UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE SISTEMAS DE INFORMAÇÃO

GABRIEL BOHRER SCHMITT

DOMAIN-DRIVEN DESIGN E PROGRAMAÇÃO FUNCIONAL NO DOMÍNIO DA INTEROPERABILIDADE DE DADOS NA ÁREA DA SAÚDE:

Aplicação de práticas de engenharia de software com o modelo FHIR

Porto Alegre
2021

GABRIEL BOHRER SCHMITT

DOMAIN-DRIVEN DESIGN E PROGRAMAÇÃO FUNCIONAL NO DOMÍNIO DA INTEROPERABILIDADE DE DADOS NA ÁREA DA SAÚDE:
Aplicação de práticas de engenharia de software com o modelo FHIR

Artigo apresentado como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação, pelo Curso de Sistemas de Informação da Universidade do Vale do Rio dos Sinos (UNISINOS).

Orientadora: Profa. Dra. Margrit Reni Krug

Porto Alegre
2021

DOMAIN-DRIVEN DESIGN E PROGRAMAÇÃO FUNCIONAL NO DOMÍNIO DA INTEROPERABILIDADE DE DADOS NA ÁREA DA SAÚDE:
Aplicação de práticas de engenharia de software com o modelo FHIR

Gabriel Bohrer Schmitt*
Profa. Dra. Margrit Reni Krug**

Resumo: Uma das principais dificuldades enfrentadas na transformação digital na área da saúde refere-se à interoperabilidade de dados. O FHIR mostrou ser um candidato promissor para estabelecer uma padronização do formato de dados de forma a permitir seu compartilhamento por organizações de saúde. Entretanto, o FHIR traz um grau de complexidade que resulta em uma curva de aprendizado íngreme para os desenvolvedores que o utilizam. Portanto, o objetivo deste trabalho foi explorar as possibilidades da aplicação do Domain-Driven Design, aliado à programação funcional, como forma de lidar com essa complexidade. Foi desenvolvida uma prova de conceito de um sistema como forma de validar essa abordagem. A principal contribuição deste trabalho foi a identificação da necessidade de se utilizar um modelo domínio específico para a aplicação que abstraia as complexidades do padrão FHIR.

Palavras-chave: Domain-Driven Design; programação funcional; FHIR.

1 INTRODUÇÃO

A área da saúde passa por profundas transformações digitais, provedores de saúde e instituições médicas adotam cada vez mais sistemas de software em seus processos, assim gerando “quantidades massivas de dados clínicos que necessitam ser compartilhados com provedores de serviços de saúde e pesquisadores com o objetivo de oferecer melhores cuidados e possibilitar avanços médicos” (MUKHERJEE et al., 2017, p. 29).

De acordo com Stan e Miclea (2018, p. 2) “para possibilitar esse compartilhamento, os dados devem obedecer a padrões e o candidato mais promissor promovido pela *Health Level Seven International* (HL7) atualmente é o FHIR (*Fast Healthcare Interoperability Resources*).” No entanto, enquanto o FHIR possibilita a representação de dados com alto grau de detalhamento e extensibilidade, ele também “requer uma curva de aprendizado profunda para sua compreensão e utilização” (HONG et al., 2017, p. 26). Olivero (2020, p. 4) acrescenta que, “considerando a alta

* Aluno do curso de Sistemas de Informação. Email: gbs149@edu.unisinos.br.
** orientadora

complexidade e tamanho dos padrões HL7, sistemas da área da saúde criados de acordo com os mesmos são menos manuteníveis e têm menos capacidade de adaptação que os sistemas comuns que não são da área da saúde."

O *Domain-Driven Design* (DDD) é uma prática popularizada no livro *Domain-driven design: tackling complexity in the heart of software* (EVANS, 2003) que, de acordo com Braun, Bieniusa e Elberzhager (2021, p. 4), tem como um de seus objetivos primários a redução da complexidade no software. Portanto, a motivação deste trabalho é explorar a utilização dessa prática, aliada às técnicas de programação funcional com uma linguagem fortemente tipada, o TypeScript (MICROSOFT, 2021), como forma de produzir uma camada de tradução entre o modelo FHIR e o modelo de domínio da aplicação.

Desta forma pretende-se responder à seguinte questão de pesquisa: é possível aprimorar o processo de desenvolvimento de sistemas na área da saúde com o padrão FHIR através da utilização de conceitos de DDD e programação funcional?

Estabeleceu-se então, o objetivo geral deste trabalho como sendo analisar a validade da abordagem DDD no contexto do desenvolvimento, documentação, complexidade, e manutenibilidade de código de sistemas que implementem a interoperabilidade de dados da área de saúde através do padrão FHIR.

De maneira a verificar a aplicabilidade prática dessa abordagem, propõe-se, como objetivos específicos do trabalho: identificar os principais padrões de escrita de código prescritos pela prática do DDD que podem auxiliar nesse contexto; verificar de que forma o paradigma de programação funcional pode facilitar a implementação desses padrões; implementar uma prova de conceito de aplicação FHIR como forma de demonstrar os conceitos apresentados; e validar a aceitação dessa proposta junto a alguns profissionais com experiência nesse domínio.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão expostos os principais conceitos abordados a partir da pesquisa bibliográfica realizada. São apresentados o padrão FHIR de interoperabilidade de dados de saúde, o DDD e a programação funcional.

2.1 Interoperabilidade e FHIR

Uma das grandes barreiras encontradas na transformação digital na área da saúde refere-se à dificuldade de integração e padronização entre diferentes aplicações para a troca de dados. A interoperabilidade é definida pela HIMSS (2019) como a capacidade de diferentes sistemas, dispositivos e aplicações de trocar e utilizar dados dentro e através de fronteiras organizacionais de forma padronizada.

A HL7, de acordo com seu website, "é uma organização sem fins lucrativos dedicada ao desenvolvimento de padrões (...) para a transmissão, integração, compartilhamento e recuperação de informações eletrônicas de saúde" (HL7, 2021). Dentre os padrões definidos pela HL7, destaca-se o FHIR, definido como "a próxima geração de *framework* de padrões criado pela HL7, [...] aproveitando os mais recentes desenvolvimentos em padrões web" (HL7, 2019).

Com o objetivo de possibilitar a interoperabilidade de dados na área da saúde, o FHIR define uma série de recursos que representam entidades a serem modeladas, como, por exemplo: paciente, medicação ou diagnóstico. "Sozinhos ou combinados esses recursos satisfazem a maioria dos casos de uso comuns" (HL7, 2019). Também é possível criar extensões para cobrir outros casos conforme a necessidade. De acordo com a especificação, "o FHIR usa uma abordagem de composição na modelagem. Com FHIR, casos de uso específicos são normalmente implementados pela combinação de recursos através do uso de referências" (HL7, 2019).

O padrão FHIR "é projetado como uma especificação de interface - especifica o conteúdo dos dados trocados entre aplicações de saúde e como essa troca é implementada e gerenciada" (HL7, 2019). Para isso propõe o uso tecnologias web modernas disponibilizando os recursos em serviços REST ou GraphQL em formatos XML, JSON ou RDF.

2.2 Domain-Driven Design

A complexidade no desenvolvimento de software pode se tornar um problema, especialmente à medida em que os sistemas crescem e novas funcionalidades são acrescentadas. Evans (2003, p. xxi) explicita a questão da seguinte forma:

> Quando a complexidade foge ao controle, desenvolvedores não conseguem mais entender o software bem o suficiente para mudá-lo ou estendê-lo com

> facilidade e segurança. Por outro lado, um bom *design* pode criar oportunidades para a exploração dessas características complexas.

O DDD “é, ao mesmo tempo, uma forma de pensar e um conjunto de prioridades com o objetivo de acelerar projetos de software que têm que lidar com domínios complicados” (EVANS, 2003, p. xxi). Ainda de acordo com Evans (2003, p. xxi), ele trabalha sobre duas premissas, a de que a maioria dos projetos de software deve ter o foco primário no domínio e na lógica de domínio e a de que os projetos de domínios complexos devem se basear em modelos.

