UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I
CENTRO DE CIÊNCIA E TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Arioston Jaerger de A. C. Junior

TEORIA DAS CATEGORIAS COMO FERRAMENTA PARA MODELAGEM DE DOMÍNIO

Campina Grande
2019

Arioston Jaerger de A. C. Junior

TEORIA DAS CATEGORIAS COMO FERRAMENTA PARA MODELAGEM DE DOMÍNIO

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Graduado em Ciência da Computação.

Área de concentração: Teoria das categorias.

Orientador: Prof. Me. Luciana de Queiroz Leal Gomes

Campina Grande

2019

É expressamente proibido a comercialização deste documento, tanto na forma impressa como eletrônica. Sua reprodução total ou parcial é permitida exclusivamente para fins acadêmicos e científicos, desde que na reprodução figure a identificação do autor, título, instituição e ano do trabalho.

C376t Cavalcante Junior, Arioston Jaerger de Araujo.
Teoria das categorias como ferramenta para modelagem de domínio [manuscrito] / Arioston Jaerger de Araujo Cavalcante Junior. - 2019.
54 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia , 2020.

"Orientação : Profa. Ma. Luciana de Queiroz Leal Gomes , Coordenação do Curso de Computação - CCT."

1. Teoria das Categorias. 2. Olog. 3. Teoria dos Processos. 4. Diagrama de Cordas. I. Título

21. ed. CDD 005.1

Elaborada por Giulianne M. Pereira - CRB - 15/714 BC/UEPB

ARIOSTON JAERGER DE ARAUJO CAVALCANTE JUNIOR

TEORIA DAS CATEGORIAS COMO FERRAMENTA PARA MODELAGEM DE DOMÍNIO

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

Aprovada em 6 de Dezembro de 2019.

Luciana de Queiroz Leal Gomes

Profa. MSc. Luciana de Queiroz Leal Gomes (DC - UEPB)
Orientador(a)

Antonio Carlos de Albuquerque

Prof. Msc. Antônio Carlos Albuquerque (DC - UEPB)
Examinador(a)

Kézia de Vasconcelos Oliveira Dantas

Profa. Dra. Kézia de Vasconcelos Oliveira Dantas (DC - UEPB)
Examinador(a)

Agradecimentos

Agradeço a minha família, pela compreensão e paciência.

À professora Me. Luciana de Queiroz Leal Gomes pelas leituras sugeridas ao longo dessa orientação e pela dedicação.

Resumo

Um dos grande gargalos no desenvolvimento de software é a comunicação, a falta de símbolos em comum entre a equipe e *stakeholders*, gera problemas na especificação, a fim de minimizar esse problema o Desenvolvimento dirigido a domínio propõem algumas abordagens como a criação de uma linguagem ubiquá. O objetivo geral do presente trabalho é explorar de que forma é possível utilizar Teoria das Categorias no desenvolvimento de softwares utilizando Desenvolvimento Dirigido ao Domínio, explorando a relação entre o resultado da sessão de Event Storming e Teoria das Categorias, entendendo como dá-se a relação entre Eventos e a representação dos mesmos com tipo.

O trabalho faz uso de Olog e Teoria dos Processos para representar o *workflow* do domínio do problema. A representação gráfica é feita com o uso de Diagrama de Cordas. O trabalho descreve uma das possíveis formas de interligar os tópicos fazendo uso de Teoria das Categorias.

Palavras-chave: Teoria das Categorias. Olog. Teoria dos Processos. Diagrama de Cordas.

Abstract

One of the biggest problems in software development is communication, the absence of symbol in common between team and stakeholders generates specification problems, to minimizer this problem Domain-Driven Design proposes some approaches as the creation of Ubiqua language. The general objective of this work is to explore a possible way to make use of Category Theory on Software development using Domain Driven Design, exploring the relation between the final results of Event Storming session and Category theory, understanding the relation between Events and their representation in types.

The work makes use of Olog and Process Theory to represent the workflow of the problem domain. The graphical representation was done with String Diagrams. Work describes one of the possible ways to connect the topics using Category Theory.

Keywords: Category Theory. Olog. Process Theory. String Diagram.

Lista de ilustrações

Figura 1 – Diagrama isomorfismo. 15
Figura 2 – Epimorfismo 15
Figura 3 – Monomorfismo 16
Figura 4 – Diagrama morfismo. 17
Figura 5 – Diagrama composição. 17
Figura 6 – Diagrama de associatividade. 17
Figura 7 – Diagrama de composição identidade. 18
Figura 8 – Diagrama de Dualidade. 20
Figura 9 – Diagrama Produto. 21
Figura 10 – Diagrama Produto comutativo. 21
Figura 11 – Diagrama Coproduto comutativo. 22
Figura 12 – Diagrama Cone. 23
Figura 13 – Diagrama Cocone. 23
Figura 14 – Diagrama Functor. 25
Figura 15 – DDD escala de abstração 30
Figura 16 – O que não pertence? 34
Figura 17 – Diagrama Cospan. 35
Figura 18 – Diagrama processo. 36
Figura 19 – Exemplos de processos usando diagramas de cordas 37
Figura 20 – Pullback 38
Figura 21 – Fatores de 30. 38
Figura 22 – Power set 39
Figura 23 – Privilégios 39
Figura 24 – Type class 42
Figura 25 – Eventos 44
Figura 26 – Domínio e Contexto 44
Figura 27 – Processos 44
Figura 28 – Pedido composição 45
Figura 29 – Diagrama de Corda. 45
Figura 30 – Olog formulário 46

Lista de tabelas

Tabela 1 – Exemplos de limites e colimites 22
Tabela 2 – Tabela Correspondência 28

Lista de abreviaturas e siglas

CCC	*Cartesian Closed Category*
DDD	*Domain Driven Design*
Id	Identidade
Epi	Epimorfismo
Monic	Monomorfismo
TC	Teoria das Categorias
Ts	Typescript
HKT	*Higher Kinder Types*

SUMÁRIO

1 INTRODUÇÃO . . . 11
1.1 Contexto . . . 11
1.2 Objetivo Geral . . . 13
1.3 Objetivos Específicos . . . 13
1.4 Estrutura do trabalho . . . 13

2 TEORIA DAS CATEGORIAS . . . 14
2.1 Função . . . 14
2.2 Morfismo . . . 14
2.2.1 Isomorfismo . . . 14
2.2.2 Epimorfismo . . . 15
2.2.3 Monomorfismo . . . 15
2.3 Categorias . . . 16
2.4 Propriedade Universal, Objeto Inicial e Terminal . . . 19
2.5 Dualidade . . . 20
2.5.1 Produto e Coproduto . . . 21
2.5.2 Limite e Colimites . . . 22
2.6 Allegorias . . . 23
2.6.1 Relações . . . 23
2.6.2 Rel . . . 24
2.6.3 Categoria Allegory . . . 24
2.7 Functor . . . 25
2.8 Cálculo e Álgebra de tipos . . . 26

3 CONCEITOS APLICADOS À MODELAGEM DE SOFTWARE . . . 30
3.1 *Domain Driven Design* . . . 30
3.2 Event Storming . . . 32
3.3 Olog . . . 33
3.3.1 Tipos . . . 33
3.3.2 Aspectos . . . 33
3.3.3 Fatos . . . 33
3.4 Analogia, metáforas e Teoria das Categorias . . . 34
3.5 Comentários . . . 35

4 MODELAGEM DE SOFTWARE UTILIZANDO TEORIA DAS CATEGORIAS, EVENT STORMING E DDD . . . 41

4.1 Proposta: Utilizando TC e DDD para Modelagem de Software . . . 41
4.2 FP-TS . . . 41
4.3 Definição do Problema . . . 41
4.4 Representação do Domínio . . . 43

5 CONSIDERAÇÕES FINAIS . . . 48

REFERÊNCIAS . . . 49

1 Introdução

1.1 Contexto

Segundo Sommerville (2011), quando falamos de Engenharia de Software, não se trata tão somente do programa, mas sim de todos os artefatos necessários para que o programa opere, portanto, existe um conjunto de atividades relacionadas, ou seja processo de software, que levam a produção do mesmo. Existem diversos processos de software diferentes, entretanto todos devem compartilhar de um *core* de quatro atividades: Especificação; Projeto e Implementação; Validação de software; Evolução. Entre as quatro atividades, a primeira é a mais cara a este trabalho, nela a funcionalidade e as restrições do funcionamento do software são definidas.

Para que o processo ocorra é necessário, que exista comunicação humana. Por comunicação tenhamos o entendimento de troca de mensagens, onde mensagens são dados estruturados e convertidos do remetente para o destinatário. Toda estrutura possui potencial de se tornar informação, quando a informação é acurada ou verdadeira, torna-se conhecimento. Para que o processo de especificação ocorra, é necessário o conhecimento do objeto. O primeiro passo é, portanto, a descrição do objeto em si. Berrisford (2014) aponta que não há significado no mundo sem descrição, verdade é um valor aplicado a nossa descrição do mundo, não há portanto uma verdade absoluta, pois uma descrição não é a realidade que ela descreve, assim há três maneiras de testar a verdade de uma teoria, descrição ou modelo: empírico, suportado por evidencias extraídas de casos de teste; lógico, pode ser deduzido de outros conceitos; social, crença amplamente difundida em nossa rede social. Berrisford (2018) explica que não entendemos a realidade diretamente, não obstante entendemos por meio das percepções e descrições, uma descrição é um modelo ou teoria da realidade. Qualquer descrição é verdadeira desde que permita ser testada ou mensurada dada uma determinada propriedade.

Berrisford (2017b) ao falar sobre as ideias de Foerster, detalha que cada individuo constrói seu próprio modelo da realidade, essa ideia é corroborada pelo aforismo de Berrisford (2017b apud FOERSTER, 2003) "O ambiente, como percebemos, é nossa própria invenção", Foerster também fala que "Não vemos aquilo que não vemos" (ELVERFELDT, 2012 apud FOERSTER, 1984). Parafraseando o aforismo, podemos entender como não percebemos aquilo que deixamos de perceber, esse fenômeno também é conhecido pelo nome de ponto cego, o *gap* causado pode falsear a descrição. Para exemplificar usamos Wallace (2009) a seguir:

> Estes dois jovens peixes estão nadando por aí, e por acaso encontram

> um peixe mais velho nadando na direção contrária, que acena para eles e diz "Bom dia, meninos, como está a água?" E os dois jovens peixes continuam nadando por um tempo, até que eventualmente um deles olha para o outro e fala: "O que diabos é água? (WALLACE, 2009)

A Comunicação requer a codificação e a decodificação da mensagem, portanto o conjunto de símbolos que forma a mensagem, deve ser comum entre os envolvidos, segundo Berrisford (2017a) nenhuma palavra , descrição ou mensagem possui um entendimento universal, assim possuem um contexto limitador. Humanos possuem a habilidade de criar palavras e atribuir sentido a elas. Fowler (2006) apresenta esta reflexão sobre contexto limitador:

> Dar nome a um padrão aumenta imediatamente o nosso vocabulário de projeto. Isso nos permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões permite-nos conversar sobre eles com nossos colegas, em nossa documentação e até com nós mesmos. O nome torna mais fácil pensar sobre projetos e a comunicá-los, bem como os custos e benefícios envolvidos, a outras pessoas. (FOWLER, 2006, tradução nossa)

Ao observar algo, é possível isolá-lo ou isolar alguma de suas características, essa ação recebe o nome de abstração. Para Dijkstra (1972) a função da abstração é criar um novo nível semântico no qual alguém pode ser absolutamente preciso.

Todas essas ideias e conceitos, existem no nosso sistema conceitual. Segundo Lakoff e Johnson (1980 apud NORVIG, 19–) esse sistema é fundamentalmente metafórico em natureza, de tal modo que tipicamente domínios abstratos são entendidos com base em domínios mais concretos. Metáforas não somente apontam semelhanças que são objetivamente verdadeiras elas também criam. Todas essas construções são usadas de maneira consciente ou não na etapa de especificação de um software, a comparação é uma ferramenta fundamental nesse processo.

Ramisch e Hudita (2018) falam que a pedra filosofal dos matemáticos é a utopia de uma teoria genérica que pode explicar todas as outras teorias, infelizmente foi provado ser impossível. Entretanto, em 1945 Sauders Mac Lane e Samuel Eilenberg estabeleceram uma teoria que se aproxima da pedra filosofal, geral o suficiente para explicar outros campos da matemática, a essa teoria deram o nome de Teoria das Categorias. Teoria das Categorias é usada em diversos campos do conhecimento desde cognição humana (PHILLIPS; WILSON, 2010), quimica (BAEZ; POLLARD, 2017), Finanças (TANEGA, 2014), entre outros.

Dentro da especificação de um software o Desenvolvimento Dirigido ao Domínio (3.1), trabalha sobre o espaço do problema criando fronteiras, delimitando o *core* do domínio e criando uma linguagem ubíqua.

Assim o uso de teoria das categorias é um caminho natural, pois com ela é possível conectar desde as estrutura do sistema como também a própria semiótica do sistema.

Documentando determinadas escolhas durante esse processo, permitindo a transmissão futura da simbologia adotada, alguns trabalhos analisando as relações metafórica e analogia no campo da teoria das categorias já foram feitos Fuyama e Saigo (2018) e Brown e Porter (2006).

