Gustavo Lino Fernandes

Proposta de Design de Serializers para Django Rest Framework: Uma abordagem guiada dos Princípios S.O.L.I.D. para Aplicações de Gerenciamento de Equipamentos Médicos

São Paulo - SP

2024

Gustavo Lino Fernandes

Proposta de Design de Serializers para Django Rest Framework: Uma abordagem guiada dos Princípios S.O.L.I.D. para Aplicações de Gerenciamento de Equipamentos Médicos

Versão Original

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Área de Concentração: Engenharia de Software

Orientador: Prof. Dr. Jorge Risco

São Paulo

2024

FICHA CATALOGRÁFICA

[Colocar na versão final do trabalho. Obter em: https://www.poli.usp.br/bibliotecas/servicos/catalogacao-na-publicacao**]**

Nome: FERNANDES, Gustavo Lino

Título: Proposta de Design de Serializers para Django Rest Framework: Uma abordagem guiada dos Princípios S.O.L.I.D. para Aplicações de Gerenciamento de Equipamentos Médicos.

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Tecnologia de Software.

Aprovado em: / /

Banca Examinadora

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

DEDICATÓRIA

O caminho para a consolidação do conhecimento é como uma trilha que serpenteia montanha acima: repleto de desafios, pedras no percurso e momentos de cansaço, mas também pontuado por paisagens que encantam e recompensam a jornada. Não é uma estrada pavimentada ou previsível, mas cada passo nos aproxima de vistas que só se alcançam com esforço e determinação.

Este trabalho é a consolidação da experiencia obtida nesta trajetória. Portanto, dedico-o à minha família, que me suportou em todos os momentos e nas adversidades mais variadas, oferecendo sustentação e força em cada etapa. E ao meu primo Andrey Evaristo Culik, meu companheiro de trilha, nesta aventura acadêmica, que compartilhou comigo o sonho e a coragem de escalar essa nova altitude do conhecimento. Obrigado por tornarem essa jornada ainda mais significativa e memorável.

AGRADECIMENTOS

À Universidade de São Paulo – USP, à Escola Politécnica da Universidade de São Paulo – EPUSP, que disponibilizou a infraestrutura, recursos e um corpo docente dedicado à transmitir e compartilhar, com maior nível de excelencia, o recurso mais estimado que a humanidade detém: Conhecimento .

Ao PECE – Programa de Educação Continuada em Engenharia que possibilitou à acessibilidade do curso de MBA em Engenharia de Software.

RESUMO

[FERNANDES, G.L]. [Proposta de Design de Serializers para Django Rest Framework: Uma abordagem guiada dos Princípios S.O.L.I.D. para Aplicações de Gerenciamento de Equipamentos Médicos]. [2024]. [49 folhas]. Monografia (MBA em Engenharia de Software). Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo. São Paulo. [2024].

A crescente complexidade no desenvolvimento de aplicações Web, demanda estratégias claras para assegurar a manutenibilidade, que contribua com o controle sustentável da dívida técnica. Estas estratégias, sobretudo, devem ser respeitadas e compreendidas por toda a equipe responsável pelo desenvolvimento. A revisão literária mostra os desafios na aplicação de Design Patterns e dos Princípios SOLIDs, assim como as dificuldades que podem ser enfrentadas ao se trabalhar com Frameworks para o desenvolvimento de sistemas Web. Este trabalho explora a aplicação dos princípios S.O.L.I.D. para sugestão de um design de serialização e desserialização, utilizando as abstrações de Serializers do Django Rest Framework (DRF), através de um estudo aplicado para sistemas de gerenciamento de equipamentos médicos. Sob este contexto, o design proposto busca acentuar as vantagens do Princípio de Responsabilidade Único e do Princípio Aberto/Fechado, podendo ser utilizado como base para a reestruturação de códigos, ou ainda para a implementação de novos sistemas, de modo que os profissionais envolvidos se beneficiem da redução de tempo necessário para modelar e desenvolver aplicações que utilizam o DRF.

Palavras-chave: Django Rest Framework (DRF), Princípios S.O.L.I.D., Serialização e Desserialização, Serializers, Manutenção de Software, Gerenciamento de Equipamentos Médicos, Design de Software.

ABSTRACT

[FERNANDES, G.L.]. [Design Proposal for Serializers in Django Rest Framework: A S.O.L.I.D. Principles-Guided Approach for Medical Equipment Management Applications]. [2024]. [49 pages]. Monograph (MBA in Software Engineering). Continuing Education Program in Engineering at the Polytechnic School of the University of São Paulo. São Paulo. [2024].

The increasing complexity of web application development requires clear strategies to ensure maintainability and foster sustainable control over technical debt. These strategies must be understood and adhered to by the entire development team to promote best practices. A review of the literature highlights the challenges in applying Design Patterns and S.O.L.I.D. principles, as well as the difficulties encountered when working with frameworks.

This study explores the application of S.O.L.I.D. principles to propose a design for serialization and deserialization using the Serializers abstractions of Django Rest Framework (DRF). It is based on a case study applied to medical equipment management systems in the healthcare sector. Within this context, the proposed design emphasizes the advantages of the Single Responsibility Principle and the Open/Closed Principle, offering a solid foundation for code restructuring or the development of new systems.

By adopting this approach, professionals can benefit from reduced time required for modeling and developing applications with DRF, while achieving greater efficiency in data management.

Keywords: Django Rest Framework (DRF), S.O.L.I.D. Principles, Serialization and Deserialization, Serializers, Software Maintainability, Healthcare Technology Management, Software Design.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

[API	Application Programming Interface]
[EC	Engenharia Clínica]
[EM	Equipamento Eletromédico]
[DRF	Django Rest Framework]
[OCP	Open/Closed Principle (Princípio Aberto/Fechado)]
[SRP	Singles Responsability Principle (Princípio de Responsabilidade Única)]

SUMÁRIO

1. INTRODUÇÃO

A reutilização de código como uma estratégia de Engenharia de Software tomou força e foi reconhecido em meados dos anos 2000 como norma a ser utilizada em novos negócios, motivada por menores custos de desenvolvimento e manutenção. A visão dos softwares como ativos valiosos para as empresas, e a necessidade de entregas com valor agregado cada vez mais ágeis, trouxe o reuso como ferramenta para aumentar o retorno sobre investimento (MARTIN, 2017).

Nesse cenário, os *frameworks* de aplicações em software, emergiram como uma ferramenta essencial, oferecendo aos desenvolvedores uma base modular reutilizável, com diretrizes projetadas para aplicações recorrentes em um determinado domínio. Seu uso permite que seja despendido maior foco para a inovação, enquanto os aspectos estruturais estão pré-definidos.

A utilização destes frameworks, como o *Django Rest Framework*, entretanto, traz consigo desafios inerentes à sua construção para a depuração das aplicações baseadas neles, uma vez que o nível de abstração e as relações entre os métodos exigem um tempo maior de aprendizado e conhecimento sobre a arquitetura a ser reutilizada (Schmidt, Gokhale, & Natarajan, 2004). Junto a isso, um custo significativo para compreender suas abstrações ou avaliar se um determinado componente é adequado para ser reutilizado, considerando a individualidade de cada projeto e seu próprio contexto de requisitos.

Projetos que estejam vinculados ao contexto do Gerenciamento de Equipamentos Médicos, trazem ainda requisitos particulares da profissão e mercado de Engenharia Clínica, onde é fundamental controlar informações sobre o ciclo de vida útil de equipamentos médicos, desde o planejamento da sua aquisição e instalação, até sua manutenção, desinstalação e descarte.

Assim, o desafio de entregar dados com rastreabilidade técnica, e detalhados de maneira uniforme entre os diferentes usuários, sejam eles técnicos e engenheiros de manutenção, gerentes de engenharia, médicos ou enfermeiros, impõe com mesma origem, desafios para manutenibilidade, e análise de soluções nos modelos de desenvolvimento de softwares que serão utilizados em atividades de Engenharia Clínica.

Assim, uma estratégia de análise que entregue um modelo documentado com o uso de diagrama de classe, facilitará a padronização da solução a ser adotada entre

diferentes desenvolvedores, assegurando que a tomada de decisão ao longo do uso do framework e do desenvolvimento seja guiada sob a ótica da manutenibilidade e uso do software.

1.1 Motivações

A crescente complexidade dos sistemas de software e a rápida evolução tecnológica têm evidenciado a importância da manutenção no ciclo de vida do desenvolvimento. Estudos indicam que a manutenção é uma das atividades que mais demandam esforços, consumindo entre **50% e 75%** do esforço total de um projeto de software típico (Krasner, 2022). Esse cenário é agravado pela **dívida técnica**, que representa o custo adicional, gerado por soluções rápidas e temporárias, que não seguem as melhores práticas de engenharia de software.

Segundo pesquisas, em média, **36% do tempo de desenvolvimento** é perdido devido à dívida técnica (Besker, Martini, & Bosch, 2017). Esse tempo desperdiçado é frequentemente empregado em análises adicionais do código-fonte e refatorações, impactando negativamente a produtividade e a capacidade de entrega de valor das equipes. Além disso, estimativas financeiras apontam que a dívida técnica global alcançou valores na ordem de **US$ 1,52 trilhão**, evidenciando o impacto econômico significativo associado a práticas de desenvolvimento inadequadas.

O uso de *design-patterns* pode levar à problemas de designs, se utilizados da maneira equivocada (Damyanov, Hristov, & Varbanov, 2024). Assim, saber reconhecer, quando e como aplicar este conceito, se mostra um desafio maior que o seu estudo e compreensão, propriamente dito. Analogamente, a aplicação dos princípios S.O.L.I.D. sem uma orientação guiada, pode impactar negativamente a qualidade do software desenvolvido por profissionais menos experientes, tornando a tentativa promissora de aplicação dos princípios, em uma ação prejudicial. Decisões mal tomadas podem levar ao aumento de complexidade da aplicação, alta sensibilidade a mudanças de contexto e baixa adaptabilidade. Tais características contrariam os benefícios almejados pelo reuso de código.

No contexto do desenvolvimento de APIs utilizando Django Rest Framework, a inadequada estruturação dos artefatos e classes responsáveis pela conversão das informações recebidas do usuário pelo sistema (desserialização), e do sistema para o usuário (serialização), pode contribuir significativamente para o aumento da dívida

técnica e dos desafios de manutenção. Os componentes que assumem essa responsabilidade, chamado de Serializers, que não seguirem princípios sólidos de design, resultarão em código acoplado, difícil de manter e evoluir.

Empresas do setor de Saúde investem em média 7.5% de seu faturamento em projetos de desenvolvimento de software e hardware, 2% a menos que a média de investimento dos demais setores da economia brasileira (ABES - Associação Brasileira das Empresas de Software, 2024). Estes desafios com dívida técnica ao longo do desenvolvimento, a baixa manutenibilidade e escalabilidade, especialmente em empresas de pequeno porte no setor saúde, podem levar ao fracasso do projeto.

1.2 Objetivo

Visando padronizar a tomada de decisão durante o desenvolvimento de Serializers utilizando o Django Rest Framework, em pequenas empresas de tecnologia para o setor da saúde, o objetivo deste trabalho será:

Definir, como solução de optimização de design, um modelo de diagrama de classe para serializers baseado nos Princípios SOLIDs: Reponsabilidade Única e Aberto/Fechado

1.3 Justificativas

A realização deste trabalho busca enfrentar problemáticas práticas no desenvolvimento de aplicações utilizando o *Django Rest Framework*. A padronização da tomada de decisão durante o desenvolvimento de componente chaves, como serializers, fundamentada nos princípios S.O.L.I.D., traz benefícios significativos, especialmente para projetos complexos e com restrição de recursos, como no setor da saúde.

