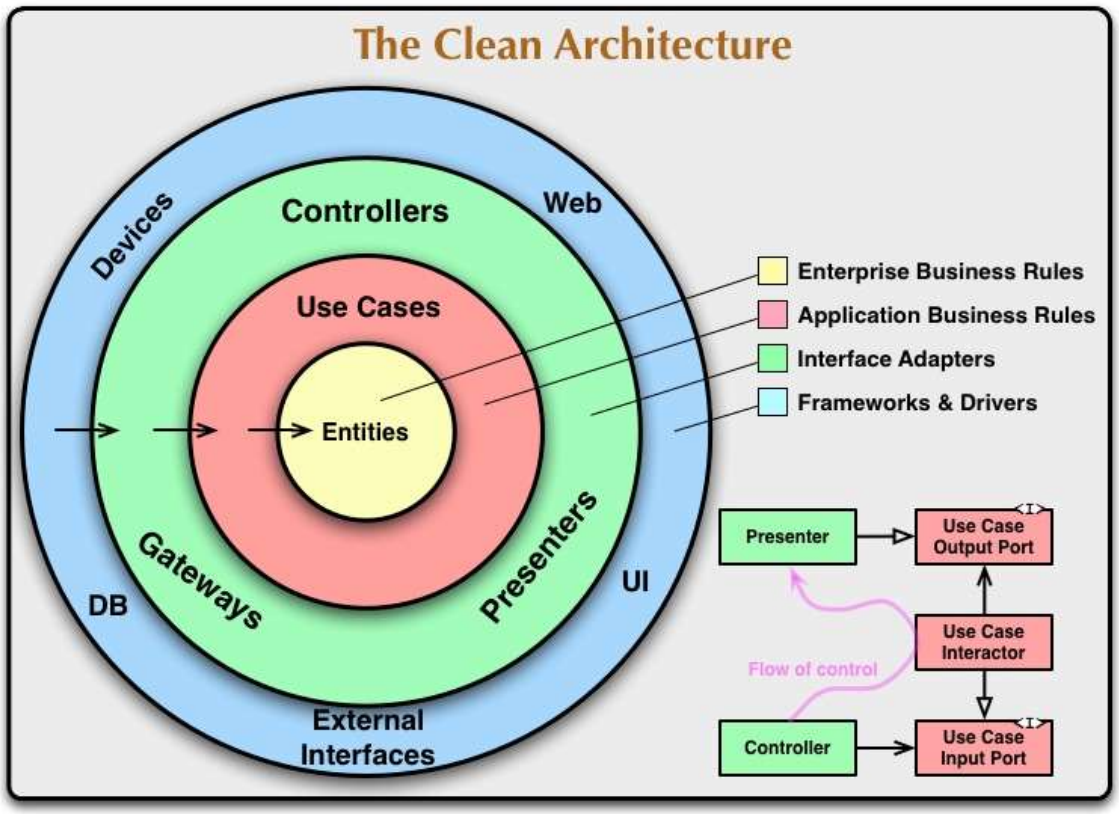
Arquitetura do Prontuário eletrônico do TCC

I- Aspéctos teóricos

A Arquitetura Limpa

(Artigo de Robert C. Martin, de 13 de agosto de 2012).



Nos últimos anos, temos visto uma grande variedade de ideias relacionadas à arquitetura de sistemas. Entre elas, podemos citar:

- A Arquitetura Hexagonal (também conhecida como Portas e Adaptadores), criada por Alistair Cockburn e adotada por Steve Freeman e Nat Pryce em seu excelente livro " Growing Object Oriented Software" (Crescendo Software Orientado a Objetos) , foi um conceito de software orientado a objetos (Amazon.com: Growing Object-Oriented Software, Guided by Tests (Addison-Wesley Signature Series (Beck)): 9780321503626: Freeman, Steve, Pryce, Nat: Livros).
- Arquitetura Cebola por Jeffrey Palermo (The Onion Architecture : part 1 | Programming with Palermo).
- Arquitetura Gritante, de um post meu do ano passado no blog.
- DCI de James Coplien e Trygve Reenskaug (Amazon.com: Lean Architecture: for Agile Software Development: 9780470684207: Coplien, James O., Bjørnvig, Gertrud: Livros).
- BCE por Ivar Jacobson, do seu livro *Engenharia de Software Orientada a Objetos: Uma Abordagem Orientada a Casos de Uso* (Amazon.com: Object-Oriented Software Engineering: A Use Case Driven Approach: 9780201544350: Jacobson, Ivar: Livros).

