`projeto/feedback-geral/relatorio-alinhamento-grupo4.md`

Relatório de Desalinhamento entre Modelagem e Implementação

Projeto SimpleHealth - Sistema Integrado de Gestão Hospitalar

Data da Análise: 13 de dezembro de 2025
Analista: GitHub Copilot
Escopo: Comparação entre documentação (pasta `docs`) e implementação (pastas `simplehealth-back` e `simplehealth-front`)

📋 Sumário Executivo

Este relatório identifica **discrepâncias críticas e moderadas** entre o que foi projetado na documentação e o que foi efetivamente implementado no código-fonte do sistema SimpleHealth. Foram identificadas **25 discrepâncias principais** distribuídas em 7 categorias.

Resumo Quantitativo

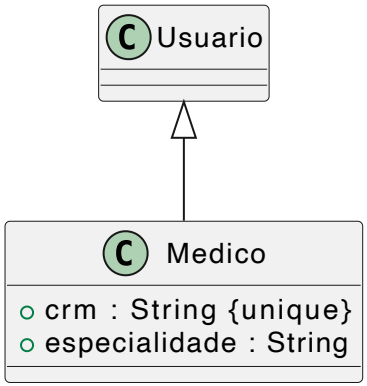
Categoria	Qtd. Discrepâncias	Severidade Predominante
Modelo de Dados	8	🔴 Crítica
Arquitetura de Persistência	3	🔴 Crítica
Camadas Arquiteturais	4	🟡 Moderada
Relacionamentos entre Entidades	4	🔴 Crítica
Casos de Uso Não Implementados	3	🟡 Moderada

Categoria	Qtd. Discrepâncias	Severidade Predominante
Nomenclatura e Padrões	2	🟢 Baixa
Atributos Faltantes	1	🟡 Moderada

1. 🔴 DISCREPÂNCIAS CRÍTICAS - Modelo de Dados

1.1 Módulo Cadastro: Hierarquia Médico vs Usuario

📘 Documentação (Diagrama de Classes):



- Médico herda de Usuario
- Usuario herda de Pessoa
- Hierarquia: Pessoa → Usuario → Medico

💻 Implementação Real:

```
// Medico.java
@Entity
```

```
public class Medico extends Pessoa {
  private String crm;
  private String especialidade;
}

// Usuario.java
@Entity
public class Usuario extends Pessoa {
  private String login;
  private String senha;
  private EPerfilUsuario perfil;
}
```

❌ **Discrepância:**

- Médico **não** herda de Usuario, herda **diretamente** de Pessoa
- Médico e Usuario são **irmãos** na hierarquia, não pai-filho
- Médico **não possui** atributos `login`, `senha` e `perfil`

💥 **Impacto:**

- Médicos não podem fazer login no sistema
- Não há gerenciamento de permissões para médicos
- Caso de Uso UC03 (Solicitar Encaixe) menciona que "Profissional de Saúde (Médico)" deve ter `temPermissaoEncaixe()`, mas Medico não herda essa funcionalidade
- Inconsistência com UC04 (Registrar Bloqueio de Agenda) que exige autenticação do Médico

✅ **Correção Necessária:**

```
// Médico deve herdar de Usuario
@Entity
```

```
public class Medico extends Usuario {
  private String crm;
  private String especialidade;
}
```

1.2 Módulo Cadastro: EventoAuditoria com Cassandra Não Implementado

📘 Documentação (Arquitetura - Seção 2.2):

```
#### Cassandra 5 (Dados de Auditoria)
**Porta**: 9042
**Entidades**: EventoAuditoria

1. **Alta Disponibilidade**: Logs de auditoria não podem ser perdidos
2. **Write-Heavy**: Milhares de eventos de auditoria por dia
3. **Time Series**: Dados ordenados por timestamp
```

💻 Implementação Real:

- ❌ Não existe classe `EventoAuditoria` em nenhum módulo
- ❌ Não há integração com Cassandra no módulo de Cadastro
- ✅ Existe Redis para pub/sub, mas não para auditoria
- ❌ Não há logs estruturados de auditoria no banco de dados

💥 Impacto:

- Rastreabilidade comprometida (quem alterou, quando, o quê)

- Requisitos de conformidade (LGPD, RN-CAD.X) não atendidos
- Impossibilidade de auditoria de acessos a dados sensíveis
- Falta de histórico para análise forense em caso de incidentes

---

1.3 Módulo Cadastro: Redis para Cache Não Implementado

📘 Documentação (Arquitetura - Seção 2.2):

```
#### Redis 7 (Cache)
**Porta**: 6380
**Por quê?**
1. **Performance**: Cache de listas de médicos disponíveis
2. **Session Storage**: Sessões de usuários logados
```

