Design de Software

Design orientado por domínio (*Domain-Driven Design*)

Baseado no material de Rubén Béjar

Habib Asseiss Neto

INSTITUTO FEDERAL
Mato Grosso do Sul

Nesta aula

- Domínio de um problema
- Modelos de domínio no desenvolvimento de software
- Obtenção de conhecimento do domínio
- Linguagem e comunicação
- Documentos e diagramas
- Vinculação entre modelo e implementação

Domínio do problema

- É a área de conhecimento a ser examinada para resolver um problema
  - Área de conhecimento à qual o software é aplicado
- Exemplos:
  - Em um sistema de controle de versão, o domínio do problema deve incluir código-fonte, texto ou arquivos binários
  - Em um programa de contabilidade, o domínio incluirá dinheiro e lançamentos contábeis
  - Em um programa de venda de ingressos para shows, o domínio inclui datas, assentos e locais de shows

Modelos de domínio

- Pode incluir uma quantidade enorme de informações
- Um **modelo desse domínio** é uma maneira de representar uma parte dessas informações de forma estruturada e simplificada
- Um modelo não é um diagrama; Um diagrama não é um modelo

Modelos de domínio

- Um modelo é uma abstração de algum conhecimento selecionado, rigorosamente estruturado
  - Que podemos expressar com um ou mais diagramas, explicações, fórmulas, códigos etc.
- Ao criar um modelo de um domínio, não buscamos o máximo de realismo, mas sim representar uma parte da realidade da maneira que melhor atenda às nossas necessidades

Pergunta

- Quantos modelos corretos de um domínio de problema podemos criar?

Modelos de domínio

- O modelo de domínio determina o design da parte principal do nosso software
  - Isto é, a parte mais estável, a que sofre menos alterações entre as versões, a parte que implementa a funcionalidade principal para o usuário

Função central do domínio

- O que é essencial para um aplicativo de software é que ele resolva problemas relacionados ao seu domínio
  - O restante dos recursos apoia essa finalidade
- Se o domínio for complexo, os desenvolvedores terão que gastar tempo analisando-o junto com especialistas e usuários
  - Isso nem sempre é uma prioridade nos projetos

Função central do domínio

- Ter um bom conhecimento do domínio é uma vantagem competitiva fundamental para trabalhar nesse campo
- Um bom modelo de domínio é a base para um aplicativo que deseja ser mantido por muito tempo
- A tecnologia (GUI, bancos de dados, conectividade de rede) está mudando muito mais rapidamente do que a maioria dos domínios de problemas

Doctors are asking Silicon Valley engineers to spend more time in the hospital before building apps

- Richard Zane, an emergency room physician, developed a program so that engineers can understand the clinician's workflow before they build their products
- RxRevu is one start-up that shadows Zane on the job.
- In the Bay Area, it's become common for doctors to invite technologists from Google and elsewhere to follow them on the job.

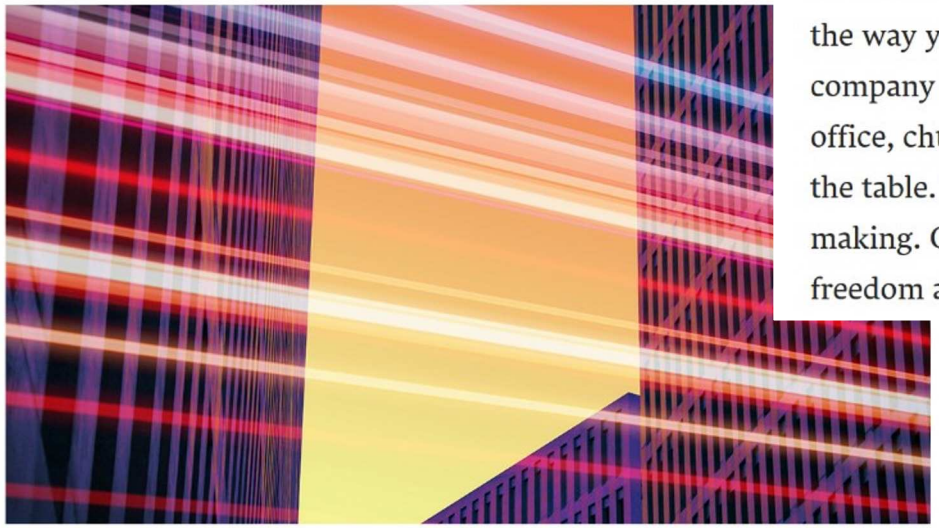
Christina Farr | @chrissyfarr

Published 9:38 AM ET Fri, 28 Dec 2018 | Updated 12:43 PM ET Fri, 28 Dec 2018

In the Digital Economy, Your Software Is Your Competitive Advantage

by Jeff Lawson

January 18, 2021



Arthur Debat/Getty Images

Summary. Many companies respond to digital competition by embracing methodologies like agile, building "innovation centers," acquiring startups, or outsourcing app development to consulting firms. But the true disruptors know that in the digital economy, whoever builds the best software wins. Companies that want to compete need to empower their developers and

That means organizations must build their own software development teams and empower developers to be creative problem-solvers. Companies can start by reskilling existing tech staff. These people are among your most valuable employees, but often are an untapped resource.

But companies also must recruit and retain top-tier software engineers. How does a non-tech company lure great developers? You must change the way you view developers. The best engineers won't work for a company that treats them like "code monkeys" — stuck in some back office, churning out code on command. Top developers want a seat at the table. Involve engineers in strategic problem solving and decision-making. Give them a voice in shaping the future of the company, and the freedom and autonomy to be creative.