Considerando todo o conteúdo apresentado até então, e considerando também os problemas existentes ao construir especificações de software aderentes ao contexto em que os usuários do software estão inseridos, surge a seguinte indagação: "De que maneira é possível construir uma especificação de software que leve em consideração o contexto que delimita esta solução?"

1.2 Objetivo Geral

O presente trabalho tem por objetivo explorar de que forma é possível utilizar Teoria das Categorias e DDD no desenvolvimento de software, a fim de minimizar a ambiguidade inerente à modelagem da solução.

1.3 Objetivos Específicos

Analisar de que de forma é possível aplicar TC na modelagem de software. Relacionar TC e DDD, explorar a relação entre o resultado da sessão de *Event storming* e TC. Entender como dá-se a relação entre Eventos e a representação dos mesmos com tipo.

1.4 Estrutura do trabalho

A apresentação do trabalho foi organizada conforme segue:

- Este capítulo apresenta a introdução do trabalho através do seu contexto, motivação e objetivos.
- O capítulo 2 faz uma breve introdução a conceitos básicos no universo de Teoria das Categorias, fazendo algumas pontes para representação dos conceitos em Typescript.
- O capítulo 3 faz uma introdução a Desenvolvimento Dirigido ao Domínio, *Event Storming* e algumas reflexões sobre a aplicação de Teoria das categorias a modelagem de domínio.
- O capítulo 4 apresenta um problema e uma possível solução para o mesmo utilizando Typescript.
- O capítulo 5 finaliza o trabalho. Dedica-se a apresentar as conclusões alçadas a partir dos estudos efetuados e da construção da solução proposta no capítulo 4.

2 Teoria das Categorias

2.1 Função

Seja a representação (x, y) um par ordenado, com propriedade essencial $(x, y) = (z, w)$. Se somente se x = z e y = w. O par ordenado representa uma relação binaria pertencendo a um Conjunto no qual todos os elementos são pares ordenados. Se R é a relação e $(x, y) \in R$, podemos interpretar que x é atribuído a y pela associação que R representa. Assim,

$$f = \{(x, y)\colon y \text{ é o codomínio x sob } f\}$$

As relações que representam funções são aquelas nas quais o primeiro elemento do par ordenado aparece uma única vez no conjunto f.

$$\text{Se } (x, y) \in f \text{ e } (x, z) \in f, \text{ então } y = z$$

Desse modo uma função pode ser representada como sendo uma tripla ordenada (A, B, R), onde $R \subseteq A \times B$ é a relação de A para B, tal que para cada $x \in A$ existe um e somente um $y \in B$ para o qual $(x, y) \in R$. Essa representação é feita com base na ideia de conjuntos, capturando a ideia, mas não todo o significado.

2.2 Morfismo

2.2.1 Isomorfismo

Em geral, a palavra isomórfico é usada no contexto matemático com o significado de indistinguível em forma (BARR; WELLS, 1995, nossa tradução, p.41). A intuição de maneira relaxada é que objetos isomórficos tem a mesma aparência - tem a mesma forma. Isso significa que cada parte de um objeto corresponde a alguma parte de outro objeto em um mapeamento de um-para-um. Matematicamente isso significa que há um mapa do objeto a para o objeto b,e há um mapeamento do objeto b para o objeto a, eles são inversos entre eles (MILEWSKI, 2019, nossa tradução, p.56-57). Barr e Wells (1995) define o conceito de inversa como:

Definição 2.2.1 *Seja $f : A \to B$ e $g : B \to A$ morfismos em uma categoria na qual $f \circ g$ é o morfismo identidade em B e $g \circ f$ é o morfismo identidade em A. Então g é uma inversa de f, e claro, f é uma inversa de g (Figura 1).*

De acordo com Spivak (2014), uma função que é bijetora sempre é um isomorfismo e todo isomorfismo é uma bijeção. Toda função bijetora é injetora e sobrejetora:

Definição 2.2.2 *Seja $f : X \to Y$ uma função. Dizemos que f é sobrejetora se, para todo $y \in Y$ existe algum $x \in X$ tal que $f(x) = y$. Dizemos que f é injetiva se, para todo $x \in X$ e todo $x' \in X$ com $f(x) = f(x')$ temos $x' = x$.*

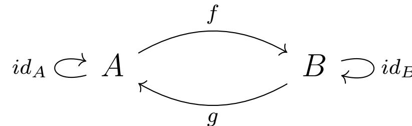


Figura 1 – Diagrama isomorfismo.

Definição 2.2.3 *Seja C é uma categoria e que A e B são dois objetos de C. Um morfismo $f : A \to B$ é dito ser um isomorfismo se ele possui uma inversa. Nesse caso, dizemos que A é isomórfico a B, escrevemos $A \cong B$.*

2.2.2 Epimorfismo

Segundo Baez (2017a) o conceito de epimorfismo é uma generalização do conceito de função sobrejetora entre conjuntos. Alguns dos jargões utilizados para se referir a um epimorfismo são 'é uma epi' ou 'é epic'.

Definição 2.2.4 *Seja $f : X \to Y$ uma morfismo em alguma categoria, se para cada objeto Z e cada par de morfismos paralelos $g_1, g_2 : Y \to Z$ (Figura 2):*

$$(g_1 \circ f = g_2 \circ f) \Longrightarrow (g_1 = g_2)$$

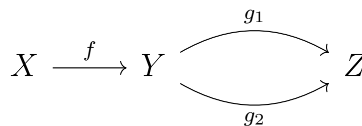


Figura 2 – Epimorfismo

2.2.3 Monomorfismo

Segundo Baez (2017b) o conceito de monomorfismo é a generalização da noção de função injetiva em conjuntos, a partir das categorias dos Conjuntos para arbitrarias categorias. Alguns dos jargões utilizados para se referir a um monomorfismo são 'é um mono' ou 'é monic'.

Definição 2.2.5 *Seja $f : X \to Y$ uma morfismo em alguma categoria, se para cada objeto Z e cada par de morfismos paralelos $g_1, g_2 : Z \to X$ (Figura 3):*

$$(f \circ g_1 = f \circ g_2) \Longrightarrow (g_1 = g_2)$$

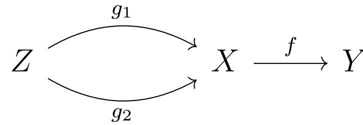


Figura 3 – Monomorfismo

2.3 Categorias

Cheng (2015) define Teoria das Categorias como uma especie de "meta-matemática". Tudo o que a matemática faz para o mundo, Teoria das Categorias faz para a matemática. Isso significa que Teoria das Categorias é intimamente relacionada com Lógica. Para Leinster (2016) Teoria das Categorias proporciona uma visão de alto nível, nesse alto nível, detalhes são invisíveis, mas padrões que antes eram impossíveis de serem notados, tornam-se visíveis.

Teoria das Categorias é o estudo da noção matemática de Categorias, o significado de categoria no contexto matemático foi introduzido por Eilenberg e MacLane (1945 apud MARQUIS, 2015), de maneira auxiliar, como uma preparação para o que eles chamaram de functor e transformação natural. Conforme Marquis (2015), a definição de Categorias evoluiu com o tempo, a definição inicial de Eilenberg e MacLane era puramente abstrata. Mais tarde Lawvere (1966 apud MARQUIS, 2015) fez uma abordagem alternativa, começou a caracterizar a categoria das categorias, e estipular que uma categoria é um objeto desse universo, essa abordagem culminou no que hoje é chamado de *higher-dimensional categories*. Mas isso é um pouco além do que necessitamos para esse trabalho,Eilenberg e MacLane (1945 apud MARQUIS, 2015), afirmam que a ideia de categoria é requerida somente pela percepção de que toda função deve ter a classe definida de domínio e uma classe definida de intervalo, pois categorias são fornecidas como domínio e intervalo de *functors*.

Uma categoria é definida em função dos seus objetos (escritos $A, B, C, ...$) e flechas (escritas $f, g, h, ...$). Onde cada flecha parte de um objeto(origem, domínio) para um objeto(alvo, codomínio). Assim uma flecha é uma ação direta:

Se uma flecha f tem como alvo um objeto b, e uma flecha g tem sua origem em b e alvo em c, então existe uma flecha $g \circ f$ que tem origem em a e alvo em c, essa ação é chamada composição (Figura 5). Badiou, Bartlett e Ling (2014) descrevem esse comportamento em função do 'ato'. Desse modo podemos entender que há uma flecha que

$$A \xrightarrow{f} B$$

Figura 4 – Diagrama morfismo.

exerce um ‘ato’ em b, enquanto outro ‘ato’ sai de ‘b’, existe uma ação que ‘liga’ a primeira com a segunda, e essa ação é a ‘junção’ das outras duas, sendo esta a única operação exigida em uma categoria. Considerando a presença de três flechas $f : A \to B$, $g : B \to C$

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ & \searrow^{g \circ f} & \downarrow g \\ & & c \end{array}$$

Figura 5 – Diagrama composição.

e $h : C \to D$, cujo domínio e codomínio são tão relacionados que pode-se aplicar ambas de maneira sucessiva para obter uma flecha de A para D. É notório que a composição de flechas em um categoria é associativa (Figura 6).

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ (h\circ g)\circ f \downarrow\downarrow h\circ(g\circ f) & g\circ f \searrow \ \swarrow h\circ g & \downarrow g \\ D & \xleftarrow{h} & C \end{array}$$

Figura 6 – Diagrama de associatividade.

Cada objeto em teoria das categorias, possui uma ação vazia, que sai de si para si mesmo, desse modo existe uma flecha que sai de a e tem como alvo a, essa flecha recebe o nome de identidade (Id), assim a identidade de a é representada por $id(a)$. Uma flecha vazia é um elemento neutro na composição (Figura 7).

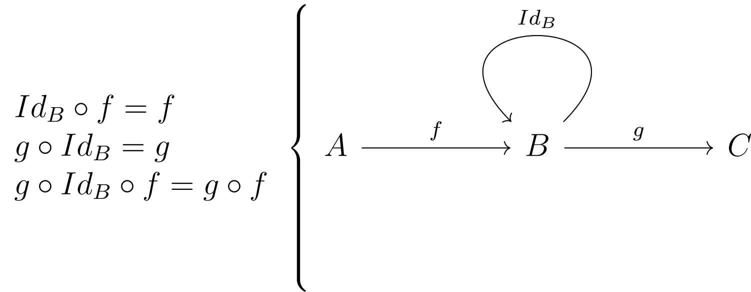


Figura 7 – Diagrama de composição identidade.

Considerando as definições apresentadas nesta seção, é possível formalizar a definição de categoria como segue:

Definição 2.3.1 *Uma* categoria **A** *consiste em:*

- *uma coleção de objetos: A, B, C ...*
- *uma coleção de flechas (morfismos): f, g, h ...*
- *uma operação que atribui cada flecha f a um objeto* $\mathrm{dom}(f)$ *e um objeto* $\mathrm{cod}(f)$*. Se $a = \mathrm{dom}(f)$ e $b = \mathrm{cod}(f)$, assim f: $a \to b$*
- *uma operação que atribui a cada par $\langle g, f\rangle$ de flechas com* $\mathrm{dom}(g) = \mathrm{cod}(f)$*, uma flecha $g \circ f$, a composição de f e g, tendo* $\mathrm{dom}(g \circ f) = \mathrm{dom}(f)$ *e* $\mathrm{cod}(g \circ f) = \mathrm{cod}(g)$*, de tal modo que obtemos: Lei da Associatividade, dada a configuração $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$, obtemos $h \circ (g \circ f) = (h \circ g) \circ f$, conforme apresentado na figura 6.*
- *uma operação que atribui a cada objeto b uma flecha $Id_b : b \to b$, chamada identidade em b, tal que para cada flecha: $f : a \to b$ e $g : b \to c$, que possuem uma operação de composição em que obtemos $Id_b \circ f$ e $g \circ Id_b = g$ e $Id_b \circ f = f$ e $g \circ Id_b = g$ (Figura 7)*

É possível representar categorias utilizando linguagem de programação, em Typescript, que é um superconjunto da sintaxe do ECMAScript 2015 (HEJLSBERG, 2019), assim podemos definir nossa categoria para Typescript como descrito abaixo:

- Objetos: são todos os tipos definidos em Typescript: string, number, Array<string>,...;
- Morfismos: são todas as funções em Typescript (Listagem 2.1);
- Morfismo identidade, é representado como uma função polimórfica (Listagem 2.2).
- Composição de Morfismo, é a usual composição (Listagem 2.3).

```typescript
declare function f<A, B>(a: A): B;
declare function g<B, C>(b: B): C;
```

Listagem 2.1 – Morfismo em Typescript

```typescript
const id = <A>(a:A) : A => a
```

Listagem 2.2 – Morfismo Identidade

```typescript
const compose = <A, B, C>(g: (b: B) => C,
                          f: (a: A) => B): ((a: A) => C) =>
                          a => g(f(a));
```

Listagem 2.3 – Morfismo Composição

2.4 Propriedade Universal, Objeto Inicial e Terminal

Em meados de 1960 Stanley Milgram liderou um experimento chamado '*The small world Problem*', que tinha por objetivo estudar a probabilidade de que dois indivíduos aleatórios se conhecessem, esse problema também pode ser visto como a distância média entre dois nós. O resultado do experimento ficou popularmente conhecido como 'Seis Graus de Separação': mensurando a distância média entre nós em redes complexas, a distância média entre você e qualquer pessoa no mundo são 4 indivíduos (BACKSTROM et al., 2011). Isso está diretamente relacionado a um importante conceito em teoria das categorias: uma vez escolhido o relacionamento, podemos procurar por um objeto especial que de alguma forma encapsula informações importantes. Matemáticos nomeiam esse objeto especial como propriedade universal:

- O número 0, o qual é o único número que quando adicionado a outro, nada acontece.
- O número 1, que quando multiplicado por outro número, nada acontece.
- O conjunto vazio, o qual é o menor conjunto possível.

