeto de estudo deste capítulo, porém serão citados para quem quiser aprofundar neste estudo.

Visão do Capítulo 5 – Métodos de Gestão e Desenvolvimento de Software

Em se tratando de um curso voltado para arquitetos de desenvolvimento de software, é importante tratar dos processos de gestão das atividades de desenvolvimento. É isto que trata esse capítulo, abordando os métodos de gestão e desenvolvimento de softwares, mostrando as melhores práticas envolvendo gestão de equipes e gestão de projetos. Uma das principais vertentes modernas para gestão de desenvolvimento de softwares são os modelos ágeis. Assim, o capítulo 5 irá tratar dos princípios e valores do Manifesto Ágil, base para o XP e *Scrum*. Estes dois processos serão estudados também no capítulo, já que têm sido adotados por diferentes empresas de diferentes segmentos e tamanhos do mercado de TI.

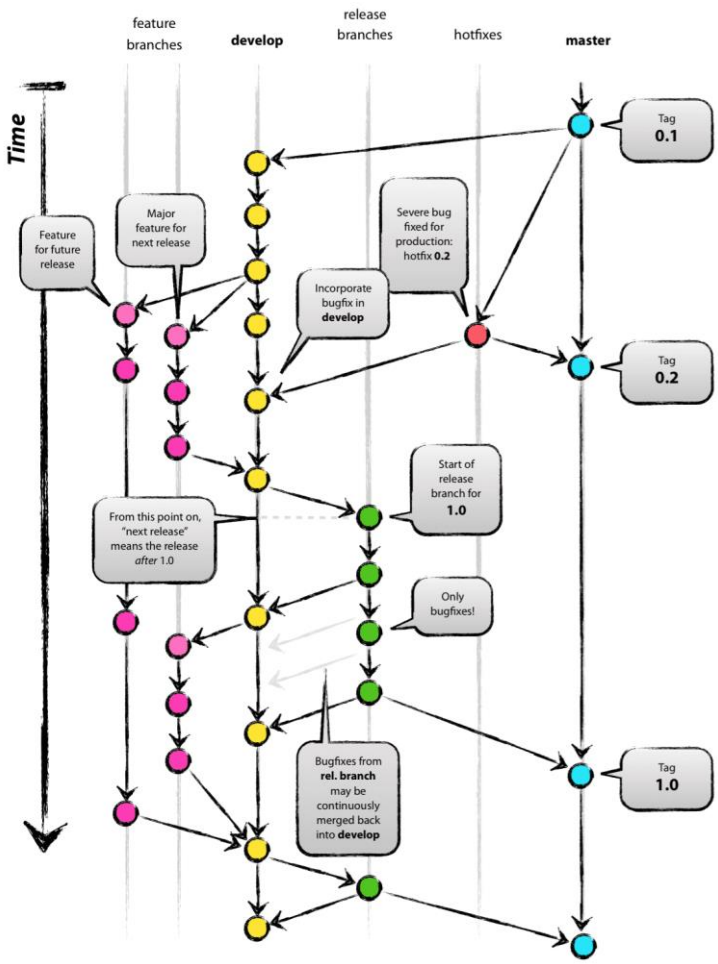
Além dos processos de desenvolvimento de software, também serão abordados processos mais voltados para gestão de ciclo de vida de produtos, como o *Customer Development* e o *Lean Startup* (RIES, 2011). Entender esses processos contribui para que o arquiteto de software tenha a habilidade de entender o mecanismo de concepção de aplicações, fim a fim em seu contexto.

Conforme falado anteriormente, este capítulo irá focar em técnicas modernas de gestão de desenvolvimento de software. Apesar da importância da UML para a história do desenvolvimento de software, entende-se que o conhecimento básico sobre esta é pré-requisito para entendimento das técnicas modernas. Assim, as características de UML não serão abordadas no decorrer do capítulo. Este capítulo também não irá tratar certificações como MPS ou CMM.

Visão do Capítulo 6 – Estratégia de Gestão de Configuração e Versionamento

Gestão de Configuração é uma das importantes disciplinas presentes no ITIL. Não por acaso também é uma das mais importantes etapas do processo de desenvolvimento de software, bem como um dos processos que mais geram retrabalho de desenvolvimento quando não funciona de acordo.

Sendo assim, este capítulo irá tratar as principais ferramentas de gestão de versões de software, apresentando um paralelo entre estas, abordando suas vantagens e desvantagens,

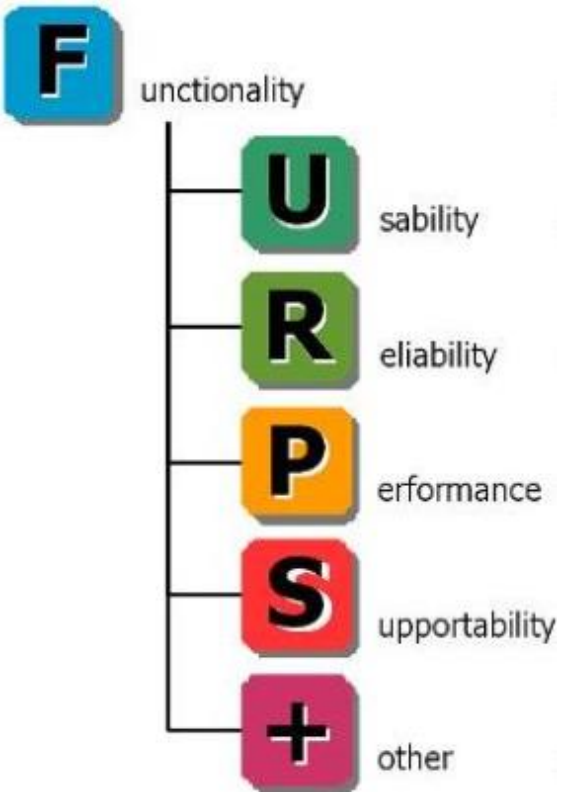


Apesar de focar nas características das principais ferramentas de versionamento de software, o capítulo 6 não irá tratar de um tutorial para utilização das mesmas. Entende-se que a operação dessas ferramentas se baseia em características específicas de cada uma delas, sendo que o importante é compreender o racional por trás de todas as ferramentas, que é o mesmo.

Visão do Capítulo 7 – Requisitos Arquiteturais

O capítulo 7 abordará as técnicas mais eficazes para extrair as necessidades dos usuários em relação às funcionalidades esperadas em um sistema. Estas necessidades são traduzidas em requisitos. Neste cenário é imprescindível que o arquiteto esteja atento e realize uma análise crítica destes requisitos.

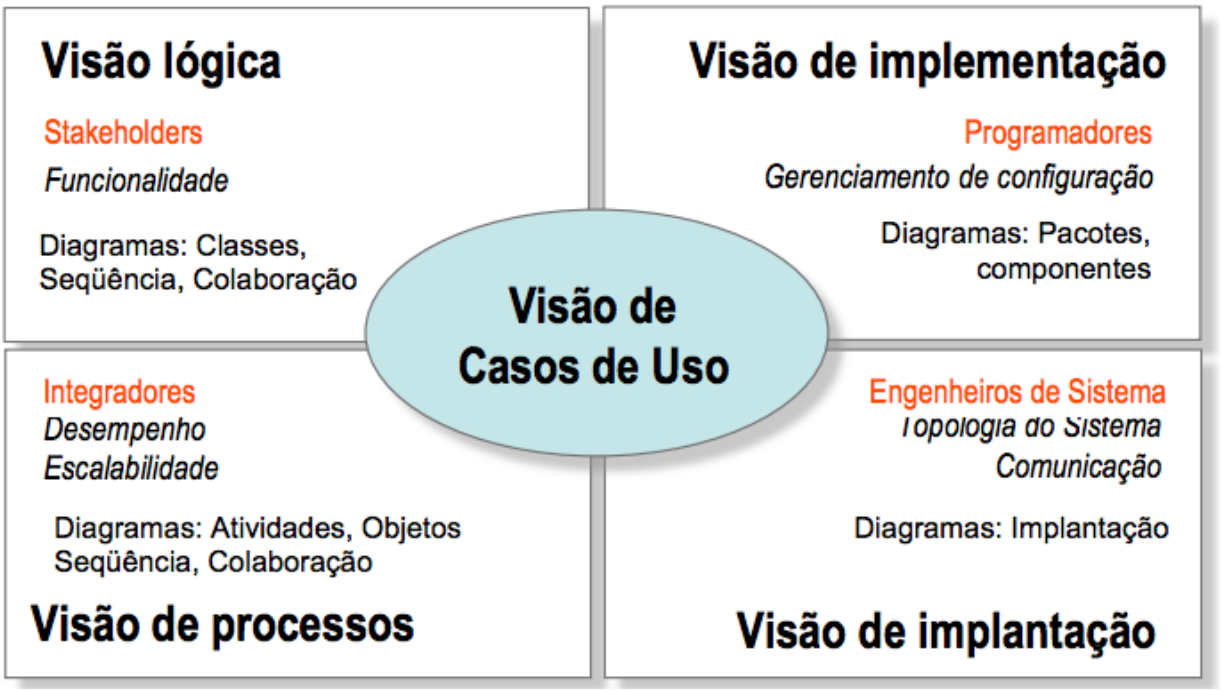
As técnicas que serão abordadas neste capítulo são a FURPS+ e SMART. Ambas técnicas são frequentemente utilizadas para se extrair requisitos que impactam na arquitetura de sistemas.



Visão do Capítulo 8 – Modelagem Arquitetural

O foco do Capítulo 8 é o modelo Visão 4+1 como forma de comunicar as funcionalidades de um sistema através de modelagem de requisitos. Este tipo de comunicação torna-se necessária a partir do momento que o arquiteto de software é responsável por dialogar com interlocutores dos mais diversos níveis, áreas e conhecimentos.

A abordagem, porém, será relativa ao processo de modelagem em si e não aos artefatos gerados. Portanto, não é foco desse capítulo tratar diagramas e documentos gerados pelos modelos, até porque estes podem ser feitos em um arquivo Word, um papel de pão ou um sistema milionário. Porém, o racional por trás da modelagem mantém-se o mesmo, em qualquer uma destas situações.

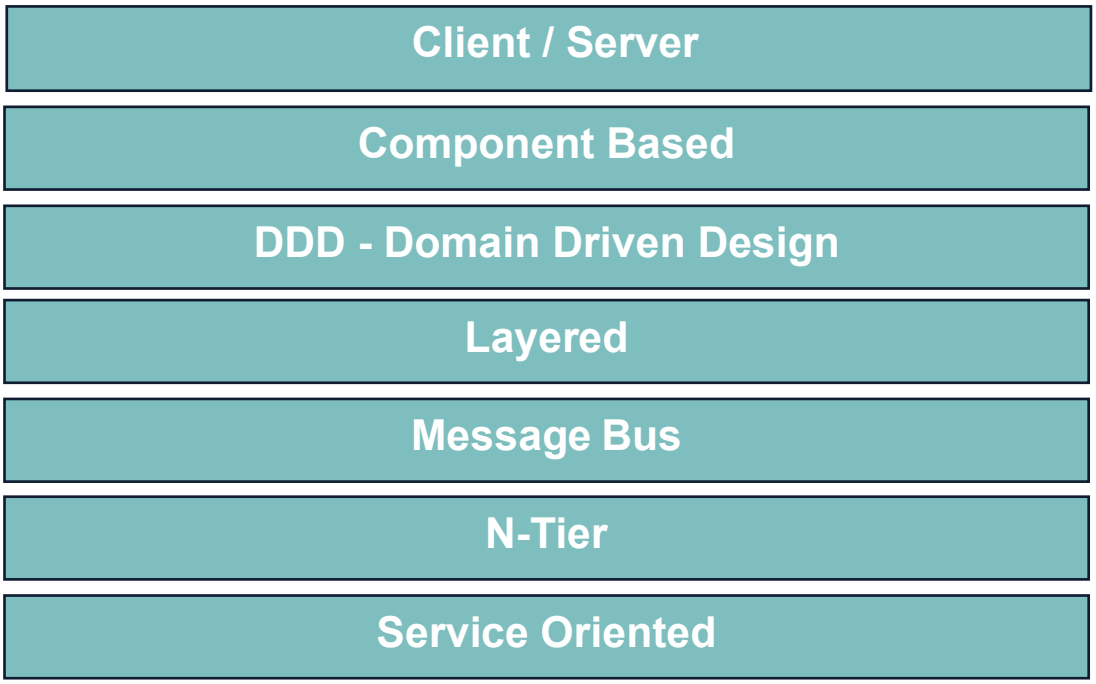


Visão do Capítulo 9 – Estilos e Padrões Arquiteturais

O capítulo 9 talvez seja o mais técnico do curso. Nele iremos tratar soluções pré-definidas, padrão de mercado, para situações comuns. O objetivo destes padrões

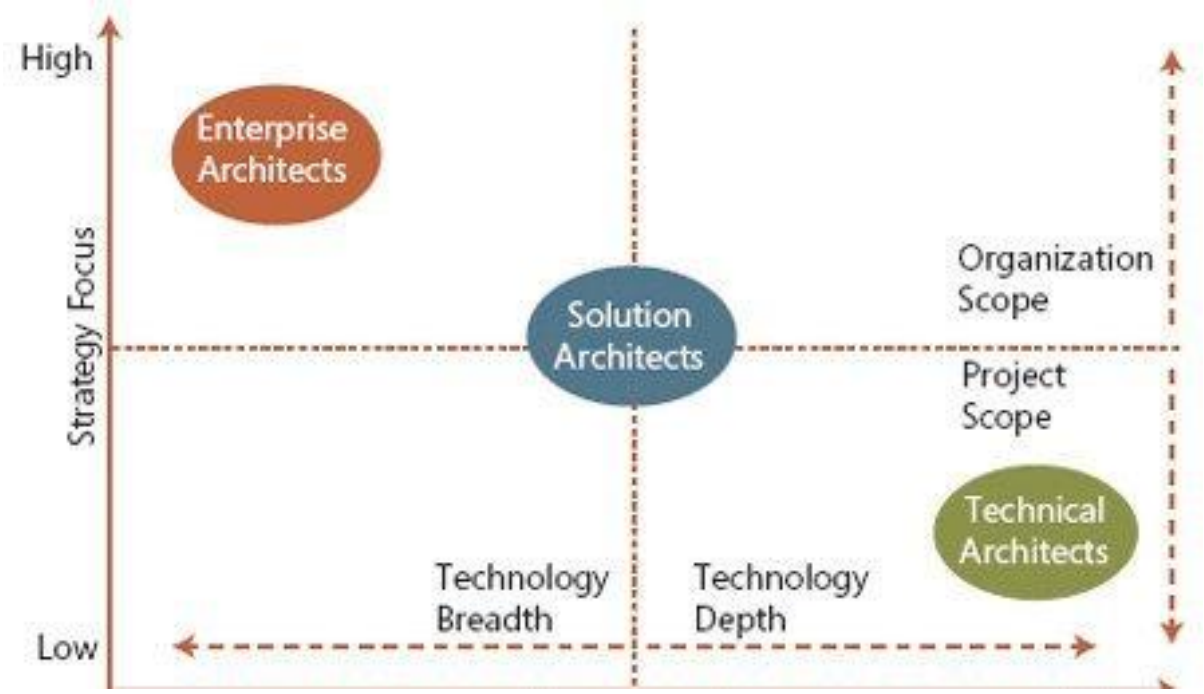
é que sejam totalmente independentes de tecnologia ou plataforma, sendo utilizados em diversas ferramentas e abordagens de construção de software.