De acordo com Kapferer e Zimmermann (2020, p. 299), desde sua introdução, os padrões táticos do DDD como *entity*, *value object*, *aggregate* e *repository*, têm sido usados para modelar domínios de negócio complexos. Os padrões relevantes no contexto deste trabalho são *entity*, *value object* e *aggregate*.

Evans (2003, p. 91) caracteriza *entity* como um objeto definido primariamente por sua identidade e que tem continuidade por um ciclo de vida. Ou seja, um objeto cujos atributos podem mudar ao longo do tempo, mas que ainda assim mantém sua identidade. Por exemplo, uma pessoa, que pode alterar seu nome e ainda assim continuar sendo identificada como o mesmo objeto, é uma *entity*.

Um *value object*, por outro lado, refere-se a “um objeto que representa um aspecto descritivo do domínio sem uma identidade conceitual” (EVANS, 2003, p. 98). Dessa forma, "um *value object* é semanticamente imutável" (GHOSH, 2017, p. 7), ou seja, uma alteração em um atributo de um *value object* resulta em um novo objeto distinto. Evans (2003, p. 98) acrescenta que *value objects* podem usados como atributos na construção de *entities* ou outros *value objects*.

Evans (2003, p. 126) define um *aggregate* como "um agrupamento de objetos associados que são tratados como uma unidade para o propósito de alterações de dados. Cada *aggregate* tem uma raiz (*aggregate root*) e um limite." De acordo com Landre, Wesenberg e Rønneberg (2006, p. 985)

> *Aggregates* definem os limites de transações no modelo. Todos os objetos membro são acessíveis somente pela navegação a partir do *aggregate root*, dessa forma limitando as referências através de transações. A partir de um *aggregate* só é permitido manter referências a outros *aggregate roots*.

Kapferer e Zimmermann (2020, p. 300) afirmam ainda que “padrões estratégicos do DDD podem ser usados para decompor o domínio do problema de um sistema de software em múltiplos subdomínios e os chamados contextos delimitados.”

Um dos padrões estratégicos, e um conceito central do DDD no âmbito deste trabalho, refere-se aos *bounded contexts* (contextos delimitados). Braun, Bieniusa e Elberzhager (2021, p. 4) descreveram um *bounded context* como algo que “define uma fronteira explícita dentro da qual um modelo específico e unificado é válido.” De acordo com Evans (2003, p. 336), “um *bounded context* demarca a aplicabilidade de um modelo particular de forma que os membros de um time tenham um entendimento claro e compartilhado do que deve ser consistente e de como se relaciona com outros contextos.” Evans (2003, p. 335) complementa: “Você pode ter que integrar seu novo software com um sistema externo, sobre o qual seu time não tem controle. Essa situação é clara para todos como um contexto distinto onde o modelo sendo desenvolvido não se aplica.” Nesses casos se faz necessária uma camada de tradução entre os modelos, que é denominada camada anticorrupção (*anticorruption layer*). Para Kapferer e Zimmermann (2020, p. 302) a função da camada anticorrupção é a de proteger o *bounded context* de mudanças em outro contexto do qual aquele dependa.

2.3 Programação funcional

Allen e Moronuki (2016, p. 2) definem programação funcional como um paradigma de programação que se baseia em funções modeladas em funções matemáticas. “O programa principal é uma função que é definida em termos de outras funções e o principal método de computação é a aplicação de funções a argumentos.” (HU; HUGHES; WANG, 2015, p. 349) Como resume Bird (2014, p. 15),

> - Programação funcional é um método de construção de programas que enfatiza funções e sua aplicação ao invés de comandos e sua execução.
> - Programação funcional usa notação matemática simples que permite que problemas sejam descritos de maneira clara e concisa.
> - Programação funcional tem uma base matemática simples que suporta um raciocínio equacional sobre as propriedades de programas.

Um dos conceitos centrais da programação funcional refere-se às funções puras em que “o resultado da execução depende exclusivamente do valor dos argumentos e nenhum estado é modificado à medida que a execução prossegue” (HU; HUGHES; WANG, 2015, p. 351). Conforme Allen e Moronuki (2016, p. 3) esse conceito também pode ser compreendido como transparência referencial, o que

significa que uma função, dados os mesmos valores a avaliar, sempre retornará o mesmo resultado, da mesma forma que na matemática. Dessa definição de pureza decorrem outros conceitos centrais para a programação funcional, como o da ausência de efeitos colaterais e o da imutabilidade dos dados.

2.4 Trabalhos relacionados

Nas pesquisas realizadas não foram encontrados trabalhos relacionando diretamente o uso do padrão FHIR com as práticas do DDD ou da programação funcional. A pesquisa foi efetuada nos bancos de dados ACM Digital Library, IEEE Xplore, Scopus e Springer Link utilizando os termos “FHIR”, “*functional programming*” e “*domain-driven design*”, assim como combinações de apenas dois de cada um dos três termos.

Por outro lado, foram localizados trabalhos reconhecendo a dificuldade na utilização dos modelos de dados na área da saúde. Olivero et al. (2020) procurou aplicar modelagem UML a padrões da HL7 e Hong et al. (2017) propôs a criação de uma ferramenta para a visualização do modelo FHIR como forma de diminuir a curva de aprendizado. Stan e Miclea (2018), por sua vez, apresentaram um exemplo de implementação com FHIR.

Sobre a aplicação do DDD em outras áreas diversas também foram encontrados alguns trabalhos, assim como: Hammarström e Herzog (2016), que relataram a experiência da aplicação de DDD à engenharia de sistemas no campo da aeronáutica; Landre, Wesenberg e Rønneberg (2006), que descreveram a aplicação de DDD na indústria petrolífera; e Rademacher, Sorgalla e Sachweh (2018), os quais analisaram os desafios da aplicação de DDD no *design* de micro-serviços.

Apesar de o DDD ser originalmente proposto usando orientação a objetos, a aplicação da programação funcional em sua implementação vem sendo bastante discutida na indústria, diversas postagens em *blogs* trazem abordagens relacionando esses dois conceitos, porém com pouco aprofundamento. Em um âmbito mais amplo, destacam-se os livros *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#* (WLASCHIN, 2018) e *Functional and Reactive Domain Modeling* (GHOSH, 2017), os quais trazem abordagens da aplicação da programação funcional ao DDD, no caso com as linguagens F# e Scala.

A Tabela 1 traz uma comparação entre os trabalhos relacionados em relação aos três eixos centrais desse trabalho: FHIR, DDD e programação funcional (PF).

Tabela 1 – Comparação de trabalhos relacionados

	FHIR	DDD	PF
Olivero et al. (2020)	x		
Hong et al. (2017)	x		
Stan e Miclea (2018)	x		
Hammarström e Herzog (2016)		x	
Landre, Wesenberg e Rønneberg (2006)		x	
Rademacher, Sorgalla e Sachweh (2018)		x	
Wlaschin, 2018		x	x
Ghosh, 2017		x	x

Fonte: elaborado pelo autor

Assim, fica evidenciado que não foi localizado nenhum outro trabalho relacionando os três tópicos.

3 METODOLOGIA DE PESQUISA

A abordagem utilizada no desenvolvimento do trabalho foi a pesquisa exploratória, definida por Azevedo, Machado e Silva (2011, p. 45) como “estudos realizados quando se tem a necessidade de identificar, conhecer, levantar ou descobrir informações sobre um determinado tema”.

Foi desenvolvida uma prova de conceito de aplicação utilizando os conceitos estudados. Guimarães (2008, p. 32) define prova de conceito como

> uma técnica que permite demonstrar que uma determinada ideia é tecnicamente possível [...]. Provas de conceito permitem demonstrar a viabilidade de construção de um sistema de software utilizando um determinado estilo arquitetural.

Assim, a aplicação foi desenvolvida com um conjunto restrito de funcionalidades como forma de permitir um aprofundamento nos temas centrais do trabalho.