Em teoria das categorias, um functor é mapeamento entre duas categorias que preserva suas estruturas, voltamos a abordar o assunto na seção 2.7.

Definição 2.4.1 *Uma propriedade universal de um objeto $c \in C$, é um functor F junto com um elemento universal $x \in Fc$ o qual define um isomorfismo natural $C(c, -) \cong F$*

O Objeto inicial, é um objeto que possui uma e somente uma flecha para todo objeto da categoria, entretanto a singularidade do objeto inicial não é necessária (caso exista). Mas garante a unicidade frente ao isomorfismo.

Definição 2.4.2 *Um objeto $i \in C$ é inicial, se para cada $c \in C$ existe um morfismo $i \to c$.*

O Objeto terminal, é um objeto que possui uma e somente uma flecha, com ele como alvo, partindo de qual quer outro objeto.

Definição 2.4.3 *Um objeto $x \in C$ é terminal se existe um morfismo $x \to 1$.É único frente ao isomorfismo, se um objeto terminal também é um objeto inicial, recebe o nome de objeto zero.*

Segundo Milewski (2015), na categoria dos **Conjuntos e funções** o objeto inicial é o conjunto vazio. Na linguagem Haskell, a representação é a função absurd :: void $\to a$, já em Typescript a função equivalente é:

```
1 declare function absurd<A>(nothing: never): A
```

Listagem 2.4 – Representação do objeto inicial em typescript

Na categoria dos **Conjuntos e funções** o objeto terminal é um *Singleton*, também conhecido como conjunto unitário, é um conjunto com exatamente um elemento. Na linguagem Typescript é definido como uma função polimórfica, que recebe qualquer valor e retorna *void*, conforme apresentado na listagem 2.5.

```
1 const unit = <A>(_:A) => { }
```

Listagem 2.5 – Representação do objeto final em typescript

2.5 Dualidade

Para cada *categoria* **C**, é possível formar uma nova *categoria* $\mathbf{C}_{op}$, chamada de *categoria* oposta ou *categoria* dual. Essas categorias possuem os mesmo objetos de **C**, mas seus morfismos são inversos, isso quer dizer que se em **C** tem uma flecha $f : a \to b$, pode-se considerar que em $\mathbf{C}_{op}$ há uma flecha $f' : b \to a$. Quando navega-se de **C** para $\mathbf{C}_{op}$, todos os conceitos são invertidos (Figura 8).

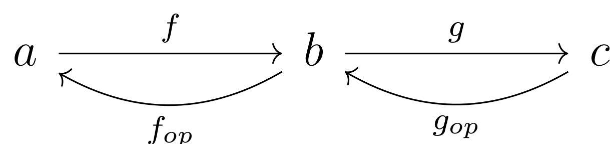


Figura 8 – Diagrama de Dualidade.

2.5.1 Produto e Coproduto

Se $\mathbf{A}$ e $\mathbf{B}$ são *categorias*, é possível formar *categorias* do produto $\mathbf{A} \times \mathbf{B}$: os objetos são pares ordenados (a, b) onde $a \in A$ e $b \in B$. Uma flecha com origem em $\mathbf{A} \times \mathbf{B}$ para $\mathbf{A}' \times \mathbf{B}'$ é um par (f, g) de um morfismo, onde $f : B \to B'$ e $g : A \to A'$.

Seja $\mathbf{A} = \{a, b, c\}$ e $\mathbf{B} = \{$■, ■, ■$\}$, o produto $\mathbf{A} \times \mathbf{B}$ é um grid 3×3. Com a definição do produto, também são definidas suas projeções, $\pi_1 : A \times B \to A$ e $\pi_2 : A \times B \to B$ (Figura 9).

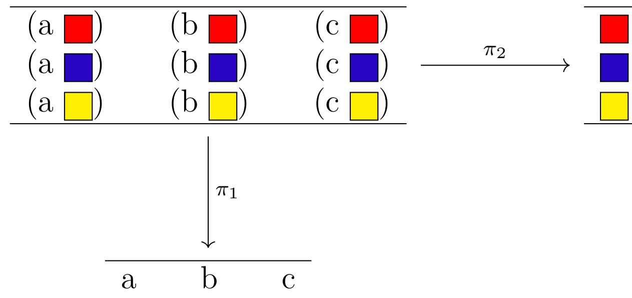


Figura 9 – Diagrama Produto.

Definição 2.5.1 *Um produto de dois objetos A e B, é um objeto $A \times B$, junto com duas flechas projeções $\pi_1 : A \times B \to A$ e $\pi_2 : A \times B \to B$, tal que para qualquer objeto C e par de flechas $f : C \to A$ e $g : C \to B$, existe exatamente uma flecha mediadora $\langle f, g \rangle : C \to A \times B$, fazendo o diagrama comutar (Figura 10).*

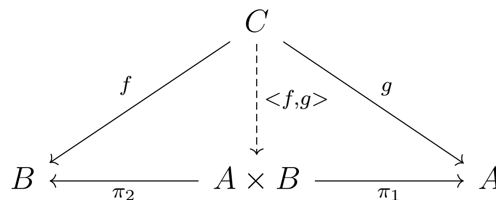


Figura 10 – Diagrama Produto comutativo.

Ao inverter a direção dos morfismos, obtemos sua dual, ou seja, o coproduto que corresponde a união disjunta.

Definição 2.5.2 *Um coproduto de dois objetos A e B é um objeto $A + B$, com duas flechas de injeção $l_1 : A \to A + B$ e $l_2 : B \to A + B$, tal que para qualquer objeto C e par de flechas $f : A \to C$ e $g : B \to C$, existe exatamente uma flecha $[f, g] : A + B \to C$, fazendo o diagrama comutar(Figura 11).*

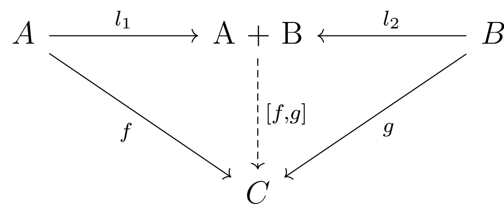


Figura 11 – Diagrama Coproduto comutativo.

O coproduto é diretamente relacionado ao *Sum type*. O tipo TrafficLight que exemplifica o coproduto é a união de três tipos, dessa maneira afirmamos que toda *TrafficLight* deve ter exatamente um dos três tipos que a constituem (Listagem 2.6).

```
1  type TrafficLight = 'Red' | 'Yellow' | 'Green'
```

Listagem 2.6 – Coproduto

2.5.2 Limite e Colimites

Limite e Colimites, podem ser entendidos como a maneira mais 'eficaz' de construir novos objetos. Limite e colimite são duais entre si, desse modo quando o limite cria um novo objeto selecionando um subconjunto o colimite o faz de maneira inversa aglutinado objetos. A tabela 1 traz alguns exemplos de limite e do seu colimite correspondente.

Limite	Colimite
Elementos únicos	Conjunto Vazio
Interseções em conjuntos	União disjunta
Pre Imagem	Não necessariamente união disjunta
Produto	Quociente

Tabela 1 – Exemplos de limites e colimites

De acordo com Badiou, Bartlett e Ling (2014), ao pensar em limite em termos geométricos, é possível visualiza-lo como uma 'posição universal' a qual permite ver todos os elementos, desse modo, definindo um diagrama D contendo os objetos a e b e uma flecha $f : a \to b$. Assim procuramos um objeto c(cone), o qual possui um propriedade universal, desse modo a e b são visíveis a c de maneira única. Desse modo o triângulo formado por $a \leftarrow c \to b$ deve comutar.

De modo inverso, é possível pensar no colimite como uma posição na qual todos os elementos podem visualizar um elemento c, dado um diagrama D contendo os objetos a e b e uma flecha $f : a \to b$. Procuramos então por um objeto c(co-cone) que seja visível a partir de a e b. De maneira análoga o triângulo $a \to c \leftarrow b$ também comuta.

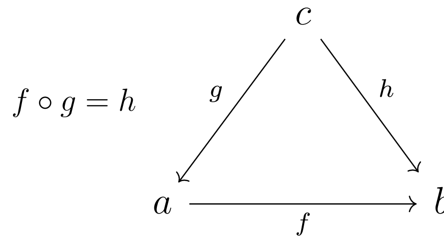


Figura 12 – Diagrama Cone.

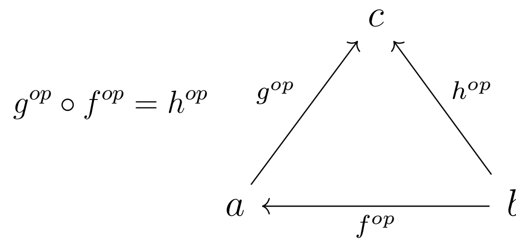


Figura 13 – Diagrama Cocone.

2.6 Allegorias

2.6.1 Relações

Uma relação é uma extensão do predicado. O predicado é por sua vez entendido em função do contexto no qual a proposição é definida. Assim, seja Y um contexto, A um tipo, pode-se estender Y em D, assim $D = Y$. Para um valor x não usado do tipo A, tem-se então que qualquer proposição em D com um predicado P em Y com variável livre x do tipo A, se t é um termo do tipo A, então temos a proposição $P\left[\frac{t}{x}\right]$ substituindo t por cada instância x em P. Também pode-se entender o predicado em termos de uma função, de tal modo que é possível estabelecer um operador $f(x_1, x_2, \ldots)$ agindo em objetos denotados $x_1, x_2, \ldots$, em um universo que retorna valores verdadeiros: True(T) ou False(F).

Definição 2.6.1 *Seja a família $(A_i)_{i:I}$ de conjunto, uma relação nessa família é um subconjunto R de um produto cartesiano $\prod_{i:I} A_i$, para o conjunto VT dos valores Verdadeiros(VT é um subobjeto classificador). Este também é um monomorfismo em subobjetos do produto cartesiano, assim:*

$$R \to \prod_{i:I} A_i$$

em uma relação binaria tem-se $A_1 \leftarrow R \to A_1$, induz uma noção natural da composição das relações em termo da composição da categoria das correspondências (GUIRE, 2017).

2.6.2 Rel

É uma categoria que tem como objetos conjuntos, seus morfismos são relações binarias em seus conjunto, **Rel** é um 2-*poset* por ser um 2-*categoria* possui um morfismo entre objetos e um morfismo entre morfismos, desse modo tem-se um morfismo $X \to Y$ o qual representa uma relação $R \subseteq X \times Y$, e um outro morfismo $R \to S$ que representa inclusões. A composição $S \circ R$ dos morfismos $R : X \to Y$, $S : Y \to Z$ é definida pela composição:

$$\{(x, z) \in X \times Z : \exists_{y \in Y} (x, y) \in R \wedge (y, z) \in S\} \subseteq X \times Z$$

A identidade $Id : X \to X$ é definida:

$$\{(x, x) : x \in X\} \subseteq X \times X$$

Se $R : X \to Y$ logo a categoria oposta é:

$$R^{op} : \{(y, x) \in Y \times X : (x, y) \in R\} \subseteq Y \times X$$

2.6.3 Categoria Allegory

Uma *Allegory* é uma categoria com propriedades que refletem a categoria *Rel* de relações. Foi inicialmente apresentada por Freyd e Scedrov (1990), que argumentaram que o cálculo categórico de relações é uma alternativa mais simples para o desenvolvimento de conceitos tradicionais expressos em linguagem, como o paradigma funcional. Uma *Allegory* é uma categoria enriquecida com três propriedades adicionais. Essas novas propriedades permitem comparar relações com uma ordem parcial, utilizando a interseção das duas relações, levando a relação para sua inversa usando o operador unário $()^{\circ}$. As propriedades de uma *Allegory* são apresentadas abaixo:

- Inclusão, supõe que qualquer par de flechas com a mesma origem, são comparáveis usando uma ordem parcial $\subseteq$ e que a composição é monótona, isso implica que a composição preserva a ordem e o reverso também preserva a ordem:

$$(S_1 \subseteq S_2) \;\; e \;\; (T_1 \subseteq T_2) \Rightarrow (S_1 \circ T_1) \subseteq (S_2 \circ T_2)$$

em *Rel*, tem-se $R : B \to A$ que é entendida como $R \subseteq A \times B$, assim inclusão de uma relação é análoga a inclusão em conjuntos.

$$R \subseteq S \equiv (\forall a, b : aRb \to aSb)$$

- *Meet*, assumindo que para todas as flechas $R, S : B \to A$ existe uma flecha $R \cap S : B \to A$, caracterizada pela propriedade universal:

$$X \subseteq (R \cap S) \equiv (R \subseteq R) \;\; e \;\; (X \subseteq S)$$

- Inversa, assumindo que para toda flecha $R : B \to A$, existe $R^\circ : A \to B$, esse operador possui as seguintes propriedades:
  - Involução
$$(R^\circ)^\circ = R$$
  - Sua ordem é preservada:
$$R \subseteq S \equiv R^\circ \subseteq S^\circ$$
  - É contravariante
$$(R \circ S)^\circ = S^\circ R^\circ$$

Bird e Moor (1997) apresenta mais um axioma que conecta todos os três operadores em uma *Allegory*. Esse axioma é conhecido como lei modular ou regra de Dedekind:

$$(R \circ S) \cap T \subseteq R \circ (S \cap (R^o \circ T))$$

2.7 Functor

É uma transformação entre categorias, preservando a estrutura da origem. Um functor F de uma categoria **C** para uma categoria **D** é uma função que atribui:

- Para cada objeto $a \in C$, um objeto $F(a) \in D$
- Para cada morfismo $f : a \to b$ um morfismo $F(f) : F(a) \to F(b)$
  - $F(Id_a) = Id_{F(a)}$, para todo objeto $a \in C$
  - $F(f \circ g) = F(f) \circ F(g)$, preserva a composição sempre que o lado esquerdo for bem definido.