Como um domínio é modelado?

1. Você se reúne com os especialistas no domínio

2. Você faz algumas perguntas: o que eles precisam do software, os princípios básicos do trabalho deles...

   - Isso pode ser feito em paralelo à captura de requisitos ou depois, com base nesses requisitos

3. Você faz um diagrama informal que lhe permite refletir o que os especialistas lhe dizem e discutir com eles.

   - Para começar, objetos-objeto, algum diagrama de interação ou autômatos geralmente funcionam

   - Em seguida, você pode passar para diagramas de classe, diagramas de sequência...

4. Você discute o diagrama com os especialistas e faz mais perguntas

Como um domínio é modelado?

5. Você repete os passos 3 a 4 e refina os diagramas, que gradualmente se tornam um modelo básico do domínio

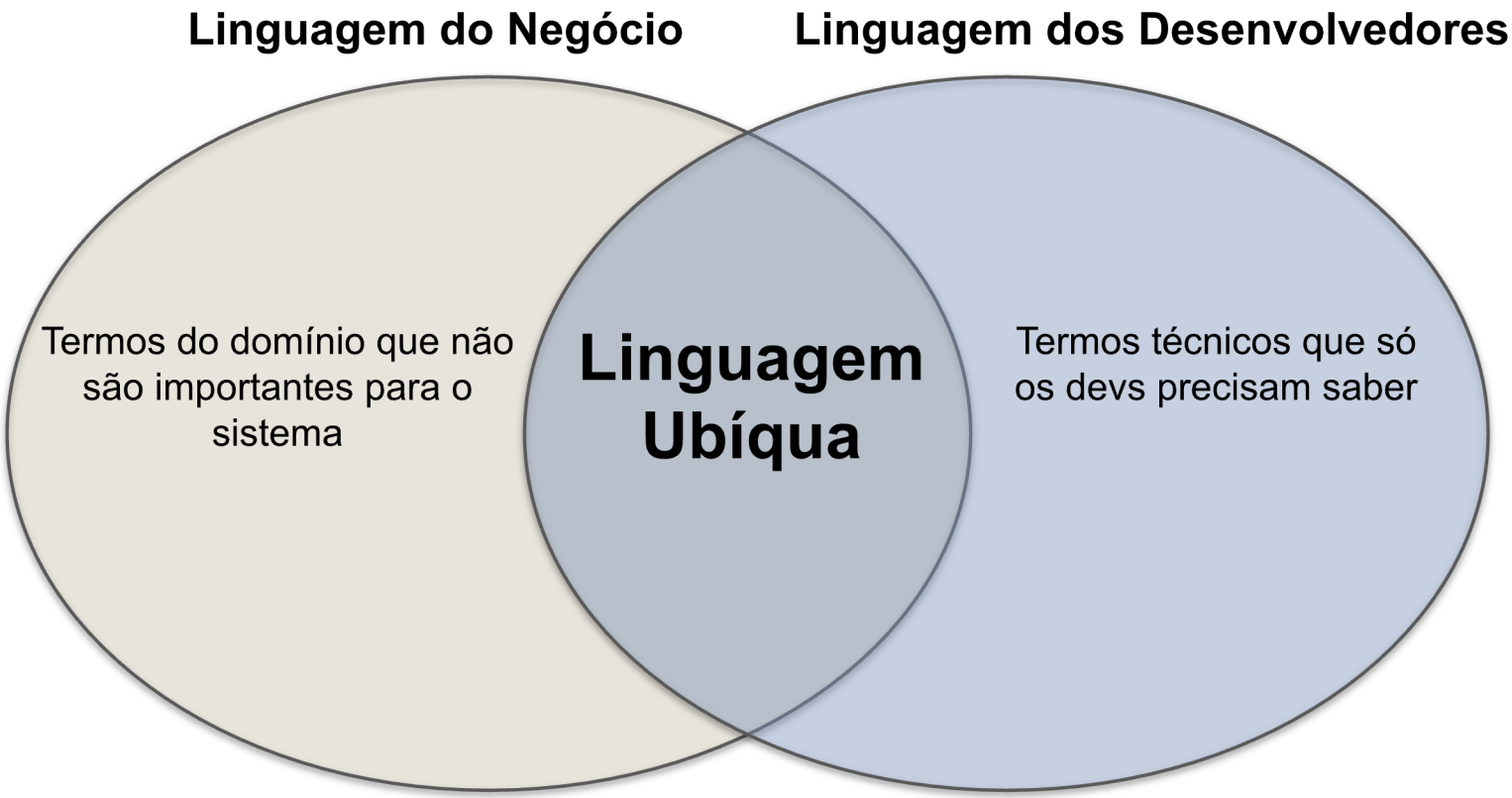
6. Você implementa um protótipo a partir de seu modelo básico

   - Ele permite que você verifique se o entendeu, se estão faltando coisas importantes a serem capturadas, dá a você algo para mostrar aos especialistas para confirmar se é a coisa certa a fazer e para obter feedback...