💻 Implementação Real:

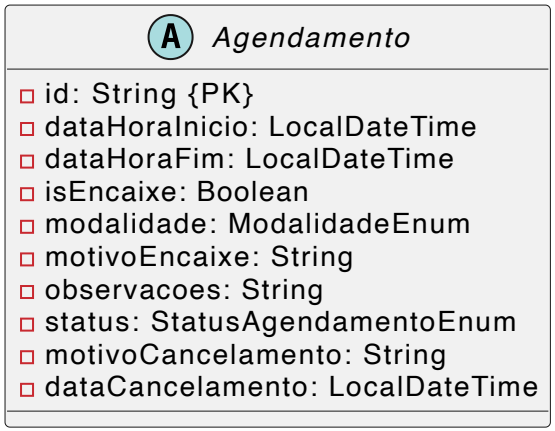
- ✅ Redis existe para **pub/sub** (EstoqueAlertaPublisher, HistoricoPublisher)
- ❌ Redis **não é usado** para cache de médicos disponíveis
- ❌ Redis **não é usado** para sessões de usuários
- ❌ Porta 6380 não está configurada (provavelmente 6379 padrão)

💥 Impacto:

- Performance degradada em consultas frequentes (lista de médicos)
- Necessidade de reautenticação constante (sem session storage)
- Aumento de carga no PostgreSQL

1.4 Módulo Agendamento: Atributos de Agendamento Divergentes

📘 Documentação (Modelagem de Classes de Projeto):



💻 Implementação Real:

```
@Document(collection = "agendamento")
public abstract class Agendamento {
  private String id;
  private LocalDateTime dataHoraAgendamento;
  private LocalDateTime dataHoraInicioPrevista;
  private LocalDateTime dataHoraFimPrevista;
  private LocalDateTime dataHoraInicioExecucao;
  private LocalDateTime dataHoraFimExecucao;
  // ... outros atributos corretos
}
```

❌ Discrepância:

- Documentação menciona apenas `dataHoraInicio` e `dataHoraFim`
- Implementação possui **5 campos temporais**:
  - `dataHoraAgendamento` (não documentado)
  - `dataHoraInicioPrevista` (vs `dataHoraInicio` doc)
  - `dataHoraFimPrevista` (vs `dataHoraFim` doc)
  - `dataHoraInicioExecucao` (não documentado)
  - `dataHoraFimExecucao` (não documentado)

📊 **Análise:**

- ✅ **Positivo**: A implementação é **mais completa** que a documentação
- ✅ Separação entre horários previstos vs executados permite tracking real
- ❌ **Negativo**: Documentação desatualizada pode confundir manutenção futura

---

1.5 Módulo Agendamento: Atributos Extras de Rastreamento

📘 **Documentação:** Não menciona atributos de rastreamento de usuários executores

💻 **Implementação Real:**

```
@Document(collection = "agendamento")
public abstract class Agendamento {
  private String usuarioCriadorLogin;
  private String usuarioCanceladorLogin;
  private String usuarioIniciouServicoLogin; // NÃO DOCUMENTADO
  private String usuarioFinalizouServicoLogin; // NÃO DOCUMENTADO
}
```

❌ **Discrepância:**

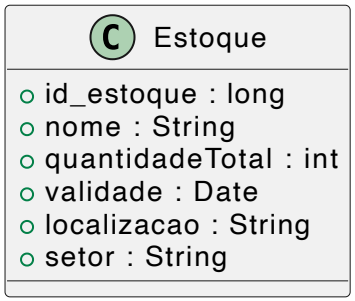
- Implementação rastreia **quem iniciou** e **quem finalizou** o serviço
- Documentação só menciona criador e cancelador

✅ **Avaliação:**

- Melhoria de rastreabilidade (positivo)
- Alinhado com RN-BAIXA.1 (rastreabilidade)
- Documentação deve ser atualizada

1.6 Módulo Estoque: Entidade `Estoque` Subimplementada

📘 **Documentação (Diagrama de Classes - Módulo Estoque):**



💻 **Implementação Real:**

```
@Table("estoque")
@Data
public class Estoque {
  @PrimaryKey
```

```
  private UUID idEstoque = UUID.randomUUID();

  @Column
  private String local;
}
```

❌ **Discrepâncias Críticas:**

- ❌ Faltam atributos: `nome`, `quantidadeTotal`, `validade`, `setor`
- ❌ Entidade `Estoque` parece representar **localização física** apenas
- ❌ Confusão conceitual: `Estoque` (local) vs `Item` (produto)

💥 **Impacto:**

- UC05 (Dar Baixa em Insumos) menciona "atualizar quantidadeTotal do Item no Estoque"
- Implementação atual: `Item` tem `quantidadeTotal`, `Estoque` só tem `local`
- Não há rastreabilidade de **qual item está em qual estoque**