Ao identificar desvios dos princípios S.O.L.I.D. em designs de serializers, o estudo destaca as práticas inadequadas que podem comprometer a qualidade do software. Essa identificação, permite que os desenvolvedores reconheçam padrões de código que prejudicam a manutenibilidade da aplicação, aumentando acoplamento, complexidade, e levando a sistemas menos flexíveis. Com este reconhecimento, é possível corrigir os desvios antecipadamente, durante a etapa de modelagem do desenvolvimento das funcionalidades.

Em seguida, a definição de modelos padronizados de solução para os serializers oferece exemplos claros e consistentes para os desenvolvedores. Esses modelos facilitam a implementação correta dos princípios S.O.L.I.D., superando barreiras relacionadas ao nível de experiência dos profissionais envolvidos. Como resultado, espera-se uma redução no tempo de desenvolvimento e, junto a isso, a melhora na colaboração entre equipes, visto que haverá soluções pré-estabelecidas a serem adotadas.

A aplicação dessa estratégia contribui diretamente para a manutenibilidade e escalabilidade das aplicações. Com um código aderente aos princípios de design, torna-se mais prático introduzir novas funcionalidades e realizar adaptações a mudanças de requisitos, diminuindo a incidência de dívida técnica ao longo do tempo no projeto. Impactando, com isso, o retorno sobre investimento, e contribuindo para a sustentabilidade do projeto a longo prazo, permitindo que as organizações respondam de forma eficaz às demandas crescentes por soluções inovadoras e de alta qualidade no setor da saúde.

1.4 Metodologia

Este trabalho utilizou o estudo de caso aplicado, através da abordagem investigativa e descritiva para a compreensão e formulações de padrões, guiados para a escolha de design no desenvolvimento de sistemas de gerenciamento de equipamentos médicos que utilizam DRF.

2. TEORIA E CONCEITOS

O processo de desenvolvimento de aplicações acessíveis via internet, baseia-se largamente em uma arquitetura que separa as reponsabilidades de interação com o usuário, do processamento das informações. Construindo duas grandes áreas: *front-end* (“parte da frente”*) e back-end (*“parte de trás”*).*

Estas áreas podem ser enxergadas como camadas de diferente proximidade com o usuário, onde o *front-end* se concentra nas informações que os usuários podem ver, acessar, e interagir, ou seja, a interface-gráfica do usuário (GUI – Graphical User Interface). Por outro lado, o *back-end* lida com o processamento das informações no servidor da aplicação, integrando o banco de dados para armazenar, consultar e alterar informações relevantes, podendo utilizar micros serviços para executar diferentes rotinas.

A integração entre o *front-end* e o *back-end*, pode-se dar a partir do uso de APIs (Application Programming Interface), interfaces de programação de aplicações. Em termos técnicos as *APIs* definem, como em um contrato, o formato de dados e endereço de comunicação entre dois dispositivos na internet via protocolo HTTP. (De, 2023).

O cientista da computação Roy Fielding, em sua tese de doutorado pela Universidade da Califórnia, trouxe horizontes a serem explorados pelas APIs, através do estabelecimento de um design de API que considere a escalabilidade de servidores web, para construir uma arquitetura que entregue:

- Uma interface consistente, e padronizada, permitindo que os clientes interajam com o servidor de maneira previsível.
- Modificação e evolução, separadas e independentes, entre cliente e servidor, onde o cliente lida com a interface do usuário, enquanto o servidor gerencia o armazenamento e a lógica de negócio.
- Independência do servidor para armazenar o estado da sessão de aplicações clientes, onde todas as informações relevantes são enviadas durante cada requisição.

- As respostas devem indicar se são ou não passíveis de armazenamento em memória *cache*, permitindo que clientes e intermediários armazenem respostas.
- Arquitetura composta de várias camadas, cada uma com responsabilidades distintas, em que o cliente dispensa a necessidade do contexto de que está conectado diretamente ao servidor final ou a um intermediário.
- Código executável ao cliente, estendendo sua funcionalidade, sob demanda.

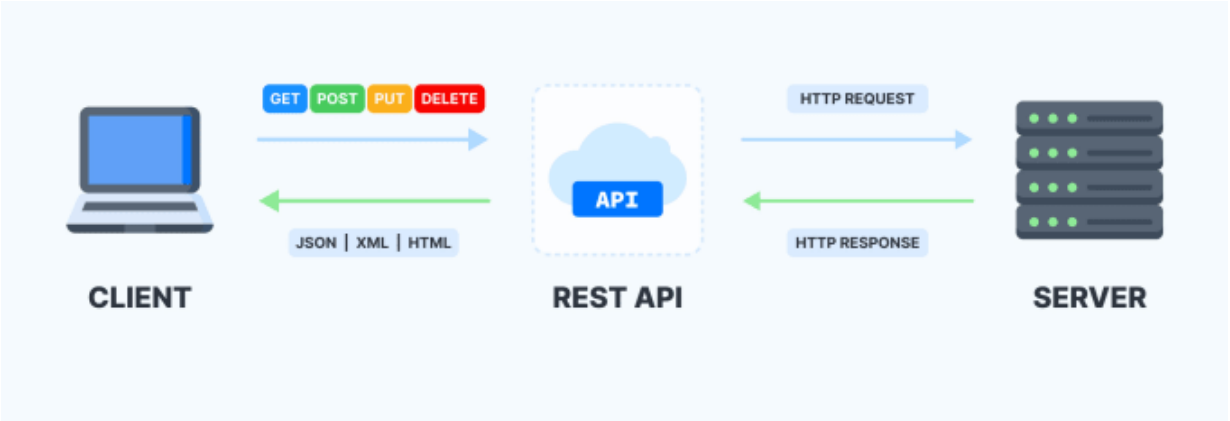


Figura 1 – Modelo REST API – Interação cliente e servidor

Esta padronização no comportamento e design de APIs, conhecida como *REST* (*Representational State Transfer),* cujo significado é “Transferência de Estado Representacional”, trouxe novas perspectivas de como a Web pode ser utilizada, muito além de armazenar e recuperar informações, evidenciando a Web como uma plataforma de aplicações (Fielding, 2000).

Com a arquitetura REST, exemplificada na Figura 1, as fronteiras entre diferentes serviços podem ser superadas, com baixo acoplamento e sistemas cada vez mais preparados para a escalabilidade.

2.1 Serializers e Django Rest Framework: Conceito e Uso

A motivação para a criação do padrão REST, através de uma arquitetura-modelo sobre como a Web deveria se comportar, que pudesse estabelecer uma plataforma

para os protocolos na web, de fato motivou a criação de bibliotecas de funções em diversas linguagens de programação como Java e Python que replicassem o conceito REST.

Tais bibliotecas, ao estabelecerem padrões arquiteturais reutilizáveis, são conhecidas como *frameworks,* integrando um conjunto de artefatos de desenvolvimento de software (classes, objetos, componentes) que colaboram entre si, aprimorando o processo de desenvolvimento através de:

- **Design reutilizável:** Guiando os desenvolvedores através dos passos necessários para criar e disponibilizar softwares de alta complexidade;
- **Implementação reutilizável:** Aproveitando o código previamente desenvolvimento, otimizando o esforço necessário e os custos durante o ciclo de vida do software.

Neste contexto, o *framework* **Django** teve seu código disponibilizado de forma aberta em 2005, entregando uma plataforma baseada na linguagem de programação Python, desenhada para tornar as tarefas comuns do desenvolvimento Web mais rápidas e fáceis de serem executadas.

Em 2011 foi criada uma extensão do Django dedicada para otimizar o desenvolvimento de Web APIs, o **Django REST Framework (DRF).** Em 2012 a versão 2.0 é disponibilizada ao público aberto.

O fluxo de informações em uma aplicação que utiliza o DRF, pode ser exemplificado conforme a Figura 2.

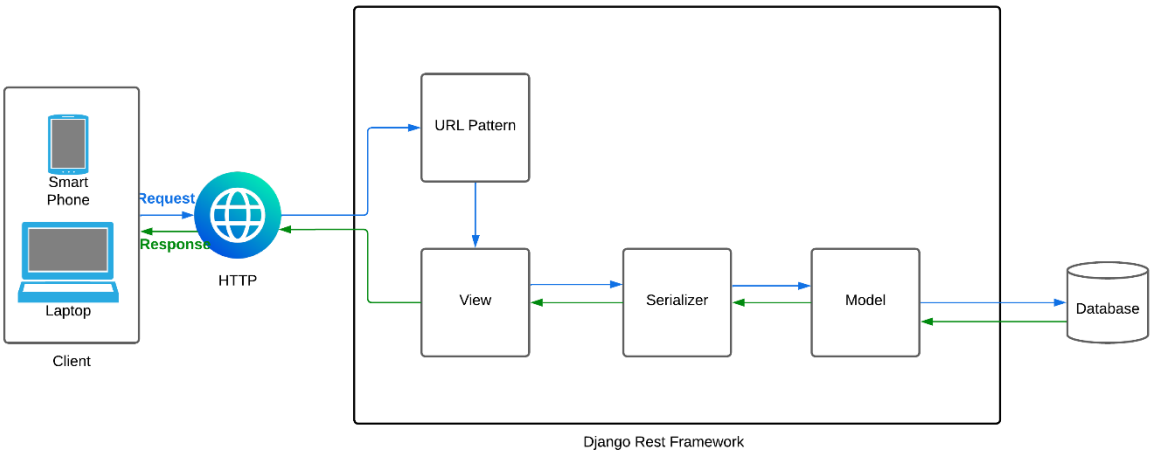


Figura 2 – Fluxo de dados em aplicações Django Rest Framework

Ao enviar uma requisição (*request*), trajeto em azul, via HTTP para um determinada *URL* (*Uniform Resource Locator*), que em português refere-se ao Localizador Uniforme de Recursos, o servidor da aplicação DRF irá tratar esta requisição através de um componente roteador de URL (*router*) que analisa o padrão do endereço (*URL Pattern*) e mapeia qual *View* que irá tratar esta requisição.

As *Views* no DRF representam a camada de apresentação e controle da API. Este artefato é responsável por receber as requisições HTTP, processá-las aplicando lógica de negócio e interagindo com outras camadas e componentes da aplicação, para então retornar uma resposta adequada ao cliente, conforme seus respectivos métodos (GET, POST, PATCH, DELETE, etc). A Figura 3 expõe através das dependências que uma View possui, tais interações entre uma requisição HTTP, a serialização (desserialização) dos dados e as repostas do servidor, em um modelo de exemplo aplicado que analisaremos em sequência suas fragilidades.