Embora essas arquiteturas variem um pouco em seus detalhes, elas são muito semelhantes. Todas têm o mesmo objetivo: a separação de responsabilidades. Todas alcançam essa separação dividindo o software em camadas. Cada uma possui pelo menos uma camada para regras de negócio e outra para interfaces.

Cada uma dessas arquiteturas produz sistemas que são:

1. **Independente de frameworks**: A arquitetura não depende da existência de uma biblioteca de software repleta de funcionalidades. Isso permite que você utilize esses frameworks como ferramentas, em vez de ter que adaptar seu sistema às suas limitações.
2. **Testável**: As regras de negócio podem ser testadas sem a interface do usuário, banco de dados, servidor web ou qualquer outro elemento externo.
3. **Independente da interface do usuário:** A interface do usuário pode ser alterada facilmente, sem alterar o restante do sistema. Uma interface web pode ser substituída por uma interface de console, por exemplo, sem alterar as regras de negócio.
4. **Independente do banco de dados:** Você pode substituir o Oracle ou o SQL Server por MongoDB, BigTable, CouchDB ou qualquer outra solução. Suas regras de negócio não estão vinculadas ao banco de dados.
5. **Independente de qualquer agência externa**: Na verdade, suas regras de negócio simplesmente não sabem absolutamente nada sobre o mundo exterior.

O diagrama no início deste artigo é uma tentativa de integrar todas essas arquiteturas em uma única ideia prática.

A Regra da Dependência

Os círculos concêntricos representam diferentes áreas do software. Em geral, quanto mais se aprofunda no conteúdo, mais complexo o software se torna. Os círculos externos representam **mecanismos**. Os círculos internos representam **políticas**.

A regra fundamental que faz essa arquitetura funcionar é *a Regra de Dependência*. Essa regra estabelece que *as dependências no código-fonte* só podem apontar *para dentro*. Nada em um círculo interno pode saber nada sobre algo em um círculo externo. Em particular, o nome de algo declarado em um círculo externo não deve ser mencionado pelo código em um círculo interno. Isso inclui funções, classes, variáveis ou qualquer outra entidade de software nomeada.

Da mesma forma, os formatos de dados usados em um círculo externo não devem ser usados por um círculo interno, especialmente se esses formatos forem gerados por uma estrutura em um círculo externo. Não queremos que nada em um círculo externo impacte os círculos internos.

Entidades

As entidades encapsulam regras de negócio *corporativas*. Uma entidade pode ser um objeto com métodos ou um conjunto de estruturas de dados e funções. Não importa, desde que as entidades possam ser usadas por diversas aplicações na empresa.

Se você não tem uma empresa e está desenvolvendo apenas um aplicativo, essas entidades são os objetos de negócio do aplicativo. **Elas encapsulam as regras mais gerais e de alto nível**. **São as que menos provavelmente serão alteradas quando algo externo mudar**. Por exemplo, você não esperaria que esses objetos fossem afetados por uma mudança na navegação da página ou na segurança. Nenhuma alteração operacional em qualquer aplicativo específico deve afetar a camada de entidades.

Casos de uso

O software nesta camada contém **regras de negócio específicas da aplicação**. Ele encapsula e implementa todos os casos de uso do sistema. Esses **casos de uso orquestram o fluxo de dados de e para as entidades** e orientam essas entidades a usar suas regras de negócio corporativas para atingir os objetivos do caso de uso.

Não esperamos que alterações nesta camada afetem as entidades. Também não esperamos que esta camada seja afetada por alterações em elementos externos, como o banco de dados, a interface do usuário ou quaisquer frameworks comuns. Esta camada está isolada dessas preocupações.

Contudo , **esperamos que alterações no funcionamento da aplicação afetem os casos de uso e, consequentemente, o software nesta camada**. Se os detalhes de um caso de uso mudarem, certamente algum código nesta camada será afetado.

Adaptadores de interface

O software nesta camada é um conjunto de **adaptadores que convertem dados do formato mais conveniente para os casos de uso e entidades**, **para o formato mais conveniente para alguma agência externa, como o banco de dados ou a Web**. É esta camada, por exemplo, que conterá integralmente a arquitetura MVC de uma interface gráfica. Os **Presenters, Views**, e **Controllers** pertencem a esta camada. Os modelos provavelmente são apenas estruturas de dados que são passadas dos controladores para os casos de uso e, em seguida, dos casos de uso de volta para os apresentadores e visualizações.