Após essa etapa foram realizadas entrevistas semiestruturadas, em plataforma virtual, seguindo um roteiro prévio com perguntas abertas e com a possibilidade de o pesquisador acrescentar, retirar ou reformular as perguntas de acordo com o

andamento da entrevista (Azevedo, Machado e Silva, 2011, p. 69). Uma vez que o projeto aborda um domínio relativamente restrito, a área de interoperabilidade de dados de saúde, optou-se por realizar as entrevistas apenas com desenvolvedores já familiarizados com o assunto, o que restringiu as opções na escolha dos entrevistados. Assim, foram realizadas entrevistas com seis desenvolvedores que atuam ou já atuaram no mesmo projeto que o autor, um sistema da área da saúde que utiliza dados no padrão FHIR.

Os entrevistados têm diferentes níveis de experiência, formação acadêmica e senioridade, e, também, diferentes graus de familiaridade com os conceitos de DDD, programação funcional e FHIR. Cinco deles atuam profissionalmente na mesma empresa de consultoria em desenvolvimento de software que o autor e um trabalha no cliente dessa empresa que desenvolve o projeto. Serão usados nomes fictícios, mantendo em sigilo o nome dos entrevistados como forma de preservar o anonimato dos mesmos. Foram entrevistados Jéssica, estagiária em desenvolvimento de software com nove meses de experiência, estudante do 4o semestre de Análise e Desenvolvimento de Sistemas; Mônica, desenvolvedora com 5 anos de experiência, formada em Análise e Desenvolvimento de Sistemas; Dario, desenvolvedor com 5 anos de experiência, formado em Análise e Desenvolvimento de Sistemas; Heitor, desenvolvedor com 4 anos de experiência, graduando em Engenharia de Software; Ernesto, arquiteto de software com 13 anos de experiência, com curso superior incompleto em Análise e Desenvolvimento de Sistemas; e Márcio, gerente de engenharia de software do cliente, com 10 anos de experiência, formado em Sistemas de Informação.

As entrevistas seguiram o seguinte roteiro: primeiramente foi realizada uma contextualização da proposta do trabalho, acompanhada de uma breve explanação sobre DDD, caso os entrevistados declarassem não ter conhecimento sobre o assunto. Em seguida a arquitetura do projeto foi apresentada em conjunto com o código fonte, detalhando a implementação dos conceitos abordados. Por fim foram realizadas perguntas com o objetivo de validar as hipóteses levantadas na concepção deste trabalho. As perguntas realizadas estão listadas na seção de discussão dos resultados.

Por fim foi feita uma avaliação qualitativa dos resultados obtidos nas entrevistas. De acordo com Azevedo, Machado e Silva (2011, p. 44), “o método

qualitativo abrange um paradigma fenomenológico – por exemplo, de interpretação e de descrição.”

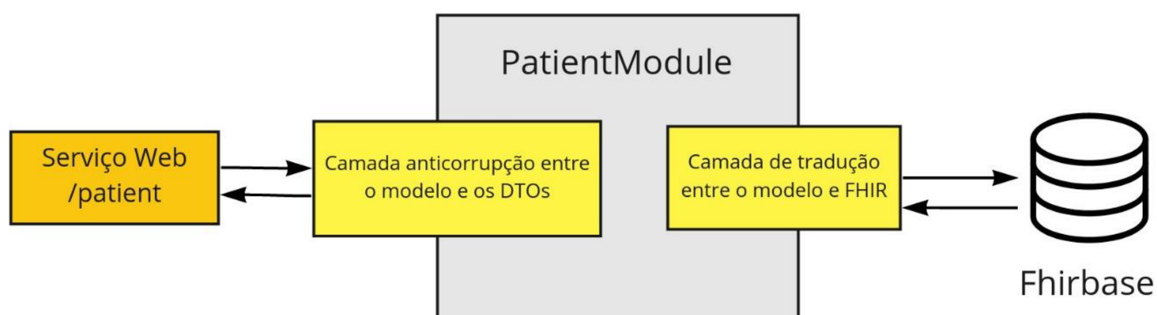
4 DESENVOLVIMENTO

O foco principal da implementação da prova de conceito foi criar um sistema com um contexto delimitado no domínio de dados de saúde. Assim foi desenvolvido o módulo *Patient*, com especial ênfase na camada anticorrupção, que garante a integridade do modelo ao tornar impossível a representação de estados inválidos, e na camada de tradução, que realiza a conversão entre os dados no modelo FHIR e o modelo da aplicação, facilitando a manipulação dos dados dentro do contexto delimitado. Foi criado um servidor web simples com as funções de criação, leitura, edição e exclusão de dados de pacientes com persistência dos dados em um banco de dados no padrão FHIR. Assim pode-se demonstrar tanto o recebimento de dados “crus” vindos do serviço na forma de DTOs, como a manipulação de dados FHIR armazenados no banco de dados. O código fonte da aplicação encontra-se disponível em https://github.com/gbs149/TCC_project.

4.1 Arquitetura

A arquitetura geral do projeto, ilustrada na Figura 1, consiste em um serviço web com uma API Rest com rotas para as operações básicas de criação, leitura, edição e exclusão:

Figura 1– Arquitetura do projeto



Fonte: elaborado pelo autor

A camada do serviço se comunica com o contexto delimitado *Patient* através da camada anticorrupção, que faz a validação e sanitização dos dados recebidos. O contexto delimitado *Patient* se comunica com o banco de dados Fhirbase (FHIRBASE, 2021) através de uma camada de tradução, responsável pela conversão entre o modelo da aplicação e o modelo FHIR. Dessa forma, de um lado a camada anticorrupção garante a integridade do modelo ao receber dados do mundo exterior e, do outro, a camada de tradução proporciona uma abstração para a manipulação de dados armazenados no padrão FHIR.

4.2 Ferramentas utilizadas

A linguagem de programação escolhida para a implementação foi o TypeScript por se tratar de uma linguagem fortemente tipada, com múltiplos paradigmas e com grande aceitação no mercado. Optou-se por utilizar a biblioteca de programação funcional *fp-ts*, que oferece uma grande gama de estruturas de dados, classes de tipos e funções auxiliares similares às existentes em linguagens funcionais como Haskell e Scala (FP-TS, 2021), uma vez que o foco deste trabalho não é a implementação de tais construções. Pelo mesmo motivo, foi usada uma predefinição de tipos FHIR para TypeScript (@TYPES/FHIR, 2021), que proporciona uma implementação dos tipos do padrão FHIR, garantindo a correta utilização do padrão.

Para a camada de persistência dos dados foi utilizado o banco de dados Fhirbase (FHIRBASE, 2021), que é uma extensão sobre o banco de dados PostgreSQL para o armazenamento e manipulação de dados no padrão FHIR. Foi ainda utilizado o *framework* web Koa (KOA, 2021), para NodeJS, além de bibliotecas de validação de datas, números telefônicos e números de CPF. Para a criação e execução de testes unitários foi utilizado o *framework* de testes Jest (JEST, 2021).

4.3 Definição do modelo

O padrão FHIR é extensível e customizável, não obrigando a utilização de um modelo específico e dando liberdade para cada aplicação definir seu próprio modelo de acordo com suas necessidades, portanto, o modelo de paciente implementado foi definido arbitrariamente como forma de exemplificar os conceitos explorados. O nome do tipo do *aggregate* que representa um paciente foi definido como *PatientModel*,

como forma de diferenciar do tipo *Patient*, usado no padrão FHIR, conforme ilustrado na Figura 2.

Figura 2 – Definição do tipo PatientModel

```
export interface PatientModel {
  id?: Id;
  active: boolean;
  birthdate: Birthdate;
  cpf: CPF;
  email: EmailContact;
  gender: GenderType;
  name: Name;
  phone: Option<PhoneContact>;
  currentAddress: Address;
}
```

Fonte: elaborado pelo autor

Pode-se notar que cada um dos campos que compõem o *PatientModel* foi definido como um *value object* cuja criação se dá por uma função que realiza a validação dos dados de entrada, portanto, a criação de um *aggregate* do tipo *PatientModel* resultará necessariamente ou em um objeto válido ou em uma lista de erros de validação. Assim pode-se garantir a integridade do modelo a nível do sistema de tipos de forma que um estado inválido seja impossível de ser representado. Como forma de simplificar a implementação da prova de conceito, os erros foram representados por valores do tipo *string*, porém, o modelo poderia ser estendido de forma que os diferentes tipos de erros também fossem representados por tipos específicos.