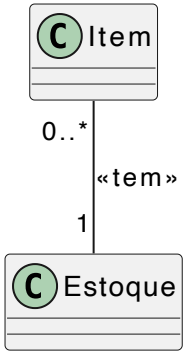
🔧 **Análise:** Parece haver um desalinhamento conceitual:

- **Documentação**: `Estoque` é entidade que agrega `quantidadeTotal` + `validade`
- **Implementação**: `Estoque` é apenas um **local físico**; `Item` já possui os atributos de quantidade

---

1.7 Módulo Estoque: Relacionamento Item ↔ Estoque Faltante

📘 **Documentação (Diagrama de Classes):**



- Item pertence a um Estoque
- Estoque contém múltiplos itens

💻 Implementação Real:

```
// Item.java
public abstract class Item {
  private UUID idItem;
  private String nome;
  private Integer quantidadeTotal;
  private Date validade;
  // ❌ NÃO HÁ referência para Estoque
}

// Estoque.java
public class Estoque {
  private UUID idEstoque;
  private String local;
  // ❌ NÃO HÁ lista de itens
}
```

❌ **Discrepância:**

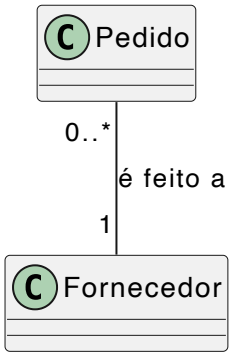
- ❌ Não há FK `estoque_id` em `Item`
- ❌ Não há lista `List<Item>` em `Estoque`
- ❌ Relacionamento **não implementado**

💥 **Impacto:**

- Impossível rastrear **onde** um item está fisicamente
- UC05 (Dar Baixa) não pode validar estoque por localização
- Dificuldade em implementar "transferência entre estoques"

1.8 Módulo Estoque: Relacionamento Pedido ↔ Fornecedor Simplificado

📘 **Documentação:**



💻 **Implementação Real:**

```
// Pedido.java
```

```
public class Pedido {
  private UUID idPedido;
  private Date dataPedido;
  private String status;
  private List<UUID> itemIds;
  private UUID fornecedorId; // ✅ Correto (FK para Fornecedor)
}

// Fornecedor.java
public class Fornecedor {
  private UUID idFornecedor;
  private String cnpj;
  // ❌ Falta: nome, telefone, email, endereço
}
```

❌ Discrepâncias:

- ✅ Relacionamento FK existe
- ❌ `Fornecedor` está **extremamente simplificado** (só tem CNPJ!)
- ❌ Faltam dados básicos: nome, contato, endereço

💥 Impacto:

- UC06 (Processar Entrada de NF) menciona “seleciona o Fornecedor”
- Não é possível exibir lista de fornecedores com nome (só CNPJ)
- Dificuldade em relatórios (“Compras por Fornecedor”)

---

2. 🔴 DISCREPÂNCIAS CRÍTICAS - Arquitetura de Persistência

2.1 PostgreSQL + Cassandra no Módulo Cadastro (Auditoria)

📘 **Documentação:**

> "Módulo Cadastro: PostgreSQL 16 (Banco Principal) + Cassandra 5 (Dados de Auditoria)"

💻 **Implementação Real:**

- ✅ PostgreSQL implementado (porta 5432 padrão, não 5430 documentada)
- ❌ Cassandra **não implementado** no módulo Cadastro
- ✅ Redis implementado para pub/sub

💥 **Impacto:**

- Persistência poliglota **parcialmente implementada**
- Benefícios de Cassandra (alta disponibilidade, write-heavy) não aproveitados
- Auditoria provavelmente implementada como logs em arquivo (se existir)

2.2 Redis com 3 Portas Diferentes (6380, 6379, 6381)

📘 **Documentação:**

```
- Cadastro Redis: 6380
- Agendamento Redis: 6379
- Estoque Redis: 6381
```

💻 Implementação Provável:

- Uso de uma única instância Redis padrão (6379)
- Separação por **namespaces** ou **prefixos de chave** ao invés de instâncias separadas

💥 Impacto:

- ❌ Menos isolamento entre módulos
- ✅ Simplificação operacional (menos serviços para gerenciar)
- ⚠️ Documentação vs realidade: decisão arquitetural não registrada

---

2.3 MongoDB com Coleções Separadas (Consulta, Exame, Procedimento)

📘 Documentação:

💻 Implementação Real:

```
@Document(collection = "consulta")
public class Consulta extends Agendamento { }

@Document(collection = "exame")
public class Exame extends Agendamento { }

@Document(collection = "procedimento")
public class Procedimento extends Agendamento { }

@Document(collection = "agendamento") // ❓ Classe abstrata também tem collection
```

```
public abstract class Agendamento { }
```

❌ **Discrepância:**

- ✅ Coleções separadas implementadas corretamente
- ❓ Classe abstrata `Agendamento` possui `@Document(collection = "agendamento")`
  - Isso pode gerar confusão: haverá collection "agendamento" com registros?
  - Ou apenas "consulta", "exame", "procedimento"?