7. Repita as etapas de 3 a 6, sempre evoluindo tanto o modelo quanto os protótipos e, à medida que as coisas ficarem mais claras, você começará o desenvolvimento real

Extração de Conhecimento de domínio

- Vincular o modelo e a implementação o mais rápido possível
- Um protótipo muito simples pode ser usado
- Desenvolve uma linguagem compartilhada (entre desenvolvedores e especialistas no domínio) com base no modelo
- Sempre chame os principais conceitos pelo mesmo nome, não use sinônimos
- Evite um modelo que seja apenas um "esquema de dados"
- O comportamento e as regras são tão importantes ou até mais importantes


Linguagem do Negócio
Linguagem dos Desenvolvedores
Termos do domínio que não são importantes para o sistema
Linguagem Ubíqua
Termos técnicos que só os devs precisam saber

Extração de Conhecimento de domínio

- Remover itens do modelo à medida que o refina é tão importante quanto adicionar novos itens
- Altere, refine, discuta, teste... iterativamente
- O trabalho é realizado em conjunto pela equipe de desenvolvedores e pela equipe de especialistas
- O conhecimento também é obtido de usuários de sistemas anteriores e da experiência anterior da equipe de desenvolvimento com esse domínio
  - Parte disso pode vir em documentos (por exemplo, análise de requisitos, manuais de aplicativos existentes...)

Extração de Conhecimento de domínio

- Com as metodologias tradicionais (em cascata), os especialistas no domínio conversam com os analistas, que digerem as informações e as passam para os programadores
  - Isso falha, em parte, devido à falta de feedback
- Os analistas criam modelos baseados apenas nos especialistas no domínio, sem a possibilidade de testar protótipos

Aprendizagem contínua

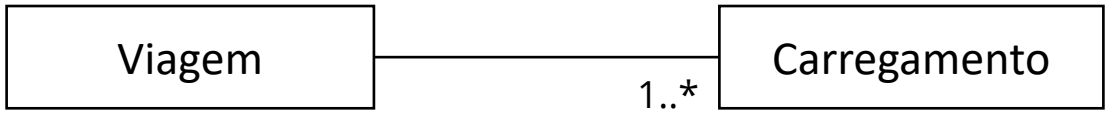
- Em um projeto, o conhecimento é fragmentado entre pessoas e documentos, e misturado com outras informações
  - O conhecimento pode ir se perdendo
- As pessoas saem, as equipes são reorganizadas, partes do sistema são terceirizadas, o conhecimento dessas partes é levado embora
- Uma equipe produtiva e consciente dos problemas praticará o **aprendizado contínuo**
  - Que inclui o aprendizado aprofundado por toda a equipe do domínio que está sendo trabalhado

Design rico em conhecimento

- Um bom modelo de domínio não é apenas um esquema de dados
- Entidades + relacionamentos + atributos **não são suficientes**
- Ele deve incluir comportamento (na forma de atividades, processos, interações...) e regras
- Para os especialistas, eles geralmente são mais complicados de explicar do que os conceitos estáticos
- E também é mais difícil para os designers se expressarem e discutirem

Exemplo:
Extração de um conceito oculto

- Um modelo de domínio muito simples para um aplicativo de reserva de espaço de carga em um navio:
  - Associe cada carga a uma viagem, registre e acompanhe esse relacionamento



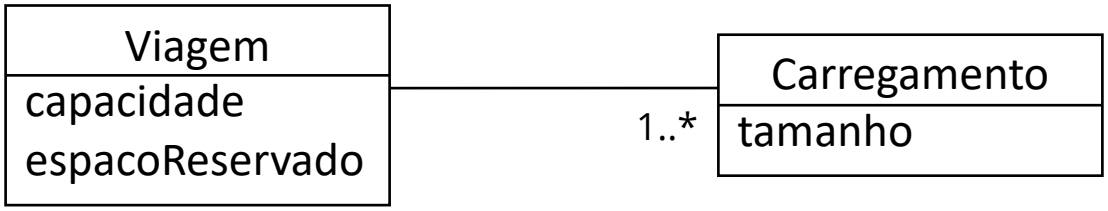

```
def makeReservation(load: Load, trip: Trip) -> int:
  confirmation = requestIdConfirmationId()
  trip.addLoad(load, confirmation)

  return confirmation
```

Exemplo:
Extração de um conceito oculto

- É comum que um navio possa ter overbooking (aceitar mais carga do que o máximo que pode ser acomodado)

- Suponha que isso seja expresso como uma porcentagem e que, no documento de requisitos, tenhamos:
  "Permitir 10% de overbooking"

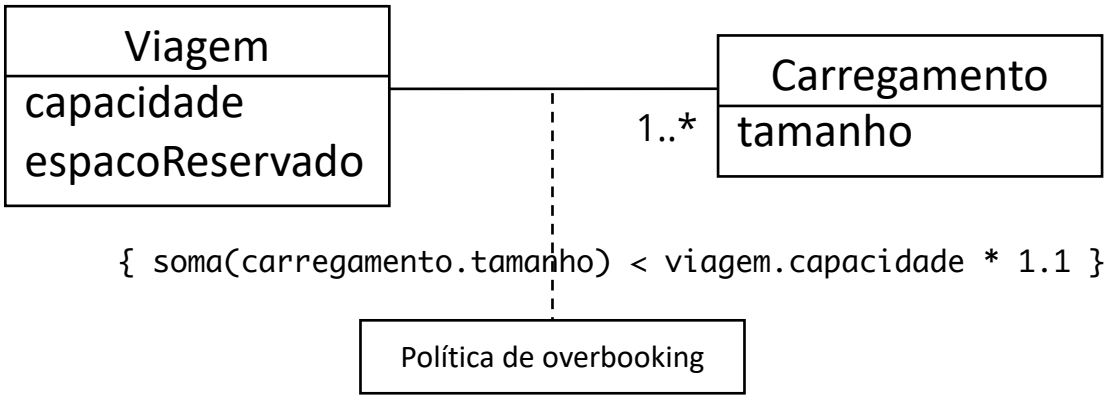



```
def makeReservation(load: Load, trip: Trip) -> int:
  maxReservation = trip.capacity * 1.1 # 10% OVERBOOKING
  if trip.spaceReserved + load.size > maxReservation:
    return -1
  confirmation = requestIdConfirmationId()
  trip.addLoad(load, confirmation)

  return confirmation
```