Uma das dificuldades encontradas refere-se ao fato de que o sistema de tipos de TypeScript é estrutural, em contraste com sistemas de tipos nominais, como em Java ou C#, o que significa que o compilador identifica os tipos baseado apenas nos seus membros e, portanto, permite certas operações sobre as quais não se pode decidir quanto à segurança em tempo de compilação (MICROSOFT, 2021). Pode-se, por exemplo, criar *aliases* de tipos da seguinte forma, *type Name = string*, porém isso permite usar de maneira intercambiável o tipo *string* e seu *alias Name*. No modelo da aplicação os tipos Id, CPF e *GenderType* têm seu valor representado pelo tipo *string* e a utilização de um alias na definição desses tipos teria permitido a criação de um

modelo inválido com qualquer valor do tipo *string* no lugar dessas propriedades. A solução encontrada, exemplificada na Figura 3, foi a utilização conjunta de predicados de tipos (B) com o conceito de *tagged intersection* para reproduzir as características de um sistema de tipos nominal. Assim, ao expor apenas o tipo CPF (A) e a função *makeCPF* (C), pode-se restringir a instanciação do tipo à utilização da função de criação que faz sua validação.

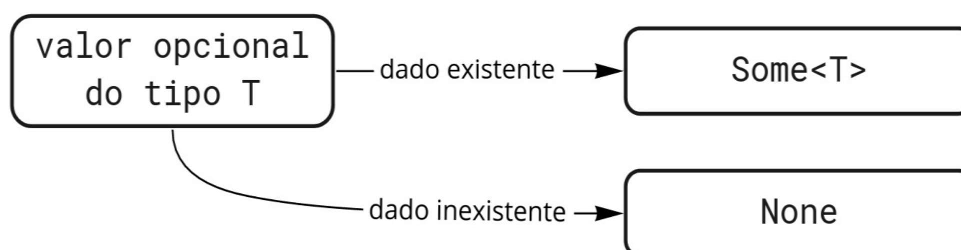
Figura 3 – Definição do tipo CPF

```
interface CPFBrand {
  readonly CPF: unique symbol;
}

export type CPF = string & CPFBrand; // A

const isCPF = (s: string): s is CPF => isValidCpf(s); // B

export const makeCPF = (s: string): ValidationResult<CPF> =>
  isCPF(s) ? right(s) : left(["Invalid CPF"]); // C
```

Fonte: elaborado pelo autor

Para a modelagem de propriedades opcionais na programação funcional, ao invés de se usar um tipo que pode assumir o valor *null* ou *undefined* é utilizada a estrutura *Option*, ilustrada na Figura 4.

Figura 4 – Diagrama tipo Option



Fonte: elaborado pelo autor

A estrutura da Figura 4, que possibilita a composição de funções de forma a evitar erros de acesso a propriedades de um valor *null* ou *undefined*, foi utilizada na

propriedade *phone* do *aggregate PatientModel*. A definição do tipo *Option* é ilustrada na Figura 5.

Figura 5 – Definição do tipo *Option*

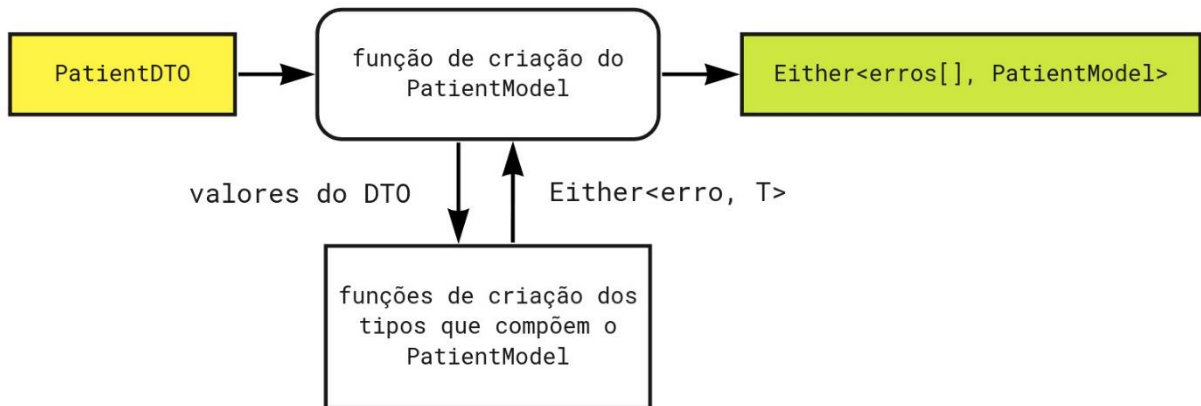
```
type Option<Tipo> = None | Some<Tipo>;
```

Fonte: elaborado pelo autor

4.4 Camada anticorrupção

A implementação da camada anticorrupção se dá na própria função de construção do tipo *PatientModel,* no qual o valor de cada campo é validado, conforme ilustrado na Figura 6. O sistema de tipos é utilizado para impedir que um tipo seja criado sem ser validado, assim a única forma de instanciar os tipos associados aos valores é através da função que realiza a validação, retornando um resultado de erro em caso de falha.

Figura 6 – Validação na criação do modelo

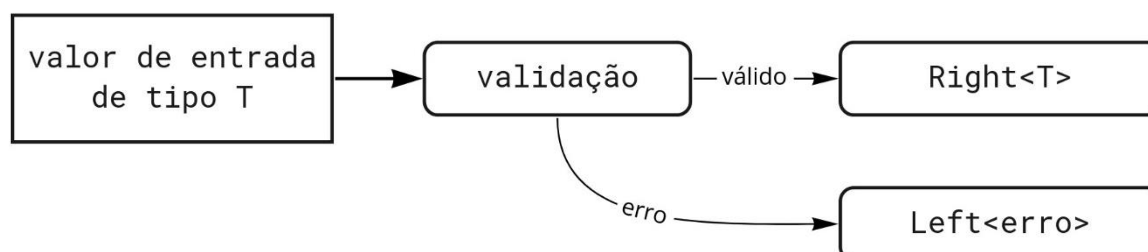


Fonte: elaborado pelo autor

A Figura 6 demonstra a utilização do tipo *Either* para a representação do resultado de funções que podem retornar um erro. Na programação funcional evita-se ao máximo a utilização de efeitos colaterais, priorizando a utilização de funções puras que garantem a transparência referencial, e, por esse motivo, não são lançadas exceções. Como forma de atender a essa regra, na validação é utilizada a estrutura

Either, ilustrada na Figura 7, que possibilita a representação de dois valores excludentes, ou um erro ou um objeto válido.

Figura 7 – Diagrama da validação com o tipo Either



Fonte: elaborado pelo autor

Para cada *value object* do tipo T, a sua função de criação retorna um tipo *Either<erro, T>*, abstraído no projeto na forma de um tipo *ValidationResult<T>*, conforme mostra a Figura 8.