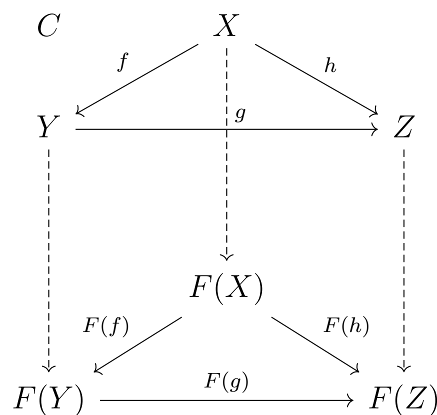


Figura 14 – Diagrama Functor.

Canti (2019b) define um Functor como um tipo construtor, que suporta operação de mapeamento 'map', 'map' pode ser usado para tornar uma função $a \to b$ em uma função $Fa \to Fb$ cujo argumento e tipo de retorno usa o tipo construtor F para representar o contexto computacional. Sua implementação em Typescript é apresentada na Listagem 2.7.

```
interface Functor<F> {
  readonly URI: F
  readonly map: <A, B>(fa: HKT<F, A>, f: (a: A) => B) => HKT<F, B>
}
```

Listagem 2.7 – Functor

2.8 Cálculo e Álgebra de tipos

Em 1934, Gödel propôs o que ele nomeou de função recursiva, seu trabalho foi publicado em maior detalhe por Stephen Kleen, no ano de 1936. Nesse mesmo ano, Alonzo Church propôs o cálculo lambda, o qual foi provado ser equivalente a função recursiva. Em 1937, Alan Turing propôs o que ele nomeou de Máquina de Turing, o qual dedicou um apêndice para mostrar a equivalência da sua máquina com o lambda tipado.

Em 1935, Gerhard Gentzen introduziu duas novas formulações lógicas: a Dedução Natural e o Cálculo Sequencial. Em 1969 William Alvin Howard foi motivado pelo trabalho de Haskell Curry, o qual observou o fato de que qualquer função tipada $(A \to B)$ pode ser lida como uma preposição $(A \supset B)$. Desse modo, Howard observou:

- Conjunção $A \ \& \ B$ corresponde ao produto cartesiano $A \times B$
- Disjunção $A \vee B$ corresponde a união disjunta $A + B$
- Implicação $A \subset B$ corresponde ao espaço da função $A \to B$
- Os predicados $\vee, \exists$ são correspondentes a Tipos Dependentes

Uma apresentação formal entre *lambda calculus* e dedução natural pode ser vista em Wadler (2015).

No ano de 1960 F. W. Lawvere desenvolveu Categorias cartesianas fechadas. Uma categoria com produtos finitos que é fechada em relação à sua estrutura monoidal cartesiana. Isso diz que existem isomorfismos (natural em A, B, C)

- $Hom_c(A, 1) \equiv \{*\}$
- $Hom_c(C, A \times B) \equiv Hom_c(C, A) \times Hom_c(C, B)$

Uma categoria Cartesiana fechada C, tal que para cada objeto $A \in C$ um $functor\,(-) \times A : C \to C$ tem um adjoint a direita , denotado $(-)^A$

$$Hom_c\,(C \times A, B) \equiv Hom_c(C, B^A)$$

No ano de 1973, Lambek publicou *Functional Completeness of cartesian categories*, CCC, no qual estressaram a conexão entre cálculo lambda de Church e as Categorias cartesianas fechadas.

O cálculo lambda pode ser sintetizado a grosso modo em termos das suas estruturas básicas, tipos, termos e equações entre termos.

- Tipo: coleção de *sorts*, obtidos pela regra:
  Sorts são tipos, 1 é um tipo, se A e B são tipos, $A \times B$ e B^A, são tipos
- Termos: para cada tipo A, atribui-se um conjunto contável $X_{i^A} : A = 0, 1, 2, 3, \ldots$. Termos são livremente gerados por variáveis, constantes e termos formadores de operação. Requer no mínimo os seguintes geradores:
  - $* : 1$
  - Se $a : A$, $b : B$, $c : A \times B$, então $\langle a, b\rangle : A \times B$, $\pi_1\,(c) : A\ \pi_2\,(c) : B$
  - Se $a : A$, $f : B^A$, $\varphi : B$ então $ev_{ab}\,(f, a) : B, \lambda_x : a^{*\varphi} : B^A$

Scott (2000) define λ-calc como uma categoria onde objetos são lambda cálculos tipados e os morfismos são traduções.

Teorema 1 *Existe um par de functor $C : \lambda - Calc \to Carl$ e $L : Carl \to \lambda - Calc$, o que configura a equivalência $\lambda - Calc \equiv Carl$.*

O functor L associado a CCC **A** sua linguagem interna, enquanto o functor C associa para qualquer lambda cálculo L, uma CCC gerada sintaticamente $C\,(L)$, onde objetos são tipos de L e flechas $A \to B$ são definidas por termos lambda $t\,(x)$ representando a prova $x{:}A \vdash t\,(x) : B$.

Lambek e Scott (1988 apud SCOTT, 2000) publicaram uma precisa equivalência entre as noções de Categorias cartesianas fechadas, cálculo lambda simplesmente tipado e lógica intuicionista. Existem outras diferentes estruturas de tipo com diferentes níveis de abstração, diferentes conceitos de tipos: tipo como conjuntos, tipo como álgebras, tipo como lattice, tipo como predicado, entre outros, cada um deles classifica algo, e possui sua correspondência em teoria das categorias (Tabela 2).

Teoria de Tipos	Teoria dos Conjuntos	Logica
A	Conjunto	Proposição
$x : A$	Elemento	Prova
$\emptyset$	$\emptyset, \{\emptyset\}$	$\bot, \top$
$A \times B$	Conjunto de Pares	$A \ \& \ B$
$A + B$	União	$A \mid\mid B$
$A \to B$	Conjunto de Funções (Hom)	$A \Rightarrow B$
$X : A \vdash B\,(x)$	Família de conjuntos	Predicado
$X : A \vdash b : B\,(x)$	Família de elementos	Prova condicional
$\prod_x : A\ B(x)$	Produto	$\forall x.B\,(x)$
$\sum_x : A\ B(x)$	União Disjunta	$\exists x.B\,(x)$
$p : x = a_y$	$x = y$	Prova de igualdade
$\sum_{x,y} : A\ x = A\ y$	Diagonal	Igualdade Relacional

Tabela 2 – Tabela Correspondência

Com essa correspondência é possível analisar a complexidade dos tipos de dados, em termos do número de instâncias que podem existir, assim o número de instância é igual a cardinalidade do conjunto. Por exemplo:

- boolean $= \{true, false\}$ possui tamanho 2
- int $= \{-2.147.483.648, \ldots, 2.147.483.647\}$ possui tamanho 4.294.967.295
- String $= \{\text{inf}\}$ infinitas instâncias
- $A \to B$ possui B^A instancia
- *CustomerType* possui tamanho 4, de acordo com a Listagem 2.8
- *Player* possui tamanho 2, de acordo com a Listagem 2.9
- *List* tem seu tamanho calculado pela Series de Taylor, de acordo com a Listagem 2.10

```
1 type CustomerType = { b: Boolean , c: Boolean }
```

Listagem 2.8 – Customer Type

```
1 interface PlayerOne {
2   kind: "PlayerOne";
3 }
4 interface PlayerTwo {
5   kind: "PlayerTwo";
6 }
7 type Player = PlayerOne | PlayerTwo
```

Listagem 2.9 – Player

```
type List<T> = { _: "Nil" } | { _: "Cons", h: T, t: List<T>}

const empty = <T>() : List<T> => ({ _: "Nil" })

const cons = <T>(head: T, tail: List<T>) : List<T> => ({ _: "Cons", h:
    head, t: tail })
```

Listagem 2.10 – List

Essas relações permitem que código seja tratado de maneira algébrica, assim uma assinatura pode ser simplificada:

$$(A \to C) \to ((B \to C) \to C)$$

Convertendo de tipo para conjunto:

$$\left(C^{\left(C^B\right)}\right)^{\left(C^A\right)}$$

$$\left(C^{\left(C^B\right)*\left(C^A\right)}\right)$$

$$C^{\left(C^{(A+B)}\right)}$$

Convertendo de conjunto para tipo:

$$(Either\,[A, B] \to C) \to C$$

O *Either* é um tipo de Dado Paramétrico que encapsula uma disjunção .

3 Conceitos Aplicados à Modelagem de Software

3.1 *Domain Driven Design*

Domain Driven Design (DDD) ou Desenvolvimento Dirigido ao Domínio é uma filosofia de desenvolvimento definida por Evans (2003). É uma metodologia que permite aos times de desenvolvimento gerenciar de maneira eficiente a construção e manutenção de software para problemas de domínio complexo (MILLETT, 2015). Como filosofia é composta por terminologia e princípios. A definição do termo DDD é abstrata , não tendo assim entendimento restrito, o mesmo principio é aplicado aos demais termos cunhando por Evans (2018), alguns mais abstratos que outros:

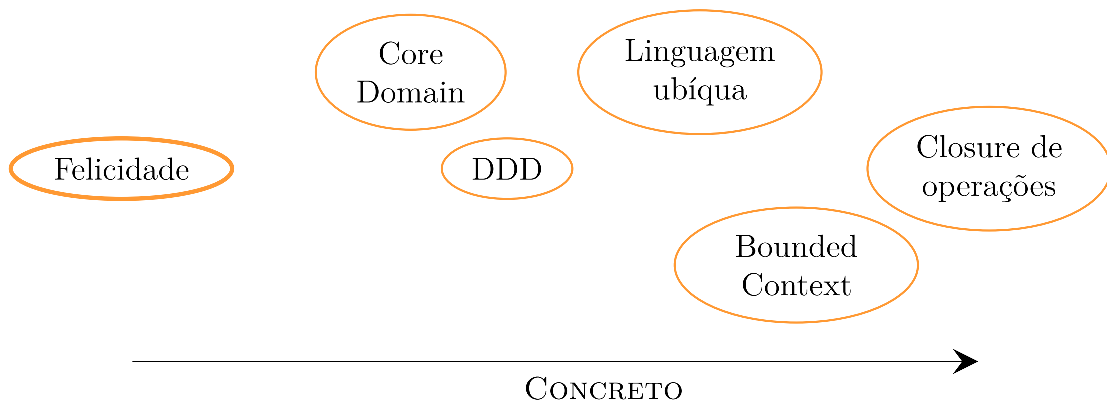


Figura 15 – DDD escala de abstração

Entre todos os princípios do DDD, Evans (2018) lista como conjunto principal:

- Foco no ***Core Domain***;
- Exploração dos **Modelos** de maneira criativa e em colaboração com os especialista e os desenvolvedores;
- Utilizar uma **Linguagem Ubíqua** em um explicito ***Bounded Context***;
- Escrever software que expresse esse modelo de maneira explícita;

O termo **Modelo** é definido por um conjunto de conceitos como abstração, seleção de dados, estabelecimento de formalismo e afirmações. Ao Imaginar um mapa Mercator, onde a abstração é a terra como uma esfera, Latitude e Longitude como a seleção dos

dados, a utilização de projeção preserva a direção entre os pontos. Não existe somente um modelo correto, mas sim modelos úteis a um cenário (EVANS, 2016).

Vernon (2013) define o termo ***Bounded Context*** como o espaço conceitual que circunscreve uma aplicação ou sistema finito. Essa fronteira existe a fim de destacar que termos, frases ou sentenças são contextualizadas. Há uma preocupação com a complexidade de definir o tamanho do **Bounded Context**, pois ao restringir de maneira rigorosa, buracos são criados, resultando em perda da definição de conceitos vitais. Ao flexibilizar as fronteiras acaba causando o acúmulo de conceitos que não reflete o domínio, causando o obscurecimento do contexto. Wlaschin (2018) por sua vez adiciona duas novas definições, complementando o entendimento, que são o **Espaço do problema** e **Espaço da solução**. Inicialmente um modelo é criado no **Espaço do problema**, posteriormente são extraídos os aspectos relevantes e depois recriados no **Espaço da solução**, nesse espaço domínio e subdomínio são mapeados para o ***Bounded context***.