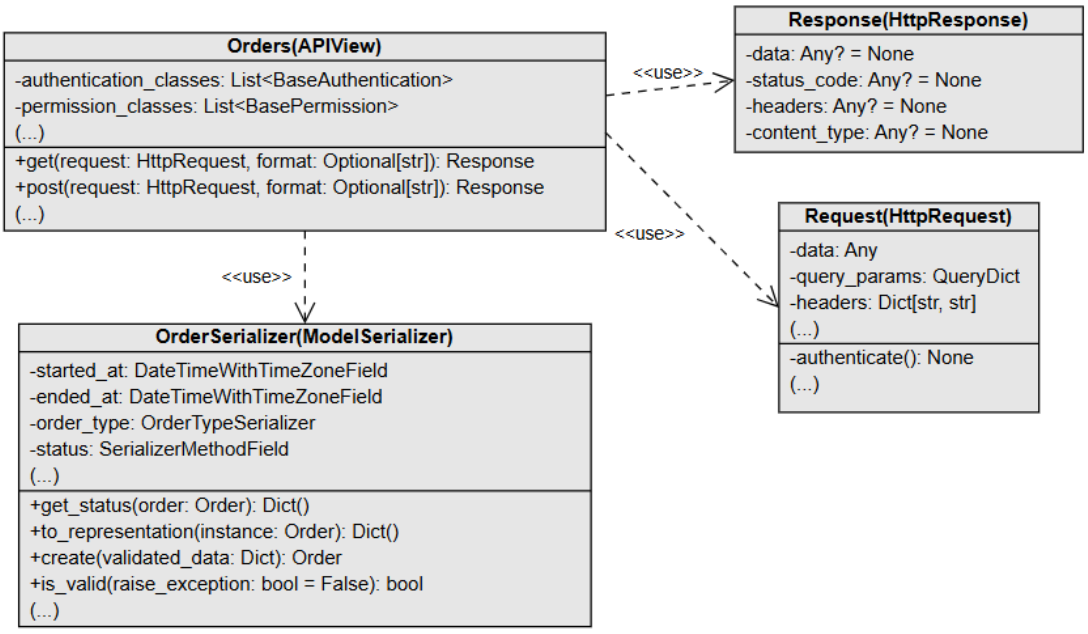


Figura 3 - Exemplo (UML) - Dependências View

Além dos parâmetros de consulta, que podem estar presentes ao final do endereço URL, em uma requisição do tipo GET, a View também será responsável por lidar e interpretar os dados de cabeçalho (headers), arquivos enviados, corpo da requisição (body), fazendo uso de serviços, assim como estruturas internas da arquitetura do DRF como os *middlewares*, e outras bibliotecas adicionais, que não iremos nos aprofundar neste trabalho.

Os serializers**,** por outro lado, convém o detalhamento e serão o foco das próximas análises, uma vez que no DRF são responsáveis por converter (serializar) os dados de objetos complexos, abstrações do Django e dados de consulta, para formatos simples, como JSON ou XML, que podem ser facilmente enviados como resposta a uma requisição de API. Eles também têm a função inversa: desserializar os dados recebidos de uma requisição, validando-os e convertendo-os de volta em objetos complexos e suas respectivas abstrações dentro do framework, para uso na lógica de negócio.

Para o DRF, pode-se destacar dois tipos de serializers:

- ***Serializer***: O tipo mais básico de serializer, onde é definido manualmente os campos que serão serializados/desserializados, entregando maior controle sobre o processo.
- ***ModelSerializer***: É uma subclasse do Serializer, mas com integração direta com os **Models** do Django. Esta subclasse gera automaticamente os campos com base no modelo especificado, simplificando o código e eliminando a necessidade de definir explicitamente cada atributo. O ModelSerializer também pode ficar responsável por persistir e atualizar os dados no banco de dados.

Os serializers, assim, são essenciais para garantir que os dados manipulados nas APIs estejam no formato correto, conforme as regras de negócio da aplicação, validando tipos de dados, formatos e obrigatoriedade de campos. Neste trabalho, consideraremos como padrão os serializers que herdam de ModelSerializer, se referindo à eles apenas como serializers.

Para o Django, os ***Models*** (Modelos) representam a camada de dados da aplicação, ou seja, são classes Python que mapeiam as tabelas do banco de dados. Cada *model* define a estrutura de um recurso no banco de dados, como um produto, usuário ou pedido, incluindo seus campos, tipos de dados e regras de validação. No DRF, os *models* são usados como a fonte principal para os dados que serão serializados e desserializados nas APIs.

.

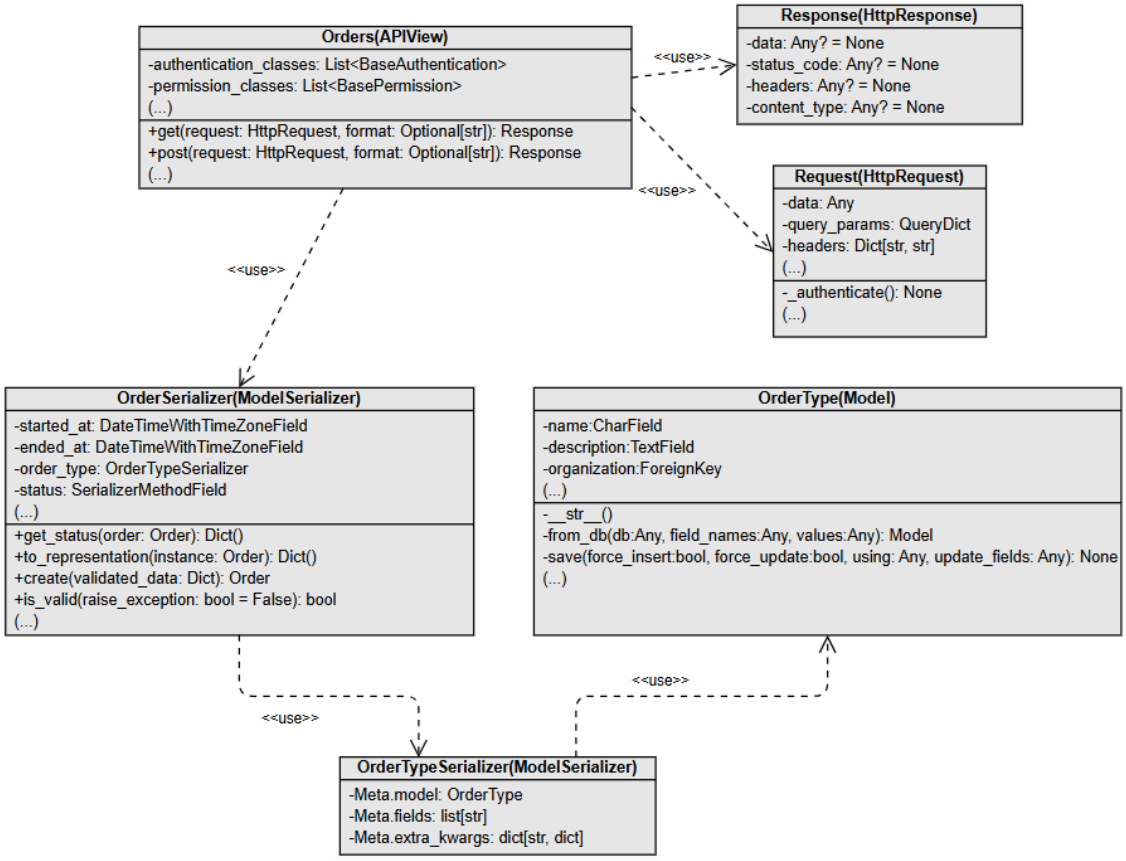


Figura 4 - Relação Serializer e Model

Um *model* típico no Django é composto por vários campos, onde cada campo corresponde a uma coluna na tabela do banco de dados. O Django fornece ainda uma série de métodos pré-definidos para facilitar o CRUD (Create, Read, Update, Delete) e outras interações com o banco de dados, tornando desnecessário o uso direto de SQL, na maioria dos casos.

2.2 Gerenciamento de Equipamentos Médicos e Engenharia Clínica

Até meados de 1920, a parcela mais representativa dos atendimentos para o cuidado de pacientes e o diagnóstico de doenças era ainda realizado por visitas residências de médicos e suas equipes, munidos de pouca infraestrutura e ferramentas para intervir ou auxiliar na melhora do quadro clínico dos pacientes, embora já houvesse centros hospitalares.

Mesmo que já houvesse aparatos desenvolvidos, muitos deles pela própria comunidade científica de médicos, o uso de tais equipamentos ainda era muito

arriscado para a vida do paciente e nem sempre entregavam a eficiência necessária. Um bom exemplo pode ser obtido quando é revisitado o histórico das técnicas para auxiliar na respiração pulmonar de vítimas de afogamento ou paradas cardiorrespiratórias (Tobin, 2013). Em 1790, já havia procedimentos e aparatos para a intubação de pacientes visando manter a respiração pulmonar por pressão positiva, utilizando foles para este propósito.

Para diversos outros propósitos, ainda mais exemplos podem ser encontrados da aplicação de conceitos de engenharia na criação de equipamentos voltados para a área da saúde, o primeiro eletrocardiograma (ECG) gravado por Willem Einthoven, a aplicação de Raio-X para obter imagens ósseas por Ratcliffe e Hall-Edwards em 1896, utilizando as descobertas de 1895 de Wilhelm Conrad Röntgen (Dyro, 2004).

A partir do crescimento tecnológico-científico e o desenvolvimento de equipamentos mais seguros e eficientes, os ventiladores pulmonares por pressão negativa, conhecidos como *iron lungs* (pulmões de ferro) de Drinker-Shaw em 1928, foi um marco importante para o período, sendo um equipamento fundamental para a vida dos pacientes vítimas da poliomielite. Com isso, a importância de um setor dedicado para manutenção dos equipamentos sensíveis de apoio à saúde já se tornava uma necessidade reconhecida na época.

Segundo Ramirez & Calil (2000), o início do conceito da Engenharia Clínica pode ser associado com o primeiro curso de manutenção de Equipamentos Eletromédicos (EMs) que foi oferecido pelas forças armadas dos EUA, em 1942 na cidade de St. Louis. O avanço então da tecnologia e dos custos relacionados ao mantenimento delas, junto as preocupações quanto à segurança do paciente, principalmente relacionados à fuga de corrente e testes, fez com que o FDA (*Food and Drug Adminstratiori*) começasse a enxergar os EMs de uma maneira normalizada tal como os medicamentos, criando procedimentos de revisão, a necessidade de registros de controle, de realizar inspeções e melhorar continuamente as práticas de manufatura.

Esta entrada do FDA na fiscalização dos EMs, começou a exigir maior empenho dos fabricantes ao disponibilizar seus produtos ao mercado, e cada vez mais discussões técnicas sobre a necessidade e coerência de uma legislação acerca do tema foi motivada. Com isso a presença de engenheiros no ambiente hospitalar se tornou cada vez mais comum.

Thomas Hargest e César Cáceres foram os principais responsáveis pela criação do termo engenheiro clínico, durante a década de 70, usando o termo para a denominação do profissional responsável pelo gerenciamento dos EMs dentro de um hospital.

No Brasil, o primeiro curso dedicado a especialização para a engenheira clínica foi entre 1993 e 1995, já havia sido levantado à época a baixa eficiência no gerenciamento dos equipamentos e sua disponibilidade para uso, representando um grande desperdício do setor da saúde. A estimativa feita por Wang & Calil (1991) era de que entre 20 e 40% dos equipamentos do país estavam desativados por falta de manutenção, peças suprimentos ou instalação.