Exemplo:
Extração de um conceito oculto

- Agora temos uma regra comercial oculta no código de um método de aplicativo
- Deveríamos torná-lo mais visível (principalmente se fosse mais complexo)
  - Para poder compartilhá-lo com especialistas no domínio sem precisar mostrar o código (que eles não entenderão).
- Mas também para deixar mais claro para um desenvolvedor que esse trecho de código está lá por causa de uma política específica (permitir 10% de overbooking)



Exemplo:
Extração de um conceito oculto

- A política de overbooking agora é explícita e a implementação é separada (e, portanto, pode ser facilmente usada em diferentes partes do aplicativo)
- Não precisaremos chegar a esse nível de detalhe para cada elemento do domínio, mas sim para os aspectos mais importantes ou mais sujeitos a mudanças

```
class OverbookingPolicy:
  def allows(self, trip: Trip, load: Load) -> bool:
    return load.size + trip.spaceReserved <= trip.capacity * 1.1


def makeReservation(load: Load, trip: Trip) -> int:
  if not overbookingPolicy.allows(trip, load):
    return -1
  confirmation = requestIdConfirmationId()
  trip.addLoad(load, confirmation)

  return confirmation
```

Linguagem compartilhada

- Cada domínio de problema tem seu próprio jargão, e os desenvolvedores também o têm
  - Um projeto precisa de uma linguagem compartilhada que seja melhor do que o menor denominador comum entre os jargões
- O modelo de domínio pode ser o núcleo de uma linguagem comum para um projeto de software
  - Por exemplo, os principais conceitos dessa linguagem corresponderão às classes do modelo de domínio e serão nomeados da mesma forma

Linguagem compartilhada

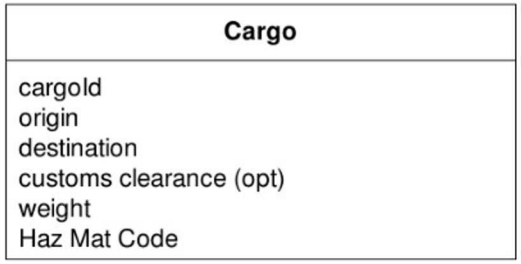
- As metodologias tradicionais enfatizam documentos de texto e diagramas detalhados para a especificação e o controle do projeto
- As metodologias ágeis enfatizam a conversação, os diagramas informais e o código-fonte para a comunicação
- Todos esses meios podem ser valiosos em diferentes ocasiões
- Mas para que qualquer um deles funcione, os conceitos que eles capturam e a linguagem em que são expressos devem ser compartilhados por todos no projeto
    - Em conversas, documentos, diagramas e códigos

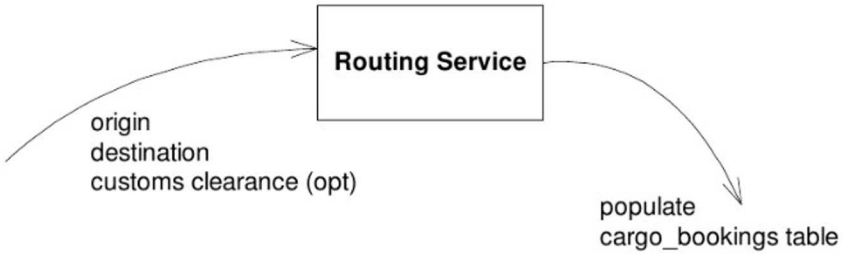
Linguagem compartilhada

- Um modelo pode ser visto como uma linguagem
  - Os nomes das classes e das principais operações constituem o vocabulário básico
  - A linguagem fornece os meios para discutir as regras que foram explicitadas no modelo
- Essa linguagem, baseada no modelo, deve ser a principal ferramenta de comunicação da equipe de desenvolvimento para descrever os artefatos, as tarefas e a funcionalidade do sistema
  - E para que os especialistas se comuniquem com a equipe de desenvolvimento, e até mesmo entre si, ao discutir requisitos ou planos para o projeto

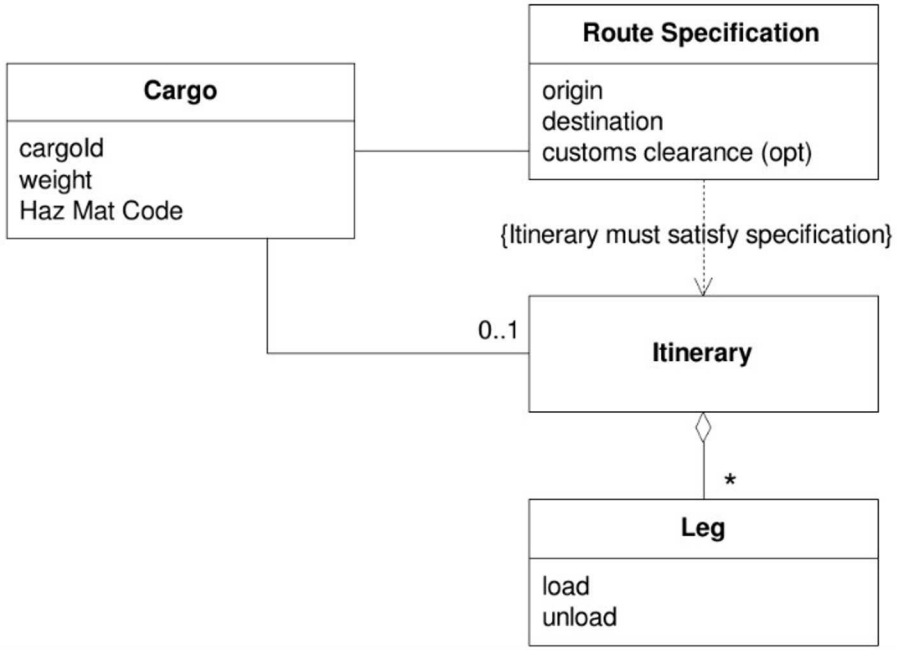
Quem fala a melhor linguagem compartilhada?

