UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALBERT KOLBERG

Melhorando coordenação inter-times através da restrição do domínio de dados em web services

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof[a]. Dr[a]. Renata Galante

Porto Alegre
2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitor de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Cláudio Machado Diniz
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro
Bibliotecária-chefe da Escola de Engenharia: Rosane Beatriz Allegretti Borges

AGRADECIMENTOS

Agradeço especialmente à minha mãe, que dedicou seus esforços prioritariamente à minha educação e à dos meus irmãos. Aos meus irmãos, expresso minha gratidão por proporcionarem referências que eu jamais teria descoberto apenas seguindo meus próprios interesses. Agradeço a todos eles por sempre terem acreditado em minha capacidade, nunca permitindo que eu duvidasse disso.

Sou grato pela oportunidade de ter estudado em uma instituição de excelência como a UFRGS, onde pude ampliar e formalizar meus conhecimentos, estabelecer novos contatos e compartilhar salas de aula com professores admiráveis.

Expresso meu carinho à professora orientadora por sua ternura e gentileza durante os meus tempos de calouro e durante o período de desenvolvimento deste trabalho.

Muito obrigado.

RESUMO

Com a popularização dos *web services* nos últimos anos, torna-se indispensável o emprego de técnicas que promovam a comunicação correta entre cliente e servidor. Problemas relacionados à interface de comunicação entre esses componentes acarretam em novos esforços de coordenação entre as suas equipes de desenvolvimento, o que encarece o projeto. Ferramentas que se baseiam em *schemas* podem contribuir para assegurar que o domínio dos dados comunicados por um componente produtor de dados ao consumidor esteja dentro do domínio válido para a execução adequada das regras de negócio. No entanto, essas soluções nem sempre são adotadas e, quando são, podem não ser empregadas corretamente. Este trabalho propõe um modelo de framework que padroniza o tratamento do domínio de dados que é disposto por interfaces de comunicação. Diferentemente das soluções baseadas em *schemas*, a solução proposta prescreve seu emprego apenas no componente consumidor de dados. Assim, atenua-se a necessidade de coordenação entre as equipes, pois o desenvolvimento pode seguir de forma robusta diante de problemas que decorrem das interfaces estabelecidas. Neste trabalho também é realizada a implementação do framework proposto em Swift, assim como a sua avaliação.

Palavras-chave: Web services. coordenação. GSD. schemas. interfaces mal formadas. troca de requisitos. memória de projeto. tolerância a falhas.

Improving inter-team coordination through the restriction of data domain in web services

ABSTRACT

With the popularization of web services in recent years, it becomes essential to employ techniques that promote correct communication between client and server. Problems related to the communication interface between these components result in new coordination efforts among their development teams, which increases project costs. Tools based on schemas can help ensure that the data domain communicated by a data-producing component to the data consumer belongs to the valid domain for the proper execution of business rules. However, these solutions are not always adopted, and when they are, they may not be used correctly. This work proposes a framework model that standardizes the treatment of the data domain determined by communication interfaces. Unlike solutions based on schemas, the proposed solution prescribes its use only in the data consumer component. Thus, the need for coordination among teams is attenuated, as development can proceed robustly in the face of problems arising from established interfaces. This work also includes the implementation of the proposed framework in Swift, as well as its evaluation.

Keywords: web services, coordination, GSD, schemas.

LISTA DE FIGURAS

Figura 2.1 Jornada dos dados em uma arquitetura cliente-servidor 13

Figura 4.1 Exemplos do tamanho do domínio de um dado em transformação 27
Figura 4.2 Exemplo de coordenação entre um produtor e um consumidor 31

LISTAGENS

2.1 Exemplo de documento XML. 17

2.2 Exemplo de documento JSON. 18

2.3 Exemplo de documento XML Schema (XSD). 19

2.4 Exemplo de documento JSON Schema. 19

2.5 Exemplo de documento GraphQL Schema. 20

4.1 JSON com a idade de uma pessoa 28

4.2 Exemplo de mapeamento sem o uso do framework 28

4.3 Exemplo de mapeamento com o uso do framework 29

5.1 Exemplo de mapeamento com o uso do framework implementado em Swift 37

5.2 Exemplo da definição de uma nova Simplificação através de um novo *factory* para o tipo Morphism 38

5.3 Exemplo da definição de uma nova Asserção através de um novo *factory* para o tipo Assertion 39

5.4 Saída da ferramenta de análise estática feita sobre a Listagem 5.8. 40

5.5 Exemplo de Domain Model Object - Página inicial 41

5.6 Exemplo de Data Transfer Object - Página inicial 42

5.7 Mapeamento ad-hoc do DTO para o DMO. 42

5.8 Mapeamento do DTO para o DMO com o framework implementado. 44

LISTA DE ABREVIATURAS E SIGLAS

HTTP	*Hypertext Transfer Protocol*
XML	*Extensible Markup Language*
JSON	*JavaScript Object Notation*
IDE	*Integrated Development Environment*
IEFT	*Internet Engineering Task Force*
GSD	*Global Software Development*
DTO	*Data Transfer Object*
DMO	*Data Model Object*
LSP	*Language Server Protocol*
AST	*Abstract Syntax Tree*
CI	*Continuous Integration*

SUMÁRIO

1 INTRODUÇÃO ... 10
2 CONCEITOS E TECNOLOGIAS UTILIZADAS ... 12
2.1 Arquitetura Base ... 12
2.2 Propriedades de Dados ... 14
2.2.1 Espaço-valor ... 14
2.2.2 Nulabilidade ... 14
2.3 Processo de Desenvolvimento ... 14
2.3.1 Global Software Development ... 14
2.3.2 Métodos Ágeis ... 15
2.3.3 Desenvolvimento Iterativo ... 16
2.3.4 Entrega contínua ... 16
2.4 Tecnologias ... 16
2.4.1 Web Services ... 16
2.4.2 Data Interchange Format ... 17
2.4.3 GraphQL ... 18
2.4.4 Schemas ... 19
2.4.5 Swift ... 20
2.4.5.1 Nulabilidade ... 21
2.4.5.2 O tipo Result ... 21
2.4.5.3 Protocols ... 21
2.4.5.4 Extensions ... 21
2.4.5.5 Generic Type Constraints ... 22
2.4.6 Swift Package Manager ... 22
3 TRABALHOS RELACIONADOS ... 23
4 PROPOSTA ... 25
4.1 Software ... 26
4.2 Coordenação ... 31
4.3 Awareness e memória de projeto ... 33
5 RESULTADOS ... 35
5.1 Implementação ... 35
5.1.1 Tecnologias ... 35
5.1.2 Framework ... 36
5.1.3 Análise estática ... 40
5.2 Avaliação ... 41
5.2.1 Complexidade ... 41
5.2.2 Reuso e Testabilidade ... 45
6 CONCLUSÃO ... 47
REFERÊNCIAS ... 49

1 INTRODUÇÃO

A coordenação inter-times desempenha um papel crítico para o sucesso de projetos de software de larga escala com equipes de desenvolvimento distribuídas. Esse aspecto torna-se ainda mais relevante quando as distâncias entre essas equipes são maiores. Nesses tipos de configuração é indispensável adotar estratégias para diminuir a tensão sobre as atividades que envolvem coordenação (TEKINERDOGAN et al., 2012). Uma das práticas mais comuns para melhorar a coordenação entre equipes é separar os detalhes das partes dos sistemas em componentes. Essa separação possibilita o agrupamento de tarefas de desenvolvimento por contexto, de modo que cada grupo possa ser atribuído a uma equipe co-localizada e especializada naquele contexto. No entanto, tarefas que envolvem a comunicação de dados entre os componentes são interdependentes por natureza e exigirão interações entre diferentes times.

As tarefas relacionadas à comunicação de dados são indispensáveis na coordenação entre equipes, tornando relevante o uso de tecnologias que promovam a troca correta de dados. Essas tecnologias evitam cenários inesperados e reduzem a necessidade de interações adicionais entre as equipes. Nesse sentido, ferramentas baseadas em *schema* permitem definir regras sobre os dados que trafegam por uma interface. A aplicação dessas regras aproxima o espaço dos dados que podem ser transmitidos do espaço dos dados considerados válidos para a correta execução de uma aplicação.

Em geral, as ferramentas baseadas em *schema* atuam sobre um formato de dados específico. Esses formatos são conhecidos como *data interchange format* e os mais populares no contexto de *web services* são o Extensible Markup Language (XML) e o JavaScript Object Notation (JSON). O XML é um formato extensivamente testado, mas é redundantemente complexo para certas aplicações (ŠIMEC; MAGLIČIĆ, 2014). O JSON tornou-se popular nos últimos anos devido à sua simplicidade, entretanto a especificação do JSON Schema, o principal padrão de *schema* para JSON, ainda não tem o grau de maturidade dos *schemas* para XML e encontra-se em estado de *draft* no IETF.

Diante dessa conjuntura, propomos um modelo de framework cuja implementação visa mitigar erros decorrentes de definições de interfaces com ambiguidades, reduzindo a tensão sobre tarefas que envolvem coordenação inter-time. O framework atua independentemente do emprego de soluções baseadas em *schema* e promove robustez e reuso por meio do tratamento padronizado de ambiguidades no domínio dos dados. Além disso, ele habilita a geração de artefatos que podem auxiliar na difusão do conhecimento entre as

equipes (*inter-knowledge*) e contribuir para memória de projeto.

No próximo capítulo, são apresentados conceitos e tecnologias que dão suporte à apresentação da proposta deste trabalho e sua implementação. No próximo, Capítulo 3, é feita uma revisão sobre coordenação e interfaces. No Capítulo 4, é proposto o modelo de framework mencionado no parágrafo anterior e, no Capítulo 5, é apresentada a implementação da proposta e a sua avaliação. Por fim, o Capítulo 6 conclui este trabalho.

2 CONCEITOS E TECNOLOGIAS UTILIZADAS

Esse capítulo está dividido em quatro subcapítulos que introduzem conceitos e tecnologias relacionadas à proposta deste trabalho. O primeiro subcapítulo descreve uma arquitetura cliente-servidor que dá suporte à apresentação da proposta e, o segundo, apresenta propriedades de dados exploradas na proposta. Já, no terceiro subcapítulo, são introduzidos conceitos do processo de desenvolvimento nos quais a proposta tem impacto. Por fim, no último subcapítulo, são introduzidas as tecnologias que fazem parte da implementação ou do ambiente onde a solução é testada.