Dentre estes padrões de mercado, serão abordados os Padrões arquiteturais de Aplicação *(Enterprise Application Patterns – EAP)* e os Padrões arquiteturais de Integração *(Enterprise Integration Patterns - EIP).* O EAP define abordagens comuns para problemas recorrentes do desenvolvimento arquitetural de aplicações corporativas. O EIP, por sua vez, contempla práticas para problemas recorrentes de integração entre sistemas.



Capítulo 2. Arquitetura de Software como Estratégia Corporativa

Neste capítulo iniciamos definitivamente a jornada pelo trabalho do arquiteto de software. Nele iremos aprofundar nessa atividade além do conhecimento puramente técnico. À medida que o profissional se torna referência em sua organização ou contexto, é natural que as pessoas ou a organização em que ele atua esperem dele um nível de responsabilidade maior, participando e liderando decisões estratégicas de tecnologia. Assim, torna-se importante entender como o trabalho do arquiteto de software pode auxiliar a atingir metas definidas pela estratégia da empresa.



A figura acima mostra diferentes níveis de atuação de um arquiteto de softwares dentro de uma organização. À medida que aumenta o nível de importância e atuação do arquiteto de software, também aumenta o seu foco em relação à estratégia da organização. Não se trata, entretanto, de uma evolução hierárquica. Pode até ser, dependendo de como a organização enxerga essa carreira, mas o importante nessa evolução é a importância e atuação estratégica do arquiteto no processo de software da empresa.

Assim, podemos entender cada um desses papéis do arquiteto de software como:

- **Arquiteto técnico (ou líder técnico)**: foca predominantemente em aspectos de implementação tecnológica e está envolvido de forma ativa na codificação. A tecnologia envolvida nesse caso já é aplicada em outras frentes da organização e ele o faz no seu escopo de projeto com expertise e detalhe. Entretanto, não se pode esperar do papel um amplo conhecimento da estratégia organizacional e do ciclo de vida completo da solução ou produto de software.

- **Arquiteto de soluções:** é alocado em um projeto ou programa (conjunto de projetos) com o objetivo de garantir a integridade técnica e a consistência da solução ao longo de todo o seu ciclo de vida e da estratégia organizacional. Arquitetos de solução não estão normalmente envolvidos de forma ativa (*hands-on*) na implementação, pois a coordenação das atividades técnicas consome a maior parte do seu tempo. Além disso, em um contexto em que existem provas de conceitos ou elevada incerteza, é recomendado que o arquiteto de soluções assuma o lugar do arquiteto técnico na implementação.

- **Arquiteto corporativo:** possui visão de toda a empresa, como o nome sugere. Preocupando-se com a visão holística de TI, desde a infraestrutura até frameworks arquiteturais de forma a otimizar resultados operacionais e entregar a estratégia corporativa. As próximas duas seções serão dedicadas a compreender o papel do arquiteto corporativo e os seus principais tópicos de atuação.

O entendimento de cada um desses papéis é importante para se planejar uma carreira como arquiteto de software. É muito comum nas organizações, que os melhores programadores (técnicos), que se destacam, sejam “promovidos” a arquiteto. Isso pode ser interessante para um arquiteto técnico, porém, nem sempre esse profissional está preparado para se envolver em questões estratégicas que um arquiteto corporativo demanda. Por isso, vamos focar em descrever mais

detalhadamente a importância do arquiteto corporativo e a atuação dele menos como um técnico e mais como um estrategista para a organização.

O papel do arquiteto corporativo está intimamente ligado à visão estratégica global da empresa. Quem efetivamente transforma as necessidades tecnológicas da organização para atingir os objetivos estratégicos em solução de software, é esse profissional.

Mas o papel do arquiteto corporativo não se resume a, por si só, definir as necessidades tecnológicas da empresa a curto, médio e longo prazo. O arquiteto corporativo tem também um papel de aconselhador tecnológico para todas as áreas da empresa. Devido à sua experiência técnica, visão global estratégica da empresa e do negócio em que a empresa está inserida, este profissional pode auxiliar tanto sua equipe quando seus gestores, entendendo como deve se comunicar com cada um destes interlocutores.

Em suma, as principais diferenças entre um nível mais técnico (arquiteto técnico) e um nível mais estratégico (arquiteto corporativo), são:

- Arquiteto técnico: se preocupa com projetos específicos de SW da empresa e como eles serão entregues da forma como foram especificados:
  - Possui visão local: em geral é alocado para um desenvolvimento específico e muitas vezes não “conversa” com outros projetos, sendo responsável por aquela entrega pontual.
  - É especialista na tecnologia do projeto: possui conhecimento profundo em todas as ferramentas e artifícios que estão sendo usados no projeto, sem necessariamente conhecer todo arcabouço tecnológico disponível na empresa.
  - É referência técnica para o projeto: a equipe do projeto enxerga no arquiteto uma pessoa para responder situações técnicas do escopo do que está sendo desenvolvido.

  - É um importante membro do projeto: assim como os desenvolvedores, os analistas, os testers, Scrum Master, etc., o arquiteto é uma peça chave para o sucesso do projeto.

  - Tem atuação operacional: O arquiteto técnico muitas vezes atua como hands-on na programação, possuindo uma visão mais operacional, objetivando a entrega de um projeto específico.

- Arquiteto corporativo: se preocupa com todos os projetos de SW da empresa e como eles irão ajudar a atingir os objetivos estratégicos:

  - Possui visão global: Conhece muito sobre o negócio da empresa. Precisão entender como todos os projetos se integram e o que se espera estrategicamente de cada uma das soluções que estão sendo desenvolvidas.

  - É especialista na estratégia de tecnologia da empresa: Conhece todas as tecnologias utilizadas em todos os sistemas da empresa. Sabe bem quais os potenciais e quais as limitações destas tecnologias e entende o melhor momento para evoluí-las ou até mesmo substituí-las.

  - Serve como líder técnico de toda empresa: A equipe de desenvolvimento da empresa espera do arquiteto um “oráculo” tecnológico, em relação ao por que foi adotada uma tecnologia em detrimento a outras. Não sabe tudo sobre tudo, mas espera-se que ele consiga direcionar a equipe a encontrar suas respostas de forma mais ágil.

  - É peça chave para a estratégia de TI da empresa: Por ter contato com a equipe de desenvolvimento (bottom-up) e com a diretoria (top-down), o arquiteto corporativo é a pessoa chave para traduzir objetivos estratégicos em soluções de software, que consigam ser desenvolvidas com excelência.

- Tem atuação estratégica: Possui uma visão completa da TI, englobando infraestrutura, linguagens, frameworks, banco de dados, sempre buscando alinhar as definições técnicas com a estratégia de longo prazo.

Em geral, os profissionais que estão no mercado querem evoluir na carreira. Isso não é diferente com a área de Tecnologia da Informação, em especial nas atividades de desenvolvimento de sistema. Seguindo uma carreira Y tradicional, chega uma hora que um profissional de sistemas tem que decidir em seguir um caminho gerencial ou um caminho técnico. Seguindo pela “perna” gerencial do Y ele poderá se aprimorar em disciplinas como Gestão de Projetos, Gestão de Processos, Gestão Estratégica de TI, entre outras. Ao optar pelo lado técnico, terá um caminho tradicional a ser seguido, principalmente por desenvolvedores: buscar o cargo de arquiteto de software ou arquiteto de sistemas. Para muitos, ocupar esse cargo é chegar ao topo da pirâmide da hierarquia de desenvolvimento de sistemas. É se tornar o oráculo dos programadores.

Em parte, isso é verdade. Para chegar a ser um arquiteto de software é preciso ter um vasto conhecimento técnico, aliado a experiências muitas vezes frustrantes de adoção de tecnologias inovadoras ou tradicionais. É preciso ter “quebrado a cara” com SGBDs que não se integram bem com determinada tecnologia, um padrão de projeto que não suporta o volume de transações quando a aplicação cresce ou um modelo arquitetural que precisou ser adaptado para um projeto mobile, por exemplo. Estas experiências, quando vivenciadas e bem assimiladas, habilitam um desenvolvedor a conseguir entender tecnicamente as proposições de um software e traduzir as necessidades de especificação no melhor modelo arquitetural possível.

Porém, um dos grandes equívocos de quem almeja a carreira de arquiteto de software é achar que, se tornando um programador excelente, já está apto para exercer a função. Ou mais, que o dia a dia do arquiteto de software se limita a desenhar soluções técnicas eficientes para problemas complexos. Essa função existe, sim, e faz parte do dia a dia do arquiteto. Porém, mais do que isso, este

profissional precisa exercer dois papéis macro fundamentais: líder/coach técnico da equipe e conselheiro técnico da alta diretoria.

O primeiro papel diz respeito à forma como a equipe de desenvolvimento enxerga o arquiteto de software. De forma natural, espera-se que este profissional exerça o papel de referência para equipe. E, como referência, subtende-se que ele saiba ouvir e dialogar com a equipe, repassando seu conhecimento quando solicitado ou quando entender que seja necessário. Pressupõe-se, também, que este profissional consiga perceber quais as deficiências técnicas da equipe, inclusive propondo treinamento para pessoas -chave que precisam desenvolver alguma competência técnica. Enfim, é função do arquiteto de software assumir o papel de líder técnico da equipe, uma pessoa com quem todos possam contar em situações de aperto ou que saiba conduzir uma equipe de excelência em desenvolvimento de software.

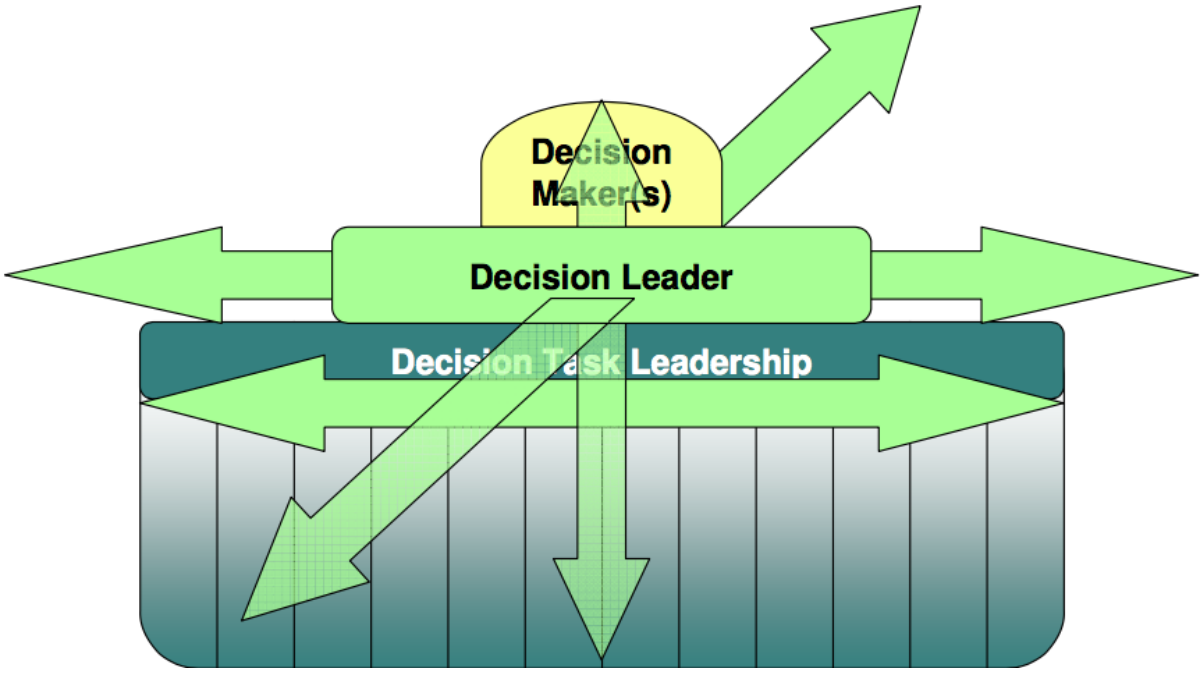
O segundo papel diz respeito à necessidade da alta diretoria em ter um respaldo técnico confiável para as necessidades estratégicas da empresa. Nesse sentido, o arquiteto de softwares precisa “sair” do universo pontual de desenvolvimento e adentrar a estratégia corporativa da empresa. Por exemplo, se ele está trabalhando em um sistema web de CRM (Customer Relationship Management), que levará seis meses para ser desenvolvido, ele precisa participar ou ter conhecimento de decisões estratégicas da empresa que pode, por exemplo, definir que existe a expectativa deste sistema se tornar um App Android e iOS em dois anos. Com essa informação, ele poderá definir a arquitetura já pensando nessa possibilidade, trazendo ganhos para todo o processo. Além disso, como advisor da diretoria, o arquiteto de software deve advertir estes sobre qualquer ação que possa colocar em risco a empresa, como por exemplo, negociar com um cliente um volume de transações que o sistema atual não irá suportar.

Capítulo 3. Liderança Arquitetural

Está cada vez mais clara a necessidade de um arquiteto de software atuar como líder. É justamente desse papel de liderança e todas as nuances que envolvem essa importante atividade, que se trata esse capítulo.

A liderança exercida pelo arquiteto de softwares muitas vezes é feita de forma natural, sem um relacionamento hierárquico. Dessa forma, algumas habilidades precisam ser intrínsecas deste profissional, dificultando que sejam algumas vezes desenvolvidas de forma artificial. Uma vez que o arquiteto de software atua em um papel estratégico, ele muitas vezes precisa tomar frente em atividades como gestão de projetos, planejamento de equipe, levantamento de dados, comunicação, entre outras.

Como já mencionado, o arquiteto de software, no papel de líder, exerce poder de persuasão em sua equipe direta, em uma atuação de liderança top-down, mas também seus pares ou seus gestores, em uma atuação de influência bottom-up.



Como líder de decisão, o arquiteto precisará intermediar direcionamentos estratégicos da alta gestão e da sua equipe, garantindo que a execução operacional esteja alinhada e atenda aos requisitos de qualidade estabelecidos pelo estratégico.

Em relação ao time de alta gestão o arquiteto será responsável, por exemplo, por direcionar os membros para que eles entendam as responsabilidades da decisão, construir comprometimento e confiança em torno da decisão a ser tomada, conduzir uma comunicação aberta e objetiva ao longo da execução da decisão, fornecer atualizações periódicas até sua conclusão, com qualidade. Por outro lado, no time em que estará envolvido como arquiteto, ele terá que, entre outras responsabilidades, organizar e monitorar as tarefas e os processos, lidar com mudanças inevitáveis e inesperadas, garantir a qualidade do trabalho, gerenciar recursos e cronograma.