🔧 **Recomendação:**

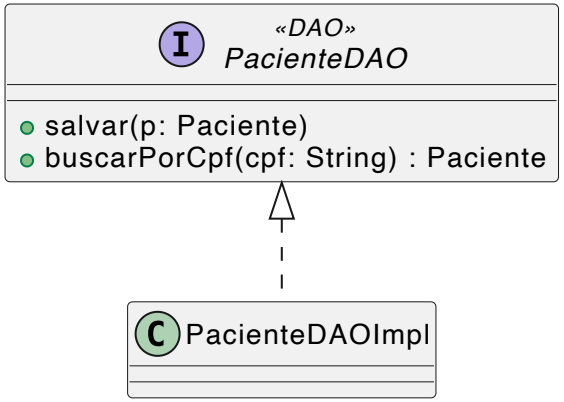
```
@Data
// ❌ Remover @Document da classe abstrata
public abstract class Agendamento { }
```

---

3. 🟡 DISCREPÂNCIAS MODERADAS - Camadas Arquiteturais

3.1 Padrão DAO vs Repository (Spring Data)

📘 **Documentação:**



- Usa nomenclatura **DAO** (Data Access Object)
- Implementações explícitas: `PacienteDAOImpl`

💻 Implementação Real:

```
public interface PacienteRepository extends JpaRepository<Paciente, Long> {
  boolean existsByCpf(String cpf);
  Optional<Paciente> findByCpf(String cpf);
}
```

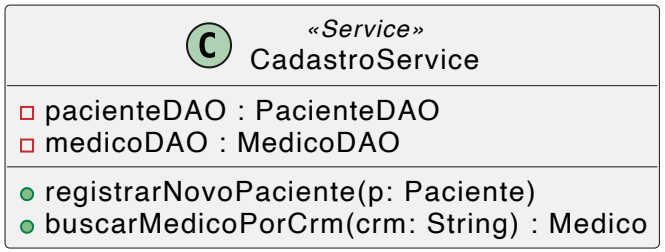
- Usa **Spring Data Repositories** (padrão moderno)
- ❌ Não há classes `*DAOImpl`
- ✅ Spring gera implementações automaticamente

📊 Análise:

- ✅ Implementação usa **best practice atual** (Spring Data)
- ❌ Documentação reflete padrão **legado** (DAO manual)
- ⚠️ Discrepância de **nomenclatura**, não de funcionalidade

3.2 Camada de Serviço: Service vs UseCase

📘 Documentação:



💻 Implementação Real:

```
// PacienteService.java - CRUD genérico
@Service
public class PacienteService {
  private final PacienteRepository pacienteRepository;
  public Paciente save(Paciente paciente) { }
  public Paciente findById(Long id) { }
  public List<Paciente> findAll() { }
}

// ConsultarHistoricoPacienteUseCase.java - Caso de uso específico
@UseCase
public class ConsultarHistoricoPacienteUseCase {
  private final PacienteService pacienteService;
  public HistoricoPacienteDTO executar(String cpf) { }
}
```

❌ **Discrepância:**

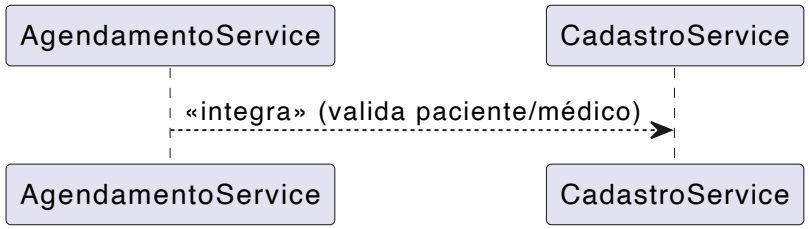
- Documentação: `CadastroService` centraliza toda lógica
- Implementação: Separação em `*Service` (CRUD) + `*UseCase` (lógica negócio)

📊 **Análise:**

- ✅ Implementação adota **Clean Architecture** (Use Cases explícitos)
- ✅ Melhor separação de responsabilidades
- ❌ Documentação não reflete essa granularidade