Desde então o papel da engenharia clínica vem sendo aperfeiçoado dentro das intuições de saúde, apesar de ainda pouco conhecida e regulamentada no país a profissão do Engenheiro Clínico, seja como consultor, na fabricação, ou interno à um EAS, as suas reponsabilidades podem ser exemplificadas e variar de acordo com seu foco, dentre outras atividades, as seguintes:

- Pré avaliação e planejamento para a aquisição tecnológica.
- Design, modificações e reparos de sistemas e EMs sofisticados.
- Gerenciamento do custo e da eficiência dos serviços de manutenção e calibração.
- Avaliação de qualidade e de segurança das tecnologias e suas funções.
- Inspeção de entrada e retorno dos EMs.
- Estabelecimento de padrões de comparação de desempenho dos EMs.
- Controle de inventário.
- Manutenção corretiva, preventiva e calibração de EMs.
- Coordenação dos prestadores de serviço e fornecedores.
- Treinamento de efetividade e uso dos EMs e suas tecnologias.
- Aplicação de engenharia para customização, modificações para pesquisas e avaliações.
- Elaboração do design e necessidades de arquitetura para a infraestrutura e instalação de tecnologias médicas.
- Desenvolvimento e implantação de padrões documentacionais para controle e registro, conforme protocolos exigidos por agências de regulação e acreditadoras de qualidade.

Para os propósitos deste trabalho, utilizaremos as informações relacionadas a manutenção de EMs. Abordando sob este contexto de Engenharia Clínica, o papel dos serializers e seus desafios de manutenibilidade para o registro e consulta de Ordens de Serviços digitais, em uma aplicação web especializada.

2.3 Desafios da Manutenibilidade de Serializers

A decisão de exibir ou não uma informação, deve ser pautada na necessidade de prover aquilo que será suficiente para atender aos requisitos do cliente. Informações adicionais podem impactar na segurança da aplicação, tanto do ponto de vista de ataques direcionados ao servidor, e seus dados sensíveis armazenados, quanto a segurança intelectual, uma vez que pode ser compreendida a arquitetura através da dinâmica das repostas e dos resultados obtidos.

Exigir que o servidor seja responsável por receber muitas informações também é preocupante, uma vez que esta necessidade será invariavelmente repassada ao dispositivo do usuário, ainda que em menor escala. Assim, a experiência do usuário pode ser prejudicada ao impor o preenchimento de muitos formulários e de processar excessivamente os dados no *front-end*.

Uma vez ponderado a sobrecarga no volume de informações sendo fornecidas do *back-end* ao *front-end* e vice-versa, convém analisar a sobrecarga no fluxo de dados, propriamente dito. O número de requisições necessárias ou o tempo que cada requisição demora para ser executada é um fator de preocupação, limitando os dados disponíveis em tela para interação e consequentemente a experiência das funcionalidades. Isso pode exigir uma complexidade extra para o desenvolvimento, sendo necessário prever etapas de carregamento adicionais, feedback ao usuário de que os dados estão sendo carregados e gerenciamento dos resultados parciais.

O artefato dos *serializers*, exposto no capítulo anterior, é parte fundamental nestes desafios, bem como no aperfeiçoamento de melhorias e praticidades no desenvolvimento e manutenibilidade para softwares que utilizam DRF. Assim, a Figura 5 exibe uma das possíveis relações que pode haver no gerenciamento de ordens de serviço (*Order*), que são responsáveis pelo registro de manutenções em equipamentos médicos, onde uma ordem de serviço, naturalmente, estará relacionada à um equipamento médico (*Product*).

Um equipamento médico, por sua vez, está relacionado à um modelo específico (não confundir com o artefato Model do DRF) e possui uma determinada situação de disponibilidade (*Status*), que varia ao longo da sua vida útil. O modelo de um equipamento médico é especificado pela sua relação única com uma tecnologia específica (*Technology*) e um fabricante (*Manufacturer*). Por fim, uma ordem de serviço será composta por suas diversas ações técnicas (*Action*) promovidas pelos usuários (*User*).

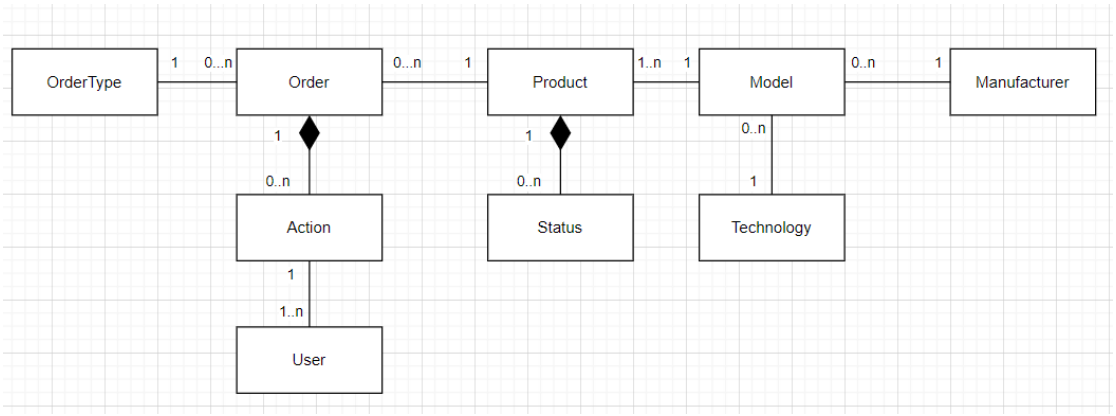


Figura 5 - Diagrama de Classe Simplificado

Nesta simples recorte de relacionamento, podemos levantar exemplos possíveis de atributos que cada classe possuiria, conforme a tabela abaixo:

Object	Fields
Order	'id', 'description', 'status', 'product', 'actions', 'user'
OrderType	‘id’, ‘name’, ‘description’, ‘organization’
Product	'id', 'name', 'serial_number', 'status', 'model', ‘manufacturer’, ‘technology’
Status (of Product)	'id', 'name'
Model	'id', 'name', 'manufacturer', 'technology'
Manufacturer	'id', 'name'
Technology	'id', 'name'
Actions	'id', 'action_type', 'performed_at', 'user'
User	'id', 'username', 'full_name'

Tabela 1 - Atributos Simplificados de Objetos em uma Ordem de Serviço

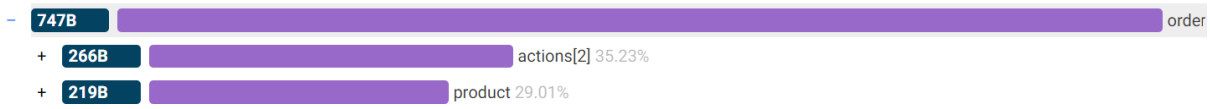


Figura 6 - Distribuição de carga da desserailização via JSON em Bytes (JSON Size Analyzer | DebugBear)

Para um *serializer* escolhido, que trata apenas dos objetos de *Actions* e *Product*, dentre os demais da estrutura vinculada à *Orders*, conforme relações da Figura 5, o JSON obtido como resposta teria um tamanho total de 747 Bytes, onde *Actions* representaria 35,23% dos bytes trafegados, mesmo em uma ordem de serviço que tenha apenas duas ações técnicas registradas. A remoção, portanto, do campo *Actions* em uma API que não necessita desta informação, promoveria uma redução de 35% no tamanho da resposta em bytes, este é uma das questões a se abordar dentro da sugestão de um design de serializers.

2.4 Princípios S.O.L.I.D

Segundo Robert C. Martin, em seu livro “Clean Architecture: A Craftsman's Guide to Software Structure and Design”, os princípios SOLID estabelecem como organizar funções e estrutura de dados em classes. Não se limitando à softwares de programação orientada a objetos, uma vez que reconhece que uma classe é um agrupamento de funções e dados, e que todo software possui estes agrupamentos, ainda que não sejam chamados de classes.

Assim, o objetivo dos princípios SOLID é criar uma estrutura de software que seja tolerante a mudanças, de fácil entendimento e com uma base de componentes que podem ser utilizados em diversos sistemas.

Através de seu no artigo “Design Principles and Design Patterns”, onde descreveu o impacto de padrões arquiteturais de aplicações, Robert expos os princípios Aberto/Fechado, Substituição de Liskov, Inversão de Dependência, Segregação de Interface e Responsabilidade Única. Posteriormente, em meados de 2004, segundo Robert, a sigla **“SOLID”**, em inglês, surgiu através de uma ideia recebida por e-mail de Michael Feathers, sugerindo reorganizar a ordem das primeiras palavras destes princípios.

A letra S, refere-se ao princípio de Responsabilidade Única (SRP - Single Responsibility Principle), onde uma classe deve ter apenas um motivo para mudar, de modo que esta mudança está diretamente relacionada à novas necessidades dos atores, que pertencem ao grupo de stakeholders ou usuários. A classe *User*, acumula responsabilidades que são descentralizadas na solução proposta pela Figura 7.

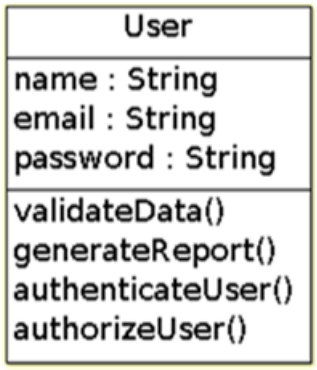


Figura 6 - Violação do Princípio de Responsabilidade Única

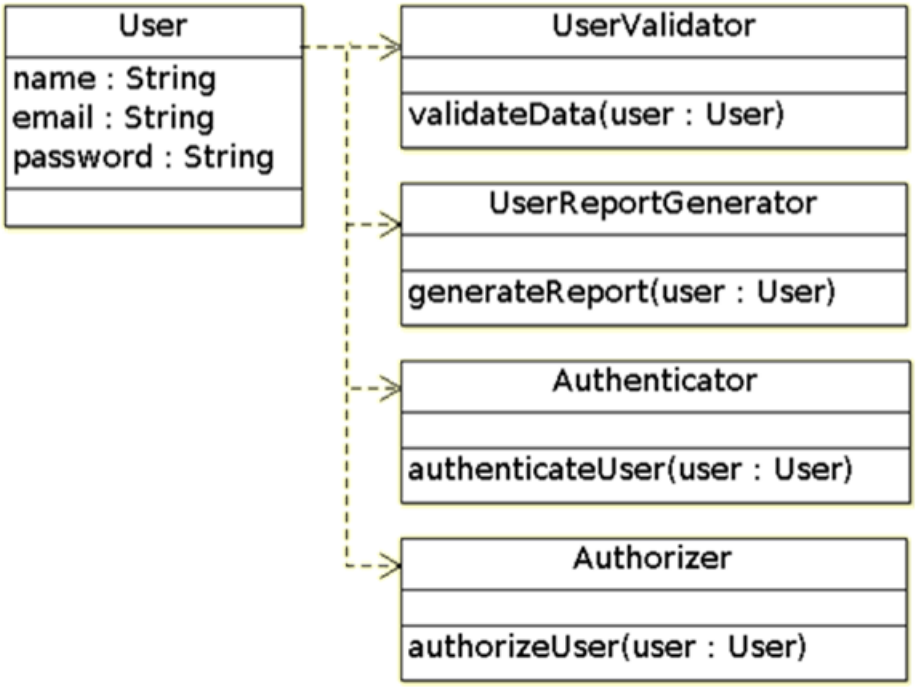


Figura 7 - Promoção do Princípio de Responsabilidade Única

O princípio Aberto/Fechado (OCP - Open/Closed Principle), contribui com a letra O, propondo que as classes devem estar abertas para extensão, mas fechadas para modificação. Isso significa que o comportamento de uma classe pode ser estendido sem alterar seu código fonte, como mostra a Figura 8, em que *Warrior*,