De forma semelhante, os **dados são convertidos, nesta camada, do formato mais conveniente para entidades e casos de uso** para o **formato mais conveniente para qualquer framework** de persistência que esteja sendo utilizado, ou seja, o banco de dados. Nenhum código dentro deste círculo deve ter qualquer conhecimento sobre o banco de dados. Se o banco de dados for um banco de dados SQL, todo o SQL deve ser restrito a esta camada e, em particular, às partes desta camada que têm relação com o banco de dados.

Nessa camada também se encontra qualquer outro adaptador necessário para converter dados de um formato externo, como um serviço externo, para o formato interno usado pelos casos de uso e entidades.

Estruturas e fatores determinantes.

A camada mais externa é geralmente composta por frameworks e ferramentas como o banco de dados, o framework web, etc. Normalmente, não se escreve muito código nessa camada, exceto o código de integração que comunica com o próximo nível de complexidade.

É nessa camada que ficam todos os detalhes. A Web é um detalhe. O banco de dados é um detalhe. Mantemos essas coisas do lado de fora, onde podem causar pouco dano.

Apenas quatro círculos?

Não, os círculos são esquemáticos. Você pode descobrir que precisa de mais do que apenas esses quatro. Não existe uma regra que diga que você precisa sempre ter apenas esses quatro. No entanto, **a Regra da Dependência** sempre se aplica. **As dependências do código-fonte sempre apontam para dentro**. À medida que você se move para dentro, o nível de abstração aumenta. O círculo mais externo representa detalhes concretos de baixo nível. **Conforme você se move para dentro, o software**

se torna mais abstrato e encapsula políticas de nível superior. **O círculo mais interno é o mais geral**.

Ultrapassando fronteiras.

Na parte inferior direita do diagrama, há um exemplo de como cruzamos os limites do círculo. Ele mostra os Controladores e Apresentadores se comunicando com os Casos de Uso na camada seguinte. Observe o fluxo de controle. Ele começa no controlador, passa pelo caso de uso e termina sendo executado no apresentador. Observe também as dependências do código-fonte. Cada uma delas aponta para dentro, em direção aos casos de uso.

Normalmente, resolvemos essa aparente contradição usando o **Princípio da Inversão de Dependência**. Em uma linguagem como Java, por exemplo, organizaríamos as interfaces e as relações de herança de forma que as dependências do código-fonte se oponham ao fluxo de controle nos pontos exatos da fronteira.

Por exemplo, considere que o caso de uso precisa chamar o apresentador. No entanto, essa chamada não deve ser direta, pois violaria a **Regra de Dependência**: Nenhum nome em um círculo externo pode ser mencionado por um círculo interno. Portanto, temos o **caso de uso chamando uma interface (mostrada aqui como Porta de Saída do Caso de Uso) no círculo interno**, e o **apresentador no círculo externo a implementa**.

A mesma técnica é usada para transpor todas as fronteiras nas arquiteturas. Tiramos proveito do polimorfismo dinâmico para criar dependências no código-fonte que se opõem ao fluxo de controle, de modo que possamos estar em conformidade com *a* Regra de Dependência*,* independentemente da direção do fluxo de controle.

Que dados ultrapassam as fronteiras?

Normalmente, os dados que atravessam as fronteiras são estruturas de dados simples. Você pode usar structs básicos ou objetos de transferência de dados simples, se preferir. Ou os dados podem ser simplesmente argumentos em chamadas de função. Ou você pode compactá-los em um mapa hash ou construí-los em um objeto. **O importante é que estruturas de dados isoladas e simples sejam passadas através das fronteiras**. **Não queremos burlar o sistema e passar entidades ou linhas de banco de dados**. Não queremos que as estruturas de dados tenham qualquer tipo de dependência que viole *a* **Regra de Dependência**.

Por exemplo, muitas estruturas de banco de dados retornam um formato de dados conveniente em resposta a uma consulta. Podemos chamar isso de RowStructure. Não queremos passar essa estrutura de linha para dentro de um círculo externo. Isso violaria *a Regra de Dependência,* pois forçaria um círculo interno a saber algo sobre um círculo externo.

Assim, **quando transmitimos dados através de uma fronteira, eles sempre o fazem no formato mais conveniente para o círculo interno**.

Conclusão

Aderir a essas regras simples não é difícil e evitará muitas dores de cabeça no futuro. Ao **separar o software em camadas e seguir a Regra de Dependência**, você criará um sistema intrinsecamente testável, com todos os benefícios que isso implica. Quando alguma das partes externas do sistema se tornar obsoleta, como o banco de dados ou o framework web, você poderá substituir esses elementos obsoletos com o mínimo de esforço.