2.1 Arquitetura Base

Nesta seção, são apresentadas uma arquitetura cliente-servidor em torno da qual é discutida a proposta. A arquitetura contém apenas as abstrações necessárias que dão suporte ao estudo, sem deixar de manter um arquétipo coerente.

As principais entidades da arquitetura são o produtor e o consumidor. O produtor é responsável por deter a fonte de verdade, assegurar a devida execução das regras do negócio e prover os dados de que o consumidor precisa. Já o consumidor tem como propósito de interface com o usuário final apenas.

A arquitetura é composta por três camadas distintas: o Data Interchange Layer, Domain Data Service Layer e o Presentation Layer. No produtor, fica aparente apenas a camada Data Interchange e no consumidor, onde a solução da proposta é demonstrada, estão presentes as três camadas citadas. A seguir, cada uma das camadas é descrita:

- Data Interchange Layer: camada responsável pela comunicação entre produtor e consumidor. Ela tem como dependência as tecnologias necessárias para serializar e deserializar os dados trafegados entre esses componentes.
- Domain Data Service Layer: camada responsável por prover os objetos da modelagem do negócio para o Presentation Layer. Tais objetos são inicializados com os dados obtidos através do Data Interchange Layer, isto é, dados oriundos da comunicação cliente-servidor.
- Presentation Layer: camada de interface com o usuário final. Os dados consumidos nessa camada são obtidos a partir dos objetos fornecidos pelo Domain Data Service.

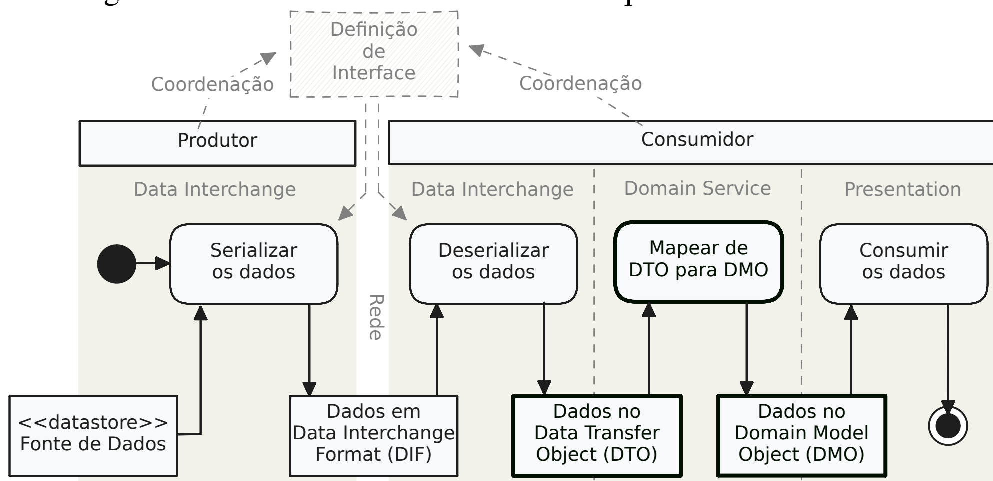
Outro aspecto importante dessa arquitetura para a discussão da solução é a clas-

sificação atribuída aos dados ou objetos que trafegam entre os componentes e camadas. São elas:

- Dados em Data Interchange Format: os dados nesse formato são os dados serializados que trafegam no canal de comunicação estabelecido entre produtor e consumidor. Esses dados, quando recebidos no consumidor, são transformados no Data Transfer Object pela ferramenta que realiza a deserialização.
- Data Transfer Object: objeto fruto imediato da deserialização de dados recebidos no Data Interchange Layer do consumidor. Esse objeto reflete diretamente a estrutura dos dados que trafegam entre produtor e consumidor e pode não representar exatamente a necessidade da aplicação no cliente. No consumidor, os dados desse objeto são mapeados para o Domain Data Object no Domain Data Service.
- Domain Model Object: objeto da modelagem do negócio no consumidor e, portanto, esse objeto possuí a estrutura final e contém os dados úteis que são manipulados para aplicação do negócio nesse componente. Esse objeto é fornecido ao Presentation Layer através da API do Domain Data Service.

O diagrama de atividades da Figura 2.1 demonstra o processo de uma requisição de dados pelo consumidor, entretanto também pode sumarizar a arquitetura. As raias externas representam os componentes e, as internas, as camadas dos componentes. As atividades (retângulos arredondados) representam a atividade central de cada uma dessas camadas e, os retângulos, representam os dados que trafegam entre as interfaces dessas camadas.

Figura 2.1 – Jornada dos dados em uma arquitetura cliente-servidor



Fonte: Próprio Autor

2.2 Propriedades de Dados

Nesse subcapítulo, são apresentadas duas propriedades relacionadas aos dados utilizadas na proposta e que, também, possuem um papel em soluções baseadas em *schemas*. As soluções baseadas em *schemas* assemelham-se com esta proposta em alguns pontos e fazem parte da discussão deste trabalho.

2.2.1 Espaço-valor

Também conhecido como domínio, representa o conjunto de todos os valores possíveis que um tipo de dados pode conter. O espaço-valor é essencial para a validação de dados que devem estar compreendidos dentro dos limites esperados.

2.2.2 Nulabilidade

Nulabilidade, ou *nullability*, refere-se à capacidade de uma variável ou elemento de dados receber um valor especial chamado nulo. Nulo representa a ausência de um valor ou um estado indefinido. Tratar com a nulidade é crucial para evitar erros inesperados em software. Verificações de nulos e mecanismos de tratamento apropriados são implementados para gerenciar situações em que os dados podem estar ausentes.

2.3 Processo de Desenvolvimento

A proposta tem potencial de contribuir positivamente nos ambientes de Global Software Development (GSD) que aplicam Métodos Ágeis. Portanto, nesse subcapítulo são introduzidos esses conceitos com ênfase nos aspectos nos quais a solução proposta tem maior influência.

2.3.1 Global Software Development

Global Software Development refere-se à prática de desenvolvimento de software entre equipes distribuídas geograficamente. Neste cenário, há desafios que surgem desde

a diferença de localização física das equipes e se estende até diferenças culturais entre elas. Essa configuração levanta desafios únicos relacionados à coordenação, colaboração e comunicação.

A coordenação é um processo fundamental em GSD que envolve organizar e sincronizar os esforços de equipes geograficamente dispersas para atingir objetivos comuns do projeto. Abrange o estabelecimento de canais de comunicação claros, gestão eficiente de fusos horários, atribuição e monitorização de tarefas através de metodologias ágeis, sensibilidade cultural a diversos estilos de trabalho e a utilização de repositórios de conhecimento centralizados.

Duas propriedades das equipes em GSD são importantes para este trabalho: inter-knowledge e awareness of others. O Inter-knowledge refere-se ao compartilhamento e troca de informações, habilidades e conhecimentos entre as equipes. Enfatiza o aproveitamento da inteligência de cada equipe de maneira a aprimorar o conhecimento e as capacidades do coletivo. Já, awareness of others centra-se na tomada de conhecimento dos integrantes de diferentes equipes, os seus papéis, pontos fortes, perspectivas a fim de ampliar o leque de apoio entre todas as equipes.

Essas propriedades são relevantes no desenvolvimento de qualquer projeto, entretanto, quando se trata de GSD, manter a qualidade dessas propriedades é um desafio devido às distâncias impostas nessa modalidade. Portanto, o sucesso do GSD exige estratégias e ferramentas únicas para gerir estes desafios e promover um ambiente de desenvolvimento produtivo.

2.3.2 Métodos Ágeis

Métodos Ágeis são uma abordagem flexível e colaborativa à engenharia de software que enfatiza o progresso iterativo, a adaptabilidade às mudanças e a satisfação do cliente. Incentiva as equipes multifuncionais a trabalharem em estreita colaboração, respondendo aos requisitos em evolução por meio de comunicação frequente e ciclos de desenvolvimento incrementais (BECK et al., 2001). Metodologias ágeis, como Scrum ou Kanban, priorizam a entrega regular de pequenos incrementos funcionais de software, permitindo melhoria contínua e maior capacidade de resposta ao feedback dos envolvidos.

2.3.3 Desenvolvimento Iterativo

O desenvolvimento iterativo, um componente central das metodologias ágeis, envolve dividir o processo de desenvolvimento de software em iterações menores. Cada iteração se concentra em entregar uma parte bem definida e testada do projeto. Este ciclo iterativo permite revisões, ajustes e refinamentos frequentes com base nas mudanças nos requisitos ou nas percepções obtidas durante o desenvolvimento. O desenvolvimento iterativo promove um processo de desenvolvimento mais adaptável e responsivo, onde as equipes podem se articular rapidamente para atender às prioridades emergentes ou incorporar o feedback dos usuários.

2.3.4 Entrega contínua

A Entrega Contínua é uma prática de desenvolvimento de software que garante que as alterações no código sejam automaticamente testadas e preparadas para produção durante todo o ciclo de vida do desenvolvimento. Ele anda de mãos dadas com o desenvolvimento iterativo, simplificando o processo de implantação. Com a entrega contínua, os incrementos funcionais de cada iteração são entregues imediata e automaticamente à produção, permitindo que as equipes lancem software com mais frequência e confiabilidade. Essa abordagem reduz a intervenção manual, minimiza os riscos de implantação e permite o encadeamento de processos de entrega.

2.4 Tecnologias

Essa seção dedica-se a introduzir as tecnologias que são discutidas neste trabalho ou que são empregadas na implementação da proposta.

2.4.1 Web Services

Os Web Services fornecem uma base para comunicação e interoperabilidade contínua entre diferentes sistemas. Basicamente, um Web Service é um protocolo padronizado que permite que diferentes aplicativos de software se comuniquem entre si pela web.

Um dos recursos marcantes dos Web Services é seu papel na habilitação da Ar-

quitetura Orientada a Serviços (SOA). Este paradigma arquitetônico gira em torno do encapsulamento de funcionalidades de negócios em serviços reutilizáveis, que podem ser acessados e utilizados de forma independente. Os Web Services, como canais para essas funcionalidades, promovem modularidade e capacidade de manutenção em ecossistemas de software complexos.

Um dos principais pontos fortes dos Web Services reside na sua capacidade de transcender as barreiras linguísticas e de plataforma. Ao aderirem a protocolos de comunicação padrão, como HTTP, eles facilitam a troca de dados em um formato universalmente compreendido.