Magician e *Dragon* implementam de diferentes formas as ações de *Character* (*move, attack e defend*) sem alterar a definição de *Character.*

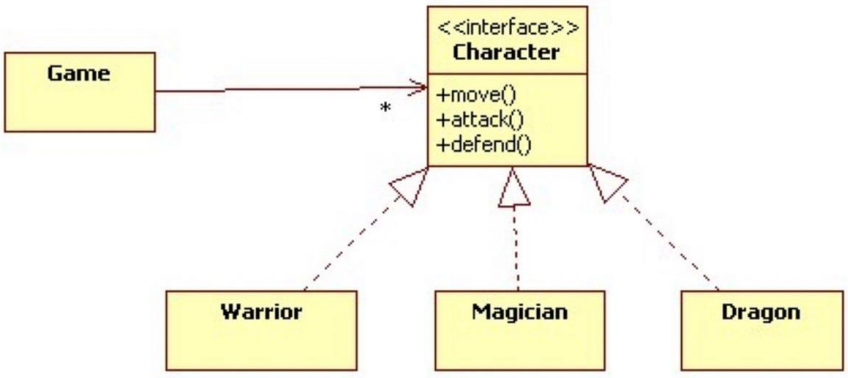


Figura 8 - Promoção do Princípio Aberto/Fechado

A terceira letra, “L”, relacionado ao Princípio de Substituição de Liskov (LSP – Liskov Substitution Principle), estabelece que objetos de uma determinada classe deve poder ser substituídos por objetos de subclasses desta. Em outras palavras, de maneira inversa, um objeto criado de uma subclasse pode ser usado no lugar de um objeto da classe superior, sustentando o comportamento esperado da aplicação sem afetar o funcionamento da lógica ou introduzindo erros como nas diferentes licenças da Figura 10. Esse comportamento não é observado no design da Figura 9, uma vez que Rectangle (retângulo) pode ter suas dimensões de altura (setH) e largura (setW) estabelecidas de forma independente, porém Square (quadrado), não.

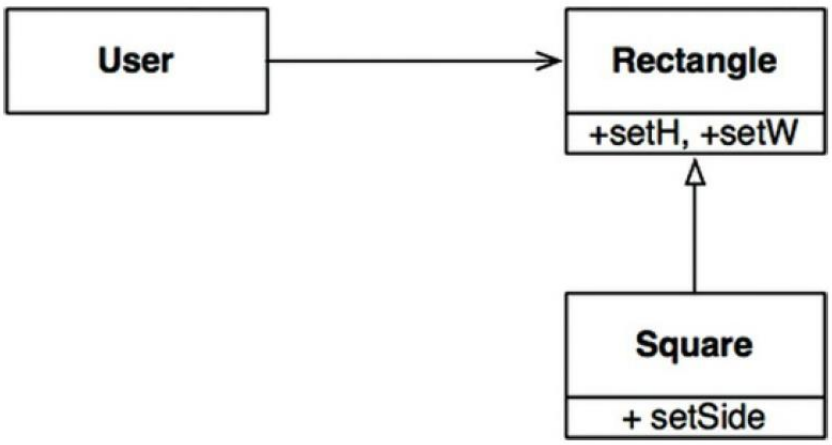


Figura 9 - Violação do Princípio de Substituição de Liskov

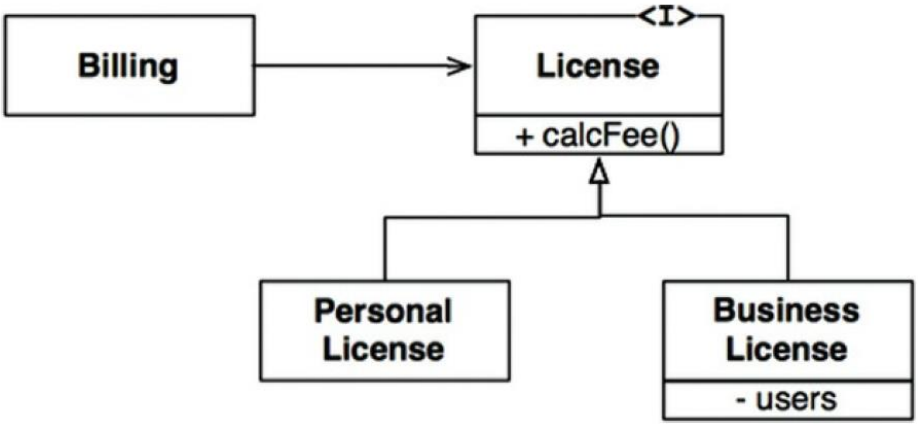


Figura 10 - Promoção do Princípio de Substituição de Liskov

A penúltima letra, “I”, refere-se ao Princípio de Segregação de Interface (ISP - Interface Segregation Principle), que entrega a relevância de evitar dependências desnecessárias através de interfaces generalistas, dando preferência à criação de interfaces de propósitos segregados. A Figura 11 mostra uma violação deste princípio ao centralizar op1, op2 e op3 para diferentes usuários ainda que estes usem apenas um dos métodos, enquanto a Figura 12 segrega as interfaces baseadas no seu uso, assegurando que caso uma alteração seja realizada em op1, não será necessário recompilar User2 e User3 em decorrência da dependência segregada.

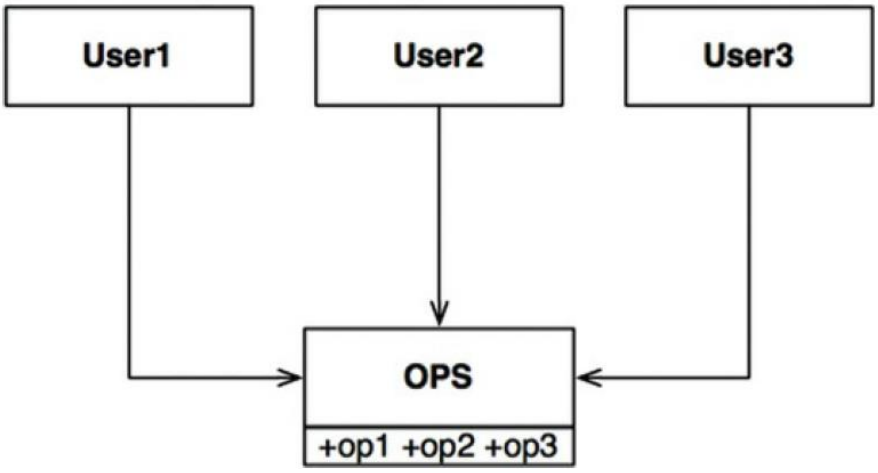


Figura 11 - Violação do Princípio de Segregação de Interface

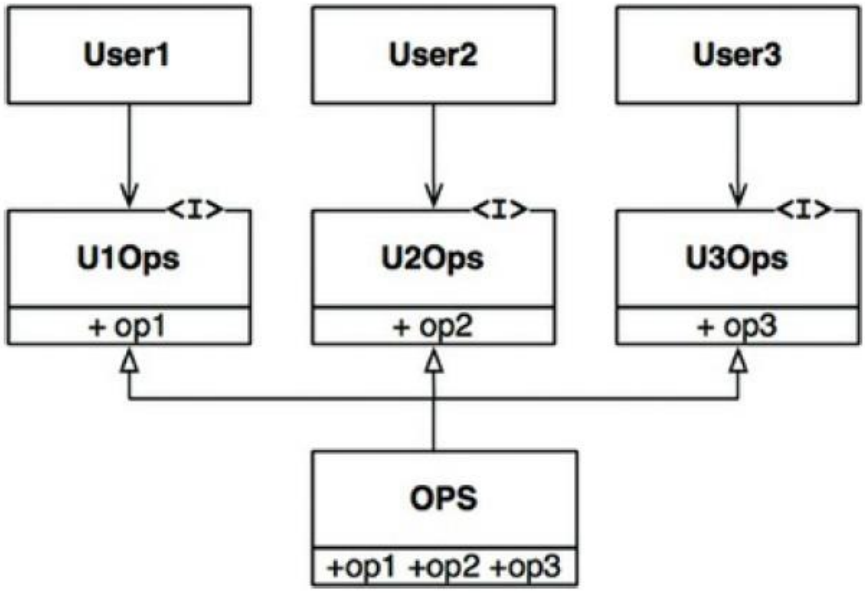


Figura 12 - Promoção do Princípio de Segregação de Interface

Por fim, a letra “D” se refere ao Princípio de Inversão de Dependência (Dependency Inversion Principle), e estabelece que os sistemas mais flexíveis serão aqueles nos quais o código desenvolvido depende apenas de abstrações e não de implementações concretas.

3. DESENVOLVIMENTO

O estabelecimento de um design base, que seja utilizado como proposta de solução para Serialização e Desserialização de dados, deve considerar a concepção da implementação do código que atenda da melhor forma os princípios SOLID.

Neste momento deve-se ter um cuidado especial em compreender a aplicabilidade das soluções de design, o DRF por atuar sob os pilares da linguagem Python, como tipagem dinâmica, não sofre das limitações de recompilação e redistribuição que o Princípio de Segregação de Interface busca resolver, evidenciando que este é um princípio relacionado mais à linguagem do que ao design das classes e solução de implementação.

O entendimento de não agregar complexidade desnecessária durante a implantação dos Serializers é fundamental para evitar que a busca pelo cumprimento dos princípios SOLIDs resulte em um código ineficiente. Deste modo, será abordado nesta recomendação, sob o ponto de vista de atender especialmente os dois princípios reconhecidos como mais relevantes para este contexto e que estão intrinsicamente relacionados:

- Princípio da Responsabilidade Única
- Princípio Aberto/Fechado

A Figura 13, expõe um recorte do modelo exemplificado anteriormente no capítulo 2.1. É observado que, independentemente do verbo utilizado no *request*, seja POST ou GET, ambos serão tratados pela mesma classe de serializer (OrderSerializer). Este componente, portanto, assumirá a responsabilidade de serializar e desserializar os dados, violando o SRP.