II- Arquitetura do Prontuário Eletrônico

A arquitetura do prontuário eletrônico foca na divisão do software em anéis concêntricos para separar interesses, onde camadas internas contêm políticas de alto nível e camadas externas contêm detalhes de implementação, conforme definido por Robert C. Martin (Tio Bob). No protótipo do TCC, essa separação se reflete seguinte um esquema arquitetural coerente, testável, independente de frameworks e preparado para evolução futura:

1. Visão Geral: como o prontuário eletrônico se encaixa na Arquitetura Limpa

O princípio fundamental da Arquitetura Limpa é que as dependências de código fonte devem apontar apenas para dentro, em direção às políticas de alto nível. A arquitetura do Prontuário Eletrônico adotada permite o **desacoplamento efetivo entre o frontend e o backend**. Isso garante que a lógica clínica **Registro Clínico Orientado por Problemas (RCOP)** não dependa da interface do usuário. Se a interface WEB mudar para mobile, ou se o banco de dados SQL for trocado, as regras de negócio (Entidades e Casos de Uso) permanecem inalteradas, respeitando a regra de que o círculo interno não deve saber nada sobre o círculo externo. Portanto, a estrutura segue a **Regra da Dependência**: **tudo aponta para dentro**, e nada nas camadas internas conhece detalhes externos como banco de dados, Docker, Kubernetes ou frameworks. Assim, “As dependências no código-fonte só podem apontar para dentro. Nada em um círculo interno pode saber nada sobre algo em um círculo externo” (Martin, 2012).

Outra diretriz diretriz a ser observada na arquitetura do Prontuário Eletrônico é a Testabilidade eficiente. Um dos objetivos da Arquitetura Limpa é a testabilidade das regras de negócio sem a necessidade de UI ou Banco de Dados. O Prontuário Eletrônico reforça esse ponto ao mencionar a implementação de um pipeline de CI/CD que inclui **execução de testes unitários e de integração automatizados**. A estrutura desacoplada facilita a criação desses testes, garantindo que a lógica do **Subjetivo, Objetivo, Avaliação e Plano (SOAP)** e a do **RCOP** funcionem corretamente antes mesmo da implantação nos contêineres Docker.

Com os mesmos critérios, a **Independência de Tecnologias** segue como princípio a ser observado na arquitetura do Prontuário Eletrônico. A Arquitetura Limpa visa criar sistemas testáveis e independentes de frameworks, UI, Banco de Dados e agentes externos. Nessa perspectiva, arquitetura do Prontuário Eletrônico segue as seguintes diretrizes:

- **Independência de Banco de Dados**: O Prontuário utiliza uma abordagem híbrida (SQL, para dados estruturados, e NoSQL, para textos livres/logs). A arquitetura limpa permite essa flexibilidade, pois as regras de negócio não estão vinculadas a um banco específico; o banco de dados é tratado como um detalhe.

- **Independência de Interface (UI):** O microsserviço de backend (Auth, Patient, EMR) não conhece a interface. Isso permite que diferentes frontends consumam as mesmas regras de negócio.

- **Independência de Agentes Externos**: O serviço de Inteligência Artificial (AI Service) opera como um microsserviço independente. Isso isola a lógica do

prontuário da complexidade dos modelos de IA (MLOps), permitindo que o modelo preditivo seja atualizado ou trocado sem quebrar o sistema principal.

1.1. Entidades (Regras de Negócio Corporativas)

No contexto do TCC, as Entidades (Camada Mais Interna) representam as regras de negócio corporativas que não mudam com facilidade. Aqui residem os modelos fundamentais do domínio da saúde, especificamente a estrutura do **Registro Clínico Orientado por Problemas (RCOP)** e o formato **Subjetivo, Objetivo, Avaliação e Plano (SOAP).** Logo, as Entidades encapsulam regras de negócio corporativas, encapsulam as regras mais gerais e de alto nível. Representam o núcleo clínico e conceitual do sistema, são estáveis, independentes de tecnologia e refletem o RCOP/SOAP, a camada clínica da aplicação. Portanto, segundo a Arquitetura Limpa, essas entidades não devem saber nada sobre o banco de dados ou a interface web e deve ser estruturada seguindo o método RCOP e o formato SOAP para qualificar a coleta de dados.