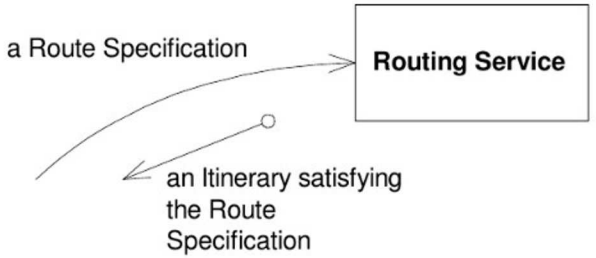
- Especialista: Quando mudamos o ponto de controle alfandegário, temos que refazer toda a rota.
- Desenvolvedor 1: OK. Excluiremos todas as linhas da **tabela de remessas** que têm os **identificadores de remessa** atuais, calcularemos a nova **rota** com o **serviço de cálculo de rota** e preencheremos novamente a **tabela de remessas**.
- Desenvolvedor 2: OK. Quando houver alguma alteração na **especificação da rota** para as **cargas** atuais, excluiremos a **rota** anterior e calcularemos uma nova **rota** com o **serviço de cálculo de rota** com base na **especificação atualizada**.





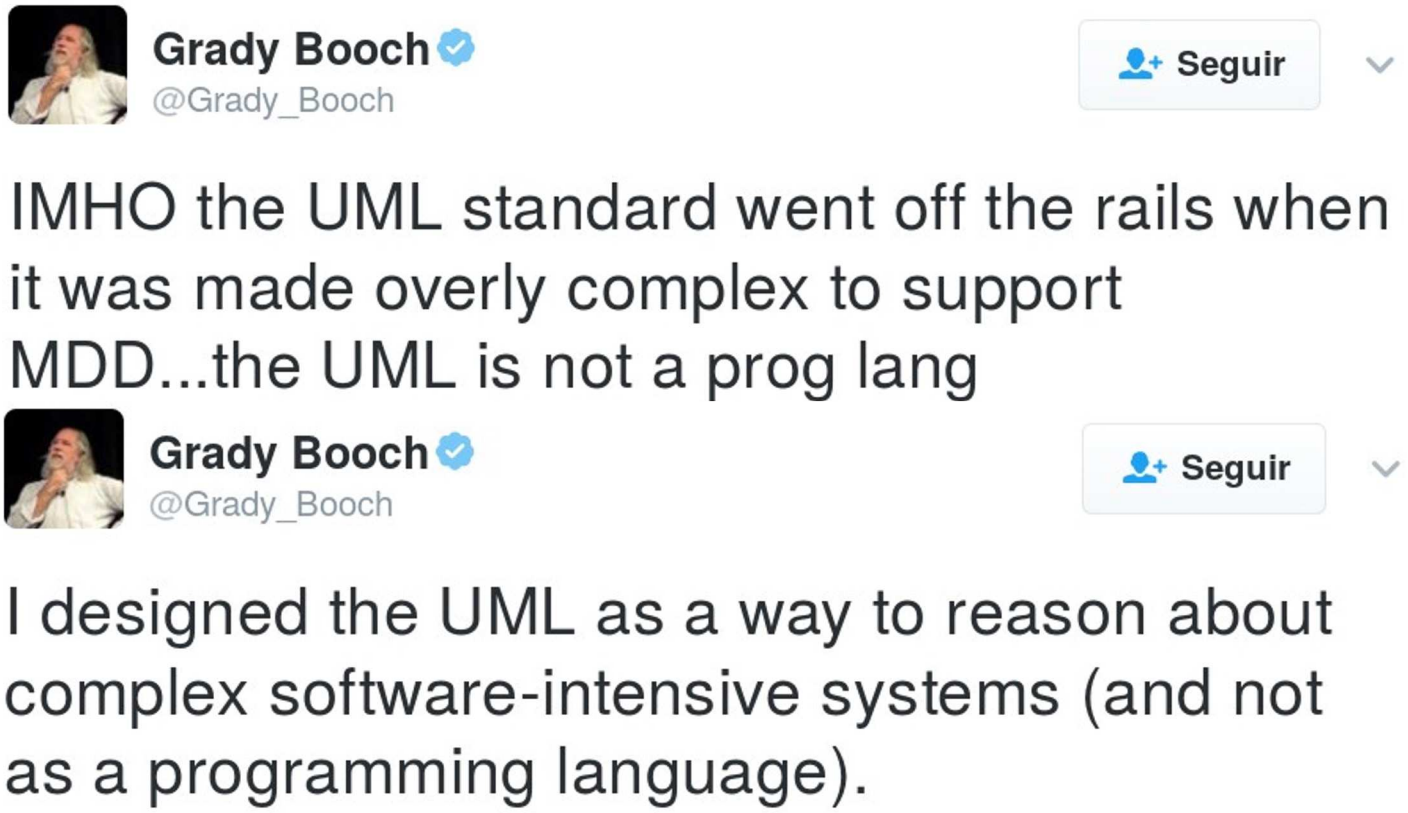
Database table: cargo_bookings





Diagramas

- Os diagramas são uma forma natural de representar conceitos e suas relações e interações
- Alguns dos diagramas UML (não necessariamente muito formais ou sintaticamente rigorosos) são uma boa opção
- Um diagrama é uma boa maneira de ancorar uma discussão
- Os problemas surgem quando se tenta representar todo o modelo em UML
  - Após um certo nível de detalhe, a UML (como todas as notações gráficas) rapidamente deixa de ser uma ajuda e passa a ser um obstáculo



- Na minha humilde opinião, o padrão UML saiu dos trilhos quando o tornaram complexo demais para dar suporte ao design orientado por modelos... a UML não é uma linguagem de programação.

- Projetei a UML como uma forma de raciocinar sobre sistemas complexos de software intensivo (e não como uma linguagem de programação).

(Grady Booch é um dos três pais da UML)

Ivar Jacobson @ivarjacobson · 2 oct.

UML needs to be scaled down to its essentials. Now after 20 years of experience in using it, scaling down should be easily done. Scaling down in a way that it can be scaled up in more complex situations is a bit more demanding but doable.

Ivar Jacobson @ivarjacobson · 19 sept.

Exactly, in 2010 I wrote a paper with Steve Cook titled "The Road ahead for UML" ubm.io/2Nn3NFt, suggesting a UML Essentials, a small but well selected subset of UML. Less than 20% of UML would cover more than 80% of the need of modelling. Too bad, not yet implemented.

- A UML precisa ser reduzida ao essencial. Agora, depois de 20 anos de experiência com seu uso, reduzi-la deve ser fácil de fazer. Reduzi-la para que possa ser ampliada em situações mais complexas é um pouco mais complicado, mas factível.

- Exatamente. Em 2010, escrevi um artigo com Steve Cook intitulado "The way forward for UML" (O caminho a seguir para a UML), sugerindo uma UML essencial, um subconjunto pequeno, mas bem escolhido, da UML. Menos de 20% da UML cobriria mais de 80% das necessidades de modelagem. É uma pena que ela ainda não tenha sido implementada.

(Ivar Jacobson é um dos três pais da UML)

Diagramas

- A UML também não é a solução perfeita para expressar o comportamento de um sistema