2.4.2 Data Interchange Format

Os Data Interchange Formats, ou formatos de intercâmbio de dados, definem o formato das mensagens que trafegam entre os ecossistemas de software, facilitando a troca contínua de informações entre diversos aplicativos e sistemas. O uso desses formatos permite que diferentes componentes de software interpretem os dados transferidos entre eles de maneira uniforme.

Dois formatos proeminentes Data Interchange Formats são o XML (eXtensible Markup Language) e o JSON (JavaScript Object Notation). Esses formatos fornecem uma forma padronizada de representar dados aos diferentes componentes de um sistema. Ambos possuem representações que são compreensíveis tanto para máquinas quanto para humanos.

O XML é caracterizado por sua estrutura hierárquica baseada em tags, é versátil e extensível. É comumente usado em cenários onde a representação baseada em documentos é crucial, como arquivos de configuração, serviços web e armazenamento de dados. A natureza legível do XML facilita a depuração e a inspeção manual dos dados.

```xml
<Person>
    <Name>Fulano</Name>
    <Age>30</Age>
    <City>Porto Alegre</City>
</Person>
```

Listagem 2.1 – Exemplo de documento XML.

Por outro lado, JSON é um formato leve e fácil de ler que ganhou imensa popu-

laridade, principalmente em desenvolvimento web e interações de API. Sua estrutura de pares de valores-chave e suporte a array o tornam conciso e adequado para representar estruturas de dados complexas.

```
{
    "Person": {
        "Name": "Fulano",
        "Age": 30,
        "City": "Porto Alegre"
    }
}
```

Listagem 2.2 – Exemplo de documento JSON.

Esses formatos são independentes de linguagem, permitindo que sistemas desenvolvidos em diferentes linguagens de programação se comuniquem. Seja transmitindo dados entre um cliente e um servidor, integrando serviços de terceiros ou armazenando definições de configuração.

2.4.3 GraphQL

GraphQL é uma linguagem de consulta para APIs que fornece uma alternativa às APIs REST tradicionais. Seu desenvolvimento iniciou-se no ano de 2012 no Facebook e foi disponibilizado em código aberto em 2015 (GraphQL Foundation, 2024).

Essa linguagem permite que os clientes solicitem apenas os dados de que precisam, eliminando a busca excessiva ou insuficiente de dados. Os clientes definem a estrutura da resposta, possibilitando uma transferência de dados mais eficiente.

As APIs GraphQL são definidas por um *schema* que descreve explicitamente os tipos de dados que podem ser consultados. Esse *schema* serve como um contrato entre o cliente e o servidor, oferecendo uma API mais previsível.

A especificação do GraphQL não estipula qual tecnologia deve ser utilizada como camada de transporte. No caso deste trabalho, é utilizada a implementação da graphql-http, que utiliza HTTP para a transferência de dados e JSON como Data Interchange Format.

2.4.4 Schemas

Um *schema* é um projeto ou especificação formal que define a estrutura, a organização e as restrições dos dados em um sistema ou contexto específico. Ele atua como um conjunto de regras que regem como os dados devem ser formatados, quais tipos de dados são permitidos e como são estabelecidas as relações entre os diferentes dados.

No contexto dos Data Interchange Formats, um *schema* torna-se especialmente significativo. Ele fornece uma maneira padronizada de definir a estrutura e as características esperadas dos dados trocados entre diferentes sistemas. Isto garante que os dados trocados entre aplicações diferentes sigam um formato consistente, promovendo a interoperabilidade e reduzindo o risco de má interpretação.

No contexto dos formatos apresentados anteriormente, dois tipos de *schema* comumente utilizados são XML Schema (XSD) usado com XML e JSON Schema usado com JSON. Na implementação presente nesse trabalho, será utilizado o GraphQL Schema.

O exemplo de documento XML Schema a seguir é um *schema* que valida o exemplo de XML descrevendo uma pessoa apresentado anteriormente (Listagem 2.1):

```
<xs:element name="Person">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="Age" type="xs:int"/>
            <xs:element name="City" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Listagem 2.3 – Exemplo de documento XML Schema (XSD).

O exemplo de documento JSON Schema a seguir é um *schema* que valida o exemplo de JSON descrevendo uma pessoa apresentada anteriormente (Listagem 2.2):

```
{
    "type": "object",
    "properties": {
        "Name": { "type": "string" },
        "Age": { "type": "integer" },
```

```
        "City": { "type": "string" }
    },
    "required": ["Name", "Age", "City"]
}
```

Listagem 2.4 – Exemplo de documento JSON Schema.

Os schemas GraphQL são escritos usando a linguagem de definição de GraphQL Schema (SDL), que fornece uma sintaxe para expressar a estrutura da API. Nele são descritas a estrutura de tipos, a relação entre esses tipos e como o conteúdo deles são acessados e modificados através da API.

O exemplo de documento GraphQL Schema a seguir também pode ser utilizado para validar o documento JSON exemplificado anteriormente (Listagem 2.2):