3.3 Integração AgendamentoService → CadastroService

📘 **Documentação:**



💻 **Implementação Real:**

```
// CriarConsultaUseCase.java
@UseCase
public class CriarConsultaUseCase {
  private final ConsultaRepository consultaRepository;
  // ❌ NÃO HÁ injeção de CadastroService/PacienteService
```

```
  public Consulta executar(ConsultaDTO dto) {
    // Validação feita apenas nos campos String (CPF, CRM)
    // NÃO valida existência no módulo Cadastro
  }
}
```

❌ **Discrepância Crítica:**

- Documentação prevê **integração entre módulos** (microsserviços)
- Implementação atual: **microsserviços independentes** (sem chamadas HTTP entre eles)
- Validação de CPF/CRM feita apenas como **string**, não valida existência real

💥 **Impacto:**

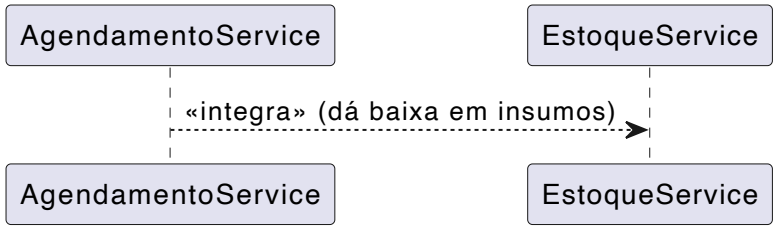
- UC02 (Agendar Consulta) passo 4: “usuário pesquisa e seleciona o Paciente (ex: por CPF) usando o CadastroService”
  - ❌ Não implementado: Agendamento não chama Cadastro
- Possibilidade de agendar com CPF/CRM **inexistentes**

🔧 **Soluções Possíveis:**

1. **API REST**: AgendamentoService faz HTTP GET para CadastroService
2. **Mensageria**: Validação assíncrona via RabbitMQ/Kafka
3. **Banco compartilhado** (anti-pattern de microsserviços)

3.4 Integração AgendamentoService → EstoqueService (Baixa em Insumos)

📘 **Documentação:**



💻 Implementação Real:

```
// FinalizarConsultaUseCase.java
@UseCase
public class FinalizarConsultaUseCase {
  private final ConsultaRepository consultaRepository;
  // ❌ NÃO HÁ integração com EstoqueService

  public Consulta executar(String id, String usuarioLogin) {
    consulta.setDataHoraFimExecucao(LocalDateTime.now());
    // ❌ NÃO dá baixa em materiais consumidos
  }
}
```

❌ Discrepância:

- UC05 (Dar Baixa em Insumos) menciona “vinculado a um procedimento, consulta”
- Implementação: **não há vínculo automático**

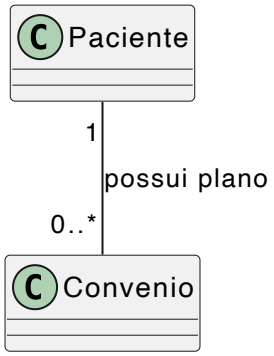
💥 Impacto:

- Baixa de insumos é **manual**, não automática ao finalizar consulta
- Possibilidade de esquecer de registrar consumo
- Estoque pode ficar **positivo fantasma**

4. 🔴 DISCREPÂNCIAS CRÍTICAS - Relacionamentos entre Entidades

4.1 Paciente ↔ Convênio (Cardinalidade Errada)

📘 Documentação:



- Um Paciente pode ter **múltiplos** Convênios

💻 Implementação Real:

```
@Entity
public class Paciente extends Pessoa {
  @ManyToOne
  @JoinColumn(name = "convenio_id")
  private Convenio convenio; // ❌ Apenas UM convênio
}
```

❌ Discrepância:

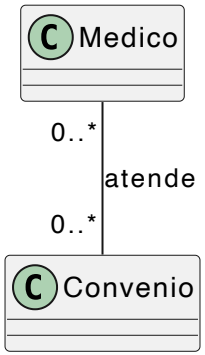
- Documentação: **1:N** (Paciente → Convênios)
- Implementação: **N:1** (Pacientes → Convênio)

💥 **Impacto:**

- Paciente não pode ter plano particular + convênio empresa
- Limitação em casos reais: “atender por particular hoje, convênio amanhã”

4.2 Médico ↔ Convênio (Relacionamento Faltante)

📘 **Documentação:**



- Relacionamento **N:N**: Médico atende múltiplos convênios

💻 **Implementação Real:**

```
@Entity
public class Medico extends Pessoa {
  private String crm;
  private String especialidade;
```

```
  // ❌ NÃO HÁ referência para Convenio
}

@Entity
public class Convenio {
  private Long id;
  private String nome;
  private String plano;
  private Boolean ativo;
  // ❌ NÃO HÁ lista de médicos
}
```