Figura 8 – Definição dos tipos Either e ValidationResult

```
type Either<Erro, T> = Left<Erro> | Right<T>;

type ValidationResult<T> = Either<NonEmptyArray<string>, T>;
```

Fonte: elaborado pelo autor

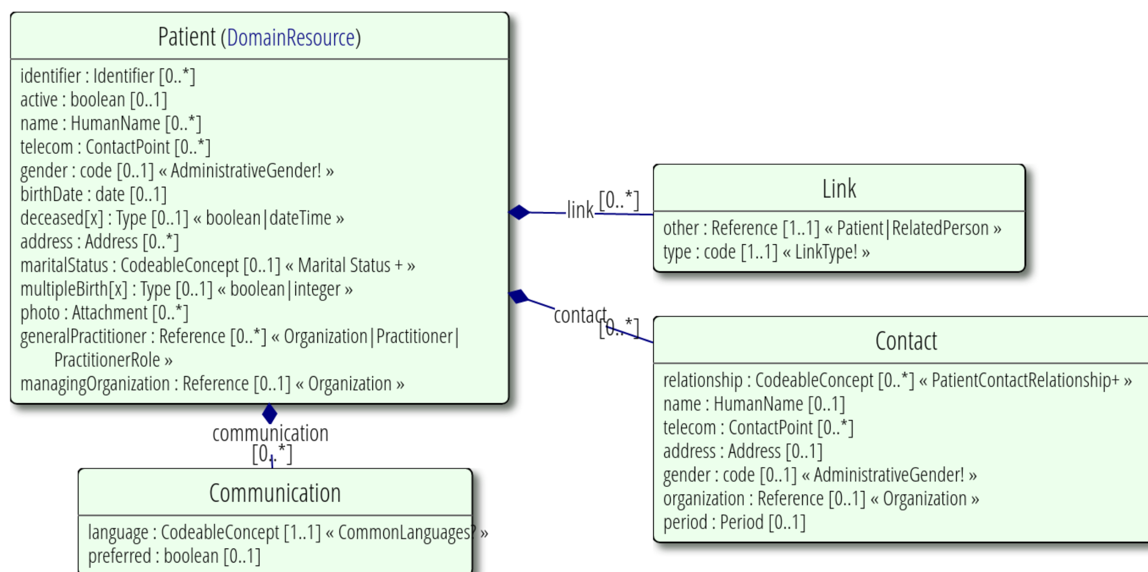
A criação do objeto raiz *PatientModel* faz a agregação dos retornos da criação dos *value objects* resultando em um tipo *ValidationResult<PatientModel>* que apresenta ou um modelo válido, ou uma lista com todos os erros de validação encontrados.

4.5 Tradução de dados no padrão FHIR

Um dos elementos centrais deste projeto foi a criação de uma camada de tradução entre o modelo da aplicação e o padrão FHIR. Optou-se por trabalhar com o recurso *Patient* por ser uma das entidades centrais em um sistema de saúde e por apresentar um grau de complexidade que demonstra com clareza a dificuldade de manipulação de dados FHIR e a necessidade da tradução para um modelo específico

para as necessidades de negócio de uma aplicação. No padrão FHIR, o recurso *Patient* é estabelecido para representar "informações demográficas ou administrativas sobre um indivíduo ou animal recebendo cuidados ou outros serviços relacionados à saúde." (HL7, 2019). A Figura 9 apresenta o diagrama UML do recurso *Patient*.

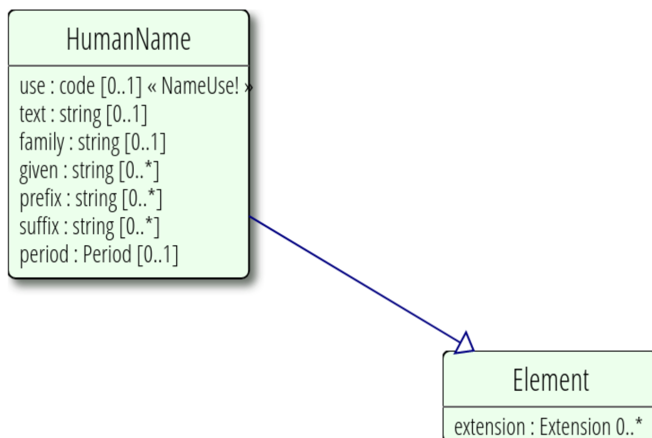
Figura 9 – Diagrama UML do recurso *Patient*



Fonte: HL7 (2019)

Como demostra a Figura 9, a estrutura é definida como um objeto composto por campos de diferentes tipos de dados, também definidos em FHIR, alguns dos quais apresentam uma construção que torna trabalhoso seu acesso e manipulação. O campo *name*, por exemplo, tem cardinalidade 0..* e é representado por objetos do tipo *HumanName* (Figura 10). Este tipo, por sua vez, é composto por campos *use*, *text*, *family*, *given*, *prefix* e *sufix*, do tipo *string,* e *period*, do tipo *Period*. Todos os campos são opcionais e os campos *given*, *prefix* e *sufix* têm cardinalidade 0..* e são, portanto, representados por listas.

Figura 10 – Diagrama UML do tipo *HumanName*



Fonte: HL7 (2019)

Conforme demonstrado na Figura 10, a modelagem possibilita que um paciente possa possuir vários nomes registrados para diferentes usos (oficial, social, usual, temporário, apelido, anônimo, antigo, solteiro), além de cada nome poder ter um período de uso, que pode ser usado em caso de troca de nome pelo paciente. A Figura 11 traz um exemplo da representação de um objeto do tipo *HumanName* no formato JSON.

Figura 11 – Exemplo do tipo *HumanName* no formato JSON

```
{
  "name": [
    {
      "use": "official",
      "text": "Prof. Dr. Carlos Roberto Santos",
      "family": "Santos",
      "given": ["Carlos", "Roberto"],
      "prefix": ["Prof.", "Dr."],
      "period": {
        "start": "1984-07-23T00:00:00Z"
      }
    }
  ]
}
```

Fonte: elaborado pelo autor

Conforme mostra a Figura 11, desta forma o acesso ao primeiro nome (campo *given*), do nome oficial e corrente de um *Patient* poderia se dar conforme ilustrado na Figura 12.

Figura 12 – Exemplo de acesso ao primeiro nome de um *Patient*

```
patient.name
  ?.find((n: HumanName) => n.use === "official"
    && n.period?.end === undefined)
  ?.given?.join(" ");
```

Fonte: elaborado pelo autor

Note-se a necessidade de concatenar as *strings* presentes no campo *given* por se tratar de uma lista e do uso da sintaxe “?.” para acesso opcional a propriedades cujos valores podem assumir o valor *undefined*. Este exemplo demonstra que isso se torna pouco prático de repetir a cada vez que se quer utilizar o dado e sugere a criação de uma abstração deste acesso para uma função. Além disso, o próprio resultado pode ser um valor *undefined*, o que acarretaria ainda mais complexidade a ser tratada no restante da aplicação. Por outro lado, o modelo da aplicação (*PatientModel*), estabelece uma cardinalidade 1..1 para o campo *name*, do tipo Name (Figura 13), o que significa que um paciente deve obrigatoriamente ter um e apenas um nome. Vale ressaltar, que mesmo que a aplicação possa permitir a troca de nome e o recurso FHIR persistido reflita isso, o modelo apresentará somente o nome corrente.

Figura 13 – Definição do tipo *Name*

```
export interface Name {
  readonly first: string;
  readonly last: string;
}
```

Fonte: elaborado pelo autor

Portanto, como mostra a Figura 13, o acesso ao primeiro nome no modelo da aplicação é bastante simplificado em relação ao nome em FHIR, conforme ilustrado na Figura 14.

Figura 14 – Acesso ao primeiro nome de um *PatientModel*

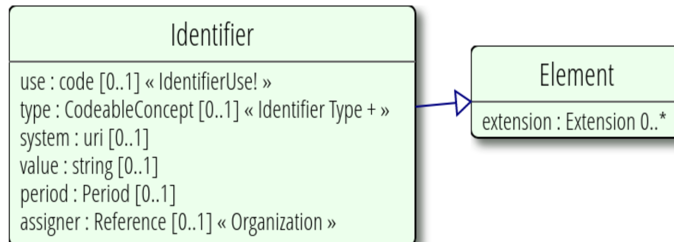
```
patient.name.first;
```

Fonte: elaborado pelo autor

Uma vez que a validação dos tipos no momento da sua criação garante ainda que um *PatientModel* terá necessariamente um nome válido ao ser acessado, pode-se dispensar o uso da sintaxe “?.” para acesso opcional de propriedades, que poderia resultar no valor *undefined*.