Linguagem Ubíqua é a linguagem compartilhada pelo time, criada com base no modelo. Desse modo, o modelo é a espinha dorsal da linguagem ubíqua. É notório que os especialistas do domínio possuem uma grande influência na linguagem decorrente do melhor entendimento do negócio. Assim como qualquer outra linguagem, a linguagem do domínio também crescerá e mudará ao longo do tempo e essa premissa também é válida para o modelo.

Closure de Operações, quando a operação retorna o mesmo tipo que seus argumentos, então a operação é fechada sob o conjunto de instancias daquele tipo. Uma operação fechada provem uma interface de alto nível sem introduzir dependência ou outro conceito. Em termos matemáticos, a soma de dois números reais, o resultado é também um número real, então os números reais são fechados sob a operação de soma.

Core Domain é a porção do domínio que diferencia a aplicação e a torna um ativo de negócio, porém não é tão fácil definir tal parte, tão pouco notá-la ou definir suas fronteiras. Desse modo, Evans (2003) sugere que os domínios adjacentes sejam resolvidos a fim de aparar o ***Core Domain***, e deixa-lo o mais claro possível. Evans (2003) define alguns conceitos e padrões organizacionais:

- Conceitos:
  - Entidade
  - Objeto Valor
  - Agregação
  - Serviço
  - Repositórios
  - Fábricas

- Padrões organizacionais:
  - Parceria
  - *Kernel* Compartilhado
  - Consumidor-Fornecedor desenvolvimento
  - Conformista
  - Camada Anti-Corrupção
  - *Open Host Service*
  - Linguagem Publicada
  - Forma separada
  - *Big Ball of Mud*

A espinha dorsal do DDD é o processo de compartilhamento e aprendizagem. Para Evans (2003) um modelador eficaz de domínio é um *knowledge cruncher*, *Knowledge crunching* não é uma atividade solitária, um time de desenvolvimento colabora com peritos do domínio, juntos desenham as informações e a trituram em uma forma útil. A interação entre os times muda a medida que trituram o modelo. O refinamento constante do modelo faz com que os desenvolvedores aprendam os princípios do negocio que estão ajudando, ao invés de criar funções mecânicas.

A importância dessas atividades é notada ao observar as leis da evolução de software, ou leis de Lehman (1996), dentre as quais destacam-se a lei do aumento da entropia e a conservação da familiaridade. A entropia de um sistema aumenta com o tempo, a menos que um trabalho especifico seja realizado a fim de reduzi-la. A lei da conservação da familiaridade (complexidade percebida) por sua vez afirma que para que um E-Programa evolua de maneira saudável, se faz necessário que todos os associados mantenham a maestria sobre o conteúdo e comportamento, crescimento excessivo diminui a maestria, portanto o crescimento médio é estaticamente invariante. Não distante, North (2010) afirma que a ignorância é o maior limitador do desenvolvimento, o desconhecimento de aspectos específicos do problema.

3.2 Event Storming

O ***Event Storming*** foi criado por Brandolini em meados de 2013, é uma técnica aplicada para acelerar a aprendizagem e modelagem do domínio do problema. Para Brandolini (2017) o desenvolvimento de software é um processo de aprendizagem e trabalhar em código é um efeito lateral. Desse modo ele propõe uma metodologia de aprendizado acelerado guiado por um facilitador, o único que realmente necessita entender o ***Event Storming***, focada no negócio e em suas regras e não em dados.

O ***Event Storming*** é baseado em passos simples, é um *workshop* organizado com um grupo de 6 a 8 pessoas, que devem ser escolhidas de maneira a diversificar o grupo com pessoas que saibam o que devem perguntar e pessoas que saibam as respostas, e um espaço suficientemente grande para que o problema possa ser modelado. A exploração do domínio deve começar a partir dos eventos do domínio. Um evento do domínio é algo significativo que aconteceu no domínio. Todos os eventos devem ser colocados na superfície destinada a modelagem em geral com *stickies* laranjas, Command são geradores de eventos que são representados por *stickies* azuis. Alguns eventos são consequências ou origem de outros, desse modo, devem ser dispostos próximos ao mesmos, o agrupamento lógico de eventos e comando recebe o nome de *Aggregate* que é representado por *stickies* amarelos. Existem também *Actors* que são atores responsáveis por executar comandos. *Actors* podem ser usuários do sistema, sistemas externos ou o tempo, são representados por *stickies* roxos, e a assim como *command* e eventos devem ser colocados próximos aos seus respectivos efeitos. Algumas casos de estudo podem ser encontrados em Vernon (2013), Wlaschin (2018), Pilimon (2018), Pilimon (2019) e Michaluk (2018).

3.3 Olog

É uma tentativa de prover um *framework* matemático para representação de conhecimento, construção de modelos científicos e armazenamento de dados usando teoria das categorias, linguagem e ferramentas gráficas.

3.3.1 Tipos

Segundo Spivak (2014) um Tipo é um conceito abstrato, representado por caixa contendo uma frase sem verbo, no singular e com um artigo indefinido.

3.3.2 Aspectos

Segundo Spivak (2014) um Aspecto de uma coisa x é uma maneira de vê-la, uma maneira como x pode ser considerada ou mensurada. Por exemplo, uma mulher pode ser considerada uma pessoa; assim “ser uma pessoa” é um aspecto de uma mulher. Um aspecto é uma função.

3.3.3 Fatos

Segundo Spivak (2014), fatos são “caminhos de equivalência”, um caminho em olog é simplesmente uma sequencia de flechas.

3.4 Analogia, metáforas e Teoria das Categorias

Segundo Brown e Porter (2006) a classificação de objetos em um contexto particular é um tema central na ciência. Como exemplo, tomemos a classificação das formas geométricas na Figura 16. Diversos atributos são notados: um possui quatro lados, outro possui um angulo de 90°. A partir desses atributos é possível classificar as figuras em diferentes classes com diferentes nomes. A figura 2 é classificada como um quadrilátero, a figura 1 como um triangulo retângulo, dependendo de qual atributo é elencado as figuras podem ou não fazer parte da mesma classe. A comparação permite a construção de especificação de conceitos ou classes, e é uma ferramenta essencial no desenvolvimento de ontologias.

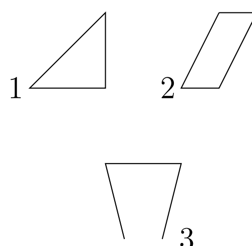


Figura 16 – O que não pertence?

Para Brown e Porter (2006) a analogia é o que eles chamam de "outro lado" da comparação e é essencial no processo "indutivo" da aquisição de conhecimento. Ao observar dois objetos ou conceitos A e B, e possuindo alguma definição sobre eles. É possível supor que algumas das especificações em A são iguais em B. A correspondência parcial, via comparação, das propriedades de A e B leva a uma analogia, um teste, experimento ou tentativa de prova ou ao começo de um processo de abstração.

Por muito tempo a ideia de que a linguagem que falamos influencia a maneira como pensamos foi aceita como verdadeira nos meios acadêmicos, porém o relativismo linguístico foi deixado de lado, e a ideia de que a linguagem que falamos influencia a maneira como categorizamos as coisas tomou seu lugar. Em termos de como pensamos, Lakoff e Johnson (1980) defendem que pensamos de maneira metafórica, assim metáforas tornam-se não somente uma ferramenta linguística mas uma maneira de construir conhecimento, pois permitem, o entendimento de domínios abstratos em termos de metáforas de um domínio mais concreto.

Lakoff e Johnson (1980) explicam o termo como um mecanismo cognitivo que nos permite raciocinar sobre algo como se fosse outro, é fundamentado e preserva a inferência através de mapeamentos de domínio. Desse modo pode-se inferir de um domínio conceitual (geometria) e raciocinar sobre outro (aritmética). Lakoff e Johnson (1980) propuseram que a aritmética foi criada com quatro fundações metafóricas diferentes, os quais criam um espaço conceitual abstrato a partir de experiencias corpóreas apresentadas abaixo. A formalização matemática para esta proposta pode ser encontrada em Guhe, Smaill e Pease

(2009).

- **Coleção de Objetos**. Descreve como por meio da interação com objetos é experienciado que objetos podem ser agrupados e que existe uma certa regularidade quando uma coleção é criada.
- **Construção de Objetos**.De maneira similar com a metáfora da Coleção de Objetos, é experienciado que pode-se combinar objetos a fim de formar novos objetos.
- **Medidas**. Essa metáfora por sua vez captura a regularidade a fim de estabelecer o tamanho de objetos.
- **Movimento ao longo do caminho**. Adiciona o conceito de movimento ao longo de um caminho.

Richards e Dolch (1936 apud OLIVEIRA, 2015) apresentam os três núcleos que compõem uma metáfora são nomeados tenor, veículo e um atributo compartilhado. O fluxo do conhecimento é do veículo para o tenor, através dos atributos comuns. Tomando a visão de Richards e Dolch (1936 apud OLIVEIRA, 2015) como base, na frase de um autor desconhecido "Políticos e fraldas, devem ser trocados freqüentemente, e pelos mesmos motivos", em **??**) há a apresentação da estrutura na Figura 17, onde políticos tem o papel de tenor T, fraldas tem o papel de veiculo V, com atributos implícitos compartilhados. Assim temos funções $f : T \to A$ e $g : V \to A$ que extraem um atributo comum A de um tenor T e veiculo V.

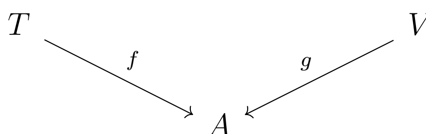


Figura 17 – Diagrama Cospan.

3.5 Comentários

DDD e *Event Storming* são entidades que existem de maneira apartada, não obstante podem ser utilizadas em conjunto. DDD é uma filosofia de desenvolvimento que permite um efetivo gerenciamento e construção de software. Como alguma de suas atividades são abstratas é possível intercambiar algumas, assim o que Evans (2003) chama de *Knowledge Crunching* pode ser facilmente substituído por seções de *Event Storming*. Ao ultilizar Event Storming o nosso foco é direcionado aos 'acontecimentos' (timeline de eventos ou *workflow*) o que nos força a entender como as mudanças acontecem no sistema. Na contramão está a modelagem convencional que possui uma visão estrutural, modela

visando comportamentos e nos dá uma visão de dependência e comunicação. Bonér (2017) fala que eventos representam fatos sobre o domínio e devem fazer parte da linguagem ubíqua do domínio. Deve ser modelado como um Evento do domínio e ajudar a definir o *Bounded Context*. Como um *Command* representa a intenção de uma ação, a ação por sua vez pode ser entendida como uma relação de causa e efeito, onde um efeito é sempre precedido por uma ação. Entendendo que um *Command* inicia um *workflow*, tratamos um *workflow* como um processo, e que um *workflow* gera um ou mais eventos do domínio.

Lahtinen e Stenvall (2017) fala que, há sistemas que possuem estado que sofrem processo que altera de alguma maneira seu estado. Falamos de processo de maneira livre, e provavelmente temos uma visão intuitiva sobre eles. Intuitivamente um processo leva um sistema de um estado para outro, assim um processo f leva A para B.

$$A \xrightarrow{f} B$$

Figura 18 – Diagrama processo.

Assim compreendendo um *workflow* como um processo, e sabendo que um processo por sua vez pode ser representado em termos de diagrama de corda (COECKE; KISSINGER, 2017), podemos então representar um *workflow* com diagrama de cordas. Diagrama de corta é um cálculo gráfico para expressar operação em categorias monoidais (DELPEUCH, 2019). É possível elevar uma categoria há uma categoria monoidal, Patterson (2017) transforma **Rel** em uma categoria monoidal equipando-a com o produto cartesiano.

Definição 3.5.1 *Um* ***Categoria Monoidal*** *é uma tripla* $(C, \otimes, I)$*:*

- C *é uma categoria.*
- $\otimes$ *é um functor* $C \times C \to C$
- I *é um objeto em* C *que age como Identidade para* $\otimes$*.*

Coecke e Kissinger (2015) descrevem a representação de processos em termos de diagrama de cordas. Um processo é uma caixa com algumas cordas na parte de baixo que representam as entradas do sistema, e algumas outras cordas na parte de cima representando a saída do sistema. Coecke e Kissinger (2015) apresentam alguns exemplos de processo (Figura 19):

Podemos formar diagramas mais complexos simplesmente ligando a saída de um processo na entrada de outro, porém isso só é permitido se os tipos de entrada e saída combinarem. Considerando o apresentado , este trabalho utilizará diagrama de cordas como

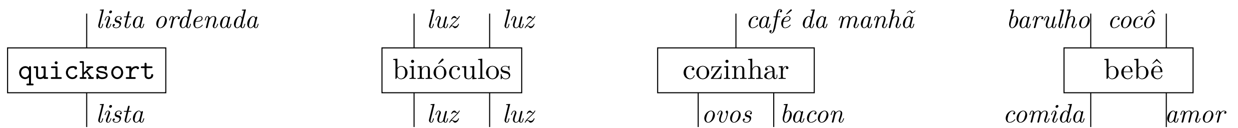


Figura 19 – Exemplos de processos usando diagramas de cordas

ferramenta para descrever *workflow*. No entendimento de Wlaschin (2018) um *workflow* sempre é contido por um único ***Bounded Context***, e nunca implementa um cenário fim-a-fim através de múltiplos contextos.