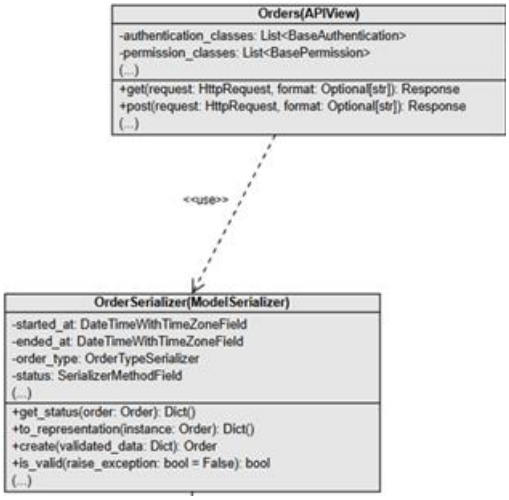


Figura 13 - Violação de responsabilidade única

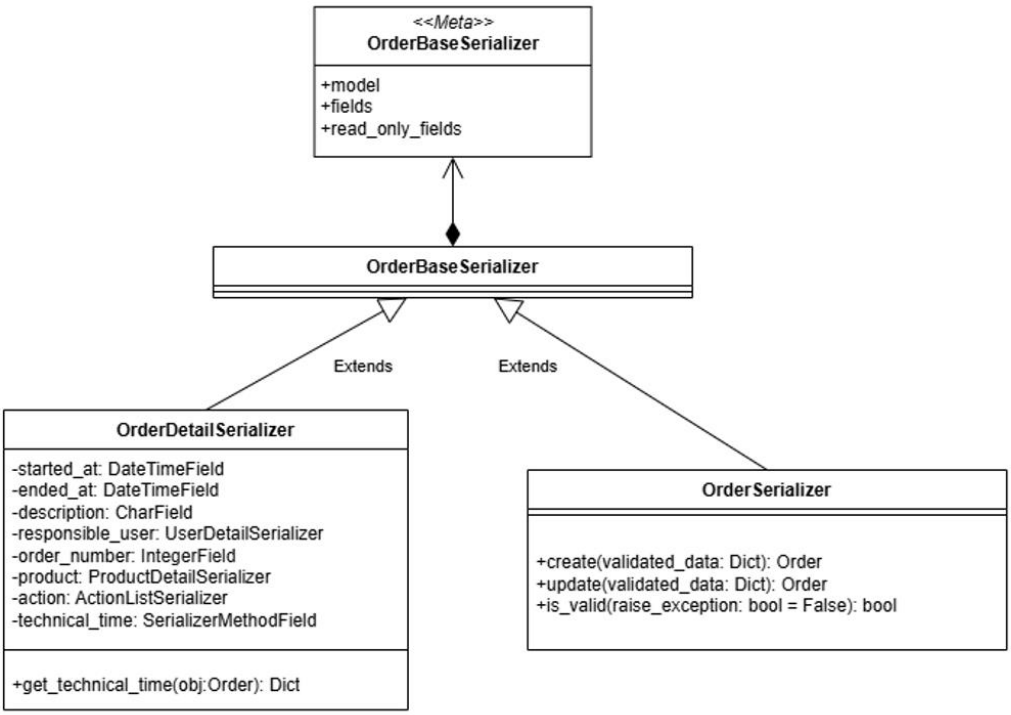
Outra fragilidade levantada pelo design exposto é a violação do OCP, uma vez que não há alternativas para detalhar melhor as informações que cada campo pode promover, ou expandir as funcionalidades ao qual o serializer está vinculado, sem alterá-lo diretamente.

Portanto, a seguir será proposta soluções que considerarão melhorias de forma progressiva, a partir da aplicação dos princípios SOLID em serializers vinculados primariamente à **Interação com o Usuário** e que possuem o objetivo de **Detalhamento de Informações.**

3.1 Serializers para Interação com o Usuário

A correspodência entre a responsabilidade do serializer e o propósito de interação com o usuário estabelece a necessidade de distinguir as responsabilidades de serializar e desserializar, primariamente, os dados.

Assim, a Figura 14 propõe um modelo diferente para *OrderSerializer*, separando as responsabildiade em *OrderDetailSerializer e OrderSerializer,* onde o primeiro é responsável pela desserialização dos modelos vinculados à uma ordem de serviço, enquanto o segundo é responsável por criar, editar e validar os dados durante a serialização de uma ordem de serviço.



.

Figura 14 - Segregação de Responsabilidade de Serialização e Desserialização (Sugestão Inicial)

Desta forma é possível assegurar que, ainda que haja mudanças provenientes na maneira com que é desejado exibir os detalhes de uma ordem de serviço, estas

mudanças não impactarão na estrutura dos campos que estão sendo gerenciados, uma vez que *OrderBaseSerializer* já estabelece o padrão de campos do modelo disponíveis para serem gerenciados e quais destes campos são apenas para leitura.

Porém, a relação de sobrecarga desnecessária exposta no capítulo 2.1**,** se mantém. A consulta de ordem de serviço dentro de uma aplicação de gerenciamento tecnológico, pode se dar em diferentes interações com o usuário, desde uma simples listagem, pelo detalhamento em formato PDF, e em diferentes visualizações (Views) relacionadas à custos de manutenção ou histórico de manutenção de equipamentos. Assim, as informações (campos) que uma ordem de serviço possui, pode ser utilizada e exibida de diferentes formas.

A criação de um serializer dedicado para os contextos em que se faz necessário detalhar **todas** as informações pertinentes à uma ordem de serviço, através de *OrderDetailSerializer*, e outro para atender os casos em que **apenas** uma listagem das principais informações é suficiente, utilizando *OderListSerializer,* assegura que a serialização é flexível o suficiente para atender suas responsabilidades particulares. Compreendendo separadamente, portanto, conforme a Figura 15, os contextos de:

1. Criação, validação e edição de dados.
2. Detalhamento.
3. Listagem.

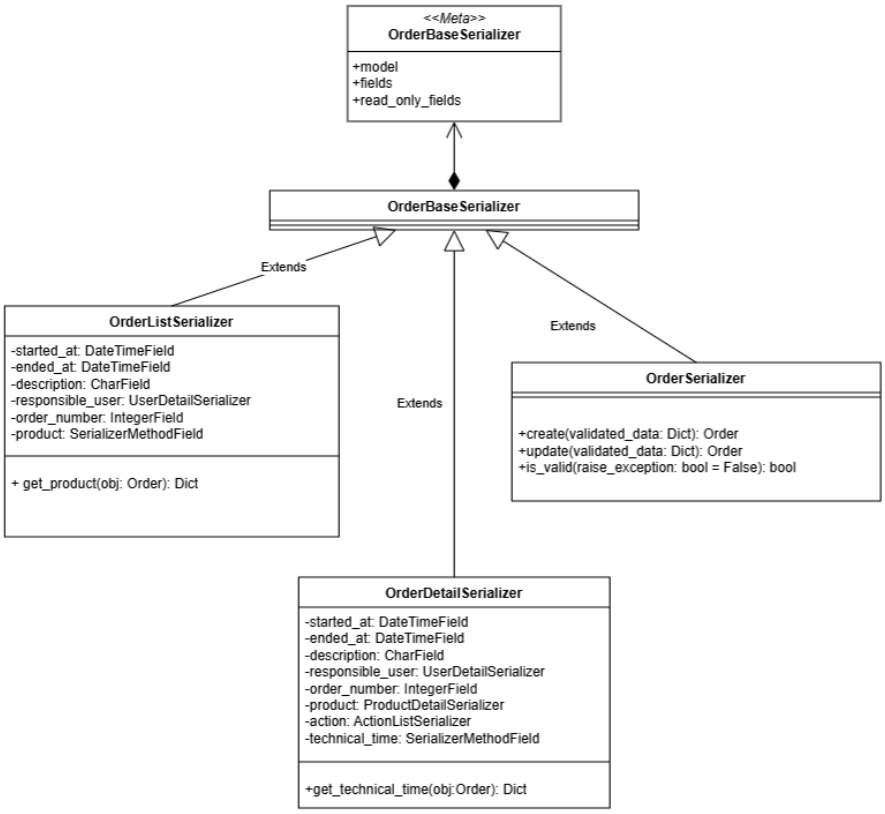


Figura 15 - Sugestão de Segregação de Responsabilidade de Serialização e Desserialização

As validações também trazem a necessidade de um olhar dedicado para assegurar o princípio de aberto para expansão e fechado para alterações. Uma vez que regras de validação costumam respeitar determinados padrões e podem ser utilizadas em diversas partes do código, como por exemplo, a comparação entre duas datas:

“A data de fechamento de uma ordem de serviço deve ser posterior à sua data de abertura”

Outras validações podem ainda assim ser necessárias, por serem específicas para em um determinado contexto, e que são incluídas (ou alteradas) ao longo do processo de melhoria contínua das funcionalidades, exemplo:

“Apenas um usuário com perfil técnico poderá ser responsável por uma ordem de serviço”

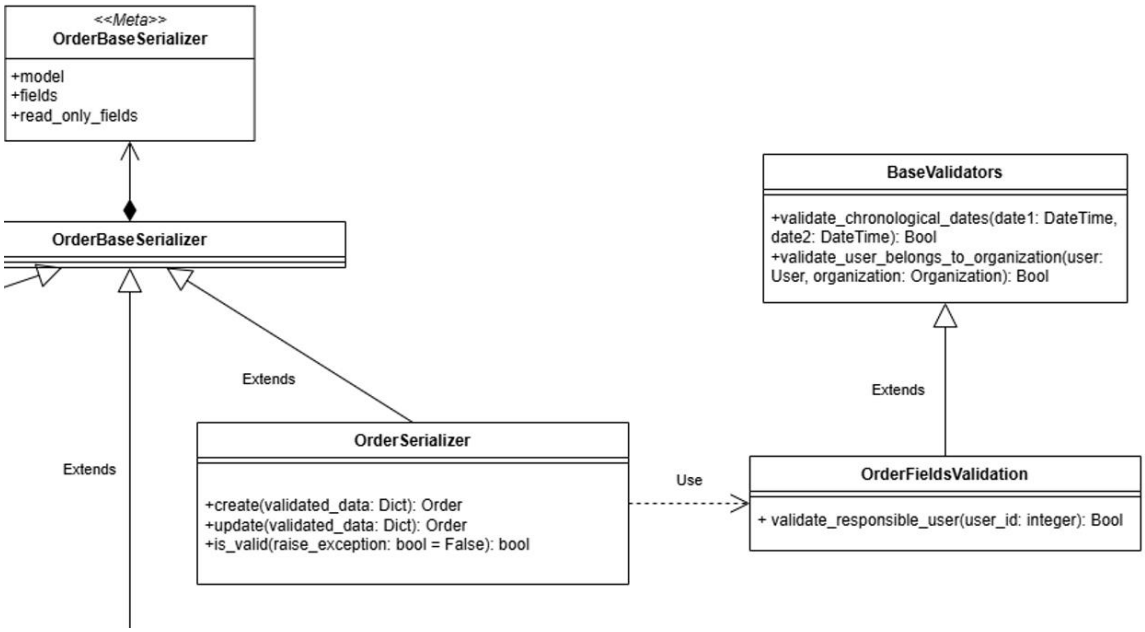


Figura 16 - Validações Genéricas e Específicas

Incluindo então uma outra classe, *OrderFieldsValidation,* vinculada ao serializer, como responsável pelos métodos de validação específicos de *OrderSerializer*, e que herda de *BaseValidators,* onde esta última possui as validações de propósitos múltiplos. De modo que OrderSerializer utilizará OrderFieldValidation de maneira padronizada, a partir da construção do método “is_valid”.

Esta abordagem é particularmente relevante pois assegura que não haverá necessidade de escrever um método específico para validar, por exemplo, que a data de fechamento é posterior à data de abertura de uma ordem de serviço, uma vez que essa lógica já está prevista em *BaseValidators* através do método “validate_chronological_dates”. Assim, as validações especificas podem ser expandidas através de *OrderFieldValidation*, para validações específicas como a que estabelece que uma ordem de serviço será criada com o vínculo apenas de usuários que possuem perfil técnico, através do método “*validade_responsible_user*”.