É importante destacar, também, que existe diversos estilos de liderança e que cada estilo pode ser mais adequado ao um tipo de equipe, projeto ou a uma cultura de empresa. Antes de mais nada, é importante destacar as diferenças entre um líder e um mero gestor. O líder é mais focado nas pessoas, suas motivações, aspirações e seu desenvolvimento. Enquanto isso, o gerente foca na estrutura e organização, lidando com planos, orçamentos e cronogramas. Não se deve classificar gerenciamento como ruim ou tradicional e a liderança como boa e moderna, ou seja, não há uma relação antagônica entre ambos, mas sim o contexto de atuação que exigirá que determinado papel e atribuições se manifestem para viabilizar resultados de forma mais adequada de acordo com a cultura do ambiente. O gerenciamento é adequado para contextos complexos, e sem uma boa gestão as empresas tendem a ser caóticas. Por outro lado, a liderança é adequada para contextos de mudança e incertezas.

Manager	Leader
Achieve plans by organizing & staffing	Achieve plans by aligning people
Creates plans and budgets	Creates visions and strategies
Maintains	Develops
Focuses on systems and structure	Focuses on people
Relies on control	Motivates and inspires trust
Asks "How" and "When"	Asks "What and Why"
Eyes the bottom line	Eyes the horizon
Imitates	Originates
Accepts status quo	Challenges it
Classic good soldier	His or her own person
Does things right	Does the right things

Diante do exposto, está claro que o contexto pode influenciar se haverá mais aspectos de gerenciamento ou liderança. Este mesmo contexto pode definir o estilo de liderança mais adequado. Iremos demonstrar alguns estilos de liderança proposto por Goleman, em seu livro Primal Leadership:

Leadership Style	How It Builds Resonance	Impact on Climate
1. Visionary	Moves people toward shared dreams	Most strongly positive
2. Coaching	Connects individual wants with organizational goals	Highly positive
3. Affiliative	Creates harmony connecting people	Positive
4. Democratic	Values inputs and gains commitment	Positive
5. Pacesetting	Meets challenging and exciting goals	When botched, highly negative
6. Commanding	Sooths fears by giving clear directions in an emergency	Often misused; highly negative

- Visionário: é um estilo de líder que está sempre atento às oportunidades e aos movimentos do mercado, aceitando facilmente os desafios, encarando os riscos como um fator necessário para o crescimento profissional. Sua

capacidade de prever os movimentos do mercado se dá através de constantes pesquisas e análises do comportamento de seu público.

- Coach: é o estilo de líder que se preocupa com a capacitação de sua equipe. Sua responsabilidade ultrapassa o papel do líder, pois ele se preocupa com a saúde física, emocional e psicológica de seus liderados. Também se preocupa com a motivação constante da equipe, preocupando que a mesma cresça profissionalmente.

- Afiliativo: é o estilo de líder que cria laços emocionais entre os membros da equipe, de forma a fazer com que as pessoas desenvolvam um sentimento de que fazem parte da empresa. Esse tipo de líder é essencial em tempos de grande estresse, quando a equipe precisa se recuperar de um trauma e reestabelecer a confiança uns nos outros.

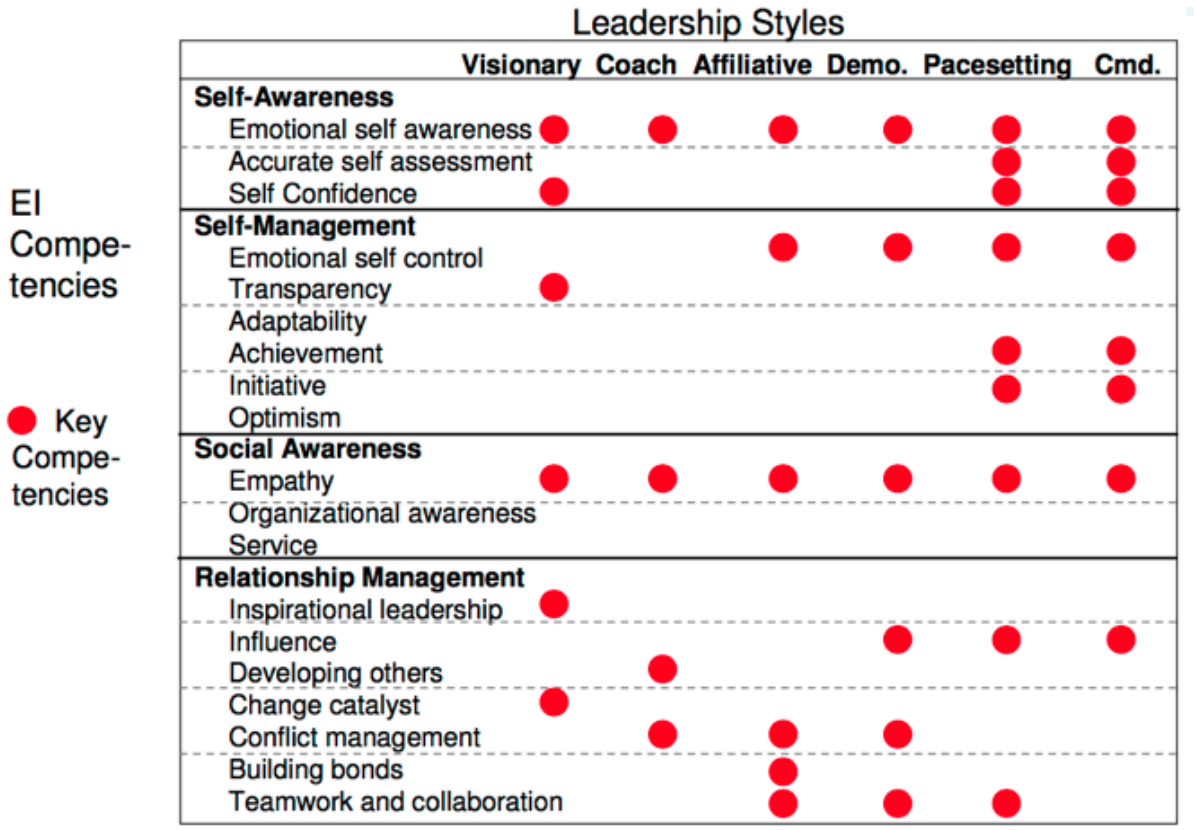
- Democrático: é o estilo de líder que acredita que a participação democrática do grupo é importante para o aperfeiçoamento dos processos internos e dos projetos da organização como um todo.

- Autoritário: é o estilo de líder que até permite que seus liderados participem da discussão, mas a decisão final será tomada por ele. Este tipo de líder geralmente gosta de correr riscos e está pronto para os resultados.

- Pacesetting: é o estilo de líder que, devido a seu perfeccionismo e detalhismo, faz com que a qualidade de entrega de sua equipe seja extraordinária, mas que pode trazer um alto nível de estresse a toda equipe.

Diferentes estilos de liderança demandam diferentes perfis de líder. Em muitas empresas os arquitetos são escolhidos a partir de destaques técnicos na equipe de desenvolvimento, quase como uma “promoção” na carreira. Assim, muitas vezes o estilo de liderança vem da forma como intrinsecamente a pessoa lida com os aspectos de liderança. Isso é perceptível pelos aspectos de inteligência emocional que demonstra. A prática da liderança, em qualquer um dos estilos, precisa estar associada aos elementos básicos da inteligência emocional e do autoconhecimento

do líder. Desta forma, é possível traçar um paralelo entre os estilos de liderança e os componentes-chave da inteligência emocional para sua eficácia.



A partir da leitura acima, é possível classificar os componentes da inteligência emocional em quatro categorias: autoconhecimento (*Self-Awareness*), autogerenciamento (*Self-Management*), conhecimento social (*Social Awareness*) e gestão de relacionamento (*Relationship Management*).

Analisando as linhas preenchidas da tabela para a categoria de autoconhecimento, é possível perceber que o autoconhecimento emocional (*Emotional self awareness*) é um componente chave em todos os estilos de liderança, ou seja, um líder em sua essência deve buscar se autoconhecer. Em seguida, a auto avaliação apurada (*Accurate self assessment*) é um componente chave para os estilos exigente e comandante, o que é justificável dada à característica detalhista e de controle desses dois estilos. Por fim, ainda nessa categoria, a autoconfiança (*Self Confidence*) se manifesta nos estilos visionário, exigente e comandante.

A categoria de autogerenciamento, por sua vez, ilustra que os líderes agregadores, democráticos e exigentes possuem esse componente como chave, com exceção do visionário e do coach. Essa ausência de competência para o visionário e o coach, se deve ao fato desses dois estilos serem bastante orientados a emoções, buscando envolver a todos pelas suas emoções e, com isso, às vezes podem se perder ao lidar com as suas próprias, deixando-as transparecer em contextos inadequados. Em seguida, a transparência (*Transparency*) é um componente chave do visionário, pois ele busca orientar e direcionar sua equipe para um objetivo compartilhado. Por fim, os componentes de realização (*Achievement*) e iniciativa (*Initiative*) estão presentes, simultaneamente, somente nos estilos exigente e comandante. Isso se deve ao fato desses dois estilos se preocuparem mais com os resultados e as decisões em si do que, necessariamente, com o envolvimento das pessoas ao longo do processo.

A terceira categoria, Conhecimento Social, apresenta a empatia se destacando em todos os estilos. Reforçando, assim, que o exercício da liderança, independente do estilo adotado e do seu contexto, deve vir acompanhado do exercício da empatia. Isto é, o líder deverá sempre buscar a percepção do outro para garantir decisões e resultados com eficácia.

A quarta e última categoria da Tabela, Gestão de Relacionamento, ressalta que o líder visionário possui a capacidade de inspirar os outros (*Inspirational Leadership*), não somente a sua equipe, mas também outros líderes! A influência (*Influence*) está fortemente presente nos líderes democrático, exigente e comandante, pois esse componente reflete a capacidade de fazer com que as pessoas realizem o que o próprio líder ou o seu grupo determina.

Ainda considerando a quarta categoria, é possível perceber que o componente de desenvolvimento dos outros (*Developing others*) está presente unicamente no coach, assim como o catalizador de mudança (*Change catalyst*) está no visionário. Mas é interessante observar que a mesma tabela referencia os estilos de coach, agregador e democrático como os mais bem preparados para lidar e gerenciar conflitos em suas equipes (Conflict Management).

Por fim, percebe-se que o líder agregador, conforme esperado, tem a habilidade exclusiva de construir laços (*Building bonds*) ao passo que ele, o democrático e o exigente possuem componentes chave focados no trabalho em equipe e na colaboração (*Teamwork and collaboration*). Evidentemente, cada um sob a sua perspectiva. Por exemplo, enquanto o agregador tem esse componente com o objetivo de desenvolver nas pessoas o sentimento de pertencer à organização, o exigente é motivado pelos melhores resultados e pela qualidade.

Capítulo 4. Decisões e Riscos

À medida que crescem as responsabilidades de um profissional, é natural que ele se exponha mais a riscos e necessite tomar decisões que muitas vezes não gostaria de tomar. Este capítulo trata de decisões técnicas, gerenciais e estratégicas que o arquiteto de softwares precisa conviver durante a execução de seu trabalho. O que alguns profissionais desconhecem é que existem técnicas para se tomar decisões de forma consciente, diminuindo o risco envolvido e aumentando a qualidade. A variável mais importante em uma decisão é a incerteza que ela gera em relação ao resultado.

A definição de decisão é a interseção entre o pensamento e a ação, estabelecendo uma decisão dentro de um contexto específico. Este ato de tomar uma decisão tem algumas características básicas:

1. **Dizem respeito ao futuro:** Não faz sentido tomar decisões sobre o passado, pois esse não pode ser alterado. A tomada de decisão tem fundamento sobre a incerteza, já que o futuro é imprevisível.

2. **Comprometem recursos:** Decisões têm como base o comprometimento de recursos, como dinheiro, tempo, hardware, software, pessoas, etc.

3. **Possuem mais de uma alternativa:** Se houver somente uma alternativa em jogo, não é necessário tomar uma decisão, basta executar a alternativa. Por isso, uma decisão sempre leva em conta pelo menos duas alternativas, sendo que é bastante comum se tomar decisões sobre mais de duas alternativas.

4. **Não tem volta:** Decisões são processos em que não se deve voltar atrás. Ou seja, uma vez tomada uma decisão, ela não pode ser desfeita. Caso seja necessário alterar uma decisão, encontra-se um processo de tomar uma nova decisão.

5. **Precisa fazer sentido para você:** Uma decisão sobre algo que não faz sentido para as partes envolvidas é uma decisão vazia. Se não é importante para os envolvidos, existe uma tendência de ignorar o risco da decisão.

6. **Maior risco leva a pior qualidade de decisão:** Quando uma decisão é tomada levando-se em consideração algum grau de incerteza, a chance de um resultado positivo diminui. De certa forma, este cenário deixa em evidência que o processo decisório pode ter sido feito à revelia.

Para se aumentar a qualidade de uma decisão, é importante entender quais são os componentes envolvidos nesse processo. Existe na literatura uma definição de seis componentes, que permeiam praticamente todos os processos decisórios. São eles: frame, alternativas, informação, valores, raciocínio lógico e compromisso com a ação. Vamos analisar cada um deste componentes.

- **Frame:**

O *frame* é uma foto do problema que se deseja resolver. Assim como em uma foto, existe uma série de elementos em uma paisagem que não se deseja dar foco. O

frame é, portanto, o recorte de um cenário maior, porém com foco em algum ponto específico. O frame pode ser dividido em partes como:

- Proposta: trata-se do que deseja resolver mediante uma tomada de decisão. Por incrível que pareça, um erro muito comum no processo de tomada de decisão é resolver o problema errado. Ou seja, se gasta um grande esforço para resolver um problema, quando há outro maior ou mais importante envolvido no processo.

- Perspectiva: trata-se de todo ambiente em volta do problema, quem é necessário envolver, o que se esperar dos envolvidos, etc. Envolver pessoas erradas no processo de decisão aumenta o risco e consequentemente leva a uma redução na qualidade esperada da decisão;

- Escopo: trata-se dos limites que definem as interfaces do problema. O escopo determina o que faz parte de um problema e o que não faz parte do problema.

- **Alternativas:**

Estando o frame definido e formalizado, sendo identificados seu escopo, perspectiva e proposta, se busca então, alternativas para sua solução. Uma das ferramentas utilizadas para identificar alternativas são os *brainstormings*. Neste tipo de reunião é possível perceber falhas na identificação de alternativas, por exemplo, quando há uma concordância de todos com relação a uma única alternativa.

Um erro comum no processo de identificar alternativas é já tomar a decisão ou pensar na decisão nesse momento. São momentos diferentes. A identificação de alternativas é apenas a formalização das possibilidades de escolha do processo decisório, e não a escolha em si.