❌ **Discrepância:**

- ❌ Relacionamento **não implementado**
- ❌ Falta tabela associativa `medico_convenio`

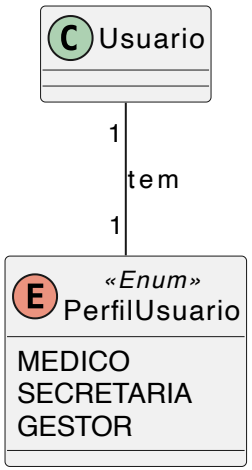
💥 **Impacto:**

- UC02 (Agendar Consulta) RN-AGENDA.2: "Se o Paciente tiver Convênio, o Agendamento deve estar associado a um Convênio que o Médico atenda"
  - ❌ **Impossível validar** essa regra de negócio
- Sistema aceita agendamentos com convênios não atendidos pelo médico

---

4.3 Usuario ↔ PerfilUsuario (Enum vs Relacionamento)

📘 **Documentação:**



💻 Implementação Real:

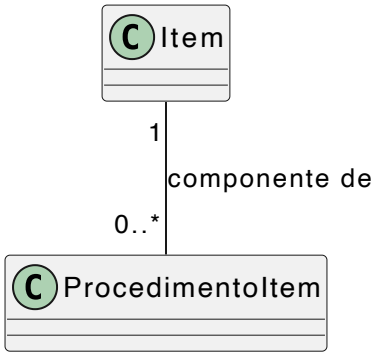
```
@Entity
public class Usuario extends Pessoa {
  @Enumerated(EnumType.STRING)
  private EPerfilUsuario perfil; // ✅ Correto
}

public enum EPerfilUsuario {
  MEDICO,
  SECRETARIA,
  GESTOR
}
```

✅ Avaliação:

- ✅ Implementado **corretamente**
- Nomenclatura: `EPerfilUsuario` vs `PerfilUsuario` (prefixo `E` no enum)

4.4 Item ↔ ProcedimentoItem (Relacionamento Presente mas Não Usado)

📘 Documentação:



💻 Implementação Real:

```
@Table("procedimento_item")
@Data
public class ProcedimentoItem {
  @PrimaryKey
  private UUID idProcedimentoItem = UUID.randomUUID();
  // ❌ NÃO HÁ campo item_id
  // ❌ NÃO HÁ campo procedimento_id
}
```

❌ Discrepância:

- Entidade existe, mas está **vazia**!
- Sem campos FK, não há como vincular Item ↔ Procedimento

💥 **Impacto:**

- Não é possível rastrear **quais itens são necessários** para um procedimento
- UC05 (Dar Baixa) menciona destino "Agendamento ID", mas não há vínculo

---

5. 🟡 DISCREPÂNCIAS MODERADAS - Casos de Uso Não Implementados

5.1 UC07: Gerar Alerta de Estoque Crítico

📘 **Documentação:**

> "UC07: O sistema verifica a quantidade de Itens no Estoque e notifica o Gestor se o saldo cair abaixo do ponto de reposição configurado."

💻 **Implementação Real:**

```
// GerarAlertaEstoqueCriticoUseCase.java
@UseCase
public class GerarAlertaEstoqueCriticoUseCase {
  private final EstoqueAlertaPublisher alertaPublisher;

  public void executar(String itemId, Integer quantidadeAtual) {
    alertaPublisher.publicarAlerta(
      "Item " + itemId + " com estoque crítico: " + quantidadeAtual
    );
  }
}

// Item.java
public abstract class Item {
```

```
  private Integer quantidadeTotal;
  // ❌ NÃO HÁ campo pontoReposicao/estoqueMinimo
}
```

❌ **Discrepâncias:**

- ✅ Use Case existe
- ❌ Falta atributo `pontoReposicao` em `Item` para comparação
- ❌ Lógica não é **automática** (precisa chamar explicitamente)
- ❌ Pré-condição UC07: "ponto de reposição configurado" → **não implementado**

5.2 UC08: Consultar Histórico do Paciente

📘 **Documentação:**

> "UC08: O sistema retorna o histórico completo de um Paciente, incluindo consultas, exames e procedimentos realizados."