3.2 Serializers para o Detalhamento de Informações

Uma vez abordada as sugestões de serialização e validação dos dados, será por fim indicada as observações pertinentes à desserialização e personalização, quanto ao detalhamento destes dados para o usuário.

O DRF possui abstrações nativas fornecidas para facilitar e personalizar a forma com que os dados podem ser desserializados em uma resposta ao usuário, comumente em formato json, são elas:

- A utilização dos serializer dos próprios objetos relacionados, como o caso de *OrderTypeSerializer* apresentado na Figura 4, através do aninhamento de serializers.
- A utilização de representações específicas através das subclasses "*serializers.StringRelatedField*", "*serializers.PrimaryKeyRelatedField*" ou ainda "*serializers.SerializerMethodField*".
- Manipular diretamente a representação final, sobrescrevendo o método "*to_representation*" do serializer.

Assim, o desenvolvedor responsável deve estar ciente das escolhas que melhor se encaixarão na resposta esperada pelo usuário através da API, se restringindo a encaminhar as informações necessárias para o consumo, sem exceder e sem restringir detalhes.

Nesta condição, não convém promover extensões de representações, realizando classes como "*OrderFieldsRepresentation*" e "*BaseFieldsRepresentation*", pois a quantidade de dependências criadas impactaria na flexibilidade da representação de cada contexto, por outro lado a própria classe de *OrderDetailSerializer* pode ser responsável por personalizar a representação de seus campos e utilizar das abstrações mencionadas anteriormente para assegurar a correta formatação da resposta à API.

Através da Figura 17, observa-se a utilização simultânea das alternativas de abstrações que o DRF disponibiliza. A avaliação do desenvolvedor para identificar se *OrderDetailSerializer* deve prover também todos os detalhes que *UserDetailSerializer*

fornece em relação às informações de um usuário para o campo de “*responsible*_user” deve ser cuidadosa, considerando que *UserDetailSerializer* contém outras informações, que podem não ser pertinentes ao contexto da API.

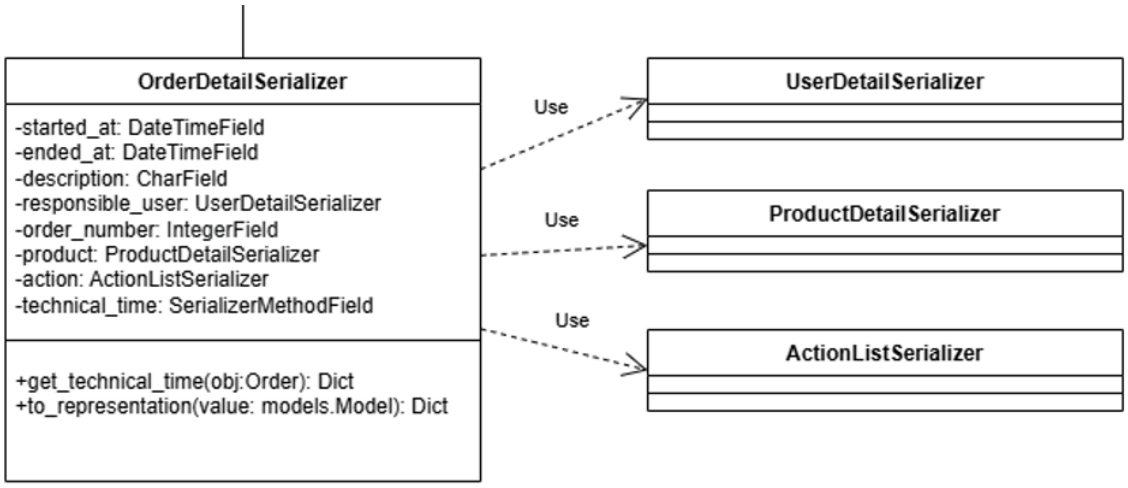
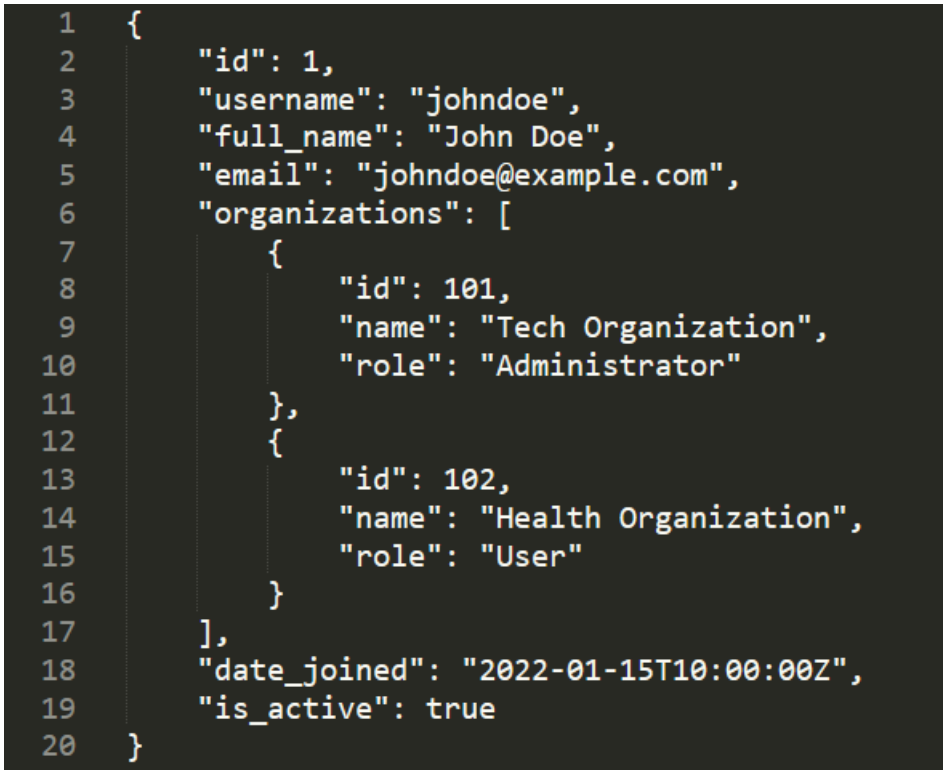


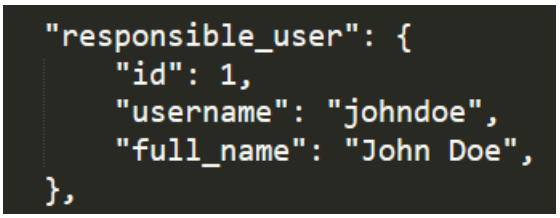
Figura 17 - Sugestão de Design para Serializers responsáveis pela Desserialização

Considerando que o *UserDetailSerializer* entrega uma resposta em formato Json, como a da Figura 17, e que informações como os campos de “*organizations*”, “*date_joined*”, “*email*” e “*is_active*”, por qualquer motivo que seja, não são desejáveis para o contexto de *OrderDetailSerializer*, a resposta esperada relacionada a este campo será apenas um recorte destes dados, representado pela resposta Json da Figura 19.



```
{
    "id": 1,
    "username": "johndoe",
    "full_name": "John Doe",
    "email": "johndoe@example.com",
    "organizations": [
        {
            "id": 101,
            "name": "Tech Organization",
            "role": "Administrator"
        },
        {
            "id": 102,
            "name": "Health Organization",
            "role": "User"
        }
    ],
    "date_joined": "2022-01-15T10:00:00Z",
    "is_active": true
}
```

Figura 18 - Exemplo de resposta padrão em Json fornecida por UserDetailSerializer



```
"responsible_user": {
    "id": 1,
    "username": "johndoe",
    "full_name": "John Doe",
},
```

Figura 19 - Campo de UserDetailSerializer utilizados por OrderDetailSerializer

Portanto, para considerar este cenário, sugere-se a utilização dos serializers responsáveis pela listagem ou detalhamento dos campos, sem precisar criar classes novas destes, buscando atingir as particularidades de cada contexto. É recomendado utilizar as abstrações que Django dispõe para tratar com tais particularidade, agregando versatilidade e assegurando o cumprimento dos requisitos da API, sem excessos e faltas.

Assim, de modo a trazer uma exemplificação concreta de como estas abstrações podem ser aproveitadas, agregando aos serializers de listagem e detalhamento, a sobrescrição do método *“__init__”* e *“to_representation”,* é possível

escolher os campos que serão utilizados destes serializers, quando aninhado a outros serializers, estabelecendo um comportamento dinâmico, conforme Figura 20 abaixo.

```python
class DynamicFieldsModelSerializer(serializers.ModelSerializer):
    def __init__(self, *args, **kwargs):
        self.requested_fields = kwargs.pop('fields', None)
        super().__init__(*args, **kwargs)

    def to_representation(self, instance):
        # Gera a representação padrão
        representation = super().to_representation(instance)

        # Filtra os campos com base em `self.requested_fields`
        if self.requested_fields:
            representation = {
                key: value for key, value in representation.items()
                if key in self.requested_fields
            }
```

Figura 20 - Desserialização Dinâmica baseado no contexto de uso do Serializer

Uma vez que essa funcionalidade pode ser útil em outros serializers, convém torná-la um modelo de serializer padrão ao ser herdado, evitando o retrabalho e duplicidade de código nas classes em que for conveniente trabalhar desta forma.

Assim, *ProductDetailSerializer* herda de *DynamicFieldsModelSerializer*, e é capaz de determinar quais campos de *ProductDetailSerializer* deseja-se utilizar em *OrderListSerializer*, por exemplo.

Um possível uso simplificado de ProductDetailSerializer, poderia ser conforme a Figura 21.

```python
class ProductDetailSerializer(DynamicFieldsModelSerializer):
    class Meta:
        model = Product
        fields = ['id', 'name', 'serial_number', 'model', 'manufacturer', 'technology']
```

Figura 21 - Exemplo de adequação de Serializer para Desserialização Dinamica

E com isso, *OrderListSerializer* pode especificar quais campos de *ProductDetailSerializer* deseja realmente utilizar, no caso da Figura 22 é utilizado apenas o ID.

```
class OrderListSerializer(DynamicFieldsModelSerializer):
    product = ProductDetailSerializer(fields=["id"])
```

Figura 22 - Exemplo de Uso de Desserialização Dinâmica para OrderListSerializer

Um racional similar poderia ser feito, naturalmente, para qualquer outra classe de serializer que precise lidar com as informações de *Product*.

4. RESULTADO

A modelagem inicialmente observada sem a aplicação direta dos princípios SOLIDs como um direcionador do design dos serializers, exposto abaixo na Figura 23, demonstra que embora a arquitetura documentada pelo DRF explicite a maneira de utilização de serializers, e entregue ferramentas úteis para o desenvolvedor trabalhar de forma ágil e prática, as fragilidades abaixo ainda podem ser expostas:

- Acúmulo de responsabilidade em *OrderSerializer*, executando em uma única classe todas as interações com o usuário (GET, POST etc), de desserialização e serialização.
- Baixa manutenibilidade, impedindo que as validações, detalhamento, listagem e demais necessidades que possam ser alteradas por requisitos de usuários, sejam abertas para expansão e fechadas para modificação, exigindo alteração direta em *OrderSerializer*.
- Pouca possibilidade de adaptação e correspondência às necessidades de interação com o usuário, com informações disponibilizadas e consumidas em sobrecarga, pela serialização aninhada.