Outro exemplo que se pode citar é a definição de um documento como o CPF em FHIR. Uma vez que o FHIR é um padrão internacional, é natural que não exista um campo específico para identificadores usados em diferentes jurisdições como é o caso do CPF. A estrutura usada nestes casos é o tipo *Identifier*, cujo diagrama UML é ilustrado na Figura 15, que representa identificadores de forma genérica.

Figura 15 – Diagrama UML do tipo *Identifier* – FHIR



Fonte: HL7 (2019)

Assim, para a representação de um CPF em FHIR pode-se usar essa estrutura com a *string* "http://rnds.saude.gov.br/fhir/r4/NamingSystem/cpf" no campo *system*, que informa o *namespace* ao qual pertence o identificador, e a *string* com o número do documento no campo *value*, conforme exemplificado, em formato JSON, na Figura 16.

Figura 16 – Exemplo de *Identifier* no formato JSON

```
{
  "identifier": [
    {
      "system": "http://rnds.saude.gov.br/fhir/r4/NamingSystem/cpf",
      "value": "12345678910"
    }
  ]
}
```

Fonte: elaborado pelo autor

O acesso ao número de um CPF em FHIR, portanto, teria que ocorrer como demonstrado na Figura 17, destacando que, assim como no acesso ao nome, é necessário o uso da sintaxe “?.” e que o resultado também pode ser o valor *undefined*.

Figura 17 – Acesso ao CPF de um *Patient*

```
patient.identifier?.find(
  (i: Identifier) =>
    i.system === "http://rnds.saude.gov.br/fhir/r4/NamingSystem/cpf"
)?.value;
```

Fonte: elaborado pelo autor

No modelo da aplicação, o acesso ao CPF seria simplesmente o acesso a uma propriedade do objeto, como ilustrado na Figura 18.

Figura 18 – Acesso ao CPF de um *PatientModel*

```
patient.cpf;
```

Fonte: elaborado pelo autor

Uma das vantagens da utilização do paradigma funcional é a possibilidade de criação de funções pequenas e simples que podem ser compostas em funções maiores, gerando abstrações complexas. Foram identificados casos que poderiam se beneficiar da definição de funções auxiliares, como por exemplo *findBySystem*, ilustrada na Figura 19, que permite localizar em listas elementos com a propriedade *system* desejada.

Figura 19 – Definição da função *findBySystem*

```
interface WithSystem {
  system?: string;
}

export const findBySystem =
  (desiredSystem: string) =>
  <T extends WithSystem>(ts: T[] = []): Option<T> =>
    findFirst<T>((t) => t.system === desiredSystem)(ts);
```

Fonte: elaborado pelo autor

Dada uma lista de objetos de tipo T com a propriedade *system*, obtém-se o primeiro que apresente o *system* desejado; como é possível que não haja nenhum objeto que atenda à condição, o tipo retornado é um *Option<T>*. Uma vez que o conceito de *system* é amplamente utilizado em FHIR, com inúmeros tipos de dados definidos com essa propriedade, essa função pode ser usada na para compor diversas funções para extrair dados, como por exemplo, o CPF de um *Patient* (Figura 20).

Figura 20 – Definição da função *getCpf*

```
export const getCpf = ({ identifier }: Patient): Option<string> =>
  pipe(
    findBySystem(CPF_NAMING_SYSTEM)(identifier),
    map((i: Identifier) => i.value)
  );
```

Fonte: elaborado pelo autor

Assim como *system* é um conceito transversal no sistema FHIR, *Period*, também é um tipo de dados utilizado na composição de diversos outros tipos e, assim, funções genéricas usando esse tipo podem ser aplicadas na composição de funções para o acesso de inúmeras propriedades, como a função *findCurrent*, cuja definição é apresentada na Figura 21, que filtra uma lista de objetos, retornando apenas aqueles que não tenham a propriedade *period.end* definida.

Figura 21 – Definição da função *findCurrent*

```
interface HasPeriod {
  period?: Period;
}

export const findCurrent = <T extends HasPeriod>(ts: T[] = []): T[] =>
  Array.filter<T>((t: T) => t.period?.end === undefined)(ts);
```

Fonte: elaborado pelo autor

Como exemplo das possibilidades de uso da composição de funções como forma de criar abstrações maiores, pode-se examinar as funções *getEmail* e *getPhone*, usadas para acessar os dados de e-mail e telefone de um *Patient*, ilustrada na Figura 22.

Figura 22 – Definição das funções *getEmail* e *getPhone*

```
const getTelecom =
  (system: string) =>
  (patient: Patient): Option<ContactDTO> =>
    pipe(
      findCurrent(patient.telecom),
      findBySystem(system),
      map(toContactDTO)
    );


export const getEmail = getTelecom(EMAIL_SYSTEM);
export const getPhone = getTelecom(PHONE_SYSTEM);
```

Fonte: elaborado pelo autor

Pode-se ver que ambas as funções *findCurrent* e *findBySystem* são usadas em sua composição, o que, aliado à aplicação parcial do parâmetro system, permite a criação de um código declarativo e enxuto.

4.6 Acesso aos dados e Webservice

Apesar de as camadas de serviço web e de acesso aos dados não serem o foco principal deste trabalho, elas serviram para validar a utilização do paradigma funcional de ponta a ponta, ressaltando as características composicionais do modelo.

Nestas camadas também foram utilizadas as construções funcionais disponíveis na biblioteca FP-TS, com destaque para *TaskEither*, utilizada em operações assíncronas que podem resultar em erro, para acesso ao banco de dados.

O módulo *PatientServiçe* estabelece as rotas expostas em um serviço REST para criação, leitura, edição e exclusão de recursos e tem conhecimento apenas dos dados no formato definido em *PatientDTO*. A validação dos dados e consequente criação do modelo é delegada ao módulo PatientController que realiza a coordenação entre o serviço web e o acesso aos dados. Também cabe ao *PatientController* realizar a tradução dos dados para o modelo FHIR, de forma que a camada de persistência, por sua vez, tenha conhecimento apenas de dados nesse padrão, estabelecendo de forma clara as fronteiras do módulo *Patient*.

Cabe ressaltar, que a abstração criada para acesso aos dados foi definida de maneira genérica, podendo ser reutilizada em outros módulos com a expansão da aplicação.

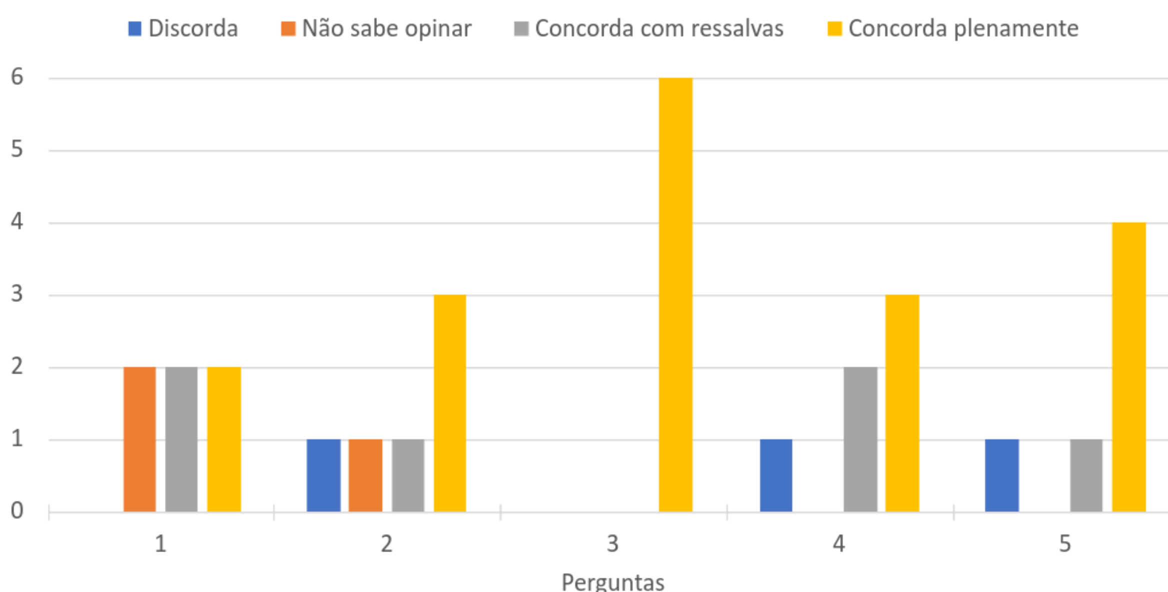
5 DISCUSSÃO DOS RESULTADOS

Analisando as entrevistas realizadas pode-se perceber que o trabalho teve uma boa aceitação por parte dos entrevistados. O Gráfico 1 exibe a distribuição do sentimento das respostas para cada pergunta, com predominância de respostas positivas ou positivas com ressalvas. Mesmo as restrições colocadas não caracterizam uma discordância total em relação à proposta.