💻 **Implementação Real:**

```
// ConsultarHistoricoPacienteUseCase.java
@UseCase
public class ConsultarHistoricoPacienteUseCase {
  private final PacienteService pacienteService;
  private final HistoricoPublisher historicoPublisher;

  public HistoricoPacienteDTO executar(String cpf) {
    pacienteService.findByCpf(cpf); // Busca paciente
    historicoPublisher.solicitarHistorico(cpf); // Publica mensagem Redis
```

```
    // ❌ Como aguardar a resposta?
    // ❌ Subscriber retorna os dados, mas como integrar no fluxo síncrono?
  }
}
```

❌ **Discrepâncias:**

- ✅ Use Case existe
- ❌ Implementação usa **pub/sub assíncrono** (Redis)
- ❌ Frontend espera resposta **síncrona** (HTTP request/response)
- ❌ Não há mecanismo de **aguardar resposta** do subscriber

💥 **Impacto:**

- Integração Cadastro → Agendamento **não funcional** em fluxo síncrono
- Possível timeout no frontend aguardando resposta

🔧 **Soluções Possíveis:**

1. Mudar para HTTP REST (síncrono)
2. Implementar pattern Request-Reply com correlation ID (Redis)
3. WebSockets para comunicação assíncrona com frontend

5.3 UC10: Controlar Validade de Itens

📘 **Documentação:**

> "UC10: O sistema identifica itens vencidos ou próximos do vencimento e gera alertas para descarte ou uso prioritário."

💻 **Implementação Real:**

```
// ControleValidadeService.java (Frontend)
@Service
public class ControleValidadeService {
  public List<Item> verificarItensVencidos() {
    // ❌ Lógica não implementada
    return new ArrayList<>();
  }
}
```

❌ **Discrepâncias:**

- ✅ Service existe no **frontend** (JavaFX)
- ❌ Método retorna lista vazia (stub)
- ❌ Backend não possui endpoint `/estoque/validade` correspondente
- ⚠️ Documentação menciona UC10, mas fluxo detalhado está ausente

---

6. 🟢 DISCREPÂNCIAS BAIXAS - Nomenclatura e Padrões

6.1 Prefixo E em Enums

💻 **Implementação:**

```
public enum EPerfilUsuario { } // Prefixo E
public enum ModalidadeEnum { } // Sufixo Enum
```

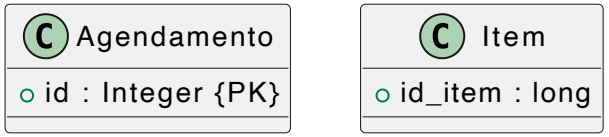
```
public enum StatusAgendamentoEnum { }
public enum TipoConsultaEnum { }
```

📊 Análise:

- ⚠️ Inconsistência: `EPerfilUsuario` vs `ModalidadeEnum`
- Documentação não especifica padrão de nomenclatura
- Sugestão: padronizar (remover prefixo/sufixo redundante)

6.2 Tipo de ID: Long vs UUID

📘 Documentação:



💻 Implementação Real:

```
// Cadastro (PostgreSQL)
@Entity
public class Paciente {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id; // ✅ Long (auto-increment)
}

// Agendamento (MongoDB)
@Document
```

```
public abstract class Agendamento {
  @Id
  private String id; // ✅ String (ObjectId do MongoDB)
}

// Estoque (Cassandra)
@Table("item")
public abstract class Item {
  @PrimaryKey
  private UUID idItem = UUID.randomUUID(); // ✅ UUID
}
```

📊 **Análise:**

- ✅ Tipos de ID **adequados** para cada banco:
  - PostgreSQL: Long (auto-increment)
  - MongoDB: String (ObjectId hex)
  - Cassandra: UUID (distribuído)
- ❌ Documentação menciona `Integer` para Agendamento (incoerente com MongoDB)

---

7. 🟡 DISCREPÂNCIA MODERADA - Atributos Faltantes

7.1 BloqueioAgenda: Falta `antecedenciaMinima` na Lógica

📘 **Documentação:**

💻 Implementação Real:

```
@Document(collection = "bloqueio_agenda")
public class BloqueioAgenda {
  private Integer antecedenciaMinima; // ✅ Atributo existe
  // ❌ Mas não é usado em nenhuma validação
}

// CriarBloqueioAgendaUseCase.java
public BloqueioAgenda executar(BloqueioAgendaDTO dto) {
  // ❌ Não valida se há agendamentos dentro da antecedência mínima
  return bloqueioRepository.save(bloqueio);
}
```

❌ Discrepância:

- ✅ Campo existe
- ❌ Lógica de validação **não implementada**

💥 Impacto:

- UC04 (Registrar Bloqueio) menciona validação de conflitos
- Possibilidade de criar bloqueios **muito próximos** de agendamentos existentes

---

8. 📊 Resumo de Impactos por Módulo

Módulo Cadastro

Discrepância	Severidade	Status Funcional
Médico não herda Usuario	🔴 Crítica	⚠️ Médicos não podem logar
EventoAuditoria ausente	🔴 Crítica	❌ Sem rastreabilidade
Redis cache não implementado	🟡 Moderada	⚠️ Performance degradada
Cassandra auditoria ausente	🔴 Crítica	❌ Alta disponibilidade perdida
Relacionamento Médico-Convênio faltante	🔴 Crítica	❌ Validação RN-AGENDA.2 impossível