Componentes incluídos:

- Problema clínico (Problem)
- Estrutura RCOP
- Estrutura SOAP (S, O, A, P)
- Paciente (dados essenciais)
- Profissional de saúde
- Regras de validação clínica (ex.: campos obrigatórios, coerência narrativa)

1.2. Casos de Uso (Regras de Negócio da Aplicação)

Casos de Uso (Camada Intermediária): Esta camada orquestra o fluxo de dados para atingir objetivos específicos da aplicação. No protótipo, os casos de uso seriam as operações realizadas pelos microsserviços, como "Registrar um novo atendimento", "Autenticar usuário no *Auth Service*" ou "Solicitar sugestão de diagnóstico ao *AI Service*". Estas regras são específicas da aplicação, mas ainda independentes de como os dados são apresentados ou armazenados. Portanto, orquestram o fluxo entre Entidades e Adaptadores. Não conhecem banco de dados, UI, APIs, Docker, nem Kubernetes.

Casos de uso principais:

- **Registrar** um novo atendimento;
- **Registrar** evolução clínica (SOAP);
- **Criar**/**editar** problema clínico (RCOP);
- **Gerenciar** prontuário longitudinal;
- **Autenticar** usuário (fluxo lógico, não tecnológico);
- **Agendar** consulta;

- **Gerar** auditoria lógica (evento clínico);
- **Solicitar** análise de IA (ex.: sugestão diagnóstica).

Os Casos de Uso orquestram o fluxo de dados de e para as Entidades e não devem ser afetados por banco de dados, UI ou frameworks.

Assim, o **EMR Service** é responsável por registros clínicos, validações de qualidade e integração com o serviço de IA.

1.3. Adaptadores de Interface (Interface Adapters)

Adaptadores de Interface (Camada Externa): No Prontuário Eletrônico, a comunicação ocorre via APIs RESTful, permitindo o desacoplamento entre frontend e backend. Na ótica da Arquitetura Limpa, essas APIs atuam como adaptadores e gateways. Elas convertem os dados recebidos via HTTP (da Web ou Mobile) para o formato que os Casos de Uso (regras de negócio do prontuário) e as Entidades (RCOP/SOAP) necessitam. Logo, o software nesta camada converte dados do formato conveniente para os **casos de uso** para o formato conveniente para alguma **agência externa, como banco de dados ou Web**. Transformam dados entre o formato interno (entidades/casos de uso) e o formato externo (HTTP, JSON, DB, IA, logs).

Inclui:

- Controllers REST dos microsserviços (.NET Core);
- DTOs de entrada e saída;
- Mapeadores (DTO ↔ Entidade);
- Repositórios (interfaces implementadas externamente);
- Presenters (formatam respostas para o frontend);
- Gateways para IA, auditoria e serviços externos.

Desse modo, a comunicação entre os serviços foi estabelecida via APIs, permitindo o desacoplamento entre frontend e backend.

1.4. Frameworks & Drivers (Detalhes Externos)

Nos Frameworks e Drivers (Camada Mais Externa) residem os detalhes técnicos que a Arquitetura Limpa recomenda manter isolados. No Prontuário Eletrônico, isso inclui o banco de dados híbrido (Relacional/NoSQL), a infraestrutura de Docker e Kubernetes, e o framework .NET Core utilizados. Desse modo, tudo que pode mudar com o tempo fica aqui. Nada dessa camada é conhecido pelas camadas internas.

Inclui:

- .NET Core (framework web);
- Docker (conteinerização);
- Kubernetes (orquestração, autoscaling, self-healing);
- Bancos SQL e NoSQL;
- Registro de contêineres;

- Ferramentas de CI/CD;
- Implementações concretas de repositórios;
- Implementações concretas de gateways (IA, auditoria).

A camada mais externa é composta por frameworks e ferramentas como banco de dados, framework web. A Web é um detalhe. O banco de dados é um detalhe.

Logo, eficácia da orquestração fica a cargo da criação dos Dockerfiles e da implantação no cluster Kubernetes.

2. Esquema Arquitetural Integrado do Prontuário Eletrônico

A estrutura final fica assim:

2.1. Entidades (Núcleo Clínico)

- RCOP;
- SOAP;
- Paciente;
- Profissional;
- Problema clínico;
- Regras clínicas gerais.

2.2. Casos de Uso

- **Registrar** evolução SOAP;
- **Criar/editar** problema;
- **Autenticar** usuário;
- **Agendar** consulta;
- **Gerar** auditoria lógica;
- **Solicitar** análise de IA.

2.3. Adaptadores de Interface

- Controllers REST (Auth, Patient, EMR, Scheduling, Audit, AI);
- DTOs;
- Presenters;
- Interfaces (repositórios);
- Gateways para IA e auditoria.

2.4. Frameworks & Drivers

- .NET Core;
- Docker;

- Kubernetes;
- SQL + NoSQL;
- Pipelines CI/CD;
- MLOps;
- Implementações concretas de repositórios e gateways.