```
type Person {
      Name: String!
      Age: Int!
      City: String!
}
```

Listagem 2.5 – Exemplo de documento GraphQL Schema.

2.4.5 Swift

Swift é uma linguagem de programação de alto nível desenvolvida pela Apple e introduzida na comunidade de desenvolvimento em 2014. A mesma é entregue (não exclusivamente) junto a IDE Xcode para o desenvolvimento de software para os dispositivos da Apple. Apesar de ser amplamente utilizada para o desenvolvimento de produtos desta empresa, a linguagem é considerada de propósito geral e sua implementação está disponível como código aberto. A empresa defende que o compilador é otimizado para desempenho e a linguagem, para a facilidade de desenvolvimento sem que um aspecto influencie negativamente o outro (Apple Inc., 2023).

Alguns aspectos dessa linguagem são relevantes para a apresentação proposta presente neste texto e para sua respectiva implementação. São eles:

2.4.5.1 Nulabilidade

Em Swift, é fortemente recomendado utilizar o tipo Optional para variáveis que podem assumir valores nulos (`nil` em Swift). A estrutura da linguagem em si conduz o desenvolvedor a verificar que o conteúdo desse tipo de variável não é nulo.

Apesar de atribuir mais robustez às aplicações escritas nessa linguagem, vale notar que Optinals, quando utilizados de forma excessiva, podem aumentar a complexidade do código desnecessariamente. Isso se dá, pois os acessos a essa variável requerem verificações, ou seja, desvios condicionais. Portanto, esse recurso deve ser utilizado somente quando necessário, isto é, quando o valor nulo desempenha um papel significante no uso daquela variável. Esse ponto será explorado com mais detalhes na proposta desse trabalho.

2.4.5.2 O tipo Result

Em Swift, o tipo Result é uma estrutura que encapsula o resultado de uma operação que pode gerar um valor ou falhar. Ele é frequentemente utilizado em situações em que uma função pode retornar um valor válido ou um erro, permitindo ao desenvolvedor tratar ambos os casos de maneira explícita. O Result é definido como uma enumeração com dois casos: success, que contém o valor resultante, e failure , que contém um erro.

2.4.5.3 Protocols

Em Swift, os *protocols* são como interfaces em outras linguagens de programação. Eles definem um conjunto de requisitos que classes, estruturas ou enumerações podem adotar, estabelecendo contratos para garantir consistência no código. Os *protocols* não fornecem implementações concretas, mas especificam o que deve ser implementado. Um tipo pode adotar múltiplos *protocols*, proporcionando flexibilidade no design do software.

2.4.5.4 Extensions

As extensions, ou extensões, em Swift são um recurso que permite aos desenvolvedores adicionar novas funcionalidades aos tipos pré-existentes, incluindo tipos de bibliotecas padrão com implantação privada. No framework que implementa a proposta desse trabalho, é esperado que os usuários do framework estendam entidades definidas no framework para ampliar de forma padronizada a cobertura de diferentes cenários em que

o mesmo pode atuar.

2.4.5.5 Generic Type Constraints

Em Swift, generic type constraints referem-se à capacidade de impor condições específicas nos tipos que podem ser utilizados como argumentos genéricos em funções, estruturas, ou classes. Essas restrições são estabelecidas através da utilização de palavras reservadas como where para definir critérios que os tipos genéricos devem atender. Por exemplo, é possível impor que um tipo genérico deva ser uma subclasse de uma determinada classe e/ou conformar a um protocolo específico. As generic constraints oferecem uma maneira de garantir que as operações genéricas sejam aplicadas apenas a tipos que possuam as características necessárias.

2.4.6 Swift Package Manager

O Swift Package Manager (SPM) é um sistema de gerenciamento e construção de dependências para projetos Swift. Desenvolvido pela Apple, o SPM simplifica o processo de distribuição e gerenciamento de módulos de código Swift, também conhecidos como pacotes. É particularmente útil para construir, testar e distribuir código Swift em diferentes projetos. O SPM é multiplataforma, de código aberto e permite o desenvolvimento de extensões pela comunidade.

3 TRABALHOS RELACIONADOS

Neste capítulo é feita a revisão de trabalhos que estudam o papel de interfaces na coordenação do desenvolvimento de software e estudam tolerância a falhas no contexto de *data interchange formats* (e.g. XML e JSON).

O estudo recente de (SIEVI-KORTE; BEECHAM; RICHARDSON, 2019) levanta um conjunto de desafios-questões acompanhadas de soluções-práticas para a concepção arquitetural de sistemas em Global Software Development (GSD) baseando-se em uma revisão sistemática validada por pares. Os desafios foram separados por temáticas recorrentes em GSD. Uma das temáticas elencadas pelos autores foi a modularidade. Segundo eles, esse tema é relevante, pois a modularidade dirige a alocação de tarefas. (HERBSLEB, 2007) aponta que a necessidade de gerenciamento de dependências entre tarefas é o fenômeno principal de GSD. Um dos desafios levantados nesse tema por (SIEVI-KORTE; BEECHAM; RICHARDSON, 2019) foi a identificação de dependências e desacoplamento de componentes a fim de permitir separação de tarefas. Como prática para tratar desse desafio, foi sugerida a promoção do desacoplamento através da definição de interfaces bem formadas.

(SOUZA et al., 2004) conduzem um estudo sobre o uso de APIs em uma organização. Eles identificam benefícios e desafios relacionados à divisão organizacional promovida pelo emprego de interfaces bem definidas. Além de verificar os benefícios da segmentação das tarefas e equipes, resultado da componentização, o trabalho destaca que essa abordagem pode levar os integrantes de uma equipe a não conhecerem os integrantes de outras e seus respectivos papéis. Essa ausência de conhecimento (*awareness of others* e *inter-knowledge*) dificulta a identificação dos responsáveis de uma interface e a comunicação com os mesmos. Os autores sugerem a busca por equilíbrio entre o isolamento do trabalho de uma equipe e a adoção de práticas e ferramentas que permitam que as equipes se conheçam o suficiente para trabalharem de forma colaborativa.

Na publicação de (EDSTROM; TILEVICH, 2012) é discutida a tolerância a falhas no contexto de *web services* e proposto um framework para implementação de tal. A solução é motivada pela tendência de não haver frameworks que implementam *web services* com tolerância a falhas incluída. Segundo os autores, esse fato é comumente justificado porque tolerância a falhas está especificamente relacionada a aplicação e, portanto, cabe ao desenvolvedor implementar. A proposta citada se assemelha com a nossa em dois sentidos: primeiro por propor uma solução lado-cliente que implementa tolerância a falhas

de forma reutilizável e, segundo, por ambas serem motivadas pela ausência de soluções semelhantes. No entanto, o trabalho mencionado se diferencia ao se concentrar nos erros de transmissão de dados, enquanto o nosso se dedica à validação dos dados transmitidos.

Os estudos de (ŠIMEC; MAGLIČIĆ, 2014), (GOYAL; SINGH; RAMKUMAR, 2017) e (ŠANDRIH; TOŠIĆ; FILIPOVIĆ, 2017) comparam o desempenho e complexidade dos formatos XML e JSON durante a transmissão dos dados. (BREJE et al., 2018) também fazem essa comparação, mas quando a validação e compressão de dados estão habilitados. As implementações para JSON apresentaram melhor desempenho na grande maioria dos testes conduzidos pelos estudos. Em geral, os resultados são justificados com o argumento de que a conversão de JSON para as linguagens de programação é mais direta, em razão de esse formato ser baseado em objetos de *JavaScript*.

Esses estudos também fornecem percepções que podem justificar por que nos últimos anos o JSON vem ganhando maior adoção. O JSON foi concebido como um Data Interchange Format. Já o XML, apesar de oferecer grande flexibilidade para transferir documentos que contêm imagens, gráficos, texto, entre outros, não é adequado para transmissão de dados mais simples, pois é desnecessariamente redundante e complexo (ŠIMEC; MAGLIČIĆ, 2014). Em suma, JSON tem um propósito específico e é simples, enquanto XML possui um propósito genérico e pode ser complicado demais para certas aplicações.

4 PROPOSTA

A proposta tem como objetivo reduzir ambiguidades no domínio dos dados que são recebidos via *web service* por um componente consumidor de dados. Isso é feito através do estreitamento do domínio dos dados que é determinado pela interface de comunicação para um domínio mais restrito. Por mais restrito, interpreta-se um domínio com cardinalidade e intersecção próxima às do domínio que contém os valores válidos para a aplicação do componente consumidor. A solução proposta realiza o estreitamento do domínio com a troca dos tipos de dados e/ou da aplicação de asserções sobre os dados recebidos de outro componente do sistema.

Soluções baseadas em *schema* possibilitam definir restrições adicionais sobre uma propriedade que são validadas quando os dados são enviados por um componente produtor ou quando recebidos pelo componente consumidor de dados. O emprego desse tipo de solução requer que as equipes responsáveis por diferentes componentes coordenem qual tecnologia é adotada e quais restrições são aplicadas aos dados que são comunicados. Caso hajam problemas quando essas decisões são exercitadas, é necessário que essas equipes coordenem-se novamente.

Assim como soluções baseadas em *schema*, a solução proposta facilita verificar os dados de propriedades de forma padronizada a fim de aproximar esses dados ao domínio válido para uma dada aplicação. Entretanto, a solução proposta envolve apenas o time consumidor de dados, de modo a não exigir coordenação no emprego da solução ou na descoberta de ambiguidades na definição da interface. Vale notar que a solução proposta não depende ou substitui *schemas*, mas age como uma camada adicional que estreita o domínio dos dados para o domínio válido.

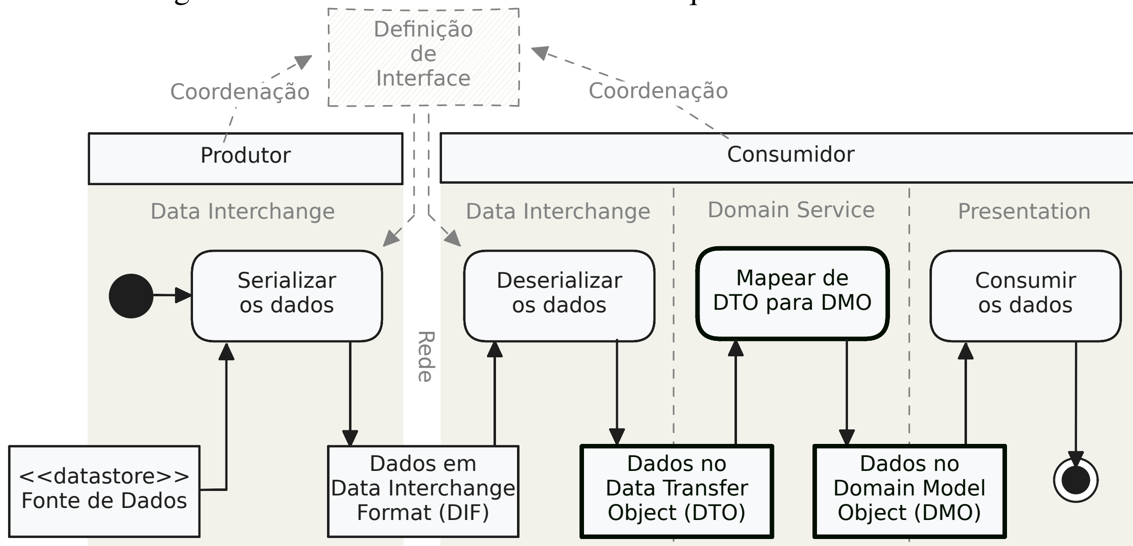
A solução consiste em um modelo de framework e não um framework em si. O modelo é utilizado para a implementação de um framework para uma dada plataforma e um dado conjunto de tecnologias. Portanto, o modelo pode ser considerado agnóstico de plataforma, mas o framework produto do modelo não. Por fins de simplicidade, o modelo do framework é mencionado apenas como framework no texto, ficando explicito a distinção entre estes dois significados quando for necessário.

Na subseção seguinte, o modelo é discutido da perspectiva da sua implementação em software. Nas duas subseções que seguem (4.2 e 4.3), são discutidos os impactos da adoção do framework para o processo de desenvolvimento.

4.1 Software

A definição de dois objetos é importante para a apresentação da proposta. São eles o Data Transfer Object (DTO) e o Domain Model Object (DMO). Eles carregam os dados consumidos pela aplicação e representam respectivamente a entrada e a saída da solução. O framework atua como um intermediário entre o DTO e o DMO, pois ele exerce o mapeamento dos dados de um DTO com um domínio possivelmente ambíguo para um DMO com um domínio mais robusto. Nos dois parágrafos seguintes, esses objetos são recapitulados e na sequência é discorrido o seu papel na solução. A Figura 2.1 mostra onde e quando o mapeamento entre esses objetos acontece no contexto de uma arquitetura simples.

Figura 2.1 - Jornada dos dados em uma arquitetura cliente-servidor



Fonte: Próprio Autor

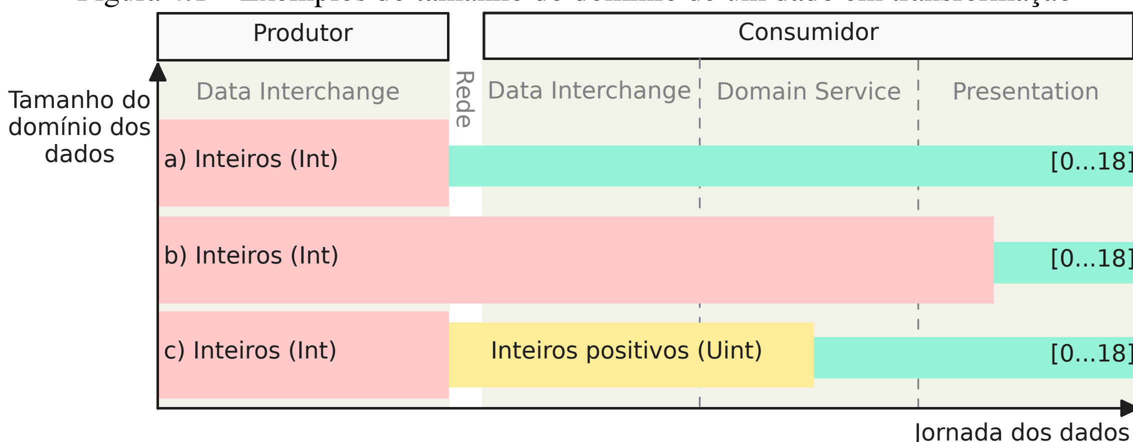
Data Transfer Object: contém dados úteis de um certo contexto para serem transferidos entre diferentes componentes de um sistema distribuído. O DTO tem a mesma estrutura no produtor e consumidor de dados. Ele é inicializado, serializado e entregue pelo produtor ao consumidor. No consumidor, esse objeto é recebido, deserializado e mapeado para um DMO. Um DTO é indiretamente o reflexo da interface de comunicação de dados acordada entre produtor e consumidor.

Domain Model Object: reflete os conceitos do domínio de negócios da aplicação e contém os dados que são consumados para a realização dos objetivos da aplicação. Esse objeto não possui necessariamente a mesma estrutura do DTO, mas é inicializado no consumidor a partir dos dados contidos no DTO.

Os tipos de dados que o DTO possui associado a suas propriedades determinam o domínio intrínseco para cada uma dessas propriedades. Por exemplo, uma propriedade do tipo *integer* poderá conter qualquer valor representável por esse tipo na linguagem de programação utilizada. Da perspectiva do código da aplicação do consumidor, essa é a única certeza a respeito do domínio. Isso ocorre, mesmo que o produtor tenha restringido os valores para um intervalo menor de inteiros ou alguma restrição tenha sido aplicada por meio de um *schema* para essa propriedade.

Em um cenário ideal, seria necessário apenas o mapeamento direto dos dados do Data Transfer Object para as propriedades do Domain Model Object para o correto funcionamento de uma aplicação. Em outras palavras, as restrições e tipos aplicados aos dados enviados por um produtor seriam suficientes para que estes dados estejam em condições válidas para o consumidor (ilustrado na Figura 4.1 a). Entretanto, um DTO é reflexo da definição de interfaces e suas tecnologias e ele pode ter um domínio maior do que aquele contento apenas os dados válidos para a aplicação (ilustrado na Figura 4.1 b e c). Outra questão é que os desenvolvedores podem não ter a segurança de que as restrições apropriadas foram aplicadas ao DTO durante sua inicialização ou transferência. Quando o domínio de um DTO é ambíguo e seus dados são mapeados diretamente para um DMO, temos uma situação propensa a falhas.

Figura 4.1 – Exemplos do tamanho do domínio de um dado em transformação



Fonte: Próprio Autor

A fim de demonstrar o problema que a solução resolve, é apresentado um exemplo simples. Suponha que o JSON do Código-fonte 4.1 é uma mensagem que deve ser enviada de um produtor para um consumidor de dados conforme a definição de uma interface:

```
{
    "age": 30
}
```

Listagem 4.1 – JSON com a idade de uma pessoa

Na definição da interface, fica acordado que não devem ser enviados valores negativos para o campo `age`. Entretanto, com as tecnologias empregadas, ainda é fisicamente possível que o produtor serialize e envie mensagens com valores negativos.

Uma forma de eliminar essa ambiguidade seria adicionar uma verificação *ad-hoc* para garantir que o valor da idade seja positivo no consumidor. Esse tipo de solução não é ideal, pois pode não estar bem especificado em que ponto da arquitetura essa verificação deve ser feita. Sem um framework ou uma prática estipulada para o projeto, esse tipo de verificação pode rapidamente se espalhar por diferentes partes do código, sem permitir reuso e dificultando a manutenção.

A partir do exemplo, fica evidente a necessidade do consumidor de atribuir restrições adicionais sobre o espaço de dados que podem ser recebidos de um produtor e tratar situações onde a aplicação de tais restrições falham. O framework proposto responde a essas necessidades de forma organizada. Para demonstrar isso, são apresentados dois trechos de código que dão sequência ao exemplo acima. O Código-fonte 4.2 apresenta o mapeamento sem o uso do framework e o Código-fonte 5.1 com o uso dele. Ambos trechos mostram o mapeamento de um objeto fruto da deserialização (Data Transfer Object) para um objetos da modelagem do domínio da aplicação (Domain Model Object).

```swift
// O data transfer object (DTO) de uma pessoa:
struct PersonDTO: Decodable {
    let age: Int? // o valor pode ser um inteiro ou nulo
}

// O domain model object (DMO) de uma pessoa:
struct PersonDMO {
    let age: Int? // o valor pode ser um inteiro ou nulo
}

// Mapeamento direto de DTO para DMO:
let personDMO = PersonDMO(age: personDTO.age)
```

Listagem 4.2 – Exemplo de mapeamento sem o uso do framework

No caso do Código-fonte 4.2, onde não é utilizado o framework, o DTO é mapeado diretamente para um DMO. Em vista disso, seria prudente verificar que o valor de `age` do DMO não é nulo e que é um inteiro positivo a fim de assegurarmos a validade dos dados. Essa verificação pode estar em qualquer parte do software ou pode nem mesmo ser criada, deixando essa verificação a merce do produtor dos dados.

```
// O data transfer object (DTO) de uma pessoa:
struct PersonDTO: Decodable {
    let age: Int? // o valor pode ser um inteiro ou nulo
}

// O domain model object (DMO) de uma pessoa:
struct PersonDMO {
    let age: UInt // o valor pode ser um inteiro positivo
}