Ao pensarmos no **Bounded Context** e **Linguagem Ubíqua** em um contexto mais abstrato como a filosofia, é difícil não fazer uma ponte com o Jogo de Linguagem, descoberto por Wittgenstein (BILETZKI; MATAR, 2018). Um dos exemplos que ele utilizou para explicar a linguagem foi o da caixa de besouros:

> Imaginemos agora que cada um de nós tem uma caixa onde apenas o dono da caixa pode ver seu conteúdo. Dentro da caixa de cada um existe um besouro e todos afirmam que, olhando para dentro de suas caixas, estão vendo um besouro. No entanto, cada um está vendo aquilo que denominam como besouro. Posso estar vendo um besouro de uma determinada espécie, outra pessoa vê um besouro de outra espécie, outra um besouro maior ou de cor diferente. Mas posso afirmar estar vendo um besouro e, na verdade, besouro para mim é um objeto qualquer que não é o inseto. Posso também afirmar estar vendo um besouro mas minha caixa está vazia (posso estar mentindo). Todos, no entanto, afirmam estar vendo um besouro em suas caixas.
>
> Mas não estamos vendo a mesma coisa e isso instaura um problema de linguagem ou de comunicação. A única solução para esse dilema é estabelecer um jogo de linguagem, onde colocamos um besouro na frente de todos e, apontando para ele, chegamos ao acordo que isto é um besouro. Então, a única solução para esse problema é um acordo coletivo sobre as palavras, conceitos e ideias. Mas ao criar esse acordo estamos criando jogos de linguagem. Wittgenstein recomenda que, ao ouvir uma palavra de alguém, não a procuremos no dicionário mas antes perguntemos à pessoa o que aquela palavra significa ("o que significa besouro para você?"). (CARNEIRO, 2014)

Spivak (2014) diz que um *fiber product* de um diagrama serve para definir um novo conceito, ser explicito diminui as chances de interpretações erradas entre diferentes grupos de pessoas. Há uma intuição de que o Jogo de Linguagem pode em certos aspectos ser expresso em termo de um *fiber product*, Mazur (2016) por outro lado remete o Jogo de Linguagem ao lemma Yoneda. São sinônimos de *fiber product*: *pullback*, *fibered product* ou *fibre product*. A Figura 20 apresenta a representação de um *fiber product* com o objetivo de expressar o jogo de linguagem.

O exemplo na Figura 20 pode parecer trivial ou até mesmo irrelevante, mas como dito por Cheng (2018) ao falar sobre lógica, cada passo dado deve ser suportado inteiramente

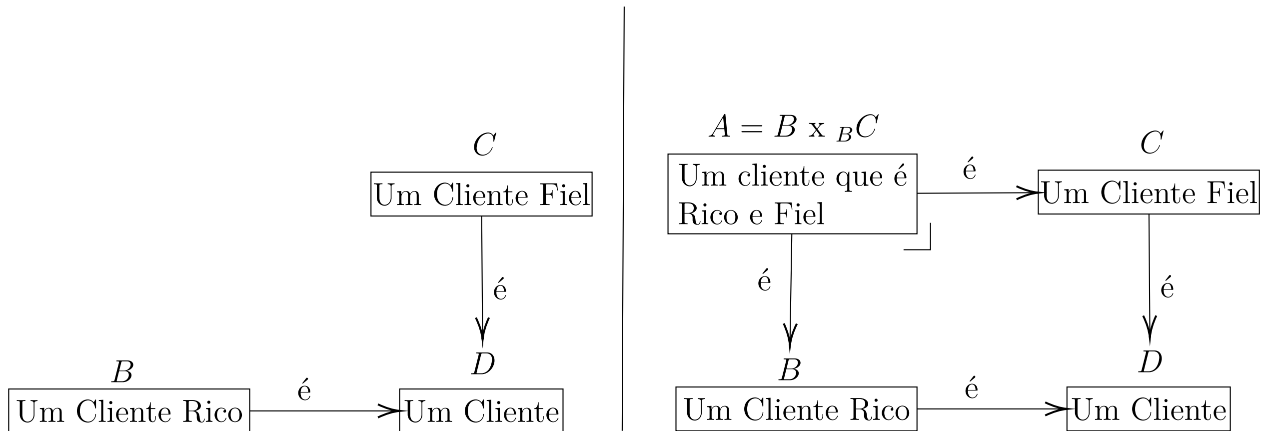


Figura 20 – Pullback

por lógica, o que significa que deve ser apenas uma descompactação de alguma definição, e deve ser óbvia e possivelmente trivial, mas quando colocadas em sucessão, pode-se chegar a um lugar novo e muito distante do início.

O grande ponto de utilizar teoria das categorias, é que observar as relações entre objetos é muito mais esclarecedor do que pensar somente em termos do objeto em si. Cheng (2018) ilustra um desses pontos ao apresentar os fatores de 30: $1, 2, 3, 5, 6, 10, 15, 30$, só que pensar em termos da lista não é esclarecedor, assim ela sugere a visualização usando diagrama de Hasse apresentado na Figura 21:

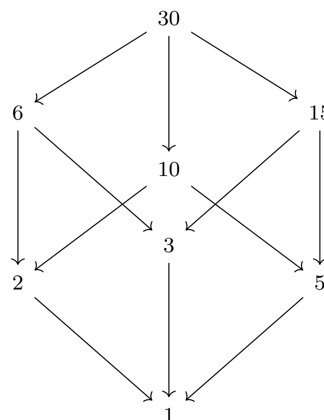


Figura 21 – Fatores de 30.

A partir do diagrama, Cheng (2018), aponta que na parte inferior temos 1, o segundo nível é formado pelos próximos menores fatores (2,3,5), o terceiro nível por sua vez possui fatores maiores do que o nível dois e os seus valores podem ser expresso pela

multiplicação dos fatores do nível 2, o mesmo acontece para o ultimo nível.Cheng então abstrai a representação tornando o *framework* aplicado para visualização dos Fatores de 30, aplicável a qualquer conjunto de três elementos, ou de maneira mas abstrata o diagrama representa as relações entre conjuntos.

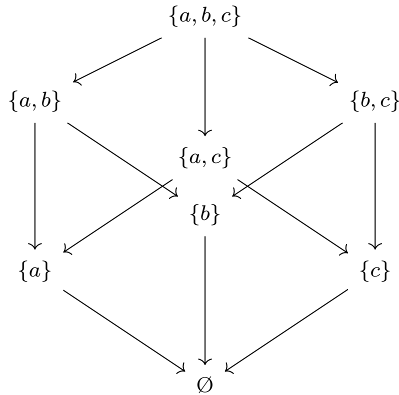


Figura 22 – Power set

Cheng (2018) considera então três tipos de privilégios: homem, rico e branco; estrutura o seguinte diagrama de Hasse apresentado na Figura 23:

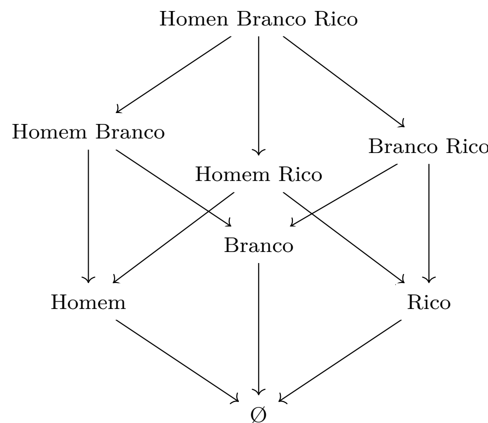


Figura 23 – Privilégios

Teoria das Categorias enfatisa o contexto no qual pensamos sobre algo, no lugar da coisa em si, isso inclui quais tipos de detalhes importam ou não na situação, o que conta como base da suposição, e o que precisa ser detalhado futuramente (CHENG, 2015), o que Cheng faz é definir o conjunto e posteriormente analisar as relações levando em consideração o contexto, cada camada hierárquica mostra o numero de privilégios, também é observado que a direção da *arrow* indica e perda de privilégios, uma descrição mais detalhada é encontrada em (CHENG, 2018). O importante a ser notado é o processo, a construção logica feita por ela para entender o privilegio, remete a Goguen (1991) que

argumenta, que para entender uma estrutura, é necessário entender os morfismos que a preservam.

Este trabalho sugere a utilização deste estilo de representação para modelagem de domínio de software, levando em consideração a aplicação de Teoria das Categorias como meio de transicionar entre diferente área e modelos, preservando assim as estruturas entre os morfismos.

4 Modelagem de Software Utilizando Teoria das Categorias, Event Storming e DDD

4.1 Proposta: Utilizando TC e DDD para Modelagem de Software

De acordo com o que foi pesquisado foram encontrados alguns conceitos que possuem relação entre si, muitas vezes não tão óbvia, como por exemplo TC e DDD. À luz desses conceitos, e na tentativa de resolver o problema de pesquisa indicado, apresentamos o seguinte processo:

1. *Knowledge cruching*: Ocorre a sessão de *Event Storming*, nessa fase pessoas vão escrever eventos de negócios nos *post-it* que serão colados na parede, outras pessoas podem responder postando sumarizações do *workflow* que são desencadeados por esse eventos. Ao final do refinamento da sessão é esperado o *workflow* da aplicação.
2. Modelagem Conceitual: Em mão do *workflow* criado, e entendendo que há uma ordenação natural, causa e efeito, é possível então representar o *workflow* em função de diagrama de cordas, onde eventos são as cordas e *command* são os processos.
3. Mapeamento da modelagem em software: Em mão do diagramas de cordas, podemos converter diretamente cada corda em *data types* e cada caixa em uma função.

4.2 FP-TS

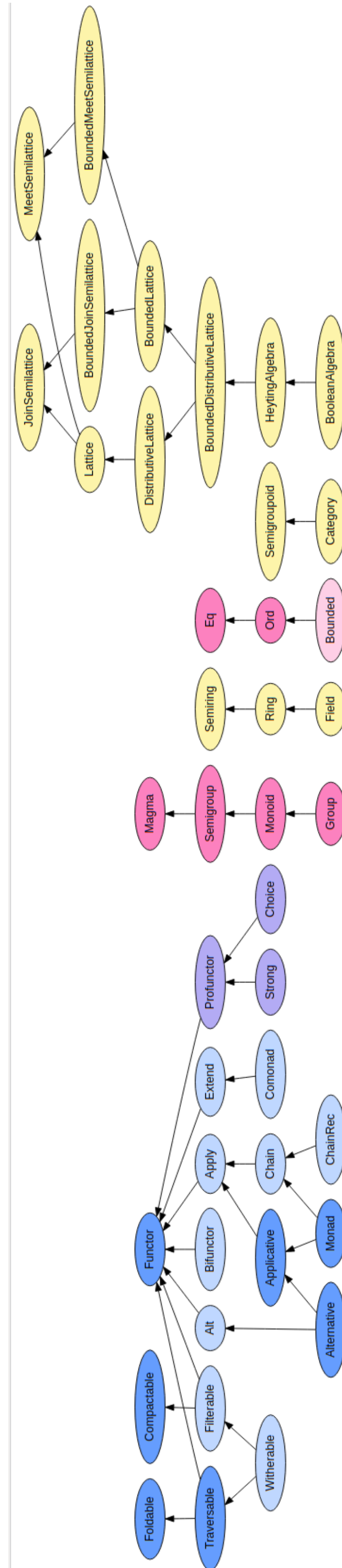
É uma biblioteca escrita em Typescript, para o desenvolvimento de aplicações puras, a biblioteca é construída em cima de abstrações de alta hierarquia. Typescript não suporta *Higher Kinder Types* por isso fp-ts emula a funcionalidade usando a proposta de Yallop e White (2014).

Type Classes (Figura 24) é um sistema de tipo que suporta *ad hoc polymorphism*, uma especie de interface que define algum comportamento, se um tipo é parte da *type class* significa que ele implementa o comportamento que a *type classe* descreve. Um exemplo de *type classe* é Functor.

4.3 Definição do Problema

O problema trabalhado é um fluxo de criação de pedidos, formulários são entregues e categorizados conforme um *checkbox* de orçamento, se marcado o formulário é um pedido

Figura 24 – Type class



Fonte: Canti (2019a)

de orçamento caso contrário é um pedido. Os dados são validados: o nome do cliente, e-mail, endereço de entrega e endereço de pagamento, os endereços devem existir. Casos haja alguma inconsistência o cliente é notificado para fazer uma retificação. O próximo passo é a validação do código de produto ferramentas sempre começam com a letra “F” seguida de 4 dígitos, engrenagens começam com a letra “E” e cinco dígitos, é verificada a existência dos itens, se algum dos códigos for inválido o formulário é considerado inconsistente. Para cada item é checado se há a disponibilidade da quantidade, a quantidade sempre é um valor natural, em caso de não disponibilidade formulário é considerado inconsistente. Cada item agora tem seu valor recuperado, e o valor total do pedido é calculado, somando o valor de cada item pedido. Por fim uma cópia é enviada para o cliente e outra para o setor de contas.