  - Os diagramas de interação e de sequência são úteis para representar alguns comportamentos particularmente importantes ou complicados, mas tentar representar a maioria das interações dessa forma só é viável em problemas de brinquedo
- Nem com restrições ou afirmações
  - Na UML, você pode representá-los como texto ou como fórmulas em OCL, mas isso não é muito visual (estamos falando de diagramas)
- A UML também não ajuda a expressar as responsabilidades dos objetos
  - A linguagem natural, no entanto, funciona bem para isso

Documentos de design

- Depois que um documento é criado, ele geralmente é desconectado do projeto
  - Por exemplo, porque a evolução do código acaba o deixando para trás
- O código não mente e seu comportamento não é ambíguo
  - Mas isso pode sobrecarregar o leitor com detalhes e ser difícil de entender
  - Além disso, os programadores não são os únicos que precisam entender o modelo
- A comunicação verbal ajuda a entender o código, mas qualquer grupo maior precisará da estabilidade e da "capacidade de compartilhamento" de alguns documentos escritos
  - Quanto maior e mais distribuída for uma equipe, mais importante será colocar mais coisas por escrito

Documentos de design

- Os documentos de design precisam fornecer significado, destacar grandes estruturas e concentrar-se nos principais elementos do programa
- Eles devem explicar os motivos que levaram ao sistema, as alternativas que foram consideradas e as restrições que foram aplicadas
- Eles devem estar vivos e escritos no idioma compartilhado
  - Um documento que não é atualizado é porque não é usado, e geralmente é porque a equipe o considera inútil
  - Documentos muito longos e/ou complexos tendem a ser difíceis de atualizar e, portanto, acabam sendo descartados

Resumindo

- Usar a UML quase como uma linguagem de programação, incluindo todos os detalhes necessários sobre o sistema, não é uma boa opção
    - Se você ainda precisasse usá-lo, teria que procurar uma maneira alternativa de representar o modelo do sistema sem tantos detalhes
- **Os diagramas e documentos de design são um meio de comunicação e explicação e facilitam o raciocínio**
    - Se forem muito detalhados, eles perdem essa utilidade: sobrecarregam o leitor com detalhes e se tornam difíceis de entender
- **Os detalhes do projeto estão no código**
    - Uma implementação bem escrita deve revelar o modelo subjacente
- **Modelo não é diagrama**
    - O diagrama comunica e explica o modelo, e o código contém os detalhes do modelo

Modelo e implementação

- A base do DDD é que **o mesmo modelo deve ser a base para implementação, design, análise e comunicação na equipe** (incluindo também especialistas em domínio)
  - E também uma maneira de explicar esse domínio
- O design orientado por domínio busca um modelo que não seja apenas uma forma de analisar o problema, mas que seja a verdadeira base para o design
  - Isso tem implicações para o código, mas também requer uma abordagem diferente para a modelagem