3. Por que essa arquitetura é essencial para o Prontuário Eletrônico?

- **Testabilidade total**: casos de uso e entidades podem ser testados sem banco, sem web, sem Docker.
- **Evolução garantida**: trocar SQL por NoSQL, trocar Kubernetes por outra solução, ou substituir o modelo de IA não afeta o núcleo.
- **Segurança e LGPD**: dados sensíveis ficam isolados em camadas externas, com regras clínicas protegidas no núcleo.
- **Escalabilidade real**: microsserviços podem escalar independentemente, sem violar a Regra da Dependência.
- **Conformidade com RCOP/SOAP**: o modelo clínico fica preservado no centro, imune a mudanças tecnológicas.

Como se ver, a arquitetura do Prontuário Eletrônico, seguindo os princípios da *clean architecture,* **facilita a manutenção** do Registro Clínico Orientado por Problemas (RCOP) de três maneiras fundamentais:

1. Isolamento das Regras de Negócio (RCOP/SOAP) A Arquitetura Limpa postula que as regras de negócio devem ser o núcleo do sistema, totalmente independentes de detalhes externos como bancos de dados ou interfaces de usuário. No projeto do Prontuário Eletrônico, o RCOP e o método SOAP representam essas regras de negócio centrais (Entidades).

- **Como facilita a manutenção:** Se as diretrizes do Ministério da Saúde sobre o RCOP mudarem (por exemplo, a adição de um novo campo no "Objetivo" ou uma nova regra de validação para a "Avaliação"), você altera apenas a camada de Entidades. Como essa camada não sabe nada sobre o banco de dados ou a interface web, a mudança é localizada e não corre o risco de quebrar o código da API ou da interface do usuário.

2. Independência de Frameworks e Tecnologias: O *Clean Coder* enfatiza que a arquitetura não deve depender da existência de bibliotecas ou frameworks específicos; estes devem ser tratados como ferramentas. No Prontuário, o RCOP é a lógica de negócio, enquanto o .NET Core, Docker e Kubernetes são mecanismos de entrega e execução.

- **Como facilita a manutenção:** Se no futuro for necessário migrar o sistema de .NET Core para outra linguagem, ou trocar o banco de dados relacional (PostgreSQL) por outro, a lógica do RCOP permanece intacta. As regras clínicas sobre como um problema de saúde é registrado e evoluído via SOAP não estão

"amarradas" à tecnologia de banco de dados ou ao framework web, permitindo atualizações tecnológicas sem reescrever a lógica clínica.

3. Testabilidade das Regras Clínicas: Um dos principais benefícios da Arquitetura Limpa é a testabilidade. Como a lógica do RCOP (regras de validação, estrutura do SOAP) é desacoplada da interface e do banco de dados, é possível escrever testes unitários rápidos e confiáveis apenas para essa lógica.

- **Como facilita a manutenção:** É possível validar se o fluxo do SOAP está correto (ex: garantir que uma "Avaliação" sempre tenha um "Problema" associado) sem precisar subir um servidor web ou conectar a um banco de dados real. Isso garante que a manutenção ou a adição de novas funcionalidades clínicas não introduzam erros na estrutura fundamental do prontuário.

Em resumo, ao aplicar a Arquitetura Limpa, a aplicação do Prontuário Eletrônico protege o "conhecimento médico" (RCOP) das mudanças frequentes no "mundo da tecnologia" (interfaces, bancos de dados, frameworks), tornando um sistema mais estável e fácil de evoluir ao longo do tempo. Assim, assa arquitetura utiliza a **conteinerização e microsserviços** (infraestrutura) para operacionalizar fisicamente o desacoplamento lógico proposto pela **Arquitetura Limpa**, garantindo que o núcleo clínico (RCOP/SOAP) permaneça protegido, testável e independente das oscilações tecnológicas externas.

Conclusão

À guisa de conclusão, pode-se inferir que **microsserviços e Clean Architecture se complementam no Prontuário Eletrônico** na seguinte perspectiva: **Microsserviços** e **Clean Architecture** (Arquitetura Limpa) atuam como estratégias complementares que resolvem problemas em níveis diferentes, de modo que enquanto os microsserviços organizam a **topologia do sistema** (macro), a Arquitetura Limpa organiza o **design interno do software** (micro).