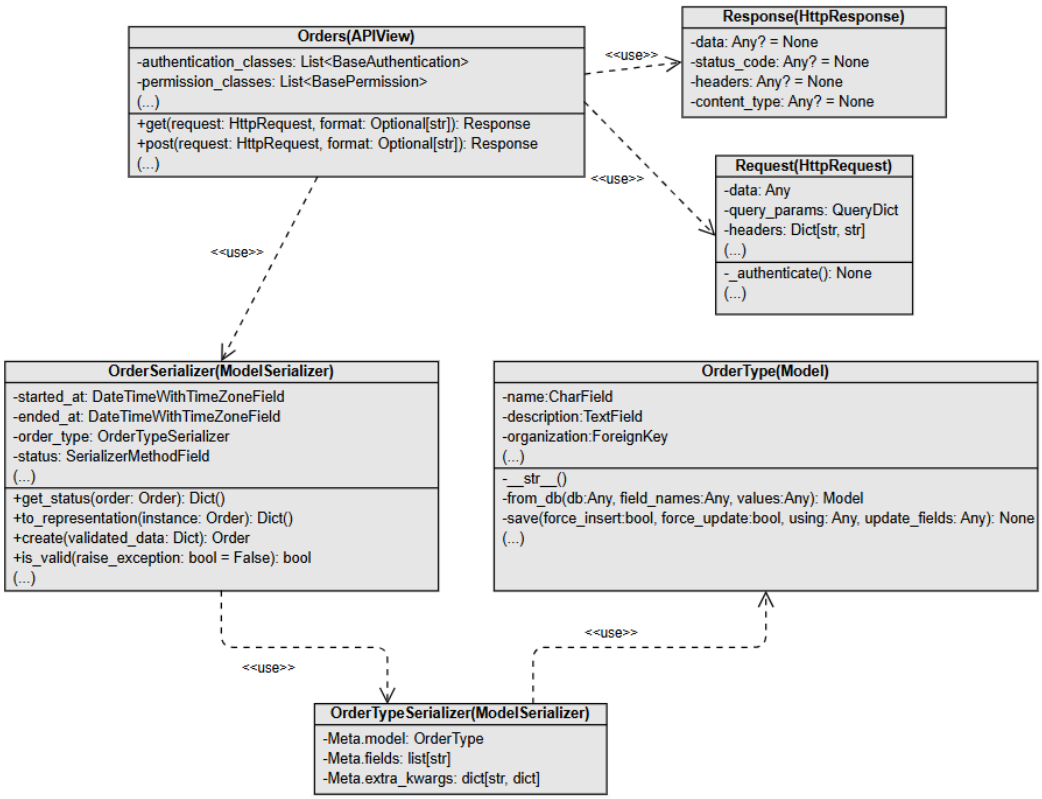


Figura 23 - Design de serializer sem aplicação dos princípios SOLIDs

De modo que, embora o framework seja bem documentado, cabe ainda ao desenvolvedor ter o cuidado necessário para a execução de um design eficiente e robusto o suficiente que evite a violação dos princípios SOLIDs.

O design proposto na Figura 23, construído no Capítulo 3, busca direcionar o desenvolvedor através de uma solução que finalmente consolida a separação da responsabilidade de desserialização e serialização, ao criar as classes de *OrderSerializer*, *OrderListSerializer* e *OrderDetailSerializer.* Além disso, entrega a possibilidade de expansão de validações específicas através de *OrderFieldsValidation,* sem a necessidade de modificar os serializers. Quanto ao controle dos dados consumidos e entregues ao usuário, o design promove o dinamismo necessário para escolher quais campos cada serializer irá entregar para a API, herdando de DynamicFieldModelSerializer, implementando a serialização aninhada ou personalizada, reforçando a aplicabilidade do Princípio de Responsabilidade Única e do Princípio Aberto/Fechado.

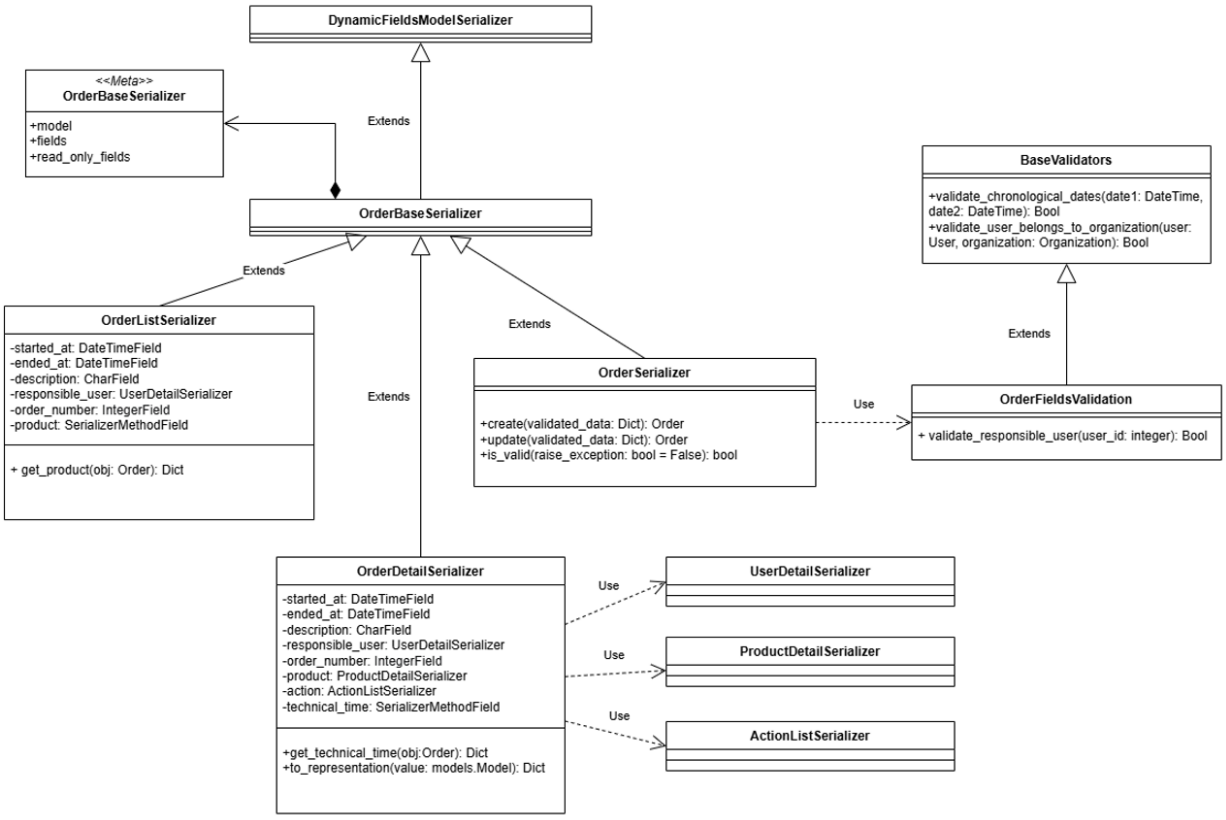


Figura 24 – Diagrama de Classes para serializers baseado nos Princípios SOLIDs: Reponsabilidade Única e Aberto/Fechado

5. CONCLUSÃO

A revisão literária promovida neste trabalho, evidenciou os desafios na manutenabilidade de softwares e os cuidados necessários para a correta interpretacão de Design Patterns e aplicação de Principios SOLID. Foi exposto o impacto que a divida técnica promove na sustentabilidade e retorno sobre o investimento, especialmente para projetos com recursos limitados.

Em um cenário onde o setor da saúde demanda por maiores investimetnos em inovação, de forma sustentável, este trabalhou entregou uma proposta de design para serialização e desserialização de dados no Django Rest Framework (DRF), guiada pelos princípios S.O.L.I.D., com ênfase na aplicação prática para sistemas de gestão de equipamentos médicos. Discutindo as estratégias que visam garantir a manutenibilidade, permeabilidade da tomada de decisão entre desenvolvedores de diferentes níveis de experiencia, promovendo a eficiência no uso de frameworks como o DRF.

A proposta demonstrou como os princípios de Responsabilidade Única e Aberto/Fechado podem ser aplicados para criar soluções mais modulares e flexíveis, reduzindo a centralização de responsabilidade e facilitando a expansão de funcionalidades. Além disso, a criação de estruturas de serializers para diferentes propósitos – como criação, detalhamento e listagem – mostrou-se uma abordagem eficiente para otimizar a compreensão do código e sua escalabilidade.

Ao propor um design que facilita a reutilização de componentes e a adoção de boas práticas, este trabalho reforça a importância de alinhar padrões de desenvolvimento à realidade dos projetos e equipes, contribuindo para entregas de sistemas mais robusto e sustentáveis.

Oportunidades futuras são levantadas com a conclusão deste trabalho, uma vez que a avaliação da efetividade do design proposto não entrou no escopo dos objetivos desta monografia, especialmente em cenários de aplicações reais, a fim de realizar a comparação com outros designs. A avaliação dos benefícios esperados quanto a adesão à escalabilidade, e o tempo necessário para o desenvolvimento das aplicações, quantificará a robustez da solução proposta e abrirá caminhos para designs ainda mais robustos.

Convém ainda, a expansão da análise para os artefatos de *Views* e *Models*, trazendo a oportunidade de unificar uma solução de design padronizada, que compreenda toda a arquitetura do DRF, não se limitando aos serializers. Por fim, o design proposto poderia ser expandido para outras áreas de aplicações além do gerenciamento de equipamentos médicos e engenharia clínica no setor da saúde.

6. REFERÊNCIAS

ABES - Associação Brasileira das Empresas de Software. (2024). *Mercado Brasileiro de Software: panorama e tendências, 2023.* São Paulo: ABES.

Besker, T., Martini, A., & Bosch, J. (2017). The Pricey Bill of Technical Debt: When and by Whom will it be Paid? *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (p. 13-23). Shangai: IEEE.

Damyanov, D., Hristov, A., & Varbanov, Z. (2024, Março). DESIGN PATTERNS OVER SOLID AND GRASP PRINCIPLES IN REAL PROJECTS. *Mathematics and Education in Mathematics*, p. 76-84.

De, B. (2023). *Organization, API Management: An Architect's Guide to Developing and Managing APIs for Your.* Bangalore,: Apress.

Dyro, J. F. (2004). *Handbook of Clinical Engineering* (1ª ed.). Academic Press.

Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architecture. Irvine: University of California.

Krasner, H. (2022). *Cost of Poor Software Quality in the U.S.: A 2022 Report.* Austin: CISQ - Consortiium for Information & Software Quality.

MARTIN, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (1ª ed.). Pearson.

Ramirez, E., & Calil, S. (2000, Dez). Engenharia Clínica: Part I - Origens (1942 - 1996). *Semina: Ci. Exatas/Tecnol.*, p. 27-33.

Schmidt, D. C., Gokhale, A., & Natarajan, B. (2004). Frameworks: Why They Are Important and How to Apply Them Effectively. *Association for Computing Machinery*, 10.

Tobin, M. J. (2013). *Principles and Practice of Mechanical Ventilation.* New York: Mc Graw Hill Medical.

Wang, B., & Calil, S. (1991, Março-Abril). Clinical Engineering in Brazil: Current Status. *Journal of Clinical Engineering*, p. 129-136.