4.4 Representação do Domínio

O primeiro passo para estruturar a solução é entender que a estruturação é *ad hoc*, ou seja é realizada para um contexto especifico. Na seção 4.3 temos a descrição do Espaço do Problema, na seção que 3.2 trata de *Event Storming*, o primeiro passo é identificar os eventos e escreve-los nos *post-its* da cor laranja, devem ser escritos no passado, assim é fácil chegar em uma possível solução como a listada abaixo:

- Pedido recebido
- Orçamento recebido
- Pedido encomendado
- Pedido Enviado

Com a listagem dos eventos, o próximo passo é nos perguntarmos o que leva esses eventos a acontecerem, como pontuado na seção 3.2 damos o nome de *command* a esse geradores de eventos e os escrevemos de maneira imperativa, chegando então ao quadro da Figura 25.

Observando os *post-its* e entendendo a disposição dos mesmos como uma relação de causa e efeito, é possível então afirmar que existem uma relação binária de ordem parcial, logo é possível assumir que a disposição dos *post-its* forma um *poset*. Ao dispor os elementos naturalmente haverá a criação de clusters, e eles nos fornecem uma visão das fronteiras dos subdomínios do espaço do problema. A descrição do problema também nos leva a um melhor entendimento das interseções entre eles, na Figura 26 atribuímos nomes conforme levantado em 4.3, são eles os domínios de recebimento de pedido e o de envio de pedido, esse subdomínios podem ser entendidos em termo do fecho do *subset* do conjunto.

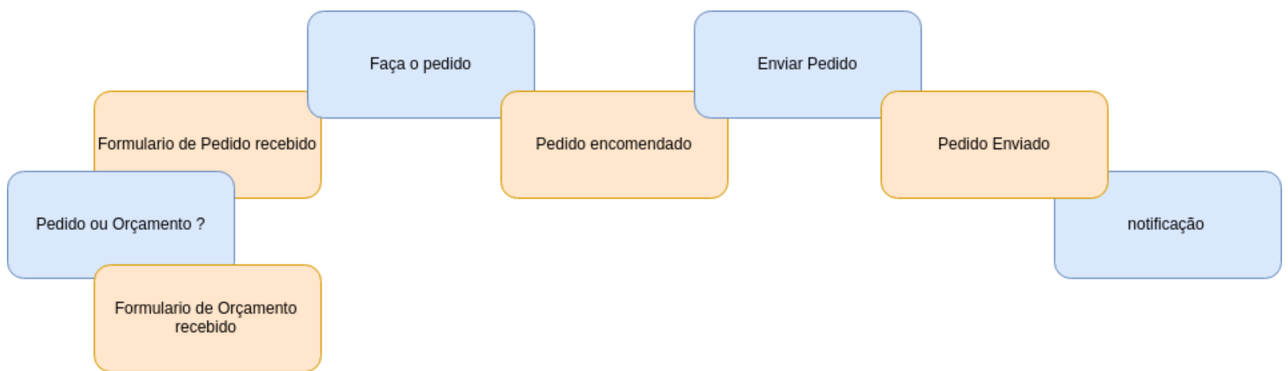


Figura 25 – Eventos

Quando saímos do Espaço do problema para o Espaço da solução temos a mudança de vocabulário, deixamos então de usar domínio para usar *bounded context* pois passamos a usar o vocabulário do DDD para descrever o domínio da solução (Figura 26).

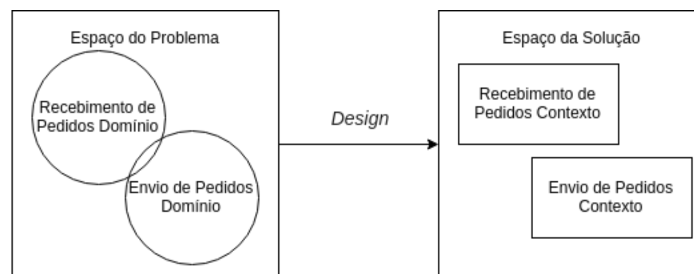


Figura 26 – Domínio e Contexto

Com os contextos definidos, vamos focar no desenvolvimento do Contexto de Recebimento de Pedidos, transformaremos os *post-it* em processos, como apresentado em 3.5. Assim consideremos que cada *command* recebe então um evento como entrada e retorna uma evento como saída, não é tão estanho propor essa abordagem visto que eventos são descritos como estados no passado.

Figura 27 – Processos

A Figura 27 apresenta a conversão dos eventos e *commands* para processos, assim nossa cordas serão os eventos nossos processos serão *command*, para compor os diagramas é necessário que a saída de um diagrama sirva como entrada de outro, a composição do processo de pedidos nos leva a Figura 28.

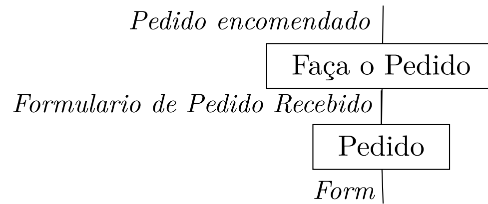


Figura 28 – Pedido composição

A Figura 28 representa o fluxo de encomenda, para melhor detalhar o fluxo podemos aplicar a substituição de diagramas como descrito por Coecke e Kissinger (2017, p.58).

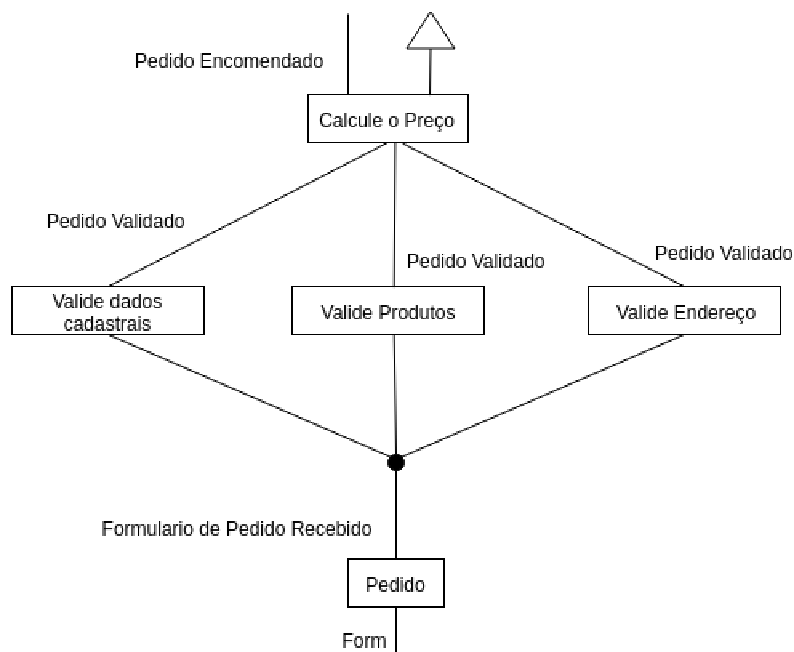


Figura 29 – Diagrama de Corda.

Na Figura 29 foi possível detalhar e especificar o fluxo de encomenda de produtos, apesar de termos descrito os processos, não temos o entendimento do que são os tipos representados pelas cordas, para a construção do entendimento como levantado na seção 3.5, faremos uso de *Olog*, assim a partir da descrição 4.3, levantamos tipos, aspectos e fatos:

Enquanto Teoria dos Processo nos permite mapear o *workflow*, Olog permite estruturar o problema em termos das relações, permitindo a criação da simbologia dentro do contexto limitador. Fazendo uso da Figura 29 onde cada caixa é uma função e cada corda é um objeto a escrita do programa dá-se com o alinhamento das funções, assim uma possível representação em termos de código:

```
1 type Form = any;
2 type OrderRecevied = any
3 type OrderSended = any;
```

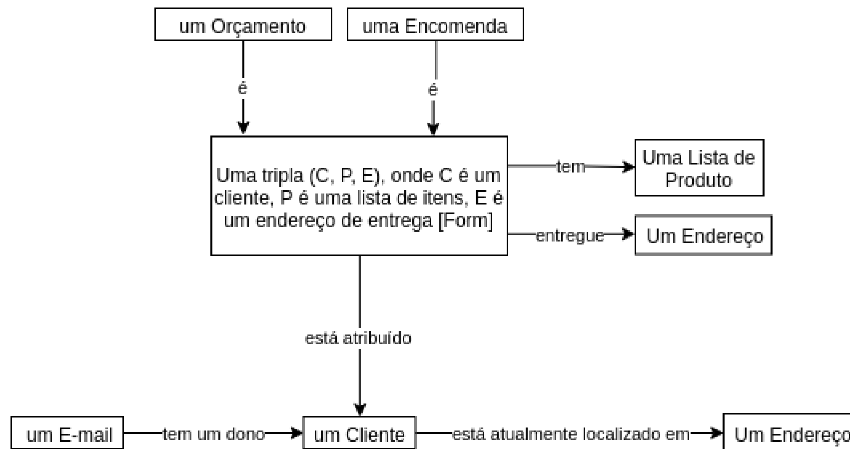


Figura 30 – Olog formulário

```
4  type OrderValidated = any;
5
6  declare function toOrderRecevied(form: Form): OrderRecevied
7
8  declare function requestOrder(form: Form): OrderRecevied;
9
10 declare function validateProducts(orderRecevied: OrderRecevied):
      OrderValidated;
11 declare function validateAndress(orderRecevied: OrderRecevied):
      OrderValidated;
12 declare function validateRegistrationData(orderRecevied: OrderRecevied):
       OrderValidated;
13
14 declare function calculatePrice(orderValidated: OrderValidated):
      OrderSended;
```

Listagem 4.1 – Função e tipos

Apesar da representação na Listagem 4.1 encaixar na apresentação na Figura 29, ela não nos da a representação do *workflow*, o primeiro passo para representa-lo é elevar as validações para *Validation* que é uma instancia de *Functor*, uma das possiveis representações é feita na Listagem 4.2.

```
1  declare function validateProducts(
2    orderRecevied: OrderRecevied
3  ): Validation<string, OrderValidated>;
4
5  declare function validateAndress(
6    orderRecevied: OrderRecevied
7  ): Validation<string, OrderValidated>;
8
9  declare function validateRegistrationData(
10   orderRecevied: OrderRecevied
```

```
11 ): Validation<string, OrderValidated>;
```

Listagem 4.2 – Validações

Como visto na Figura 29 as validações são executadas em paralelo, a estrutura que nos permite tal comportamento é a *Applicative*, desse modo chegamos a representação da validação:

```
1  import { sequenceT } from 'fp-ts/lib/Apply';
2  import { getArraySemigroup } from 'fp-ts/lib/Semigroup';
3  import { getApplicative, Validation } from 'fp-ts/lib/Validation';
4
5  declare function toOrderValidated(orderRecevied: OrderRecevied):
      OrderValidated
6
7  function validate(orderRecevied: OrderRecevied): Validation<string[],
      OrderValidated>{
8      return sequenceT(getApplicative(getArraySemigroup<string>()))(
9          validateProducts(orderRecevied),
10         validateAndress(orderRecevied),
11         validateRegistrationData(orderRecevied)
12     ).map(_ => toOrderValidated(orderRecevied));
13 }
```

Listagem 4.3 – Validação

O ultimo passo é a criação do workflow, que vai consistir em seguir os passos da Figura 29:

```
1
2  import { lift } from "fp-ts/lib/Functor";
3  import { getArraySemigroup } from "fp-ts/lib/Semigroup";
4  import {
5    getMonad,
6    Validation,
7    validation
8  } from "fp-ts/lib/Validation";
9  function workflow() {
10   return (form: Form) => {
11     const { chain, ap, of } = getMonad(getArraySemigroup<string>());
12     const orderRecevied = ap(of(toOrderRecevied), of(form));
13     return chain(orderRecevied, validate).map(calculatePrice);
14   };
15 }
```

Listagem 4.4 – Programa

5 Considerações Finais

O desenvolvimento do presente estudo possibilitou uma exploração da aplicação de Teoria das Categorias no desenvolvimento de Software com o uso de DDD, com o objetivo de minimizar problemas de ambiguidade na comunicação e representação do problema. Tal abordagem deu-se considerando o uso de Diagrama de cordas e *Olog*.

Primeiramente, assumimos que a resultante de uma sessão de *Event Storming* é um digrafo. Assim aplicamos functor que leva cada nó para uma corda e cada aresta do grafo para um caixa na categoria dos Processos, apesar de intuitivo, o processo não foi formalizado e não houve a definição de qual é a categoria inicial que representava o resultado da sessão de *Event Storming*. Como levantado no trabalho, em um determinado nível de abstração ou elencando uma característica podemos agrupar objetos, no caso da seções de *Event Storming*, elencamos que haviam objetos que geravam eventos e eventos gerados, e dessa maneira simplificamos os tipos finais em dois grupos, um representado por um nó e o outro por arestas.

A partir da representação obtida na categoria dos processos, fizemos a expansão das cordas utilizando *olog*, cada corda é expandida a fim de explicar o que ela representa e do que é formada. Por fim fizemos uso do resultado dos diagramas transcrevendo cada corda como um tipo de dados e cada caixa como uma função, criando assim as assinaturas de métodos que representam o *workflow* da aplicação.

O uso de Teoria das Categorias, aumenta a complexidade da representação do domínio da solução, o uso de linguagem natural por mais que aberta a ambiguidade é menos complexa e mais rápida, a necessidade de representar as relações em termos de funções acaba por aumentar a complexidade, um exemplo pode ser visto na Figura 30, a tripla (C,P,E) é ligada diretamente a uma lista de produtos, não ao produto em si, pois ao fazer uso da relação direta com o produto deixaríamos de ter uma função, problemas como esse tornam a representação mais complicada, pois exige um maior cuidado ao definir as relações.