Abaixo, detalha-se como essas abordagens se complementam no contexto do Prontuário Eletrônico:

1. Separação de Responsabilidades (Macro vs. Micro)

- **Microsserviços (Nível de Sistema):** O Prontuário divide o sistema em domínios de negócio independentes, como *Auth Service*, *Patient Service*, *EMR Service* e *AI Service*. Isso permite que cada serviço seja implantado, escalado e mantido separadamente usando Docker e Kubernetes.
- **Clean Architecture (Nível de Código):** Dentro de cada um desses microsserviços, a Arquitetura Limpa organiza o código em camadas concêntricas (Entidades, Casos de Uso, Adaptadores). Isso garante que as regras de negócio críticas (como o protocolo RCOP e o método SOAP) fiquem isoladas no centro, protegidas das camadas externas.
- **Complementaridade:** Os microsserviços evitam que o sistema se torne um monólito difícil de implantar, enquanto a Arquitetura Limpa evita que o código interno de cada serviço se torne uma "lasanha" difícil de manter.

2. Independência Tecnológica e Evolução

- **Decisões de Infraestrutura:** A arquitetura de microsserviços permite que o Prontuário Eletrônico utilize tecnologias heterogêneas adequadas a cada tarefa (ex: .NET Core para o backend clínico e Python para o *AI Service*). O Kubernetes orquestra essa diversidade.
- **Independência de Frameworks:** A Arquitetura Limpa assegura que o núcleo da aplicação não dependa desses frameworks ou bancos de dados. Por exemplo, o *EMR Service* utiliza uma abordagem híbrida de banco de dados (Relacional + NoSQL). A Arquitetura Limpa permite que a persistência seja tratada como um detalhe na **Camada de Frameworks e Drivers,** de modo que a lógica do SOAP não precise saber se o dado está salvo em SQL ou Mongo.

3. Testabilidade e Qualidade (CI/CD)

- **Pipeline de CI/CD:** O projeto do Prontuário Eletrônico implementa pipelines de Integração e Entrega Contínua para automatizar testes e construção de imagens Docker.
- **Facilitação dos Testes:** A Arquitetura Limpa complementa isso ao tornar o sistema intrinsecamente testável. Como as regras de negócio (Entidades e Casos de Uso) são desacopladas da UI e do Banco de Dados, é possível rodar testes unitários rápidos e confiáveis no pipeline de **CI** antes mesmo de empacotar o contêiner. Isso aumenta a confiabilidade da imagem Docker que será enviada ao registro.

4. Interfaces e Comunicação

- **Adaptadores de Interface:** No Prontuário, a comunicação entre os serviços ocorre via APIs RESTful. Na ótica da Arquitetura Limpa, essas APIs funcionam como **Adaptadores de Interface** ou **Gateways**. Elas traduzem as requisições HTTP externas para o formato que as Entidades (RCOP/SOAP) entendem, mantendo a regra de dependência onde "o código interno não sabe nada sobre o externo".

Em resumo, os **microsserviços** permitem que o protótipo escale horizontalmente e tenha alta disponibilidade via Kubernetes, enquanto a **Clean Architecture** garante que, dentro de cada contêiner, o código permaneça modular, testável e fácil de alterar sem quebrar as regras de negócio clínicas fundamentais.

4. Esquema arquitetural em ciclos concêntricos

Com base no conceito clássico apresentado por Robert C. Martin em *The Clean Architecture* e nas especificações técnicas do protótipo segue abaixo o esquema arquitetural em camadas concêntricas.

Este diagrama ilustra como as regras de negócio clínicas (RCOP/SOAP) são protegidas no centro, enquanto tecnologias como Docker, Kubernetes e Bancos de Dados permanecem nas camadas externas como detalhes de implementação.

Diagrama da Arquitetura do Protótipo (Estilo Clean Architecture)

CAMADA 4: FRAMEWORKS E DRIVERS (Detalhes Externos)

- Banco de Dados (PostgreSQL / NoSQL)

- Interface Web (Frontend React/Blazor)
- Infraestrutura: Docker & Kubernetes
- Dispositivos Externos / Navegadores

CAMADA 3: ADAPTADORES DE INTERFACE (Gateways & Controllers)

- Controllers .NET Core
- Repositórios (Interfaces)
- Serializadores JSON
- Presenters (DTOs)
- APIs RESTful

CAMADA 2: CASOS DE USO (Regras da Aplicação)

- Criar Atendimento
- Gerar Sugestão IA
- Validar Identidade
- Agendar Consulta
- Orquestrar MLOps

CAMADA 1: ENTIDADES (Regras de Negócio)

- Modelo RCOP (Problemas de Saúde)
- Protocolo SOAP (S, O, A, P)
- Regras de Domínio do Paciente
- Regras de domínio de usuário

Detalhamento das Camadas no Contexto do TCC

1. Entidades (Círculo Central - Amarelo na figura original)

- **Definição:** Representam as regras de negócio corporativas que existiriam mesmo sem software.
- **No Protótipo:** Aqui residem os conceitos fundamentais da prática clínica descritos no TCC: a estrutura do **RCOP** (Registro Clínico Orientado por Problemas) e os objetos que compõem o método **SOAP** (Subjetivo, Objetivo, Avaliação, Plano). Estas regras não mudam se o banco de dados ou a interface mudarem.