As perguntas realizadas foram as seguintes:

1. A utilização conjunta de DDD e programação funcional facilita o desenvolvimento e manutenção da aplicação?
2. A utilização de DDD e programação funcional pode ajudar a reduzir o número de bugs introduzidos no software?
3. A manipulação de dados no padrão FHIR é facilitada pela tradução para um modelo de aplicação?
4. A utilização de conceitos de programação funcional, como *Either* e *Option*, reduz a complexidade de desenvolvimento?
5. A utilização da camada anticorrupção (validação dos dados) torna a aplicação mais segura?

Gráfico 1 – Distribuição das respostas



Fonte: elaborado pelo autor

Para Ernesto e Márcio, os dois entrevistados mais experientes, também foi realizada a seguinte pergunta: "A aplicação destes conceitos aumenta muito a curva de aprendizado para um time de desenvolvimento formado por desenvolvedores júnior?".

Do ponto de vista do autor, ao longo do desenvolvimento do projeto, algumas hipóteses foram se confirmando ao passo que outras dúvidas surgiam. Por um lado, ficou evidente que a manipulação de dados se torna extremamente mais simples após a conversão do padrão FHIR para o modelo de domínio da aplicação. A abstração de alguns conceitos e regras de negócio específicos da aplicação permitiam a utilização de um código mais simples e enxuto. Por outro lado, a utilização das construções funcionais providas pela biblioteca fp-ts trouxe desafios para alguém que não havia utilizado na prática esse tipo de estrutura. A falta de familiaridade com a programação funcional fortemente tipada se mostrou uma barreira a ser transposta e a quantidade reduzida de documentação e recursos de aprendizado sobre o assunto dificultou o avanço em alguns momentos. Essas impressões do autor foram, em grande medida, confirmadas nas entrevistas.

Os entrevistados foram unânimes ao concordar que o acesso e manipulação de dados ficam mais fáceis no modelo de domínio em relação aos dados no padrão

FHIR. Nesse sentido, o entrevistado Márcio destacou a necessidade da aplicação das regras de negócio com restrições na camada de tradução como forma de simplificar o modelo e o entrevistado Ernesto ressaltou que essa simplificação se dá ao permitir abstrair elementos que não são relevantes ao domínio da aplicação. Ernesto referiu-se, ainda ao domínio da aplicação como um subdomínio do domínio médico em geral, o que permitiu realizar essa abstração, impedindo que a complexidade do modelo FHIR fosse trazida para as outras camadas da aplicação. O entrevistado Heitor mencionou que, apesar de a documentação do FHIR ser muito boa, a manipulação dos dados em geral se torna confusa.

Quanto ao uso da programação funcional, as opiniões foram mais divididas. Heitor foi o mais reticente nesse item, destacando as dificuldades que teve nas oportunidades em que trabalhou com programação funcional e afirmando que há uma curva de aprendizado íngreme quando se começa a aplicar esses conceitos. Dario, Ernesto e Márcio concordam que há uma curva de aprendizado, porém acreditam que, após esse período de adaptação, o desenvolvimento e, principalmente, a manutenção sejam simplificados. Ernesto afirmou que o tempo investido no aprendizado desses conceitos é um investimento que se paga à medida que o projeto cresce. Márcio ressaltou que, do ponto de vista prático, não é comum encontrar desenvolvedores no mercado familiarizados com programação funcional e que, em projetos com pressão por cumprimento de prazos, pode ser difícil justificar o tempo investido. Márcio destacou ainda a importância de o projeto contar com algum desenvolvedor sênior com experiência nesses idiomas para 'limitar danos' e evitar que desenvolvedores que ainda não têm domínio de programação funcional acabem introduzindo defeitos pelo uso incorreto de suas construções.

Curiosamente, os entrevistados com menos experiência com programação funcional foram os que manifestaram opinião mais positiva quanto a facilidade de desenvolvimento e manutenção, porém é possível que essa opinião seja um reflexo justamente da falta de familiaridade com as complexidades da composição de tipos monádicos, como *Option* e *Either*, uma vez que o tempo da apresentação do projeto foi reduzido e não foi possível detalhar as dificuldades encontradas no desenvolvimento.

Outra questão em que as opiniões foram divididas refere-se a se o uso de DDD e programação funcional ajudariam na redução da quantidade de defeitos (*bugs*) introduzidos no software. Os entrevistados Ernesto, Mônica e Jéssica entenderam

como vantajoso o uso da programação funcional. Jéssica mencionou que erros relacionados a valores *undefined* são eliminados com o uso do tipo *Option*. Mônica e Ernesto destacaram o tratamento de erros com o tipo *Either*, que reduz a dificuldade no entendimento do fluxo da aplicação. Ernesto ainda levantou mais duas questões, a da imutabilidade de dados, aplicada no paradigma funcional, que reduz defeitos principalmente em aplicações com concorrência, e o fato de poder ter componentes bem testados isoladamente que são compostos em componentes maiores, o que permite o reuso de código e reduz a possibilidade de introdução de defeitos.

Por outro lado, Dario, Heitor e Márcio afirmaram não acreditar que apenas o uso de DDD e programação funcional seja suficiente para reduzir defeitos. Heitor e Márcio destacaram que a cobertura por testes é mais importante para determinar o não surgimento de defeitos, embora Márcio acredite que esse estilo possa facilitar a criação de testes concisos e, portanto, a redução dos defeitos, acrescentando que nesse caso existe uma correlação, mas, não necessariamente, uma relação de causalidade.

Outro aspecto que todos destacaram relaciona-se ao código, o qual consideraram bem organizado e fácil de compreender, o que o autor acredita ser um reflexo da utilização dos conceitos de DDD na estruturação do módulo *Patient* em suas diferentes camadas.

6 CONCLUSÃO

Como forma de concluir o presente trabalho, retoma-se a questão de pesquisa, a qual motivou o desenvolvimento do projeto, sendo ela: é possível aprimorar o processo de desenvolvimento de sistemas na área da saúde com o padrão FHIR através da utilização de conceitos de DDD e programação funcional? Após a análise do resultado da execução da prova de conceito e do retorno obtido através das entrevistas realizadas, pode-se concluir que a resposta é afirmativa, pois foi possível demonstrar que o DDD é uma abordagem válida, especialmente ao permitir a criação de um modelo de domínio simplificado para a aplicação, abstraindo muitas das complexidades inerentes ao modelo FHIR. A utilização da programação funcional também trouxe benefícios para a compreensão do código e redução de defeitos, com a ressalva, nesse caso, de que se trata de um paradigma ainda pouco conhecido e

estudado que exige um certo grau de adaptação por grande parte dos desenvolvedores.

Com relação aos objetivos específicos, pode-se identificar alguns dos padrões de código prescritos pelo DDD que auxiliaram nesse contexto. Utilizou-se o padrão estratégico *bounded context*, em conjunto com o *anticorruption layer*, para estabelecer um contexto em que o modelo de domínio *Patient* foi definido com garantias de validade e integridade. Para a definição desse modelo, os padrões *aggregate*, *entity* e *value object* foram aplicados. É importante ressaltar que a prática do DDD vai muito além da mera aplicação de padrões de código, abarcando todo o ciclo de vida do sistema, envolvendo desde a coleta de requisitos até a definição da arquitetura geral do projeto e o gerenciamento dos times de desenvolvimento, o que ficou além do escopo do trabalho.