Como trabalhos a serem desenvolvidos a partir deste, sugere-se os seguintes:

- Formalização da representação do resultado da seções de *Event Storming* em termos de categorias. Bem como a formalização do Functor que faz a ligação com a categoria dos processos.
- Exploração da relação entre Olog e a Categoria dos Processos.

Referências

BACKSTROM, L. et al. Four Degrees of Separation. *arXiv:1111.4570 [physics]*, nov. 2011. Citado na página 19.

BADIOU, A.; BARTLETT, A.; LING, A. *Mathematics of the Transcendental*. [S.l.]: Bloomsbury Publishing, 2014. ISBN 978-1-4411-5043-1. Citado 2 vezes nas páginas 16 e 22.

BAEZ, J. *Epimorphism*. 2017. Https://ncatlab.org/nlab/show/epimorphism. Citado na página 15.

BAEZ, J. *Monomorphism*. 2017. Https://ncatlab.org/nlab/show/monomorphism. Citado na página 15.

BAEZ, J. C.; POLLARD, B. S. A Compositional Framework for Reaction Networks. *Rev. Math. Phys.*, v. 29, n. 09, p. 1750028, out. 2017. ISSN 0129-055X, 1793-6659. Citado na página 12.

BARR, M.; WELLS, C. *Category Theory for Computing Science*. [S.l.]: Prentice Hall, 1995. (Prentice-Hall International Series in Computer Science). ISBN 978-0-13-323809-9. Citado na página 14.

BERRISFORD, G. *The Philosophy of Systems*. 2014. Http://grahamberrisford.com/AM%204%20System%20theory/Philosophical%20dichotomies.htm. Citado na página 11.

BERRISFORD, G. *Introducing Description Theory*. 2017. Citado na página 12.

BERRISFORD, G. *Von Foerster's Ideas On Second Order Cybernetics, and Disambiguating Self-Organisation*. 2017. Http://grahamberrisford.com/AM%204%20System%20theory/The%20claims%20of%202nd%20order% Citado na página 11.

BERRISFORD, G. *The Science and Philosophy of Systems*. 2018. Citado na página 11.

BILETZKI, A.; MATAR, A. Ludwig Wittgenstein. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Summer 2018. [S.l.]: Metaphysics Research Lab, Stanford University, 2018. Citado na página 37.

BIRD, R.; MOOR, O. de. *Algebra of Programming*. London ; New York: Prentice Hall, 1997. (Prentice-Hall International Series in Computer Science). ISBN 978-0-13-507245-5. Citado na página 25.

BONÉR, J. *Events-First Domain-Driven Design - Reactive Microsystems [Book]*. 2017. Https://www.oreilly.com/library/view/reactive-microsystems/9781491994368/ch04.html. Citado na página 36.

BRANDOLINI, A. *50,000 Orange Stickies Later*. 2017. Citado na página 32.

BROWN, R.; PORTER, T. Category Theory: An abstract setting for analogy and comparison. In: *What Is Category Theory? Advanced Studies in Mathematics and Logic*. [S.l.: s.n.], 2006. p. 257–274. Citado 2 vezes nas páginas 13 e 34.

CANTI, G. *Core Concepts - Fp-Ts*. 2019. Https://gcanti.github.io/fp-ts/introduction/core-concepts.html. Citado na página 42.

CANTI, G. *Module Functor*. 2019. Https://gcanti.github.io/fp-ts/modules/Functor.ts.html. Citado na página 26.

CARNEIRO, A. d. M. R. *Wittgenstein: Jogos de linguagem e os besouros nas caixas*. 2014. Citado na página 37.

CHENG, E. Cakes, Custard and Category Theory: Easy Recipes for Understanding Complex Maths. In: *Cakes, Custard and Category Theory: Easy Recipes for Understanding Complex Maths*. [S.l.]: PROFILE BOOKS LTD, 2015. ISBN 978-1-78125-287-1. Citado 2 vezes nas páginas 16 e 39.

CHENG, E. *The Art of Logic in an Illogical World*. [S.l.]: Basic Books, 2018. ISBN 978-1-5416-7250-5. Citado 3 vezes nas páginas 37, 38 e 39.

COECKE, B.; KISSINGER, A. Categorical Quantum Mechanics I: Causal Quantum Processes. *arXiv:1510.05468 [quant-ph]*, out. 2015. Citado na página 36.

COECKE, B.; KISSINGER, A. *Picturing Quantum Processes*. [S.l.]: Cambridge University Press, 2017. ISBN 978-1-107-10422-8. Citado 2 vezes nas páginas 36 e 45.

DELPEUCH, A. *String Diagram in nLab*. 2019. Https://ncatlab.org/nlab/show/string+diagram. Citado na página 36.

DIJKSTRA, E. W. *The Humble Programmer*. 1972. Https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html. Citado na página 12.

EILENBERG, S.; MACLANE, S. General Theory of Natural Equivalences. *Transactions of the American Mathematical Society*, v. 58, p. 231–294, 1945. ISSN 00029947. Citado na página 16.

ELVERFELDT, K. von. Observation and Distinction: The Underlying Method. In: *System Theory in Geomorphology*. 1. ed. [S.l.]: Springer Netherlands, 2012. p. 13–19. ISBN 978-94-007-2821-9. Citado na página 11.

EVANS, E. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0-321-12521-5. Citado 4 vezes nas páginas 30, 31, 32 e 35.

EVANS, E. *Eric Evans — Tackling Complexity in the Heart of Software*. 2016. Citado na página 31.

EVANS, E. *Keynote: DDD Isn't Done: A Skeptical, Optimistic, Pragmatic Look*. 2018. Citado na página 30.

FOERSTER, H. *Understanding Understanding*. [S.l.]: Springer Science+Business Media New York, 2003. ISBN 978-0-387-95392-2. Citado na página 11.

FOERSTER, H. V. *Observing Systems*. [S.l.]: Intersystems Publications, 1984. (The Systems Inquiry Series). ISBN 978-0-914105-19-0. Citado na página 11.

FOWLER, M. *Neologism*. 2006. Https://martinfowler.com/bliki/Neologism.html. Citado na página 12.

FREYD, P.; SCEDROV, A. *Categories, Allegories*. [S.l.]: Elsevier Science, 1990. (North-Holland Mathematical Library). ISBN 978-0-08-088701-2. Citado na página 24.

FUYAMA, M.; SAIGO, H. Meanings, Metaphors, and Morphisms: Theory of Indeterminate Natural Transformation (TINT). *arXiv:1801.10542 [math, q-bio]*, jan. 2018. Citado na página 13.

GOGUEN, J. A. A categorical manifesto. *Math. Struct. Comp. Sci.*, v. 1, n. 1, p. 49–67, mar. 1991. ISSN 0960-1295, 1469-8072. Citado na página 39.

GUHE, M.; SMAILL, A.; PEASE, A. A Formal Cognitive Model of Mathematical Metaphors. In: *KI*. [S.l.: s.n.], 2009. Citado na página 35.

GUIRE, R. M. *Relation*. 2017. Https://ncatlab.org/nlab/show/relation. Citado na página 23.

HEJLSBERG, A. *TypeScript Is a Superset of JavaScript That Compiles to Clean JavaScript Output.: Microsoft/TypeScript*. 2019. Microsoft. Citado na página 18.

LAHTINEN, V.; STENVALL, A. Towards a unified framework for decomposability of processes. *Synthese*, v. 194, n. 11, p. 4411–4427, nov. 2017. ISSN 0039-7857, 1573-0964. Citado na página 36.

LAKOFF, G.; JOHNSON, M. *Metaphors We Live By*. Chicago: University of Chicago Press, 1980. ISBN 0-226-46801-1. Citado 2 vezes nas páginas 12 e 34.

LAMBEK, J.; SCOTT, P. *Introduction to Higher-Order Categorical Logic*. [S.l.]: Cambridge University Press, 1988. (Cambridge Studies in Advanced Mathematics). ISBN 978-0-521-35653-4. Citado na página 27.

LAWVERE, F. W. The category of categories as a foundation for mathematics. In: *Proceedings of the Conference on Categorical Algebra*. [S.l.]: Springer Science+Business Media New York, 1966. p. 1–20. Citado na página 16.

LEHMAN, M. M. Laws of software evolution revisited. In: GOOS, G. et al. (Ed.). *Software Process Technology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. v. 1149, p. 108–124. ISBN 978-3-540-61771-6 978-3-540-70676-2. Citado na página 32.

LEINSTER, T. Basic Category Theory. *arXiv:1612.09375 [math]*, dez. 2016. Citado na página 16.

MARQUIS, J.-P. Category Theory. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Winter 2015. [S.l.]: Metaphysics Research Lab, Stanford University, 2015. Citado na página 16.

MAZUR, B. Thinking about Grothendieck. p. 2, 2016. Citado na página 37.

MICHALUK, M. *Ddd-by-Examples/Factory: The Missing, Complete Example of Domain-Driven Design Enterprise Application Backed by Spring Stack*. 2018. Https://github.com/ddd-by-examples/factory. Citado na página 33.

MILEWSKI, B. *Products and Coproducts*. 2015. Citado na página 20.

MILEWSKI, B. Category Theory for Programmers. p. 499, 2019. Citado na página 14.

MILLETT, S. *Patterns, Principles and Practices of Domain-Driven Design*. [S.l.]: John Wiley & Sons, 2015. Citado na página 30.

NORTH, D. *Introducing Deliberate Discovery | Dan North & Associates*. 2010. Https://dannorth.net/2010/08/30/introducing-deliberate-discovery/. Citado na página 32.

NORVIG, P. *Review of "Metaphors We Live By*. 19–. Https://norvig.com/mwlb.html. Citado na página 12.

OLIVEIRA, J. N. Metaphorisms in Programming. In: KAHL, W.; WINTER, M.; OLIVEIRA, J. (Ed.). *Relational and Algebraic Methods in Computer Science*. Cham: Springer International Publishing, 2015. v. 9348, p. 171–190. ISBN 978-3-319-24703-8 978-3-319-24704-5. Citado na página 35.

PATTERSON, E. Knowledge Representation in Bicategories of Relations. abs/1706.00526, p. 73, 2017. Citado na página 36.

PHILLIPS, S.; WILSON, W. H. Categorial Compositionality: A Category Theory Explanation for the Systematicity of Human Cognition. *PLOS Computational Biology*, Public Library of Science, v. 6, n. 7, p. 1–14, jul. 2010. Citado na página 12.

PILIMON, J. *Event Storming and Spring with a Splash of DDD*. 2018. Https://spring.io/blog/2018/04/11/event-storming-and-spring-with-a-splash-of-ddd. Citado na página 33.

PILIMON, J. *Ddd-by-Examples/Library: A Comprehensive Domain-Driven Design Example with Problem Space Strategic Analysis and Various Tactical Patterns*. 2019. Https://github.com/ddd-by-examples/library. Citado na página 33.

RAMISCH, C.; HUDITA, I.-M. A comprehensive application of category theory to semantics of modelling language. p. 10, 2018. Citado na página 12.

RICHARDS, I.; DOLCH, E. *The Philosophy of Rhetoric*. [S.l.]: Oxford University Press, 1936. (Bryn Mawr College. Mary Flexner Lectures). ISBN 978-0-19-500715-2. Citado na página 35.

SCOTT, P. Some aspects of categories in computer science. In: HAZEWINKEL, M. (Ed.). [S.l.]: North-Holland, 2000, (Handbook of Algebra, v. 2). p. 3–77. Citado na página 27.

SOMMERVILLE, I. Introdução. In: *Engenharia de Software*. 9. ed. São Paulo: Pearson, 2011. p. 3–5. ISBN 978-85-7936-108-1. Citado na página 11.

SPIVAK, D. I. *Category Theory for the Sciences*. [S.l.]: MIT Press, 2014. ISBN 978-0-262-02813-4. Citado 3 vezes nas páginas 15, 33 e 37.

TANEGA, J. A. Default Invariance A Naïve Category Theory of Law and Finance. In: *From Hubris to Disgrace: The Philosophy, Politics and Economics of Finance in the early 21st century*. [S.l.]: Routledge, 2014. Citado na página 12.

VERNON, V. *Implementing Domain-Driven Design*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2013. ISBN 0-321-83457-7 978-0-321-83457-7. Citado 2 vezes nas páginas 31 e 33.

WADLER, P. Propositions As Types. *Commun. ACM*, v. 58, n. December 2015, p. 75–84, 2015. ISSN 0001-0782. Citado na página 26.

WALLACE, D. F. *This Is Water : Some Thoughts, Delivered on a Significant Occasion about Living a Compassionate Life*. New York: Little, Brown, 2009. ISBN 978-0-316-06822-2 0-316-06822-5. Citado 2 vezes nas páginas 11 e 12.

WLASCHIN, S. *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. 1st. ed. [S.l.]: Pragmatic Bookshelf, 2018. ISBN 78-1680502541. Citado 3 vezes nas páginas 31, 33 e 37.

YALLOP, J.; WHITE, L. Lightweight Higher-Kinded Polymorphism. In: HUTCHISON, D. et al. (Ed.). *Functional and Logic Programming*. Cham: Springer International Publishing, 2014. v. 8475, p. 119–135. ISBN 978-3-319-07150-3 978-3-319-07151-0. Citado na página 41.