- **Informação:**

Esse elemento atua ligado ao segundo, levantando e detalhando informações para cada alternativa criada. Por exemplo, durante um processo de decisão de uma equipe sobre qual banco adotar na arquitetura, uma alternativa foi o Cassandra e

outra foi o Datomic, entretanto não há ninguém na equipe que tenha conhecimento em banco NoSQL (primeiro) ou banco imutável (segundo), o que pode ou não inviabilizar a alternativa, caso seja necessário buscar um profissional externo à equipe.

Esse elemento racionaliza uma etapa em que precisa levantar as informações necessárias para a decisão de forma a ponderar alternativas, mitigar riscos e analisar viabilidades.

Um ponto importante nessa etapa é que se tenha foco nas incertezas mais críticas do processo de decisão. Além disso, vale ressaltar que qualquer dado é diferente de informação, pois dados são estabelecidos no passado e não dizem respeito ao futuro. Com isso, dados de bases históricas, por exemplo, devem ser considerados, levando em conta o contexto em que foram gerados.

- **Valores:**

Os valores de uma decisão podem ser definidos como o conjunto de preferências dos envolvidos no processo. Em geral, uma decisão é um processo de trade-off, ou seja, um lado cede para o outro “ganhar”. Os valores atuam junto com as informações levantadas para cada alternativa. Ao fazer uma escolha por uma alternativa, pode se estar levando em conta aspectos como a tolerância pessoal, do projeto ou da organização, em relação aos riscos.

- **Raciocínio lógico:**

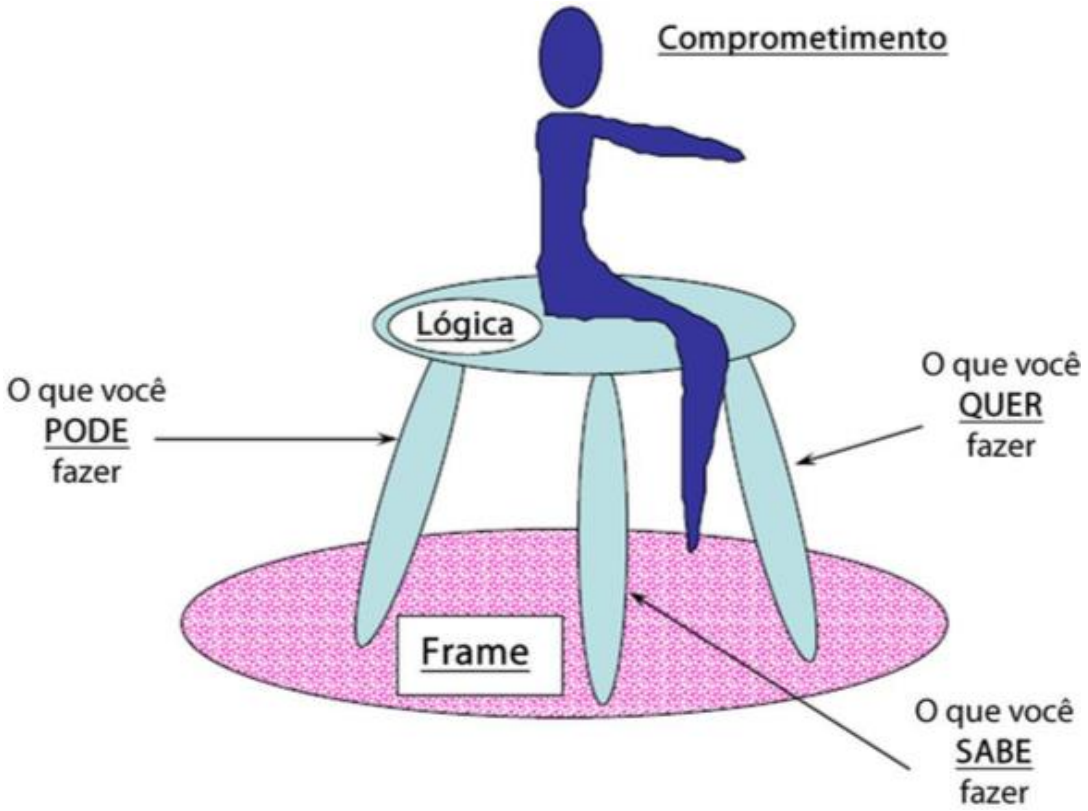
O raciocínio lógico é o elemento que une o frame, as informações, os valores e as alternativas. Trata-se do processo de tomar a decisão em si. Por mais simples que possa parecer, esse elemento pode ser um grande complicador no processo de tomada de decisão. Um problema comum ocorre quando há um número grande de alternativas. Raciocinar sobre duas alternativas é relativamente simples, pois os cenários imagináveis de consequência da tomada de decisão são facilmente elaborados. Porém quando o número de alternativas é grande, dificulta decidir porque há sempre o que avaliar melhor.

Outro problema para o raciocínio lógico aparece principalmente quando há na equipe envolvida e na decisão um gestor e seus subordinados. Existe uma tendência natural de se concordar com a alternativa escolhida pelo gestor, seja por medo ou por respeito excessivo. Em outros casos, escolhe-se “irracionalmente” uma alternativa apenas com o objetivo de evitar conflitos.

- **Compromisso com a ação:**

Este elemento do processo decisório representa a alocação dos recursos necessários para garantir que a decisão seja tomada de forma efetiva.

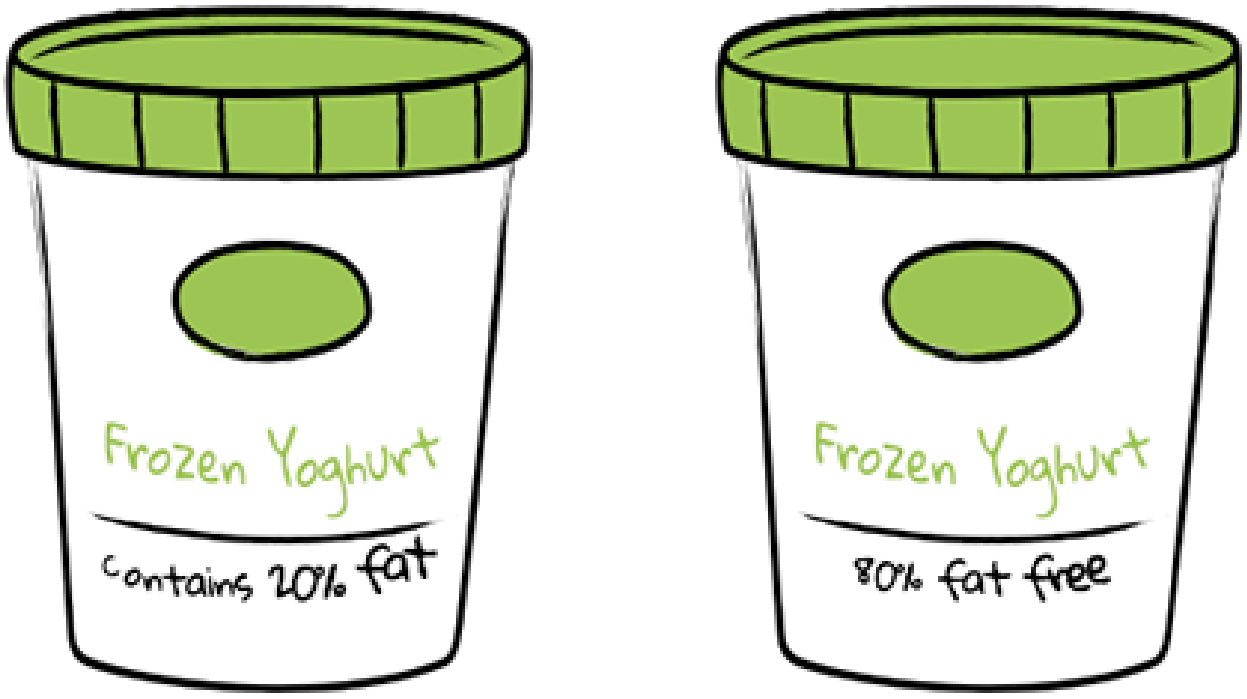
Trata-se do compromisso de cumprir o acordado. De outra forma, a decisão é vazia, pois não adianta decidir o que não será executado.



De forma lógica, o processo de tomada de decisão, quando decomposto nestes componentes, parece simples de ser executado. Porém, existe um “componente” implícito nesse processo muitas vezes renegado, porém extremamente atuante, que é o cérebro humano e suas características não lógicas. Estes

“componentes” apresentam uma característica que é a tendência de tomar decisões ou agir de maneira ilógica. Nós frequentemente vemos o mundo sob a nossa percepção. Assim, distorções do contexto ou das características das informações disponíveis, podem influenciar nossas decisões. Esta característica é documentada e estudada, fazendo parte dos vieses cognitivos. Alguns dos vieses importantes que aparecem no processo decisório serão detalhados a seguir.

- **Framing:**



Framing é a tendência de perceber a informação da forma como ela é recebida, sem considerar alternativas ou diferentes percepções. Na figura acima, percebe-se que a forma de comunicar a mesma informação leva a percepções completamente diferentes. O Iogurte tem 20% de gordura ou ele é 80% sem gordura? Dependendo da forma como a informação é apresentada, poderá ocorrer um julgamento de valores que podem também levar a uma decisão diferente.

- **Paralisia de análise:**

A base para entendimento desse viés é que quanto maior o número de opções, nós demoraremos mais ainda para decidir. Ou pior, continuaremos a analisar e gerar mais opções.

Este viés foi inicialmente comprovado através de um experimento científico. Em uma mesa de cinco opções de doces, verificou-se que as pessoas demoravam alguns segundos para escolher dentre as opções. Quando a mesa aumentou para 30 opções, as pessoas demoravam vários minutos e algumas até desistiam.

Esse viés diz respeito ao comportamento humano de que, ao ter um maior o número de opções, há uma maior demora ou uma paralisia no processo decisório.

- **Status Quo:**

Este viés trata da tendência das pessoas em preferir manter as coisas ou situações como elas estão. Alguns exemplos práticos em que é possível facilmente identificar esse viés em ação:

– Consertando um carro que continua quebrando.

– Retendo um profissional improdutivo.

– Mantendo um código que precisa ser refatorado.

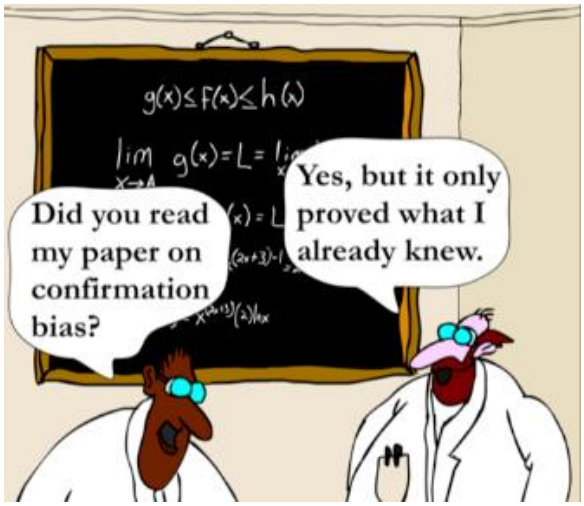
– Mantendo uma arquitetura que precisa ser reestruturada.

- **Custo afundado (sunk cost):**

O viés de custo afundado é percebido quando tomamos decisões como forma de justificar decisões passadas. Muitas pessoas consideram o dinheiro já gasto como prerrogativa para justificar decisões e investimentos futuros.

Por exemplo, esse viés pode ser facilmente percebido quando há necessidade de investir mais dinheiro em algo em que já foi investida uma quantia razoável, mas há um alto de risco de fracasso. Por exemplo, você investiu 90% das suas reservas em um app e 1 mês antes do lançamento apareceram mais dois na Google Play, que fazem exatamente o mesmo. Você investe os 10% restantes?

- **Evidência de confirmação:**



Este viés se manifesta pela tendência humana de levantarmos informações que são consistentes com o que nós queremos ouvir ou encontrar, e retiramos evidências contraditórias.

Por exemplo, Peter acha que sua feature da Sprint irá melhorar a estabilidade da conexão Wi-Fi do smartphone. Ao final da Sprint, houve melhora de 10% na conectividade, mas ao todo foram implementadas três features relacionadas por diferentes desenvolvedores (Peter acredita que sua feature foi a principal responsável pela melhoria).

Uma dica importante para evitar esse viés, é envolver no processo de decisão pessoas externas ao problema, com o objetivo de avaliar e validar informações que fujam a esse componente.

Capítulo 5. Métodos de Gestão e Desenvolvimento de Software

Este capítulo aborda as metodologias ágeis para gestão de processos e projetos em desenvolvimento de software, bem como a gestão de ciclo de vida de produtos, sempre alinhados aos objetivos estratégicos da empresa. Apesar de ser uma responsabilidade que geralmente não compete ao arquiteto de software, é importante que ele conheça essas metodologias, com o objetivo de conseguir identificar em seu contexto organizacional qual seria a ideal para entrega de valor da sua equipe.

A base para as duas principais metodologias ágeis de desenvolvimento de software, Scrum e XP, é o Manifesto Ágil. Este manifesto contempla os valores e princípios sobre os quais todos os processos ágeis são baseados.

O manifesto ágil pode ser acessado em www.manifestoagil.com.br, e foi elaborado e mantido pelos principais profissionais envolvidos na cultura ágil de desenvolvimento de software. Vários deles são criadores de metodologias como Scrum, XP, Crystal, entre outras. Os valores fundamentais presentes no manifesto são:

- A colaboração dos clientes acima da negociação de contratos.
- O funcionamento do software acima de documentação abrangente.
- Os indivíduos e suas interações acima de procedimentos e ferramentas.
- A capacidade de resposta a mudanças acima de um plano pré-estabelecido.

Além disso, o manifesto define 12 princípios que direcionam os processos ágeis e suas práticas:

- Até mesmo mudanças tardias de escopo no projeto são bem-vindas.
- Colaboração com clientes mais do que negociação de contratos.
- Softwares funcionais são entregues frequentemente (semanal, ao invés de mensal).

- Rápida adaptação às mudanças.
- Softwares funcionais são a principal medida de progresso do projeto.
- Design do software deve prezar pela excelência técnica.
- Cooperação constante entre as pessoas que entendem do 'negócio' e os desenvolvedores.
- Garantir a satisfação do cliente, entregando rápida e continuamente software funcionais.
- Software funcional mais do que documentação extensa.
- Projetos surgem por meio de indivíduos motivados, devendo existir uma relação de confiança.
- Simplicidade.
- Indivíduos e interações mais do que processos e ferramentas.
- Responder a mudanças mais do que seguir um plano.

O ponto importante de conhecer os valores e princípios acima, é que, para adotar processos ágeis em uma organização, deve-se buscar um alinhamento da cultura da empresa desde o nível estratégico até o operacional.

Assim, a adoção de um processo ágil não começa, necessariamente, pela adoção das atividades e artefatos, mas por uma mudança organizacional bem endereçada.