Modelo e implementação

- As metodologias mais tradicionais criam um "modelo de análise" para entender (analisar) o domínio do problema
  - Esse modelo não leva em consideração aspectos de design ou implementação de software
- Em seguida, são criados modelos de projeto que têm alguma correspondência com o modelo de análise, mas que levam em conta alguns aspectos de implementação
  - Em geral, os dois modelos acabam sendo bastante diferentes, pois abordam problemas diferentes
  - O conhecimento sobre o domínio do problema é perdido na mudança da análise para a modelagem do projeto

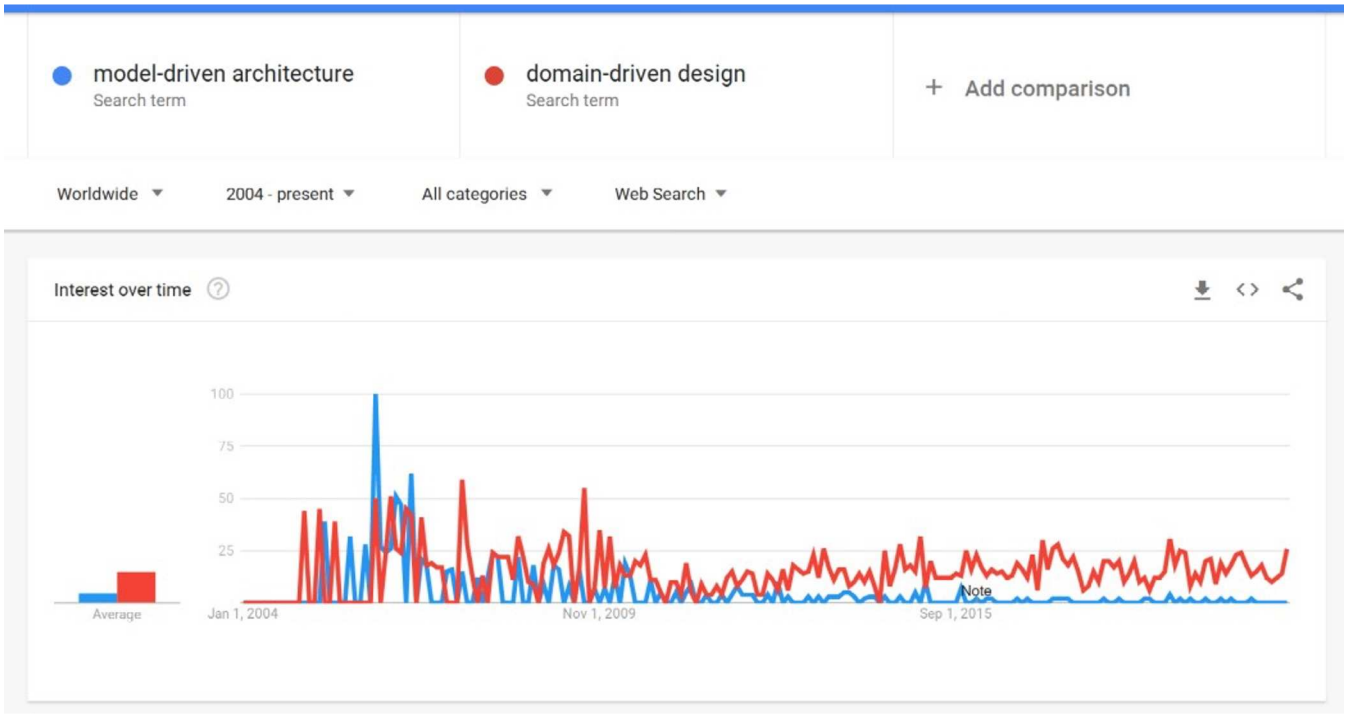
Modelo e implementação

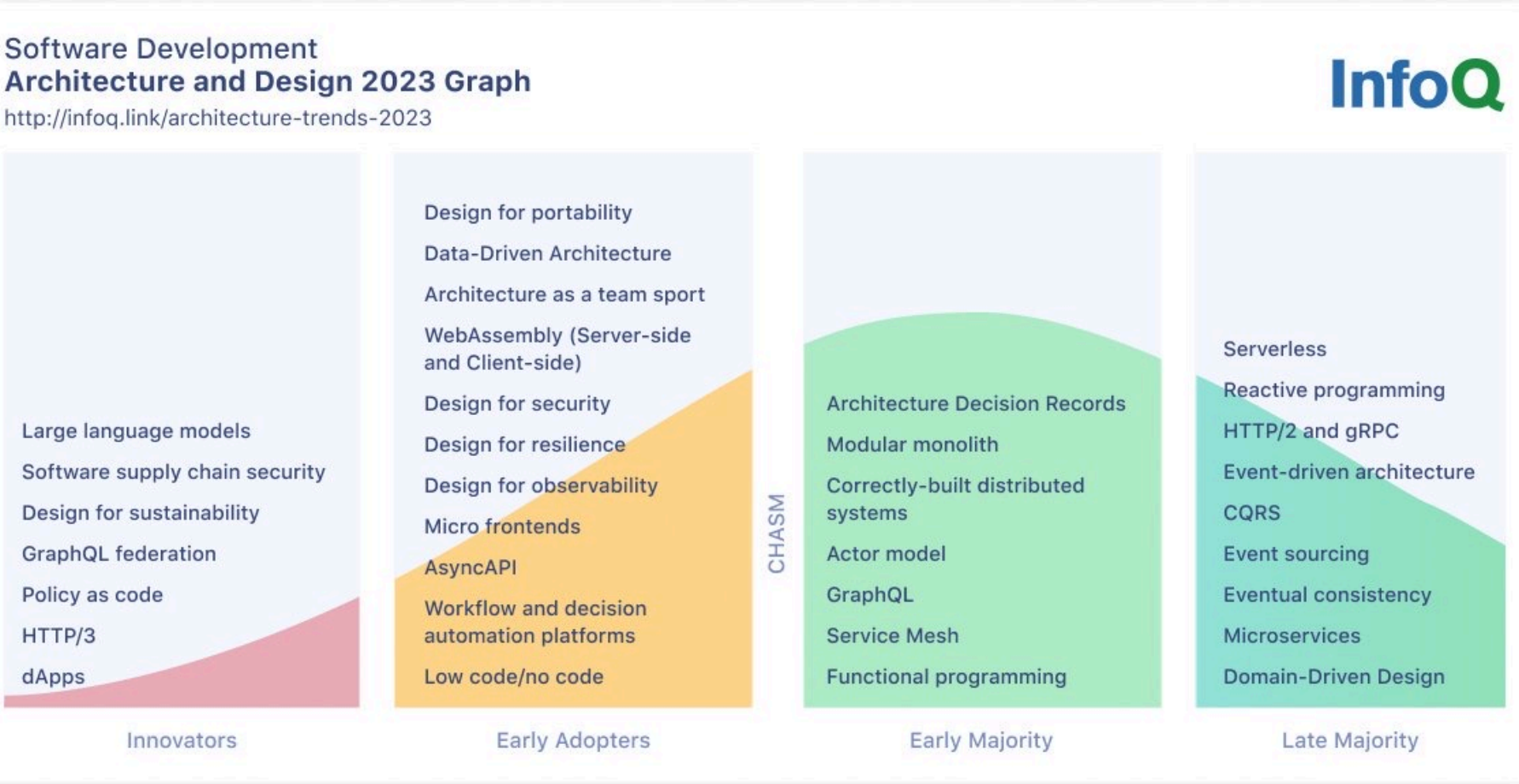
- Manter o relacionamento entre os dois modelos é caro, portanto, o relacionamento é perdido
- Tudo o que surgir no projeto/implementação do interesse do domínio nunca será incluído no modelo de análise
  - O que, nessas metodologias, costuma ser considerado quase imutável, apesar do fato de ter sido criado quando o problema era menos conhecido
- Como resultado, o modelo de análise é abandonado logo depois que você começa a escrever o código, e você tem que reinventá-lo (mal) à medida que avança, e com menos acesso a especialistas no domínio

Arquitetura Orientada a Modelos

- Para tentar corrigir alguns dos problemas dessa visão tradicional (mas sem alterar sua essência), a OMG propôs em 2001 a Arquitetura Orientada a Modelos (MDA)

- Quase 20 anos depois, essa prática não é de uso comum e o interesse por ela reflete isso





A tendência é que o DDD se torne uma prática comum no design e na arquitetura de software

Modelo e implementação

- Se o design não estiver claramente relacionado ao modelo de domínio, esse modelo terá pouco valor
- Por outro lado, mapeamentos complexos entre modelos e elementos de design são difíceis de expressar e, acima de tudo, difíceis de manter à medida que o design evolui
- Uma divisão entre análise e projeto que impeça que essas duas atividades se alimentem mutuamente é uma boa maneira de obter um software ruim

Modelo e implementação

- Visão tradicional
  - A análise deve capturar os conceitos do domínio da forma mais expressiva e abrangente possível