Módulo Agendamento

Discrepância	Severidade	Status Funcional
Atributos temporais extras	🟢 Baixa	✅ Funcional (melhor que doc)
Integração com Cadastro ausente	🔴 Crítica	⚠️ Validação CPF/CRM fraca
Integração com Estoque ausente	🟡 Moderada	⚠️ Baixa manual
UC08 assíncrono mal implementado	🟡 Moderada	⚠️ Timeout provável

Módulo Estoque

Discrepância	Severidade	Status Funcional
Estoque subimplementado	🔴 Crítica	⚠️ Confusão conceitual
Relacionamento Item-Estoque faltante	🔴 Crítica	❌ Rastreabilidade comprometida

Discrepância	Severidade	Status Funcional
Fornecedor simplificado	🟡 Moderada	⚠️ CNPJ apenas
ProcedimentoItem vazio	🟡 Moderada	❌ Vínculo procedimento-item inexistente
UC07 sem pontoReposicao	🟡 Moderada	⚠️ Alerta manual

---

9. 🎯 Recomendações Prioritárias

Críticas (Corrigir Imediatamente)

1. **Médico herdar de Usuario** → Habilitar login de médicos
2. **Implementar relacionamento Médico ↔ Convênio** → Validar RN-AGENDA.2
3. **Implementar validação cross-module** → Agendamento validar CPF/CRM no Cadastro
4. **Adicionar FK estoque_id em Item** → Rastreabilidade de localização
5. **Completar atributos de Fornecedor** → Nome, contato, endereço

Moderadas (Planejar Sprint Futura)

6. Implementar EventoAuditoria com Cassandra
7. Adicionar Redis para cache de médicos disponíveis
8. Implementar baixa automática de insumos ao finalizar consulta
9. Adicionar campo `pontoReposicao` em Item + lógica UC07
10. Completar atributos de Estoque ou refatorar conceito

Baixas (Documentação)

11. Atualizar diagramas com atributos temporais extras de Agendamento
12. Documentar uso de Spring Data Repository vs DAO
13. Padronizar nomenclatura de enums

14. Atualizar portas Redis documentadas

---

10. 📁 Anexos

A. Arquivos Analisados

Documentação:

- `docs/documentos-finais-definitivos/3.1. Documento de Visão do Projeto/Documento de visão do projeto.md`
- `docs/documentos-finais-definitivos/3.4. Classes de Análise/3.4. Classes de Análise_Diagrama de Classes.md`
- `docs/documentos-finais-definitivos/3.7_3.9_3.10_Modelagens/3.7. Modelagem de Classes de Projeto/3.7. Modelagem de Classes de Projeto.md`
- `docs/documentos-finais-definitivos/3.6. Arquitetura do Sistema - Lógica e Física/3.6. Arquitetura do Sistema - Lógica e Física.md`
- `docs/documentos-finais-definitivos/3.2_3.3_Casos de uso/3.3. Descrição detalhada de cada Caso de Uso/3.3. Descrição detalhada de cada Caso de Uso.md`

Código-Fonte Backend:

- `simplehealth-back/simplehealth-back-cadastro/src/main/java/com/simplehealth/cadastro/domain/entity/*.java`
- `simplehealth-back/simplehealth-back-agendamento/src/main/java/com/simplehealth/agendamento/domain/entity/*.java`
- `simplehealth-back/simplehealth-back-estoque/src/main/java/com/simplehealth/estoque/domain/entity/*.java`
- `simplehealth-back/**/application/service/*.java`
- `simplehealth-back/**/application/usecases/*.java`
- `simplehealth-back/**/infrastructure/repositories/*.java`

Código-Fonte Frontend:

- `simplehealth-front/**/controller/*.java`
- `simplehealth-front/**/service/*.java`
- `simplehealth-front/**/model/*.java`

---

11. 🔍 Metodologia de Análise

1. **Extração de Requisitos:** Leitura detalhada dos documentos de Visão, Casos de Uso e Diagramas de Classes
2. **Mapeamento de Entidades:** Comparação entre classes projetadas e classes implementadas
3. **Análise de Relacionamentos:** Verificação de cardinalidades, FKs e navegabilidade
4. **Validação de Arquitetura:** Comparação entre decisões documentadas (persistência poliglota, portas) e implementação
5. **Rastreamento de Casos de Uso:** Verificação de implementação de Use Cases documentados
6. **Análise de Padrões:** Identificação de padrões arquiteturais (DAO vs Repository, Service vs UseCase)

Fim do Relatório