- **Scrum:**

Uma das metodologias ágeis mais usadas atualmente no processo de desenvolvimento de software é o Scrum. Talvez pela fama de ser usada em empresas de destaque do Vale como Google, Facebook, Spotify, etc., o Scrum começou a ser adotado massivamente por startups e vem cada vez mais sendo adotado por grandes e tradicionais empresas.

Uma definição formal do Scrum pode ser dada por *"um framework no qual as pessoas podem endereçar problemas complexos, enquanto de forma produtiva e criativa entregam produtos de alta possibilidade de valor."*

Scrum é um framework no qual as pessoas podem endereçar problemas complexos, enquanto de forma produtiva e criativa entregam produtos de alta possibilidade de valor, tendo como característica ser uma abordagem:

- "Lightweight";
- Simples de entender;
- Difícil de dominar.

Scrum é fundamentado na teoria de controle do processo empírico. O Empirismo assegura que o conhecimento vem da experiência e da tomada de decisão a partir do que é conhecido. Desta forma, é possível identificar no Scrum os três pilares do processo empírico:

- Transparência: "Aspectos significativos do processo devem estar visíveis aos responsáveis pelos resultados. Esta transparência requer aspectos definidos por um padrão comum, para que os observadores compartilharem um mesmo entendimento do que está sendo visto."
- Inspeção: "Os usuários Scrum devem, frequentemente, inspecionar os artefatos Scrum e o progresso em direção a detectar variações. Esta inspeção não deve, no entanto, ser tão frequente a ponto de atrapalhar a própria execução das tarefas. As inspeções são mais benéficas quando realizadas de forma diligente, por inspetores especializados no trabalho a se verificar."
- Adaptação: "Se um inspetor determina que um ou mais aspectos de um processo desviou para fora dos limites aceitáveis, e que o produto resultado será inaceitável, o processo ou o material sendo produzido deve ser ajustado. O ajuste deve ser realizado o mais breve possível para minimizar mais desvios."

> Scrum Teams deliver products iteratively and incrementally, maximizing opportunities for feedback. Incremental deliveries of "Done" product ensure a potentially useful version of working product is always available. - The Scrum Guide - Developed and sustained by Ken Schwaber and Jeff Sutherland

O time Scrum é bastante simples em sua definição, sendo composto por três elementos: o Product Owner, o Scrum Master e a equipe de Desenvolvimento.

- Product Owner: "É o responsável por maximizar o valor do produto e do trabalho do Time de Desenvolvimento. O Product Owner é a única pessoa responsável por gerenciar o *Backlog* do Produto. O gerenciamento do Backlog do Produto expressando claramente cada item, o seu valor para o cliente e a sua prioridade."

- O Time de Desenvolvimento: "Consiste de profissionais que realizam o trabalho de entregar uma versão usável, que potencialmente incrementa o produto "pronto" ao final de cada Sprint, sendo que eles são estruturados e autorizados pela organização para ordenar e gerenciar seu próprio trabalho, com base nas metas estabelecidas pelo Product Owner e as regras do processo determinadas pelo Scrum Master."

- Scrum Master: "É o responsável por garantir que o Scrum seja entendido e aplicado, de forma que o time como um todo esteja aderente à teoria, as práticas e regras do Scrum. O seu papel deve ser de servo-líder para o Time Scrum."

E como fica o arquiteto de software dentro do time Scrum? Em geral, o arquiteto faz parte do time de desenvolvimento. Em muitos casos o arquiteto é o líder funcional ou hierárquico desse time. Um arquiteto corporativo, devido a sua atuação estratégica, quando se trata de um produto estritamente tecnológico, pode assumir até a função de Product Owner, em situações específicas.

Com relação aos eventos Scrum, por se tratar de uma metodologia de gestão de projetos de software, em que um dos recursos valiosos é o cronograma, estes são

sempre delimitados por tempo. São cinco eventos que compõe o Scrum: Sprint, Sprint Planning, Daily Meeting, Sprint Review e Sprint Retrospective.

- Sprint: “Assim como o guia do processo afirma, “o coração do Scrum é a Sprint”, sendo de um mês ou menos, durante o qual uma versão potencialmente utilizável do produto é criada.

É importante mencionar que durante uma Sprint, de acordo com o processo, não se pode:

- Diminuir metas de qualidade.
- Realizar mudanças que possam pôr em perigo o objetivo da Sprint.
  - Escopo pode ser clarificado e renegociado entre o Product Owner (PO) e o Time de Desenvolvimento.
- Cancelamento da Sprint: somente o PO tem autoridade para cancelar a Sprint caso o objetivo se torne obsoleto. O que pode ocorrer, por exemplo, quando há uma mudança de direção da organização ou do projeto.
- *Sprint Planning*: “O trabalho a ser realizado na Sprint é projetado na reunião de planejamento. Este plano é criado de forma colaborativa com o todo o Time Scrum. O tempo da reunião é proporcional à duração da Sprint. Para Sprints de um mês, são necessárias oito horas de duração”.
- *Daily Meeting*: “É uma reunião de 15 minutos, bem objetiva e sucinta, com o objetivo de sincronizar as atividades para as próximas 24 horas. O ideal é que seja mantida no mesmo horário e local, todos os dias. Todos os envolvidos devem descrever suas metas pessoais em um modelo: o que foi feito ontem, o que será feito hoje ou se está com algum impedimento em suas atividades.”
- *Sprint Review*: “Nessa reunião é realizada a inspeção da entrega da Sprint e a adaptar o Backlog do produto, se necessário. É idealmente de quatro horas, para um mês de Sprint.”

- *Sprint Retrospective*: “é a oportunidade para o Time Scrum inspecionar a si próprio e criar um plano de melhorias a serem aplicadas na próxima Sprint. Novamente, é uma reunião, “*time-boxed*”, mas, no caso, três horas para um mês de Sprint.”

Com relação aos artefatos Scrum, eles se diferem bastante de metodologias tradicionais, como a Cascata (Waterfall). Um dos princípios das metodologias ágeis é gerar documentação apenas suficiente para a execução do processo sem falhas. Assim, a quantidade de artefatos utilizados no Scrum é bastante reduzida, sendo resumidos em: Product Backlog, Sprint Backlog e Increment.

- Product Backlog: “É uma lista ordenada de tudo que deve ser necessário no produto, e uma referência dos requisitos para qualquer mudança a ser feita no produto. O Product Owner é o seu responsável, incluindo seu conteúdo, disponibilidade e ordenação. O Backlog deve listar todas as características, funções, requisitos, melhorias e correções que formam as mudanças que devem ser feitas no produto nas versões futuras.”

- Sprint Backlog: “É um conjunto de itens do Backlog do Produto, selecionados para a Sprint, juntamente com o plano para entregar o incremento do produto e atingir o objetivo da Sprint. O Backlog da Sprint é a previsão do Time de Desenvolvimento, sobre qual funcionalidade estará no próximo incremento.”

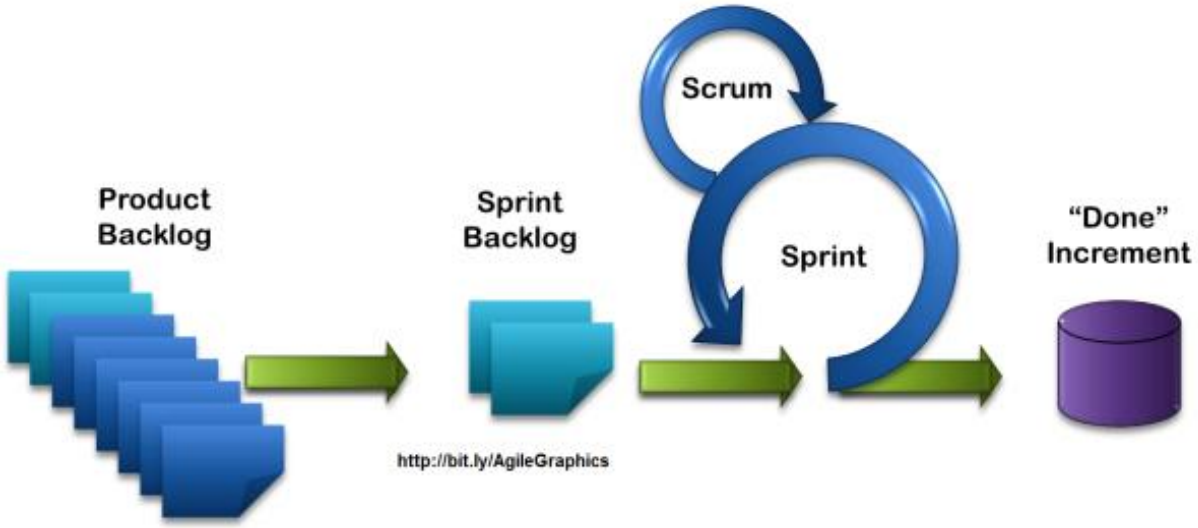
- Increment: “é a soma de todos os itens do Backlog do Produto completados durante a Sprint, e o valor dos incrementos de todas as Sprints anteriores. Ao final da Sprint, um novo incremento deve estar “Pronto”, o que significa que deve estar na condição utilizável e atender à definição de “Pronto” do Time Scrum. Este deve estar na condição utilizável, independente do Product Owner decidir por liberá-lo realmente ou não.”

Definition of "done"

It is important that the product owner and the team agree on a clear definition of "done".

VERY important!

A figura abaixo mostra de forma resumida e gráfica o ciclo de vida de um projeto Scrum, com seus processos e artefatos.



- **XP – Extreme Programming:** O XP foi uma das primeiras metodologias ágeis de desenvolvimento de software a se popularizar. Desenvolvida por Kent Beck, foi apresentado com maior ênfase em 2000 através do livro Extreme Programming Explained. Neste livro, Kent define o XP como: "a style of software development focusing on excellent application of programming techniques, clear communication, and teamwork which allows us to accomplish things we previously could not even imagine."

Os valores que norteiam o XP são herdados do Manifesto Ágil, sendo os quatro mais destacados: comunicação, feedback, coragem e simplicidade.

- **Comunicação**: "XP foca em construir um entendimento pessoa a pessoa do problema, com o uso mínimo de documentação formal e com o uso máximo de interação "cara a cara" entre as pessoas envolvidas no projeto. Práticas como

a programação em pares, testes e comunicação direta com o cliente, têm o objetivo de estimular a comunicação entre gerentes, programadores e clientes."

- **Feedback**: "Os programadores obtêm feedback sobre a lógica dos programas escrevendo e executando casos de teste, enquanto os clientes realizam testes de aceitação, eles também fornecem feedback para a equipe desenvolvedora."

- **Coragem**: "É um valor um pouco vago, mas se refere à ideia de assumir riscos e romper com abordagens tradicionais da engenharia de software. Alguns exemplos desse valor são a refatoração de código, a reescrita de componentes e de partes da arquitetura, entre outros."

- **Simplicidade**: "XP sugere que cada membro da equipe adote a solução mais simples que possa funcionar. O objetivo é fazer aquilo que é mais enxuto hoje e criar um ambiente em que o custo de mudanças no futuro seja baixo."

O XP utiliza de quatro variáveis de controle para garantir o sucesso do projeto conforme planejado:

- **Custo**: valor financeiro do projeto, seja ele com gastos com hardware, software, pessoal, etc.

- **Tempo**: tempo entre a especificação do sistema (início do projeto) e sua entrega pronta.

- **Qualidade**: é relativo ao nível de aceitação do sistema em relação ao que foi planejado versus o que foi entregue.

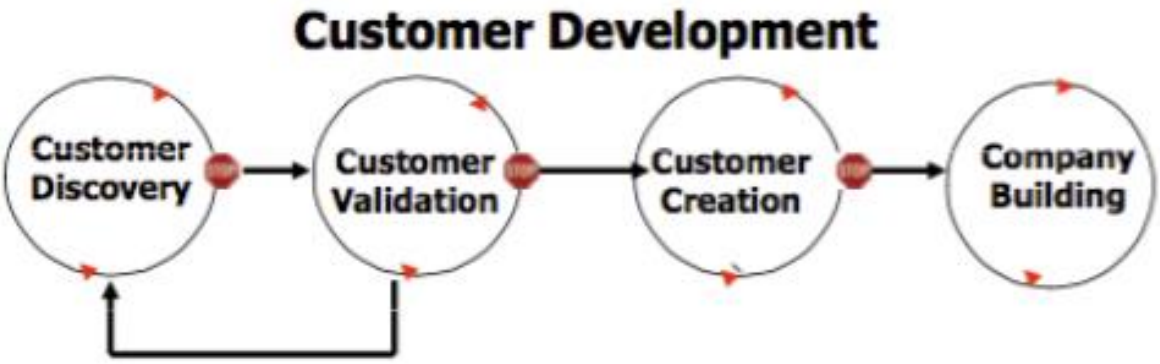
- **Escopo**: é a definição de todas as funcionalidades que serão entregues após o sistema pronto.

No XP a qualidade é uma variável não negociável, ou seja, parte-se do pressuposto que o projeto deve ser entregue na exata qualidade em que foi demandado. O tempo e custo são definidos, ou seja, não há como ser alterado por fatores internos. Sendo assim, na metodologia XP, para manter o equilíbrio do projeto,

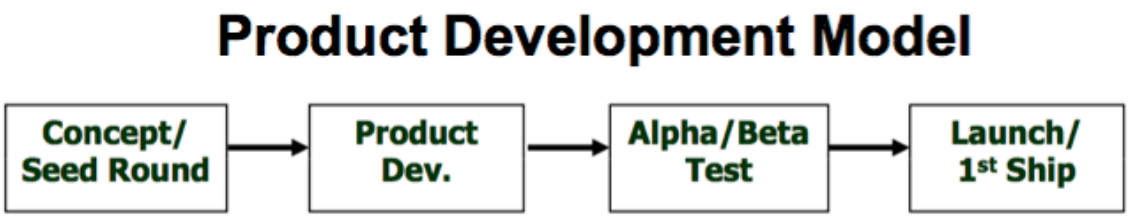
o escopo passa a ser o elemento negociável. Caso seja necessária a diminuição de escopo, as funcionalidades menos valiosas devem ser adiadas ou canceladas.

- **Customer Development:**

Saindo um pouco da área de projetos de software e entrando na área de produtos, é importante para o arquiteto de software entender, do ponto de vista estratégico, todas as preocupações que estão envolvidas ao se criar e manter um produto. Por mais que este arquiteto não vá atuar como empreendedor com seu próprio negócio, ele terá esse tipo de atuação dentro da empresa que trabalha, sendo às vezes responsável por definir estratégias de produto com a visão tecnológica.



Neste sentido, essa é a visão do Customer Development, técnica criada por Steve Blank, que descreve um ciclo contínuo para descobrir problema e solução de menor custo, que resolva o problema do cliente. O objetivo desta técnica é criar produtos que capturem ou gerem necessidades em seus clientes, mapeando quem é o cliente, suas necessidades e, sobretudo, se ele estaria disposto a pagar por uma solução.



O modelo tradicional de desenvolvimento de produtos passa por fazes e etapas bem definidas, que vão desde a concepção até o lançamento. O problema desta abordagem é que uma questão extremamente importante não é perguntada logo no início: alguém quer esse produto?