  - O projeto deve especificar os componentes a serem desenvolvidos/integrados com as tecnologias disponíveis para implementar um sistema eficiente e correto
- **O DDD exclui a necessidade de modelos separados de análise e design** e propõe um único modelo que atenda a ambas as finalidades
  - A criação desse modelo será mais complexa, mas nos permitirá criar um software mais coerente, mais bem documentado e de melhor manutenção
  - Expressaremos esse modelo único com diagramas UML e com código

Paradigmas de modelagem e ferramentas de apoio

- Para que isso funcione, é necessária uma correspondência fiel entre o modelo e a implementação
  - A linguagem de programação precisa oferecer suporte a conceitos diretamente análogos aos que usamos em outras expressões (por exemplo, diagramas) do modelo
- A orientação a objetos atende a esse requisito
  - O avanço crucial da orientação a objetos é que ela permite que os mesmos conceitos sejam usados para analisar um problema e implementar uma solução para esse problema
- A orientação a objetos é o paradigma de análise/design/implementação mais bem estabelecido e é adequada a domínios de problemas muito diversos

Modelagem e Implementação

- Na visão tradicional, o desenvolvimento de software era visto como um tipo de fabricação
  - Os engenheiros qualificados deveriam projetar e os trabalhadores menos qualificados deveriam produzir
- Essa metáfora é catastroficamente incorreta:

  O desenvolvimento de software como um todo é um projeto, não uma manufatura, e requer altas habilidades

Modelagem e Implementação

- Os responsáveis pela modelagem do domínio do problema não podem ser totalmente separados do código
  - Isso os impediria de conhecer em primeira mão os problemas que surgem ali, e esse feedback é essencial para criar um bom modelo do domínio
- Aqueles que escrevem o código devem se sentir corresponsáveis pelo modelo e entendê-lo bem, ou o software acabará sendo criado sem esse modelo (sem as vantagens do DDD)
- Todos os desenvolvedores precisam estar envolvidos (embora em níveis diferentes) nas discussões sobre o modelo e no contato com especialistas no domínio