Outro objetivo específico foi verificar de que forma o paradigma de programação funcional pode facilitar a implementação desses padrões e aqui o resultado também foi positivo, com o uso de padrões de programação funcional como *Either* e *Option*, bem como a natureza composicional deste estilo de programação, permitiram a escrita de um código conciso, de fácil compreensão e com garantias de correção proporcionadas pelo sistema de tipos da linguagem TypeScript.

A implementação da prova de conceito foi realizada com sucesso, utilizando o paradigma funcional de ponta a ponta no sistema e aplicando os padrões do DDD citados anteriormente. O resultado foi um código organizado e fácil de testar, manter e estender.

As entrevistas foram muito ricas e validaram a maior parte das hipóteses deste trabalho. A proposta do projeto foi bem recebida e a implementação da prova de conceito obteve a aprovação dos entrevistados. Foram levantados pontos de atenção quanto à utilização da programação funcional na prática, em especial com relação a desenvolvedores sem experiência com este paradigma, confirmando a impressão que o autor teve durante o desenvolvimento.

Um desdobramento futuro que se vislumbra como continuação deste trabalho é o desenvolvimento de uma biblioteca para uso em conjunto com o padrão FHIR com a definição de funções genéricas para acesso e manipulação dos dados de forma adaptável a diferentes casos de uso. Outro ponto de evolução que pode ser abordado é o aprofundamento no estudo das práticas do DDD com aplicação ao domínio da

saúde, indo além da aplicação de padrões de código e envolvendo as etapas de levantamento de requisitos e documentação do projeto.

Abstract: One of the main difficulties faced in the digital transformation in health care relates to data interoperability. FHIR has appeared as a promising standard candidate to allow sharing data between health organizations. However, FHIR presents a degree of complexity that results in a steep learning curve for developers using it. Therefore, the objective of this work has been to explore the possibilities of applying Domain-Driven Design, in tandem with functional programming, as a way of dealing with this complexity. A proof-of-concept system was developed to validate this approach. The main contribution of this work has been in identifying the need to make use of a domain specific model for the application, which abstracts the complexities of the FHIR standard.
Keywords: Domain-Driven Design; functional programming; FHIR.

REFERÊNCIAS

ALLEN, C.; MORONUKI, J. **Haskell Programming from First Principles**. [*S. l.*] Lore Pub, 2016.

AZEVEDO, Debora; MACHADO, Lisiane; SILVA, Lisiane Vasconcellos da (org.); **Métodos e procedimentos de pesquisa**: do projeto ao relatório final. São Leopoldo: Editora Unisinos, 2011.

BIRD, R. **Thinking functionally with Haskell**. Cambridge, UK: Cambridge University Press, 2014.

BRAUN, S.; BIENIUSA, A.; ELBERZHAGER, F. Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems. In: WORKSHOP ON PRINCIPLES AND PRACTICE OF CONSISTENCY FOR DISTRIBUTED DATA, 8., 2021, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2021. (PaPoC ’21).

FP-TS – Typed functional programming in TypeScript. [*S. l.*], [2021?]. Disponível em: https://gcanti.github.io/fp-ts/. Acesso em: 06 nov. 2021.

Evans, E. J. **Domain-driven design**: tackling complexity in the heart of software. Boston, MA, USA: Addison-Wesley Professional, 2003.

FHIRBASE – Open Source Database for HL7 FHIR. *In*: Health Samurai. Los Angeles, CA, USA, [2021?]. Disponível em: https://www.health-samurai.io/fhirbase. Acesso em: 14 nov. 2021.

GHOSH, D. **Functional and reactive domain modeling**. Shelter Island, NY, USA: Manning Publications Co., 2017.

GUIMARÃES, Júlio Henrique dos Nogueira e Oliveira. **Método para manutenção de sistema de software utilizando técnicas arquiteturais**. 2008. Dissertação

(Mestrado em Sistemas Digitais) - Escola Politécnica, University of São Paulo, São Paulo, 2008. doi:10.11606/D.3.2008.tde-29012009-134316.

HAMMARSTRÖM, P.; HERZOG, E. Experience from integrating Domain Driven Software System Design into a Systems Engineering Organization. **INCOSE International Symposium**, [*S. l.*], v. 26, n. 1, p. 1192–1203, 2016.

HIMSS. **HIMSS dictionary of health information technology terms, acronyms, and organizations**. Boca Raton, FL, USA: CRC Press, 2019.

Health Level Seven International (HL7). Health Level Seven International - Homepage. [*S. l.*], c2021. Disponível em: http://www.hl7.org/. Acesso em: 10 jun. 2021.

Health Level Seven International (HL7). FHIR v4.0.1. [*S. l.*], c2019. Disponível em: http://hl7.org/fhir/. Acesso em: 10 jun. 2021.

HONG, N.; WANG, K.; YAO, L.; JIANG, G. Visual FHIR: an interactive browser to navigate hl7 fhir specification. In: IEEE INTERNATIONAL CONFERENCE ON HEALTHCARE INFORMATICS (ICHI), 2017., 2017. **Anais...** IEEE, 2017. p. 26–30.

HU, Z.; HUGHES, J.; WANG, M. How functional programming mattered. **National Science Review**, [*S. l.*], v. 2, n. 3, p. 349–370, 2015.

JEST – delightful JavaScript Testing. [*S. l.*], c2021. Disponível em: https://jestjs.io/. Acesso em: 14 nov. 2021.

KAPFERER, S.; ZIMMERMANN, O. **Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling**. [*S. l.*]: SciTePress, 2020. 299-306 p.

KOA – next generation web framework for node.js. [*S. l.*], [2021?]. Disponível em: https://koajs.com/. Acesso em: 14 nov. 2021.

LANDRE, E.; WESENBERG, H.; RØNNEBERG, H. Agile Enterprise Software Development Using Domain-Driven Design and Test First. In: COMPANION TO THE 22ND ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS AND APPLICATIONS COMPANION, 2007, New York, NY, USA. **Anais...** Association for Computing Machinery, 2007. p. 983-993.

LANDRE, E.; WESENBERG, H.; RØNNEBERG, H. Architectural Improvement by Use of Strategic Level Domain-Driven Design. In: COMPANION TO THE 21ST ACM SIGPLAN SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 2006, New York, NY, USA. **Anais...** Association for Computing Machinery, 2006. p. 809–814.

MICROSOFT. The TypeScript Handbook. [*S. l.*], c2021. Disponível em: https://www.typescriptlang.org/docs/handbook/. Acesso em: 18 out. 2021.

MUKHERJEE, S.; RAY, I.; RAY, I.; SHIRAZI, H.; ONG, T.; KAHN, M. G. Attribute Based Access Control for Healthcare Resources. In: ACM WORKSHOP ON

ATTRIBUTE-BASED ACCESS CONTROL, 2., 2017, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2017. p. 29–40. (ABAC ’17).

OLIVERO, M. A.; DOMÍNGUEZ-MAYO, F.; PARRA-CALDERÓN, C. L.; ESCALONA, M.; MARTINEZ-GARCÍA, A. Facilitating the design of HL7 domain models through a model-driven solution. **BMC Medical Informatics and Decision Making**, [*S. l.*], v. 20, p. 1–18, 2020.

RADEMACHER, F.; SORGALLA, J.; SACHWEH, S. Challenges of Domain-Driven Microservice Design: a model-driven perspective. **IEEE Software**, [*S. l.*] v. 35, n. 3, p. 36–43, 2018.

STAN, O.; MICLEA, L. Local EHR management based on FHIR. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATION, QUALITY AND TESTING, ROBOTICS (AQTR), 2018., 2018. **Anais...** IEEE, 2018. p. 1–5.

@TYPES/FHIR. [S. l.], 2021. Disponível em: https://www.npmjs.com/package/@types/fhir. Acesso em: 06 nov. 2021.

WLASCHIN, S. **Domain Modeling Made Functional**: tackle software complexity with domain-driven design and F#. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.