O avanço tecnológico mostra que, atualmente, praticamente qualquer aplicação é tecnicamente possível de ser construída. A questão relevante não é se é possível fazer algo, e sim se é necessário fazer algo, empregar recursos humanos e financeiros para uma tarefa.

O processo de Customer Development é definido através de quatro etapas: Customer Discovery, Customer Validation, Customer Criation e Customer Building.

- *Customer Discovery*: "Nesta etapa do processo, Blank descreve como a empresa deve encontrar o alinhamento entre o problema e a solução a ser construída. Alguns valores são fundamentais nessa etapa:
  - *Stop Selling, Start Listening*: Dentro da organização há apenas opiniões, não fatos. Para encontrar os fatos, o empreendedor deve buscá-los fora.
  - *Test your hypothesis:* Duas hipóteses são fundamentais: a concepção do problema e do produto, que devem ser ambas testadas. Nessa etapa, a empresa (ou o empreendedor) deve buscar quais são os maiores problemas do cliente, e verificar se o produto de fato resolve esses problemas, dentro do ponto de vista dos possíveis clientes.
- *Customer Validation*: É o segundo passo do processo de Customer Development e tem como principal objetivo responder à questão: *os clientes pagarão pelo produto?* Deve-se entender como funciona o ciclo de vendas e o modelo financeiro. O processo de vendas e de distribuição do produto deve ser validado, sendo que o objetivo dessa etapa é encontrar um modelo de vendas adaptável, escalável e, então, validá-lo.
- Customer Creation: Uma vez que o modelo de negócio foi encontrado, a etapa agora é buscar por investimento e gerar maior demanda por vendas. Já é

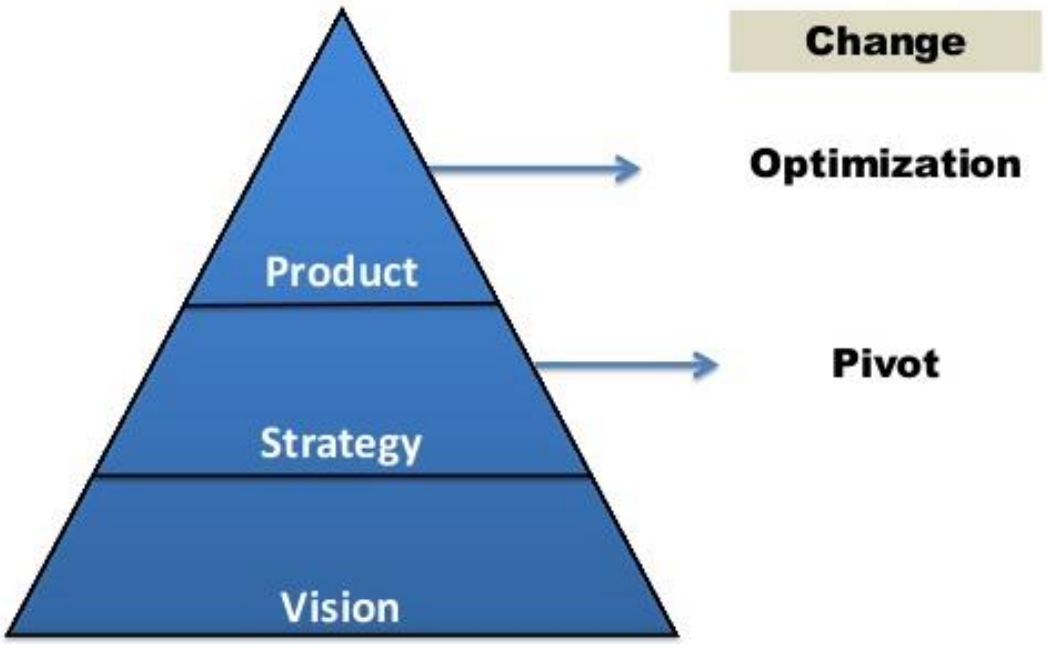
possível ter uma perspectiva da engenharia financeira proporcionada pelo produto e negócio. Essa etapa também deve descrever os mercados em que o produto estará focado, assim como seus riscos e concorrentes.

- Customer Building: Por fim, de acordo com o Steve Blank, essa etapa marca o fim da transição entre o foco do aprendizado para o foco na execução. Nessa etapa o objetivo é ser o melhor e o líder de mercado, atingindo o grande público.

- **Lean Startup:**

Lean Startup é uma metodologia criada por Eric Ries, baseada em Customer Development e que objetiva reduzir a incerteza e evitar o desperdício de tempo e dinheiro durante o processo de construção de um produto de tecnologia.

A maneira como esse objetivo é alcançado é através da diminuição dos ciclos de desenvolvimento do produto, usando experimentação de hipóteses de negócio, ciclos de liberação do produto e aprendizagem validada. A base para isso é a construção de um produto minimamente viável ou MVP. O MVP é um protótipo que possa ser utilizado diante de potenciais clientes, com o objetivo de validar a hipótese de mercado central do empreendedor e do produto que está sendo idealizado. Não necessariamente é a menor versão imaginada para o produto.



A base da estratégia e visão do Lean Startup é a pirâmide: visão, estratégia e produto:

- Visão: É a idealização do produto, o Product/Market fit em que a ideia está posicionada. Dificilmente muda.
- Estratégia: É o direcionamento encontrado para viabilizar o negócio, focado em aspectos de escalabilidade, monetização e viabilização da visão. Nesse ponto, quando é necessário mudar a estratégia, realiza-se um conceito do Lean Startup chamado de “pivô”.
- Produto: O produto é a materialização da estratégia e caso seja necessário ajustá-lo, é dito que se faz uma “otimização”. Por exemplo, em uma feature, para deixá-la mais adequada ao mercado e ao modelo de monetização, ou para deixar o produto mais escalável de acordo com a estratégia.

Capítulo 6. Estratégias de Gestão de Configuração e Versionamento

Se você trabalha com desenvolvimento de sistemas, certamente em algum período de sua carreira deve ter se deparado com alguma perda de dados, por mínimo que seja. Enquanto você é a única pessoa a trabalhar em um código, esse problema pode ser minimizado com backups automatizados, por exemplo. Mas a partir do momento em que várias pessoas trabalham na mesma classe ou porção do código, o uso de backups para essa função pode mais atrapalhar do que ajudar.

Diante disso, se tornam cada vez mais populares e indispensáveis os sistemas de controle de configuração e versionamento. Este capítulo trata sobre as necessidades acerca da gestão de configuração e das ferramentas que fornecem estratégias de configuração e versionamento.

Dentro do ITIL, existe uma disciplina que fala sobre a Gestão de Configuração. Essa disciplina é uma área da engenharia de software responsável pelo controle de versão, o controle de mudanças e a auditoria de configurações. Não é difícil imaginar os casos em que essa área é imprescindível, de forma que sua correta aplicação pode determinar o sucesso de um projeto:

- Como proceder quando o cliente reporta um bug e não pode atualizar para a versão mais nova?
- Como proceder se um desenvolvedor já terminou seu desenvolvimento e quer iniciar um novo, porém esse novo desenvolvimento não pode estar no código a ser lançado quando todos os demais terminarem suas tarefas?
- Como distribuir uma correção para todas as versões disponíveis do sistema?

Essas e outras questões acerca do desenvolvimento de software são resolvidas através de sistemas de controle de versão, como veremos a seguir.

Os benefícios dos sistemas de controle de versão vão além do maior controle sobre o código em produção e desenvolvimento. Podemos citar como benefícios, entre outros:

- Backup e Restore: Arquivos são salvos à medida que são editados.
- Sincronização: Os arquivos podem ser compartilhados entre várias pessoas e elas ainda assim estarem com a última versão disponível.
- Undo: Em caso de perda de estabilidade ou desorganização, é possível retornar para a última versão estável a qualquer momento.
- Track Changes: É possível identificar o motivo das alterações em cada arquivo.
- Sandboxing: Em caso de grandes mudanças, é possível fazer à parte e de forma controlada.
- Branch e Merge: É possível isolar os arquivos, fazer alterações, testar e, somente após isso, reintegrar à versão estável.

Apesar do Controle de Versionamento ser um conceito, existem diversas ferramentas no mercado que aplicam esse conceito em funcionalidades e facilitam o trabalho de controle de versão. Vamos falar de algumas das mais utilizadas: CVS, SNV, Git e Mercurial.

- **CVS:**

O Concurrent Version System – CVS – é uma das primeiras ferramentas disponíveis para controle de versão e amplamente utilizada por muito tempo. Criada na década de 80, o CVS inicialmente lidava com conflitos entre dois programadores, permitindo apenas que a última versão submetida fosse utilizada. Hoje o CVS já trabalha com o conceito de branch, facilitando o controle das versões. É considerado o sistema de controle de versão mais maduro, pois foi desenvolvido há bastante tempo e não recebe muitos pedidos de novas funcionalidades.

- **SVN:**

O Apache Subversion – SVN – foi criado após o CVS e distribuído pelo projeto Apache. O motivador de sua criação foi corrigir alguns problemas no sistema do CVS, mantendo compatibilidade com ele. O SVN emprega um conceito de operações

atômicas, assim, as alterações feitas no código-fonte são aplicadas ou não, o que significa que não há mudanças parciais que poderiam deixar o código em um estágio intermediário e inconsistente.

- **Git:**

Talvez hoje a ferramenta de controle de versão mais utilizada no mundo. Foi idealizado para funcionar de forma totalmente distribuída, inclusive na Internet. Criado pela equipe de Linus Torvalds, o Git não centraliza o código em um servidor, pelo contrário, o distribui em diferentes repositórios. O Git, ainda, possui uma grande variedade de ferramentas para auxiliar o desenvolvedor, sobretudo, pelo fato de cada instância possuir um histórico próprio de alterações.

- **Mercurial:**

É um concorrente direto do Git, com funcionalidades semelhantes. É desenvolvido em python, o que garante uma boa aderência na comunidade desta linguagem. O Mercurial possui características semelhantes ao SVN, tornando a curva de aprendizado favorável para quem já está familiarizado com o SVN e deseja utilizar um gerenciador distribuído.

Além destas, existem dezenas de outras ferramentas de controle de versão, e certamente outras dezenas surgirão ao longo do tempo. Mais importante que a ferramenta é compreender o conceito de controle de versão e os benefícios que a sua adoção proporciona. Dessa forma, até uma solução caseira pode ser ideal dependendo do caso. É imprescindível para uma organização que realiza engenharia de software que ela utilize uma estratégia de gestão de configuração e versionamento adequada. Diante disso, o arquiteto de software, como líder estratégico e referência do desenvolvimento, deve saber definir, aplicar e, até mesmo, auditar a estratégia de gestão de configuração a ser seguida.

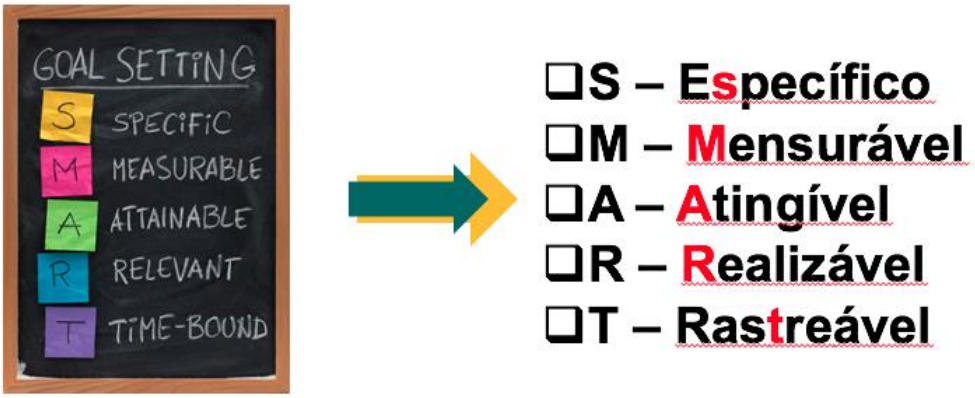
Capítulo 7. Requisitos Arquiteturais

Um dos papéis do arquiteto de software, em especial o arquiteto corporativo, é transformar as necessidades de negócio em uma arquitetura do software, que reflita estas necessidades atuais, mas que também esteja preparada para crescimentos e desdobramentos da estratégia ao longo do tempo. Existe diversas técnicas e ferramentas que auxiliam na identificação e documentação destes requisitos arquiteturais. Neste capítulo, iremos analisar duas delas: SMART e FURPS+.

Levantamento de requisitos arquiteturais não é uma tarefa simples. Geralmente ela consome tempo e esforço. Apesar do preço que se paga, é uma atividade imprescindível para o projeto. É como construir uma casa sem o projeto arquitetônico e estrutural. É possível? Sim. Mas a chance de dar errado é grande.

- **S.M.A.R.T:**

S.M.A.R.T foi proposto por Mike e Barry no artigo *SMART Requirements,* como o objetivo de tornar os requisitos de sistema comparáveis com objetivos da equipe. S.M.A.R.T é um acrônico em que cada uma das letras representa um escopo para: Specific, Measurable, Attainable, Realisable e Traceable.



Specific – Significa que um requisito deve refletir exatamente aquilo que é necessário ser feito. Portanto ele deve ser:

- Sem ambiguidade.
- Consistente: mesma terminologia de outros requisitos.

- Simples: falar de somente um requisito.

Measurable significa que um requisito deve se traduzir em uma funcionalidade que, após construída, deve ser possível ser medida

Attainable significa que o requisito deve se traduzir em uma funcionalidade viável de ser implementada, não estando além da percepção humana ou soluções teóricas.

Realisable significa que o requisito pode ser contemplado, considerando as restrições das quais o projeto e o sistema estão sendo desenvolvidos.

- É considerada a parte mais difícil do S.M.A.R.T.
- Geralmente Atingível e Realizável são considerados em paralelo.

Traceable significa que um requisito deve ser rastreado a partir da sua concepção ao longo da sua especificação, design, implementação e testes.

- Determina as fontes dos requisitos (Pessoas ou Instituições).
- Justificativas de negócio.
- Relações entre eles (sobretudo em caso de impacto).
- Suas criticidades.

O critério SMART é considerado simples do ponto de vista conceitual, mas sua aplicação exige cautela ao analisar cada requisito seguindo os seus cinco princípios. Sob o ponto de vista do arquiteto de software, é importante que ele tenha uma análise crítica sobre os requisitos de forma a garantir que eles estejam claros o suficiente para sua equipe e, sobretudo, para que sua arquitetura de software entregue o valor esperado para cada um.