2. Casos de Uso (Círculo Vermelho na figura original)

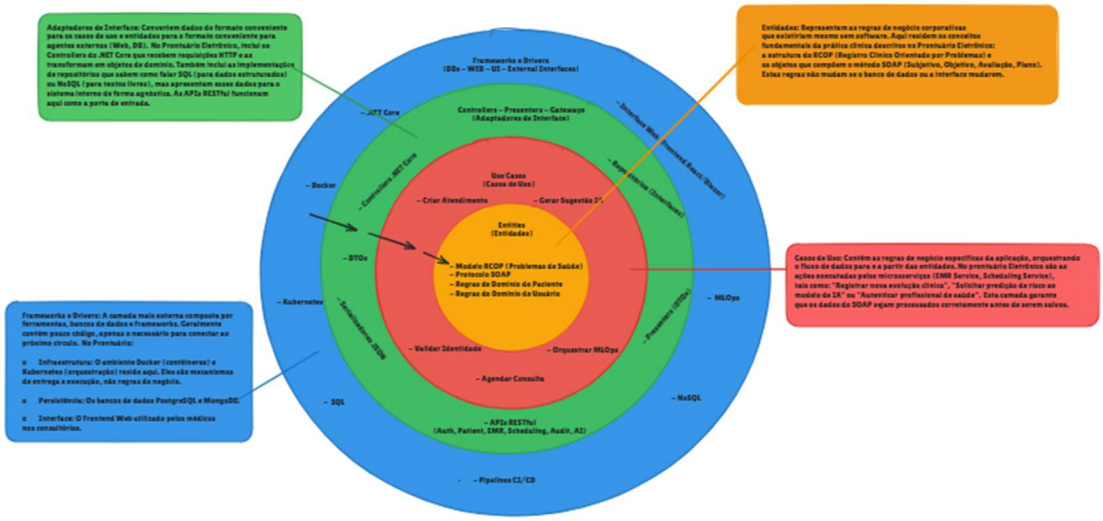
- **Definição:** Contêm as regras de negócio específicas da aplicação, orquestrando o fluxo de dados para e a partir das entidades.
- **No Protótipo:** São as ações executadas pelos microsserviços (*EMR Service*, *Scheduling Service*), tais como: "Registrar nova evolução clínica", "Solicitar predição de risco ao modelo de IA" ou "Autenticar profissional de saúde". Esta camada garante que os dados do SOAP sejam processados corretamente antes de serem salvos.

3. Adaptadores de Interface (Círculo Verde na figura original)

- **Definição:** Convertem dados do formato conveniente para os casos de uso e entidades para o formato conveniente para agentes externos (Web, DB).
- **No Protótipo:** Inclui os **Controllers do .NET Core** que recebem requisições HTTP e as transformam em objetos de domínio. Também inclui as implementações de repositórios que sabem como falar SQL (para dados estruturados) ou NoSQL (para textos livres), mas apresentam esses dados para o sistema interno de forma agnóstica. As APIs RESTful funcionam aqui como a porta de entrada.

4. Frameworks e Drivers (Círculo Azul na figura original)

- **Definição:** A camada mais externa composta por ferramentas, bancos de dados e frameworks. Geralmente contém pouco código, apenas o necessário para conectar ao próximo círculo.
- **No Protótipo:**
  - **Infraestrutura:** O ambiente **Docker** (contêineres) e **Kubernetes** (orquestração) reside aqui. Eles são mecanismos de entrega e execução, não regras de negócio.
  - **Persistência:** Os bancos de dados PostgreSQL e MongoDB.
  - **Interface:** O Frontend Web utilizado pelos médicos nos consultórios.



A Regra de Dependência: Conforme enfatizado por Martin, as dependências de código apontam apenas para dentro. O modelo RCOP (Entidade) não sabe que está rodando dentro de um contêiner Docker (Framework), nem que seus dados estão em um PostgreSQL. No entanto, o Controller (Adaptador) conhece o Caso de Uso, e o Caso de Uso conhece a Entidade. Isso permite a substituição de tecnologias externas (ex: trocar o banco de dados) sem impactar a lógica clínica do RCOP.