// Mapeamento de DTO para DMO com o framework:
let personDMO = try PersonDMO(
    age: personDTO.age.notNull().positive()
)
```

Listagem 4.3 – Exemplo de mapeamento com o uso do framework

A primeira diferença que notamos no Código 5.1, onde é empregado o framwork, é que a propriedade `age` do DMO é um inteiro positivo não opcional em vez de um inteiro opcional. Portanto, a complexidade do tipo no DMO com o uso do framework é menor que a complexidade do tipo no DMO sem o uso do framework. Dessa maneira, esse objeto pode ser consumido em diferentes camadas da aplicação do consumidor sem carregar consigo uma complexidade que surgiu de ambiguidades nos dados que foram recebidos de um produtor.

A segunda diferença no trecho está na inicialização do DMO a partir do DTO. Nesse ponto, são encadeadas duas chamadas para métodos do framework: `notNull()` e `positive()`. Esses métodos efetuam um mapeamento não direto entre os dados do DTO e o DMO. São eles que possibilitam o mapeamento de valores do domínio mais complexo do DTO para o domínio menos complexo do DMO. Na prática, isso significa que, quando a propriedade `age` é consumida, não é necessário fazer verificações adicionais, pois o dado já foi validado pelo mapeamento.

Os métodos exemplificados, assim como outros contidos no framework, são aplicados quando as restrições inerentes estabelecidas pelo tipo de uma propriedade de um DTO não são satisfatórias para assegurar o devido funcionamento da aplicação. Cada método desses desempenha simultaneamente as seguintes capacidades no mapeamento dos dados de um DTO para as propriedades de um DMO:

- **Simplificação**: transforma o tipo do dado para outro tipo mais restritivo ou menos complexo. Por tipo mais restritivo ou menos complexo, entende-se um tipo que configura um espaço-valor menor a uma propriedade; ou seja, uma propriedade que possui um domínio mais restrito para o dado que carrega. No exemplo, o tipo da propriedade `age` foi mapeado de inteiro opcional para inteiro não opcional e, na sequência, mapeado de inteiro para inteiro positivo. O efeito da Simplificação fica evidente ao comparar as diferenças entre os tipos presentes no DTO e DMO.
- **Asserção**: é uma restrição adicional sobre um mapeamento. Esse papel entra em ação, quando a Simplificação não ocorre ou não é suficiente. Ou seja, quando não há uma mudança de tipo para restringir os dados ou a mudança de tipos não restringe suficientemente o espaço-valor. No exemplo, a propriedade é restringida através da Simplificação para conter apenas inteiros positivos. Entretanto, se fosse necessário restringir a idade para contemplar apenas menores de idade (ilustrado na Figura 4.1 c), seria necessário fazer uma Asserção para garantir que só sejam válidos inteiros abaixo de dezoito ou definir um novo tipo mais restritivo e aplicar a Simplificação.
- **Tratamento**: seleciona entre tratativas pré-definidas no framework qual o tratamento deve ser aplicado quando o dado entregue pelo produtor não está no espaço-valor do tipo simplificado ou quando a Asserção sobre esse dado falha. Há quatro formas de tratamento: substituir por valor padrão, advertir, lançar erro e personalizado. No exemplo, não foi especificada uma estratégia, então a estratégia de lançar erro é aplicada.
- **Registro**: a chamada do método em si no código-fonte é um Registro estático de

que os dados entregues pelo produtor não são restritos o suficiente para o consumidor. Indiretamente, isso revela uma discordância no acordo de interface entre produtor e consumidor. Quando um tratamento é acionado em tempo de execução, é feito o Registro dinâmico da ocorrência. Na subseção 4.3, é discutida a importância desses registros para o processo de desenvolvimento.

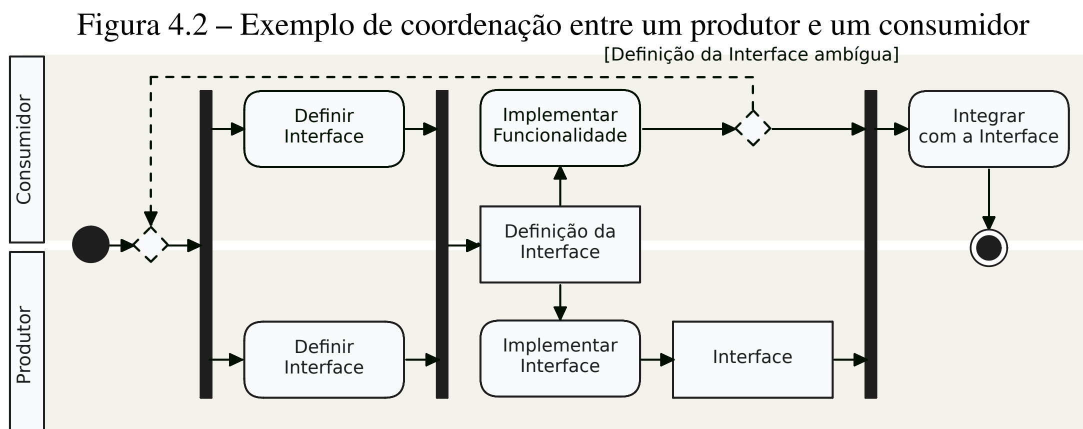
O uso do framework revela o DTO não apenas como um trasportador de dados, mas como um objeto mais complexo. O DMO é revelado não somente como um objeto que possui os dados úteis, mas um objeto mais simples com dados validados que dispensam verificações adicionais. Dada essa perspectiva, é importante reduzirmos ao máximo o ciclo de vida que o DTO possui e as partes do software que ele é acessado. Como efeito disso, as ambiguidades herdadas da interface de comunicação ficam isoladas.

O ciclo de vida de um DTO deve começar na deserialização e encerrar no mapeamento para o DTO (no caso da Figura 2.1, isso ocorre no Domain Service). Como o framework trata desse mapeamento, também fica bem determinado que parte da arquitetura ele é acessado e que parte dela tem ele como uma dependência.

4.2 Coordenação

Imagine dois times separados com custo de coordenação alto em razão da distância geográfica e cultural entre eles. Um é o time produtor e o outro é o consumidor. O consumidor é responsável apenas por apresentar dados aos usuários finais e o produtor é responsável por acessar os dados na fonte de dados do sistema e entregá-los conforme as necessidades do consumidor. As equipes coordenam-se para definir a interface de comunicação e trabalham de forma paralela, conforme exemplificado na Figura 4.2.

Figura 4.2 – Exemplo de coordenação entre um produtor e um consumidor



Fonte: Próprio Autor

Pode ocorrer que, ao longo do desenvolvimento, o consumidor identifique que as restrições aos dados definidos na interface original não são mais suficientes para assegurar o funcionamento robusto da aplicação. Do ponto de vista do consumidor, vamos considerar dois desfechos para a continuidade do desenvolvimento: coordenar novamente com o produtor, como visto no fluxo tracejado da Figura 4.2, ou aplicar localmente as restrições e verificações necessárias (evitando o fluxo tracejado).

No primeiro caso, o custo está associado ao imprevisto de coordenação. Já no segundo caso, há um aumento no custo de manutenção devido ao aumento da complexidade do código. Nesse último caso, podemos considerar que, em uma próxima interação prevista com o produtor, a interface seja alterada conforme necessário. Todavia, o consumidor arca com o custo de refatorar o código devido às alterações da interface ou com o custo de manutenção que surge ao longo do tempo em função da complexidade que foi adicionada com alterações locais.

A solução proposta nesse trabalho ameniza as consequências de optar pelo segundo desfecho, pois isola a complexidade das restrições adicionais e oferece tolerância a falhas quando essas restrições não são atendidas. Como o framework que implementa a solução encapsula a aplicação das restrições e seu tratamento, ele proporciona reuso e simplifica o refatoramento do código, diminuindo os custos de manutenção.

O framework é opinativo em relação a onde e como deve ocorrer a transformação do domínio dos dados recebidos para o domínio do consumidor. Também fica bem estabelecido que o domínio dos dados recebidos pelo produtor é o espaço-valor do Data Transfer Object (DTO) e que o domínio do consumidor é o espaço-valor das propriedades do Domain Model Object (DMO). Sendo assim, os métodos de mapeamento do framework atuam como a fronteira entre esses dois domínios e fica claro o local para refatoração decorrente de alterações na interface.

A vantagem dessa separação é mais expressiva quando consideramos desenvolvimento ágil. É comum que, nesse ambiente, seja priorizado começar o desenvolvimento mais brevemente em oposição a dedicar um grande esforço com especificações que precedam o desenvolvimento. Consequentemente, é natural existirem mais ambiguidades na interface num estágio inicial do desenvolvimento e que essa interface venha a ser iterada.

Como existe uma separação nítida entre os dados recebidos dos dados que são consumidos, é possível que os desenvolvedores de um componente consumidor comecem o seu desenvolvimento idealizando os dados do seu DMO sem se preocupar em manter um espelhamento preciso com os dados que serão recebidos de um produtor. Em um processo

de desenvolvimento que envolva entrega contínua, essa flexibilidade e a tolerância a falhas da solução facilitam que entregas sejam feitas com segurança, mesmo que ainda não haja um estreitamento ideal entre o domínio dos dados entregues pelo produtor com o domínio válido para o consumidor.

4.3 Awareness e memória de projeto

A solução proposta visa diminuir a necessidade não planejada de coordenação inter-times com o objetivo de desbloquear o progresso do time que a emprega. Entretanto, menos coordenação estende-se em menos interações inter-time que, por sua vez, pode levar a uma diminuição na consciência que diferentes times têm sobre outras partes do projeto, seus desenvolvedores e problemas (*awareness of others*). Esse distanciamento promove a diminuição na difusão do conhecimento em torno do projeto e seus desenvolvedores.

Quando a solução viabiliza que uma equipe de desenvolvimento conclua uma iteração de desenvolvimento sem a necessidade de coordenação prévia com outra equipe para ajustar uma interface, isso pode resultar na falta de ciência por parte da segunda equipe em relação às imperfeições encontradas naquela interface. O não bloqueio e a não necessidade imediata de alinhamento destas equipes podem culminar no esquecimento dessa questão. Contudo, os registros estáticos e dinâmicos gerados com o framework são artefatos que podem ser utilizados para mitigar esses sintomas decorrentes da não coordenação entre as equipes.

Os Registros estáticos presentes no código-fonte são indiretamente um registro do desalinhamento entre as necessidades do consumidor de dados e a interface de comunicação. Com o uso de uma ferramenta de análise estática de código, podem ser extraídos esses registros em forma de documento. Então, essa documentação pode ser difundida entre os times e servir como um artefato de troca de requisitos. Outro tipo útil de registro, são os Registros dinâmicos, que anotam a ocorrência de erros na execução das restrições sobre os dados recebidos pelo consumidor. Esses registros expostos através de um serviço de *logging* podem estar disponíveis em um painel remoto que é acessível por todos os envolvidos, de forma a aumentar a observabilidade do software.

Note que esses artefatos de comunicação, além de contribuírem para *awareness* e observabilidade, podem ser utilizados para memória de projeto. Os Registros estáticos podem ser gerados toda vez que uma versão do software é submetida ao gerenciador de

versões. Isso gera um histórico que pode ser analisado ao decorrer do tempo e que pode ser utilizado, por exemplo, para observar a evolução do estreitamento entre os domínio dos dados transitados via uma interface com o domínio válido para a aplicação.

5 RESULTADOS

Neste capítulo, aborda-se a implementação da solução proposta no Capítulo 4 e sua avaliação. No primeiro subcapítulo são expostas às implementações de um framework e de uma ferramenta de análise estática. No segundo subcapítulo, a implementação do framework é avaliada.

5.1 Implementação

O objetivo da implementação é concretizar a solução do framework proposto e da solução de análise estática proposta. A solução do framework consiste em padronizar o mapeamento de um DTO para um DMO eliminando ambiguidades através das operações de mapeamento da proposta (Simplificação, Asserção, Tratamento) e do respectivo log delas (Registro dinâmico). A solução da análise estática consiste em registrar (Registro estático) as chamadas para o framework a fim de gerar artefatos úteis para o processo de desenvolvimento.

Nesse subcapítulo, são citadas as tecnologias utilizadas e, então, é dado foco a implementação do framework e da ferramenta de análise estática.

5.1.1 Tecnologias

O framework e o cliente foram escritos na linguagem de programação Swift como pacotes do Swift Package Manager, permitindo a integração do framework nas plataformas da Apple e no sistema operational Windows. Foram utilizadas apenas bibliotecas despachadas com o kit de desenvolvimento básico do Swift. O servidor foi implementado em JavaScript para execução em Node.js. O servidor tem como dependências o Express para fornecer a API de HTTP e o http-graphql para a implementação do servidor de GraphQL.

É importante notar que o Data Transfer Object e o Domain Model Object são mencionados como objetos no texto. Entretanto, no contexto da implementação da proposta, esses "objetos"são implementados como structs em Swift. Isso significa que essas entidades assumem semântica de valor ao invés de semântica de referência. Não obstante, o termo objeto continuará a ser utilizado para o DTO e o DMO a fim de dar continui-

dade à terminologia anterior. O termo "object"no nome dessas entidades não deve ser interpretado como uma instância de uma classe em Swift.

5.1.2 Framework

O usuário do framework se relaciona com três entidades fundamentais para realizar o mapeamento de DTOs para DMOs:

- `DataTransferObject`: quando um tipo assina esse protocolo, as propriedades dele se tornam mapeáveis pelo framework. Os tipos que conformam esse protocolo automaticamente ganham a capacidade de criar um `MappingContext` a partir de uma implementação padrão (*protocol extension*).
- `MappingContext`: permite realizar as operações de mapeamento de forma encadeada sobre as propriedades de um DTO e extrair o resultado dessas operações a fim de popular um DMO. Possuí como dependência uma instância que implementa o protocolo `MappingLogger`.
- `MappingLogger`: dependência injetada na inicialização do `MappingContext`. Através de uma instância desse tipo, o `MappingContext` reporta o andamento das operações de mapeamento. Essa entidade realiza a capacidade de Registro da solução.

Como visto, o `MappingContext` é o cerne da implementação, pois esta entidade viabiliza as operações de mapeamento. A seguir, são relacionadas cada uma das operações com a entidade da implementação que a realiza:

- `Morphism`: tipo que contém uma função que realiza a Simplificação. Ela tem como entrada um valor oriundo do DTO e como saída um `Result` contendo o valor traduzido para um novo tipo no caso de sucesso da operação ou um erro caso contrário.
- `Assertion`: tipo que contém uma função que realiza a Asserção. Ela também tem como entrada um valor do DTO, mas um erro opcional como saída. A ausência de valor na saída (`nil`) representa a ausência de um erro.
- `ErrorHandling`: tipo que contém uma função que realiza o Tratamento. Tem como entrada um erro e o dado que falhou na execução da função de um `Morphism` ou `Assertion`. A saída é um `Result`.

Cada elo do encadeamento de operações do `MappingContext` que envolve um `Morphism` ou `Assertion`, pode ser opcionalmente combinado com um `ErrorHandling` que seja compatível. A compatibilidade é determinada pela conformidade do tipo de entrada das operações com as restrições do tipo (*generic type constraints*) de entrada da implementação do `ErrorHandling`.

No final do encadeamento, o resultado da execução das operações pode ser extraído de duas formas:

- Como um valor do tipo `Result`: esse modo é útil para os casos em que é prevista na modelagem a impossibilidade de obter um determinado dado como um estado válido para a aplicação.
- Como um valor: nesse modo, quando todo o encadeamento de operações é executado com sucesso, o dado é retornado diretamente. No caso de alguma execução falhar, é lançada uma exceção. Esse modo é útil quando um dado deve estar obrigatoriamente presente para a aplicação do negócio.

A Listagem 5.1 demonstra a aplicação das funcionalidades descritas acima no mapeamento de uma propriedade de um DTO para um DMO. A seguir, o exemplo é descrito linha a linha:

```
1 let userDMO = UserDMO(
2  name: try userDTO
3   .makeMappingContext()
4   .focus(in: \.name)
5   .map(.asMandatory)
6   .ensure(.lenght(within: 2...20), onError: .slice(at: 0..<20))
7   .getValue()
8   // [...]
9 )
```

Listagem 5.1 – Exemplo de mapeamento com o uso do framework implementado em Swift

1. Chamada do construtor do DMO.
2. Argumento do construtor para o valor da propriedade `name` do DMO do tipo `String`; essa propriedade que é obrigatória para a instanciação do DMO. O try marca a possibilidade da expressão subsequente lançar uma exceção.
3. Criação do contexto de mapeamento a partir da instância de um DTO (`userDTO`).

Essa função recebe como argumento uma instância de um `MappingLogger`. Nesse caso, está sendo utilizado um argumento padrão e, portanto, ele está implícito na chamada.

4. Seleção da propriedade do DTO da qual é extraído o dado. A propriedade `name` do DTO é uma string opcional (`String?`).
5. Simplificação da propriedade do DTO do tipo `String?` para o tipo `String`; a função `map` permite a chamada de uma Simplificação; o token `.asMandatory` é de um *factory* que instancia um `Morphism`. Essa Simplificação transforma a saída de um tipo opcional para um tipo não opcional. Como não há um Tratamento combinado à Simplificação, na ocorrência de um erro, o contexto de mapeamento não será capaz de retornar um dado no final do mapeamento.
6. Asserção sobre a propriedade `name` do DTO que, neste ponto, após a Simplificação é do tipo `String`. A função `.lenght(within:)` é um *factory* que instancia uma `Assertion` que verifica o comprimento da string. No caso, é aceito o comprimento na faixa de dois até vinte caracteres. A Asserção está combinada com um Tratamento e, portanto, é instanciado um `ErrorHandling` a partir do *factory* `.slice(at:)` que realiza a truncagem da string. Se ela falhar, o mapeamento não é capaz de retornar um dado.
7. A função `getValue()` marca o final do mapeamento. Se todas as Simplificações e Asserções são executadas com sucesso ou se todas as que falham são tradas com sucesso, o dado do DTO que foi restringido através das operações é retornado. É lançada uma exceção se, no final do contexto, não houver um dado disponível. Alternativamente, o contexto poderia ser encerrado com a função `getResult()` e, então, o retorno seria do tipo `Result` de forma a evitar uma exceção no caso de erro na execução das operações.

O framework pode ser estendido para solucionar um domínio maior de ambiguidades. Essa capacidade se dá através da possibilidade dos usuários do framework de definirem novos *factories* que geram instancias diferentes de `Morphism`s, `Assertion`s e `ErrorHandling`s além daquelas que já são despachados com o framework. A Listagem 5.2 demonstra um `Morphism` sendo estendido com um novo *factory* para a instanciação de um novo `Morphism` que converte datas representadas em strings conforme o RFC3339 para um objeto que representa datas na biblioteca `Foundation` de Swift. No exemplo é reaproveitado outro *factory* fixando alguns parâmetros conforme o padrão do RFC.

```swift
extension Morphism where Input == String, Output == Date {
 static var asRFC3339Date: Self {
  asDate(
   dateFormat: "yyyy-MM-dd'T'HH:mm:ssZZZZZ",
    locale: .init(identifier: "en_US_POSIX"),
    timeZone: .init(secondsFromGMT: .zero)
  )
 }
}
```

Listagem 5.2 – Exemplo da definição de uma nova Simplificação através de um novo *factory* para o tipo Morphism

Cada *factory* das operações de mapeamento pode estar definido para atuar dentro de um contexto de tipos, esse contexto é definido pelo tipo ou tipos que se enquadram nas restrições (*generic type constraints*) que são atribuídas aos tipos genéricos da operação. No caso da Listagem 5.2, o *factory* está restrito a instanciar um `Morphism` que mapeia uma String para um Date. No exemplo da Listagem 5.3, são apresentadas restrições mais relaxadas que viabilizam que tal `Assertion` atue sobre qualquer tipo que conforme ao protocolo `Comparable`. Essa `Assertion` permite verificar se o dado está contido dentro daquela faixa de valores.

```swift
public extension Assertion where T: Comparable {
 static func range<R>(_ range: R) -> Self where R:
   RangeExpression, R.Bound == T {
  Assertion { value in
   if range.contains(value) {
    return .success
   } else {
     return MappingError.assertionError()
   }
  }
 }
}
```

Listagem 5.3 – Exemplo da definição de uma nova Asserção através de um novo *factory* para o tipo Assertion

A API do framework se beneficia - em especial devido ao uso substancial de tipos

genéricos - da inferência de tipos do SourceKit/LSP e do compilador para proporcionar sugestões de *auto-complete* e permitir que os tipos fiquem implícitos no *call site*. Esse aspecto foi um fator determinante na decisão de abstrair as operações como tipos concretos, pois a inferência dos tipos não é possível com protocolos ou funções de primeira classe.

Atente que, apesar das operações serem implementadas em tipos concretos, esses não perdem a sua qualidade de interface e permitem o *command pattern*, pois esses tipos apenas empacotam uma função de primeira classe que pode ser definida durante a inicialização desses tipos. Vale notar também que *casts* não são empregados na implementação e qualquer incompatibilidade entre os tipos das operações são revelados em tempo de compilação.

5.1.3 Análise estática

Conforme visto na proposta, a adoção do framework por uma equipe de desenvolvimento abre a oportunidade para a criação de artefatos para troca de requisitos e memória de projeto. Cada vez que o desenvolvedor escreve uma chamada para API, ele está documentando no código-fonte um cenário onde o *backend* e as tecnologias que implementam a interface entre cliente e servidor não entregam o dado na sua forma final.

Para atuar como aliado do framework, foi implementada uma ferramenta simples de linha de comando que analisa o texto do código e gera um relatório das chamadas de mapeamento. A ferramenta pode ser incorporada uma *pipeline* de CI e gerar relatórios quando atualizações do código são integradas ao repositório do mesmo.

A implementação da ferramenta é baseada na biblioteca SwiftSyntax da Apple. Essa biblioteca permite a tradução de código para uma estrutura de dados em árvore chamada Abstract Syntax Tree que representa o código-fonte ao nível léxico e sintático. Ela também possibilita que o usuário especifique uma categoria sintática da linguagem e caminhe ao longo da árvore, visitando nodos daquela categoria e extraindo as informações necessárias.

A Listagem 5.4 mostra um exemplo de saída da ferramenta que foi gerada sobre a Listagem 5.8.

```
1 > \.analyzer
2 Usage: analizer <declarations_source_path> <calls_source_path>
3 > \.analyzer ..\framework\Sources\Framework\ example.swift
4 Mapping operations in example.swift:
```

```
Operation:                ErrorHandling:   Line:
----------------------------------------------
Morphism.asMandatory      -                4
Morphism.asRFC3339Date    -                9
Assertion.notBlank        -                12
Assertion.notBlank        fallback         16
Assertion.notBlank        -                19
```

Listagem 5.4 – Saída da ferramenta de análise estática feita sobre a Listagem 5.8.

5.2 Avaliação

Nesse subcapítulo, a implementação do framework é avaliada quanto à complexidade, o reuso e a testabilidade.

5.2.1 Complexidade

Nessa seção é definido um caso de aplicação implementado com e sem o uso do framework com o objetivo de demonstrar os benefícios que ele traz em termos de complexidade. O caso consiste de uma funcionalidade da tela inicial de um banco digital. Ele deminstra diferentes tipos de dados e estruturas que usualmente são utilizadas no contexto de um *frontend*. A Listagem 5.5 deminstra a modelagem de dados para o domínio da funcionalidade no cliente, ou seja, o Domain Model Object.

```
public struct HomeFeatureDMO {
 public let welcomeMessage: String?
 public let userSection: UserSection
 public let balanceSection: BalanceSection
 public let investmentsSection: InvestmentsSection?
 public let newsSection: NewsSection
 public let transactionsSection: TransactionsSection

 public typealias TransactionsSection = Result<[
   TransactionSummary], Error>
 public typealias NewsSection = [Result<NewsArticleSummay, Error
   >]
```

```
11 }
```

Listagem 5.5 – Exemplo de Domain Model Object - Página inicial

O DMO possui alguns de seus tipos de dados como a fixação de um genérico de outro tipo. São eles os tipos da biblioteca padrão do Swift `Result<T, Error>` e o `Optional<T>`, este também representado por `T?`. Assuma que optionals representam um dado opcional e results representam dados que são desejados, mas sua ausência é tolerada. Os demais dados são considerados obrigatórios para a devida aplicação do negócio.

Em um universo "ideal", o Data Transfer Object, objeto que é reflexo direto da interface com o servidor, teria tal forma que permitiria um mapeamento direto ao Domain Model Object e, portanto, a tarefa do *frontend*, no que tange a apresentar os dados na tela, resumir-se-ia a popular os componentes de interface de usuário com os dados do DTO. Entretanto, como discutido anteriormente, nem sempre essa é a realidade.

A Listagem 5.6 apresenta um possível Data Transfer Object que serve os dados para a funcionalidade em questão.

```
1 public struct HomeFeatureDTO: DataTransferObject {
2  public let message: String?
3  public let userInfo: UserInfo?
4  public let balanceInfo: BalanceInfo?
5  public let investmentsInfo: InsvestmentInfo?
6  public let news: [ArticleInfo]?
7  public let transactions: [TransactionInfo]?
8 }
```

Listagem 5.6 – Exemplo de Data Transfer Object - Página inicial

Note que os tipos no DTO do exemplo não possuem um mapeamento semântico claro com os tipos utilizados no DMO, pois todas as propriedades são consideradas opcionais (tipificado pelo uso do "`?`"). Partindo das expectativas expressas nos tipos do DMO, apenas `message` e `investmentsInfo` seriam opcionais.

Sendo assim, torna-se necessário que o desenvolvedor do cliente verifique pela ausência dos dados das propriedades do DTO que não são opcionais para a realização do negócio. A seguir são apresentadas duas listagens, a Listagem 5.7 demonstra uma intervenção ad-hoc para o tipo de problema mencionado acima e entre outros. Já a Listagem 5.8 demonstra a intervenção com a utilização do framework para o mapeamento do mesmo DTO:

```
typealias Article = HomeFeatureDMO.NewsArticleSummay
let newsDMO: [Result<Article, Error>]
if let newsArticles = homeDTO.news {
 newsDMO = newsArticles.map { article in
  let formatter = DateFormatter()
   formatter.locale = Locale(identifier: "en_US_POSIX")
   formatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZZZZZ"
   formatter.timeZone = TimeZone(secondsFromGMT: 0)
   let date: Result<Date, Error>
   if let someDate = formatter.date(from: article.date) {
    date = .success(someDate)
   } else {
    date = .failure(MappingError.mapFailed("Invalid date."))
   }
   let isBlank = { (str: String) in
    str.trimmingCharacters(in: .whitespacesAndNewlines).isEmpty
   }
   guard !isBlank(article.title) else {
    return Result<Article, Error>.failure(
     MappingError.assertionFailed("Blank content.")
   )
  }
  var summary = article.summary
  if isBlank(summary) {
   guard !isBlank(article.id) else {
    return Result<Article, Error>.failure(
     MappingError.errorHandlingFailed("Blank content.")
    )
   }
   summary = "Full article at https://news.com/\(article.id)"
  }
  return .success(
   Article(
    date: date,
    title: article.title,
    text: article.summary
```

```
    )
  )
 }
} else { newsDMO = [] }
```

Listagem 5.7 – Mapeamento ad-hoc do DTO para o DMO.

A verificação ad-hoc utiliza construções "vanila"da linguagem para realizar as verificações, não expressando padronização e a possibilidade de reuso. Um segundo desenvolvedor desenvolvendo uma segunda funcionalidade pode não entender esse trecho de código como um padrão do projeto e, portanto, diante de um cenário similar, implementar sua própria versão para aquele fim, aumentando a complexidade do código e não explorando reuso.