- **FURPS+:**

FURPS+ é um acrônimo para Funcionality, Usability, Reliability, Performance e Supportability. Trata-se de uma ferramenta de aquisição e documentação de Requisitos Arquiteturais, que objetiva identificar o requisito como “uma condição ou capacidade com a qual um sistema deve estar em conformidade; quer ele seja diretamente derivado da necessidade do usuário, ou estabelecido em um contrato, padrão, especificação ou outro documento formalmente imposto”. Assim, os requisitos podem ser ordenados como:

- Functionality: Essa categoria representa os principais requisitos funcionais que impactam a arquitetura.
- Usability: Considera aspectos de interatividade, design e experiência de uso. É importante ressaltar que a usabilidade é sim um fator determinante para o sucesso de um produto.
- Reliability: Diz respeito à disponibilidade do sistema (*Up Time*), precisão de cálculos e tolerância a falhas.
- Performance: Diz respeito à capacidade do sistema em processar tarefas, como o tempo de resposta de funcionalidades, inicialização, encerramento e restauração de *backups* e falhas.
- Supportability: Está relacionado à capacidade do sistema em ser testado, adaptado, mantido, compatibilizado, parametrizado, escalado, internacionalizado e implantado.
- + Desenho: Está relacionado ao projeto do sistema como, por exemplo, especificar de antemão o banco de dados que deverá ser utilizado.
- +Implementação: Especifica restrições de implementação, como utilizar bibliotecas nativas ou de terceiros.
- + Interface: Especifica integrações e contratos de acesso para interoperabilidade. Por exemplo, acessos via REST, WebService entre outras.

- + Físico: Especifica restrições de hardware e de implantação como, por exemplo, espaço de armazenamento necessário, RAM entre outros.

Capítulo 8. Modelagem Arquitetural

Vimos no capítulo 5 que é muito importante envolver o cliente, interno ou externo, durante todo o processo de desenvolvimento. Com o Customer Development, o cliente é o todo do desenvolvimento e ele deve ser sempre consultado a cada passo. Porém, muitas vezes é difícil para o cliente entender requisitos, principalmente os mais técnicos. Até mesmo na área de TI alguns elementos precisam ser transformados em modelos, para o entendimento geral.

Este capítulo trata destes modelos, principalmente utilizando a visão 4+1 para modelagem.

Modelos são uma simplificação da realidade, oferecendo, com isso, uma perspectiva para se antecipar e detalhar requisitos, informações e riscos. A modelagem de sistemas traz uma série de benefícios, dentre os quais podemos destacar:

- Compreensão de sistemas complexos: Um modelo pode reduzir em escala a complexidade de um projeto de forma a focar nos aspectos mais relevantes para uma determinada etapa. Por exemplo, uma maquete na construção civil ou um diagrama de entidades e relacionamentos na engenharia de software

focam nos conceitos e elementos fundamentais para se estabelecer a composição do projeto.

- Explorar alternativas e levantar informações: Assim como mencionado no capítulo 4, o processo de tomada de decisão de qualidade passa pelo adequado levantamento de alternativas de solução, e suas informações e riscos inerentes. Diante disso, um modelo auxilia ao representar o projeto de forma a explorar essas alternativas e instigar o levantamento de informações relevantes para cada etapa.

- Estabelecer a fundação para a implementação: Conforme mencionado anteriormente, ao se estabelecer um modelo ou uma representação do que o projeto deverá ser, é possível ter um alinhamento de todos os envolvidos, no que deverá ser implementado.

- Capturar requisitos com precisão e comunicar com maior objetividade: Ao modelar um determinado requisito, por exemplo, extraindo suas entidades e conceitos principais, pode-se validar o entendimento e comunicar com as partes interessadas que, de fato, o entendimento está correto e os próximos passos podem ser tomados.

Um modelo arquitetural de software contém a definição de todas as partes que compõe um software, e como esses elementos interagem entre si. Também descreve o comportamento destas partes de forma isolada. Para isso, o processo de modelagem é orientado através dos princípios abaixo:

- Escolha do modelo mais adequado: Um modelo deve atender à necessidade do contexto em que ele está inserido e para qual público alvo ele se destina. Por exemplo, os principais interessados em um diagrama de implantação são os profissionais que decidirão sobre a infraestrutura do projeto.

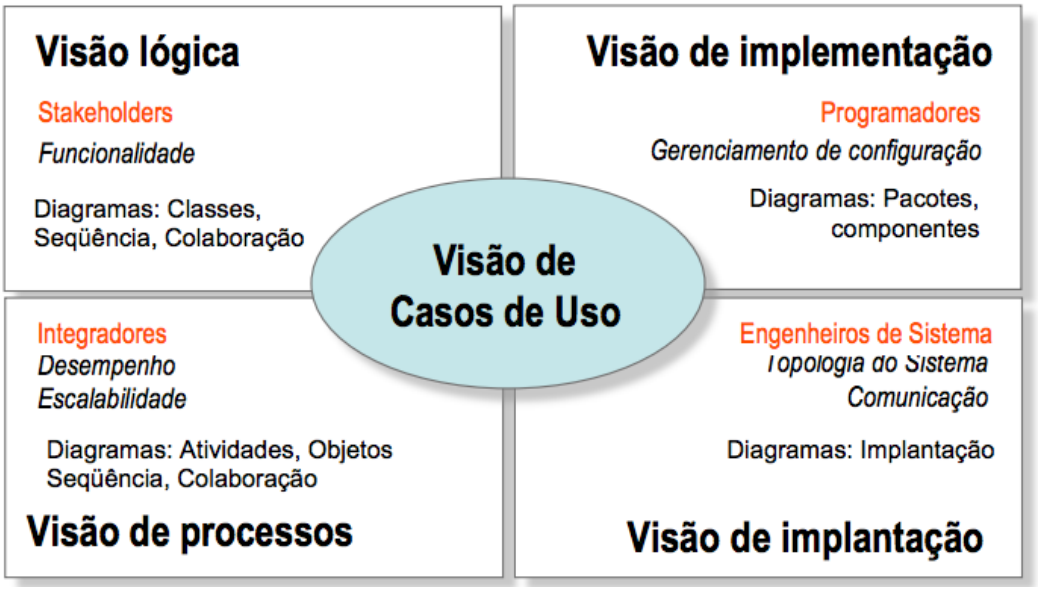
- Utilizar níveis diferentes de precisão e abstração: Cada modelo deve ter um objetivo bem definido. Por exemplo, modelar as entidades do projeto, os casos de uso arquiteturais, entre outros. Assim o modelo deve atender ao seu objetivo

e abstrair os elementos que não são relevantes, para expressar determinada informação ao interlocutor.

- Procurar conectar o modelo à realidade: Pode parecer óbvio, mas um modelo deve, acima de tudo, estar conectado à sua realidade. Ou seja, deve ser uma representação fiel ao seu contexto, considerando entidades, linguagem e notações que fazem parte do escopo do objetivo ao qual está resolvendo.

- Nenhum modelo único é suficiente: Esse último princípio está relacionado ao fato de um modelo atender a um único objetivo e interlocutor. Assim, um modelo que se destina a explicitar informações para os desenvolvedores de software poderá ser diferente do modelo que apresenta informações do mesmo projeto para analistas de negócio e cliente final.

Neste cenário, a visão 4+1 utiliza como suporte a UML, para poder modelar a arquitetura de um software como bem entender, e comunicar seus componentes estruturais e comportamentais.



- Visão Lógica: Possui como foco as funcionalidades que o sistema proporciona aos usuários finais. Os digramas da UML geralmente utilizados são de classes, atividade e estados.

- Visão de Implementação: Ilustra o sistema do ponto de vista do desenvolvedor, e se concentra na gestão do software. Diagramas de componentes e de Pacote são geralmente utilizados para representar essa visão.

- Visão de Processos: Lida com os aspectos dinâmicos do sistema, explicando os processos do sistema, como eles se comunicam e comportamentos de "*runtime*". Um diagrama frequentemente utilizado, é o de atividade.

- Visão de Implantação ou Física: Possui como foco a topologia de implantação da arquitetura em sua camada física, assim como as conexões entre os componentes físicos. O diagrama de implantação da UML é frequentemente utilizado para expressar essa visão.

- Visão de Casos de Uso ou de Cenários: Classificada como a 5ª visão, ela expressa a arquitetura utilizando um subconjunto de casos de uso. Ela descreve uma sequência de iterações entre objetos, processos, além de servir para validar o projeto da arquitetura. Ela também serve como ponto de partida para testes, protótipos arquiteturais e provas de conceito.

A partir das vantagens e objetivos da modelagem descritos nesta seção, assim como o direcionamento proporcionado pela Visão 4+1, é possível ao arquiteto de softwares fornecer os fundamentos do produto a ser construído, tendo como objetivo validar e representar de forma aderente à realidade os aspectos capturados e significantes à arquitetura de software em desenvolvimento.

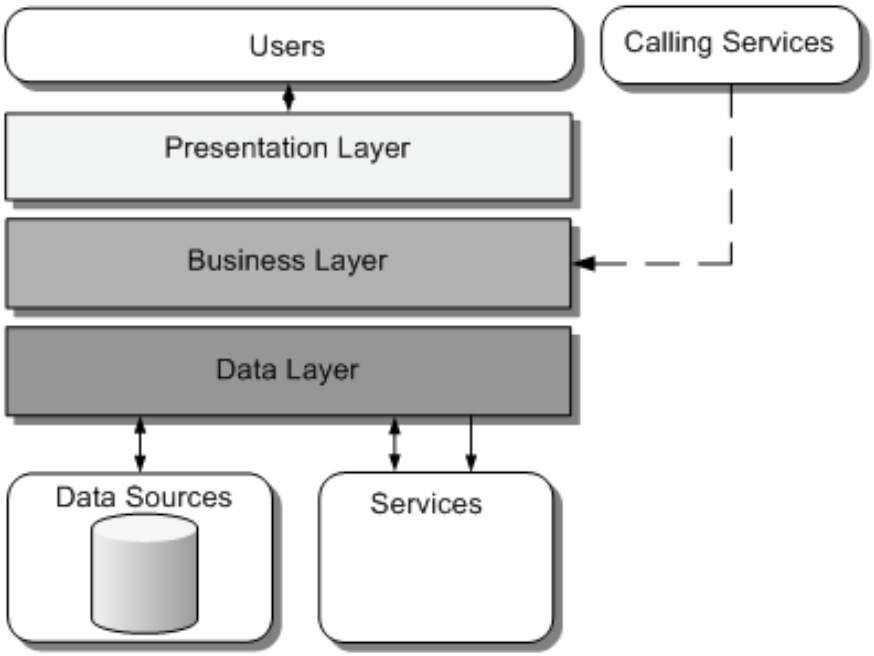
Capítulo 9. Estilos e Padrões Arquiteturais

À medida que os sistemas crescem e ficam complexos, ou então quando um sistema já nasce grande, percebe-se que muitos elementos de software são comuns em várias etapas. Além disso, muitas necessidades funcionais e arquiteturais já foram pesquisadas e desenvolvidas de forma otimizada por outros desenvolvedores, que porventura documentaram suas soluções e disponibilizaram para a comunidade. Este capítulo trata destas características comuns de sistemas que faz com que pertençam a uma mesma família, e das soluções comuns para problema recorrentes em desenvolvimento de sistemas. Estas abstrações ou padrões são independentes de linguagem de programação, uma vez que seu objetivo é descrever e padronizar problemas genéricos, independentemente da plataforma tecnológica utilizada.

Uma definição de estilo arquitetural, segundo David Garlan e Mary Shaw é "(...) uma família de sistemas em função de um padrão de organização estrutural. Mais especificamente, um estilo arquitetural determina o vocabulário de componentes e conectores que podem ser utilizados em instâncias desse estilo, juntamente com um conjunto de restrições sobre como elas são combinadas". Vários softwares podem (e devem) compartilhar o mesmo estilo arquitetural, como forma de simplificar o desenvolvimento e facilitar o entendimento da documentação. Existem vários estilos arquiteturais disponíveis no mercado. Como forma de apresentar mais detalhadamente alguns deles, iremos detalhar o funcionamento de quatro estilos: Layered, Message-bus, N-Tier e Service-Oriented.

A seguir serão abordados alguns dos principais estilos presentes e em alta no mercado. São eles: *Layered, Message-bus, N-Tier / 3-Tier e Service-Oriented.*

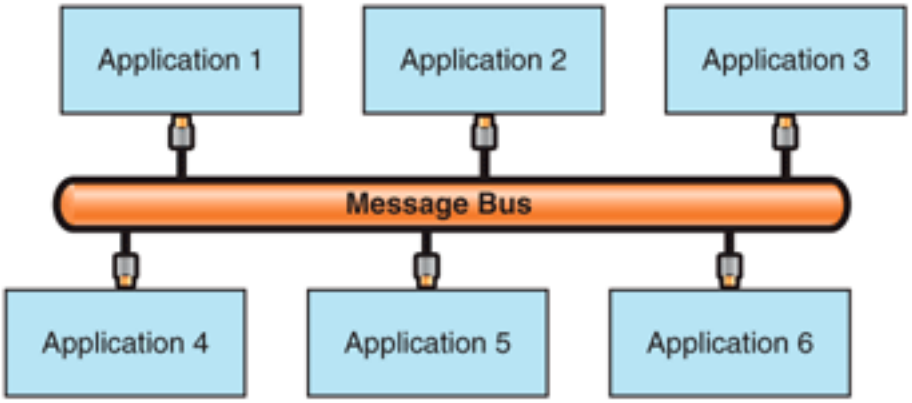
- **Layered:**



O estilo arquitetural layered tem como objetivo agrupar funcionalidades relacionadas dentro de uma aplicação em camadas. Este estilo tem como benefícios:

- Abstração: Cada camada abstrai a visão e detalhes da camada inferior para a camada superior, de forma que cada uma tenha as suas responsabilidades bem definidas, focando no relacionamento entre elas.
- Gerenciamento: A separação em camadas auxilia na identificação de dependências e manutenção do código, de forma organizada em seções.
- Desempenho: A possibilidade de distribuir as camadas em nodos físicos separados (*tiers*) aumenta a escalabilidade, tolerância a falhas e desempenho.
- Reuso: Camadas inferiores não possuem dependências das camadas superiores, apenas disponibilizam métodos e serviços para que as superiores as acessem de forma coerente e bem definida.
- Isolamento: Permite isolar atualizações de tecnologia e manutenções, com o objetivo de reduzir risco e minimizar o impacto no sistema como um todo.

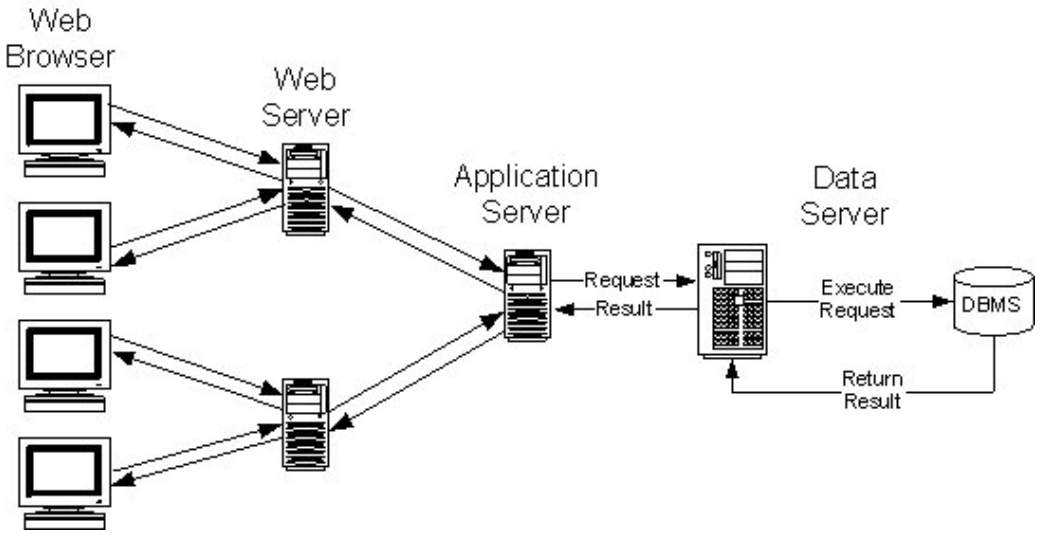
- **Message-bus:**



O estilo Message-bus é uma arquitetura em que existe uma troca de mensagens, utilizando um ou vários canais de comunicação. Este estilo arquitetural permite que ocorram alterações em cada aplicação, sem que afete a comunicação com as demais, desde que a interface com o barramento seja mantida. Esta arquitetura possui as seguintes vantagens:

- Flexibilidade: As alterações de uma aplicação e suas complexidades de comunicação ficam abstraídas das demais aplicações envolvidas.
- Baixa complexidade: A complexidade das aplicações é reduzida, pois elas precisam basicamente compreender como se comunicar com o barramento e ele, por sua vez, ficará responsável por entregar as mensagens e processos às aplicações destinatárias.
- Escalabilidade: Múltiplas instâncias das aplicações podem ser conectadas ao barramento para processar múltiplas requisições, assim como o próprio barramento pode ser escalado sem impactar as demais aplicações.
- Baixo acoplamento: Tendo em vista que cada aplicação expõe suas interfaces de comunicação ao barramento, não há dependência das aplicações em si, e as atualizações e mudanças podem ser realizadas por meio da exposição das mesmas interfaces.

- **N-Tier:**



O estilo arquitetural N-tier divide a aplicação em camadas, que podem rodar na mesma máquina ou em máquinas separadas. Este modelo naturalmente permite maior disponibilidade e escalabilidade, já que cada camada roda de forma independente. Esta arquitetura possui as seguintes vantagens:

- Flexibilidade: Devido ao fato de cada nodo poder ser gerenciado e escalado de forma independente, a flexibilidade é aumentada.

- Manutenção: Novamente a independência entre os nodos faz com que cada um possa ser atualizado e alterado sem que isso ocasione em problemas na aplicação como um todo.

- Escalabilidade: O fato de cada nodo ser baseado em uma camada da aplicação faz com que seja possível escalá-la com razoável facilidade. Por exemplo, criando mais nodos e realizando balanceamento via DNS.

- Disponibilidade: Ao escalar, e permitir que mais de um novo opere o mesmo serviço, melhora-se também a disponibilidade da aplicação como um todo. Pois, caso um serviço fique indisponível, o outro poderá assumir as requisições até que o primeiro seja reestabelecido.

- **Service Oriented:**

O Service Oriented Architecture – SOA – fornece as regras de negócio através de um conjunto de serviços de negócios. Estes serviços podem estar na empresa ou fora dela. Para isso funcionar, os serviços são descritos através de interfaces baseadas em padrões e previamente divulgadas, baseadas na troca de mensagens. São vantagens deste estilo arquitetural:

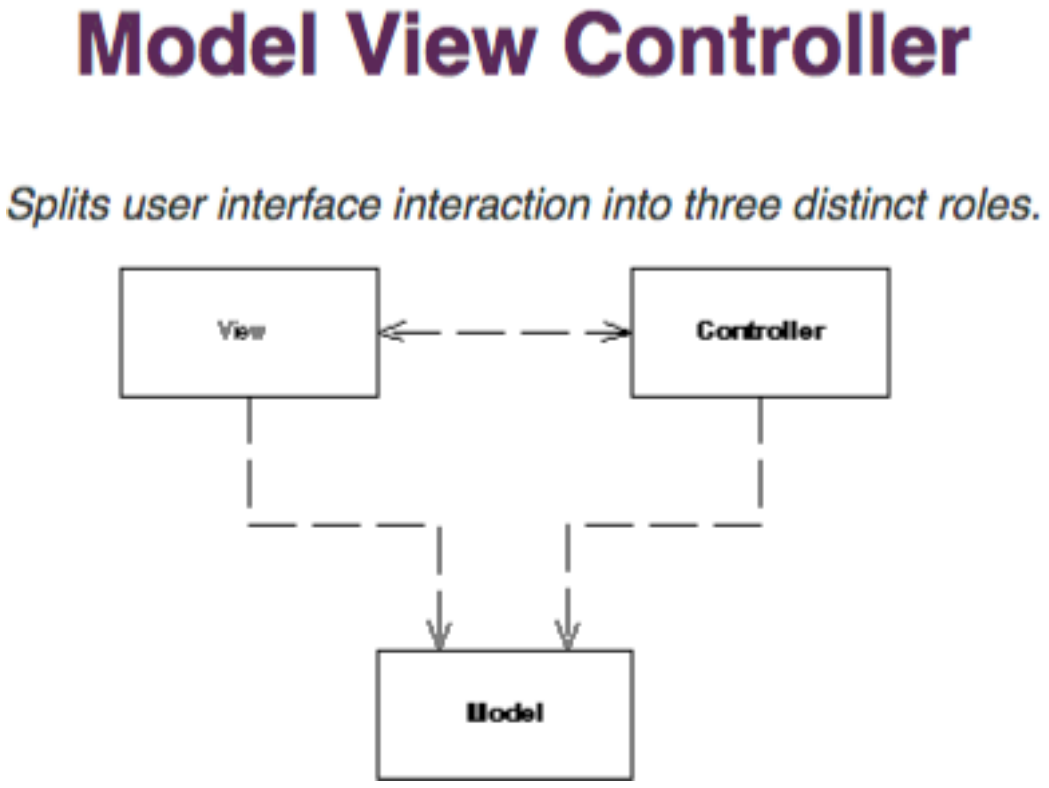
- Alinhamento de domínio: Reutilização de serviços comuns com interfaces padronizadas aumenta as oportunidades de negócios e reduz o custo.
- Abstração: Serviços são autônomos e contratuais, ou seja, possuem toda a implementação necessária para responder as requisições a partir dos parâmetros especificados em suas interfaces.
- Interoperabilidade: Porque os protocolos e formatos de dados são baseados em padrões de mercado, o provedor e o consumidor do serviço podem ser construídos e implantados em diferentes plataformas.
- Racionalização: Serviços podem ser granulares ao ponto de proporcionar uma funcionalidade específica, em vez de duplicar a mesma funcionalidade em diversas aplicações.

- Publicação: Serviços podem expor as suas descrições, que permitem que outras aplicações e serviços possam localizá-los.

Além dos estilos arquiteturais, que representam um padrão de organização estrutural dos sistemas, existem outros artefatos de engenharia de software para padronizar e garantir a qualidade do produto final. Dois deles são os Padrões de Aplicação - EAP e Padrões de Integração – EAI.

Os EAPs são soluções arquiteturais para problemas conhecidos, que foram desenvolvidas, testadas e disponibilizadas no mercado para qualquer um utilizar. Devido a estas características, os EAPs são independentes de tecnologia. O mesmo padrão pode ser aplicado em .Net, Java, PHP, Python, etc. Martin Fowler descreve 52 destes padrões no livro “Patterns of Enterprise Application”, que estão disponibilizados em: http://martinfowler.com/eaaCatalog/.

Um dos padrões mais conhecidos e utilizados, sobretudo em aplicações Web, é o Model-View_Controller – MVC:

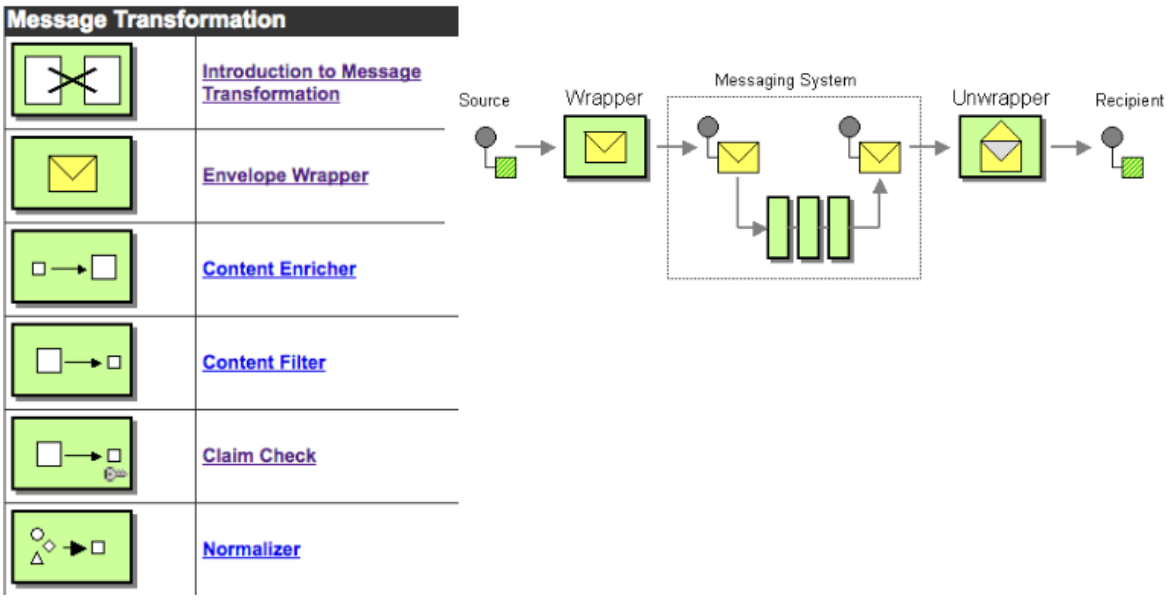


O padrão MVC é também um estilo arquitetural baseado em camadas (Layered). Nesse a camada "*View*" é responsável pela interface de usuário, enquanto a camada "*Controller*" realiza processamento de instruções da camada de View, repassando-os para o Model como, por exemplo, regras de negócio elaboradas, e transformações de dados. A camada "*Model*", por sua vez, é responsável pelo fornecimento de dados para a aplicação.

Perceba que em momento algum na descrição do MVC, informa que ele deva ser desenvolvido em PHP, ou que deva estar em disco ou memória. Como todo EAP, ele é independente de linguagem, plataforma ou infra.

Da mesma forma que os EAPs, os Enterprise Application Integration – EAI descrevem problemas comuns de integração entre sistemas, os EAIs são organizados em categorias como, *Patterns, Routing, Transformation, Endpoints, Channels, Management e Integration Styles*.

Como exemplo, EAIs da categoria *Transformation Patterns* descrevem diversas maneiras de alterar o conteúdo de uma mensagem, seja removendo, adicionando ou modificando dados.



Alguns dos EAIs do tipo Transformation Patterns e suas respectivas funções são:

- *Envelope Wrapper*: Por exemplo, encapsular os dados da aplicação dentro de uma camada de abstração, seja um formato específico ou um processo de criptografia para que ela trafegue pelo ambiente de mensagens e seja desencapsulada no destino.

- *Content Enricher*: Por exemplo, pode-se durante o meio de envio da mensagem, utilizar uma fonte de dados externa para acrescentar informações faltantes com o objetivo de entregar a mensagem ao receptor.

- *Content Filter*: Pode-se usar um filtro de conteúdo para remover itens de dados sem importância ou privados de uma mensagem, deixando apenas os itens necessários ao receptor.

- *Claim Check*: Durante o tráfego de mensagens, pode-se exigir uma autenticação para entregá-la ao receptor, ou para realizar uma transformação de dados.

- Normalizer: Realiza o processamento de mensagens que são semanticamente equivalentes, mas são provenientes de diferentes fontes e apresentam formatos diferentes.

Por fim, mais do que decorar a lista (grande) de EAPs e EAIs, é necessário que o arquiteto de softwares entenda que qualquer sistema que ele projete, haverá partes componentes do mesmo que alguém já tenha pensado em uma solução otimizada. Não há necessidade de reinventar a roda.

Conclusão

Chegamos, assim, ao fim do curso de Princípio e Práticas em Arquitetura de Software. A principal mensagem que esse curso passa, é que a atividade do arquiteto de software é complexa, estratégica e com a (cada vez mais) alta dependência dos negócios pela TI, é uma atividade imprescindível.

Há muito tempo o arquiteto de softwares deixou de ser o técnico experiente e passou a ser uma função com altos salários e alta importância. Em muitas empresas esse cargo tem nomes imponentes, como CTO ou CIO. Mas a chave de sucesso para um bom arquiteto de software é conseguir mesclar o conhecimento técnico com boas práticas de gestão e liderança, sabendo extrair o máximo conhecimento e produtividade de sua equipe.

Referências

BECK, Kent. *Extreme programming explained:* embrace change. Addison-Wesley Professional, 1999.

COOPER, Brant; VLASKOVITS, Patrick. *The entrepreneur's guide to customer development:* a" cheat sheet" to The Four Steps to the Epiphany. Cooper-Vlaskovits, 2010.

FOWLER, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

GROVE, Andrew S. *High output management*. New York: Vintage Books, 1985.

HOHPE, Gregor; WOOLF, Bobby. *Enterprise integration patterns:* Designing, building, and deploying messaging solutions. Addison-Wesley Professional, 2004.

JEFFRIES, Ron; ANDERSON, Ann; HENDRICKSON, Chet. *Extreme programming installed*. Addison-Wesley Professional, 2000.

KRUCHTEN, Philippe B. *The 4+ 1 view model of architecture*. USA: IEEE software, v. 12, n. 6, p. 42-50, 1995.

LANKHORST, Marc M.; PROPER, Henderik Alex; JONKERS, Henk. The architecture of the archimate language. In: *Enterprise, business-process and information systems modeling*. Springer Berlin Heidelberg, 2009.

LEVITT, Raymond E. *Towards project management 2.0. Engineering Project Organization Journal*, v. 1. Published online, 2011.

MANNION, Mike; KEEPENCE, Barry. *SMART requirements.* New York: ACM SIGSOFT Software Engineering Notes, v. 20, n. 2, p. 42-47, 1995.

PRESSMAN, Roger S; MAXIM, Bruno. *Software engineering:* a practitioner's approach. McGraw-Hill Education, 2014.

RIES, Eric. *The lean startup.* New York: Crown Business, 2011.

RUP, IBM. *Rational Unified Process*. Engenharia de Software, 2003.

SCHWABER, Ken; SUTHERLAND, Jeff; BEEDLE, Mike. *The definitive guide to scrum:* the rules of the game. University oh St. Andrews, 2013.

UMAR, Mahrukh; KHAN, Naeem Ahmed. Analyzing non-functional requirements (nfrs) for software development. In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, p. 675-678. IEEE, 2011.