```
let newsDMO = try homeDTO
 .makeMappingContext()
 .focus(in: \.news)
 .map(.asMandatory)
 .mapElement { context in
  Result{
   HomeFeatureDMO.NewsArticleSummay(
    date: context.date
    .map(.asRFC3339Date)
    .getResult(),
   title: try context.title
    .ensure(.notBlank)
    .getValue(),
   text: try context.summary
    .ensure(
     .notBlank,
      onError: .fallback(to: {
        let id = try context.id
        .ensure(.notBlank)
        .getValue()
       return "Full article at https://news.com/\(id)"
      }()
     )
    )
```

```
25       .getValue()
26     )
27   }
28 }
29 .getValue()
```

Listagem 5.8 – Mapeamento do DTO para o DMO com o framework implementado.

Já, com o uso do framework, fica expresso de forma autocontida do que se trata aquelas linhas de código: a criação de um contexto de mapeamento a partir de um DTO, seguido de transformações, verificações e medidas de tolerância a falhas que resultam em um DMO. O framework encapsula toda a lógica que antes estava exposta para uma única interface que exprime um propósito claro.

Outro ponto é que as verificações ad-hoc poderiam não ter um local específico na arquitetura vazando para o Presentation Layer ou outras partes da arquitetura. Nesses casos, a prática de separação de responsabilidades seria desrespeitada. A separação entre o DMO e DTO que o framework sugere poderia até mesmo não existir e o que é abstraído como DTO ser diretamente o objeto consumido no Presentation Layer.

Quando o framework é empregado, o desenvolvedor é condicionado a reutilizar as operações disponíveis sem introduzir mais complexidade nas camadas mais internas do projeto. Se uma operação não existe, o desenvolvedor precisa implementa-la através da extensão de uma entidade, já que faz parte do framework, de maneira a isolar a complexidade e estimular o reuso daquele trabalho.

5.2.2 Reuso e Testabilidade

Reuso e testabilidade são facilitados no framework através da abstração das operações de mapeamento como interfaces, ou seja, protocolos ou *wrappers* de funções de primeira classe em Swift. Essas interfaces permitem que a API permaneça a mesma para cumprir e expressar o mesmo objetivo geral, mas com diferentes estratégias.

As funções `map(_ morphism:onError:)` e `ensure(_ assertion: onError:)` expressam a capacidade de mapear e verificar um dado associado a uma medida de tolerância a falhas. Esses são objetivos invariáveis que essa API expressa, entretanto esse objetivo pode ser cumprido de diferentes formas para diferentes tipos de dados que a entrada e a saída do mapeamento podem assumir. Nesse âmbito, a

variabilidade semântica é desejada e é proporcionada pelas diferentes instâncias que os `Morphism`s, `Assertion`s e `ErrorHandling` podem assumir.

Como discutido anteriormente, essa flexibilidade viabiliza que usuários estendam o alcance do framework para atacar o mesmo problema que a API expressa, mas considerando novos cenários. Por exemplo, diante de uma propriedade de um DTO de um determinado tipo `struct` que se repete em diferentes DTOs dos serviços do Data Model Service Layer, um cliente pode implementar uma nova estratégia de mapeamento, testa-la e reaproveita-la nos diferentes serviços sem a necessidade de reescrever o mapeamento e seus testes.

6 CONCLUSÃO

Neste trabalho, foi abordado o papel crucial das interfaces de comunicação no contexto da coordenação de equipes distantes. Discutiu-se a contribuição de ferramentas baseadas em *schemas* para auxiliar nesse cenário, reconhecendo, contudo, que não são infalíveis e nem sempre aplicadas na prática. Discutiu-se também como a resolução de interfaces mal formadas demanda novos esforços de coordenação, aumentando o custo de desenvolvimento.

Considerando essa problemática, foi exemplificado como uma equipe cliente de uma interface poderia não ter o seu desenvolvimento bloqueado mesmo quando confrontada com ambiguidades nos tipos associados aos dados daquela interface. Tal desbloqueio do desenvolvimento se daria por meio de transformação e verificação dos dados como uma solução unilateral no próprio cliente. Essa abordagem, embora vantajosa por não implicar em esforço de coordenação com as demais equipes envolvidas com a interface, poderia resultar na difusão dessas transformações e verificações pelos diferentes componentes da arquitetura do cliente, aumentando a complexidade e o custo de manutenção desse software. Ademais, há o risco de que os problemas da interface sejam negligenciados, dado que o desenvolvimento continuaria.

Foi proposta, então, uma solução que consiste na padronização das transformações e verificações no cliente por meio de um framework que dirige o isolamento da complexidade citada e promove o reuso. A proposta também sugere uma solução para o negligenciamento dos problemas na interface: uma ferramenta que faz a análise estática do código e gera um relatório baseado nas chamadas da API do framework. Essa abordagem se baseia na ideia de que as chamadas à API são indiretamente o registro das divergências entre as necessidades do cliente e a realidade dos dados que são entregues. Esses relatórios podem ser alcançáveis às diferentes equipes envolvidas através de automações na *pipeline* de um CI. Adicionalmente, o framework também habilita que o tratamento de erros na execução das transformações e verificações sejam alinhadas às suas chamadas.

Como resultado deste trabalho, o framework proposto e a ferramenta de análise foram implementados em Swift utilizando apenas bibliotecas oficialmente distribuídas para essa linguagem de programação. Além disso, foi implementada uma aplicação cliente e um servidor básico com GraphQL via HTTP que simulam um caso da problemática apresentada nesse texto e sua respectiva solução com o emprego do framework. Observe que essa implementação demonstra a natureza unilateral do framework, uma vez que existe

exclusivamente no cliente, e seu caráter complementar ao atuar em conjunto com outra solução como o GraphQL Schema. Note também que, embora a implementação avaliada estabeleça mapeamentos específicos, a proposta em si não tem a intenção de fixar tais definições. Isto é, a implementação apresentada representa apenas uma possível instância da proposta. Cada implementação é encarregada de estabelecer os mapeamentos que são úteis para o seu domínio.

Uma forma para avançar nesta pesquisa consiste na avaliação do impacto do emprego do framework no processo de revisão por pares, especialmente durante a integração de novos incrementos em um repositório de código. A hipótese a ser investigada é se a clara separação dos problemas intrínsecos da implementação do cliente dos problemas derivados da interface de comunicação, promovida pelo framework, pode acelerar a revisão do incremento ao possivelmente auxiliar na identificação dos responsáveis externos dos problemas.

Outra sugestão consiste em uma simples extensão da capacidade de Registro do framework com a adição de um parâmetro obrigatório nas chamadas da API para compelir a justificativa do mapeamento dos dados. Essa documentação *in locu* também poderia ser extraída com a análise estática do código-fonte e servir de insumo técnico e motivador para a criação de tarefas aos responsáveis externos.

Também é relevante considerar se a incorporação do framework, especialmente devido à sua funcionalidade de Registro, facilita a identificação da fonte de problemas relacionados a interfaces para aqueles que não estão diretamente ligados à programação do cliente, como gerentes de projeto e equipes de controle de qualidade. Essa característica poderia tornar o processo de manutenção mais efetivo, evitando a alocação de tarefas para equipes que não possuem, ultimamente, o domínio do problema a ser solucionado.

REFERÊNCIAS

Apple Inc. **The Swift Programming Language**. 2023. Acesso em: 03 de fevereiro 2024. Disponível em: <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/aboutswift>.

BECK, K. et al. **Manifesto for Agile Software Development**. 2001. Disponível em: <http://www.agilemanifesto.org/>.

BREJE, A.-R. et al. Comparative study of data sending methods for xml and json models. **International Journal of Advanced Computer Science and Applications**, v. 9, 01 2018.

EDSTROM, J.; TILEVICH, E. Reusable and extensible fault tolerance for restful applications. In: **2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications**. [S.l.: s.n.], 2012. p. 737–744.

GOYAL, G.; SINGH, K.; RAMKUMAR, K. A detailed analysis of data consistency concepts in data exchange formats (json ‘i&’ xml). In: **2017 International Conference on Computing, Communication and Automation (ICCCA)**. [S.l.: s.n.], 2017. p. 72–77.

GraphQL Foundation. **GraphQL Documentation**. 2024. Disponível em: <https://graphql.org/>.

HERBSLEB, J. D. Global software engineering: The future of socio-technical coordination. In: **Future of Software Engineering (FOSE ’07)**. [S.l.: s.n.], 2007. p. 188–198.

ŠANDRIH, B.; TOŠIĆ, D.; FILIPOVIĆ, V. Towards efficient and unified xml/json conversion-a new conversion. **IPSI BgD Transactions on Internet Research (TIR) vol**, v. 13, 2017.

SIEVI-KORTE, O.; BEECHAM, S.; RICHARDSON, I. Challenges and recommended practices for software architecting in global software development. **Information and Software Technology**, v. 106, p. 234–253, Feb 2019. ISSN 0950-5849.

SOUZA, C. R. B. D. et al. Sometimes you need to see through walls: a field study of application programming interfaces. In: **Proceedings of the 2004 ACM conference on Computer supported cooperative work**. Chicago Illinois USA: ACM, 2004. p. 63–71. ISBN 978-1-58113-810-8. Disponível em: <https://dl.acm.org/doi/10.1145/1031607.1031620>.

TEKINERDOGAN, B. et al. Architecting in global software engineering. **ACM SIGSOFT Software Engineering Notes**, v. 37, n. 1, p. 1–7, Jan 2012. ISSN 0163-5948.

ŠIMEC, A.; MAGLIČIĆ, M. Comparison of json and xml data formats. In: **25th Central European Conference on Information and Intelligent Systems**. [S.l.: s.n.], 2014. p. 272–275.