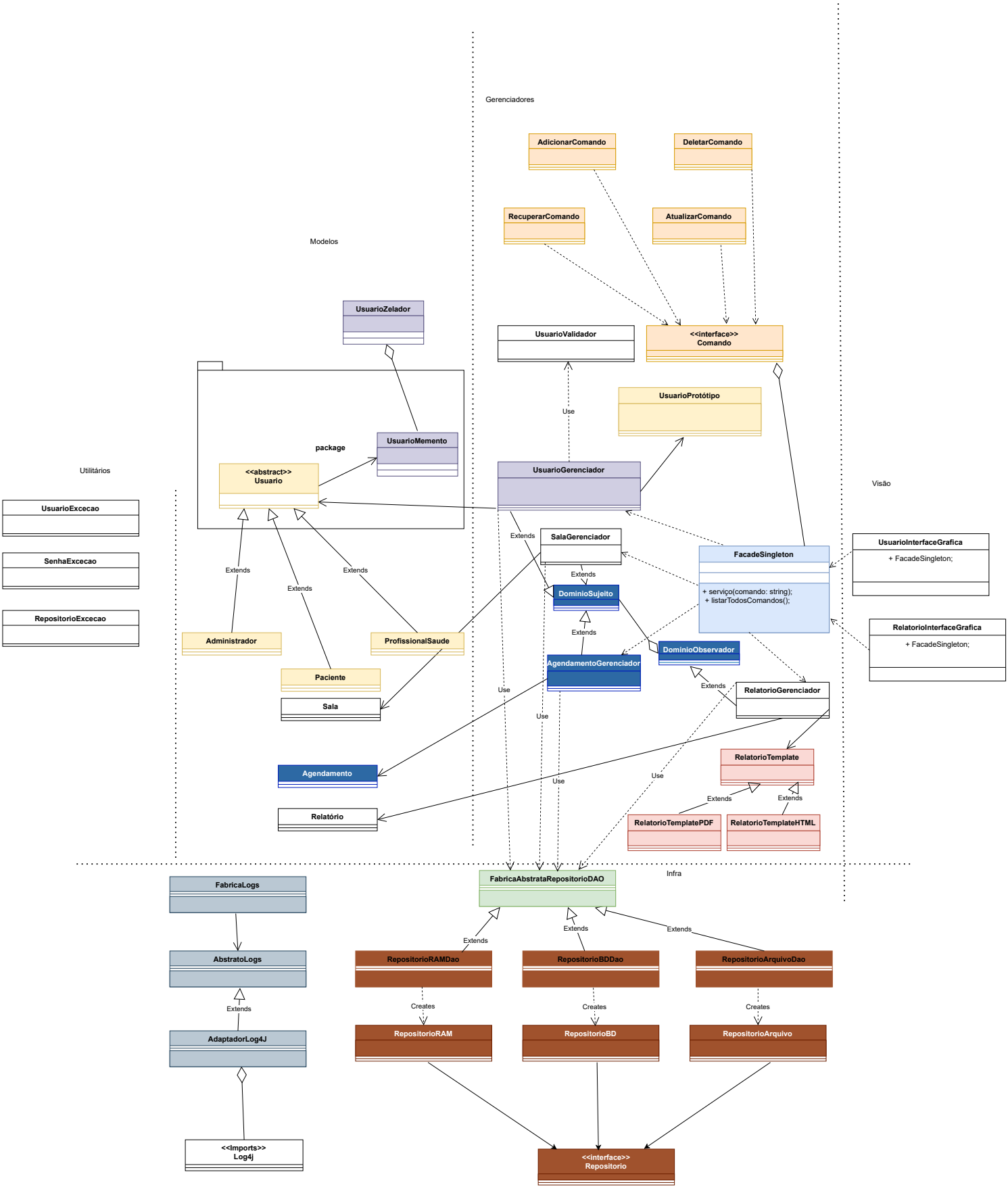

Gerenciadores
AdicionarComando
DeletarComando
RecuperarComando
AtualizarComando
Modelos
UsuarioZelador
UsuarioValidador
<<interface>>
Comando
UsuarioProtótipo
Use
package
UsuarioMemento
<<abstract>>
Usuario
UsuarioGerenciador
Utilitários
Visão
UsuarioExcecao
SenhaExcecao
RepositorioExcecao
Extends
SalaGerenciador
FacadeSingleton
+ serviço(comando: string);
+ listarTodosComandos();
UsuarioInterfaceGrafica
+ FacadeSingleton;
DominioSujeito
RelatorioInterfaceGrafica
+ FacadeSingleton;
Administrador
ProfissionalSaude
Paciente
Sala
AgendamentoGerenciador
DominioObservador
RelatorioGerenciador
Agendamento
Relatório
RelatorioTemplate
RelatorioTemplatePDF
RelatorioTemplateHTML
Infra
FabricaLogs
AbstratoLogs
AdaptadorLog4J
<<Imports>>
Log4j
FabricaAbstrataRepositorioDAO
RepositorioRAMDao
RepositorioBDDao
RepositorioArquivoDao
Creates
RepositorioRAM
RepositorioBD
RepositorioArquivo
<<interface>>
Repositorio

Diagrama C4 Nível 1: Contexto - Sistema Cirurgia Sem Fronteiras

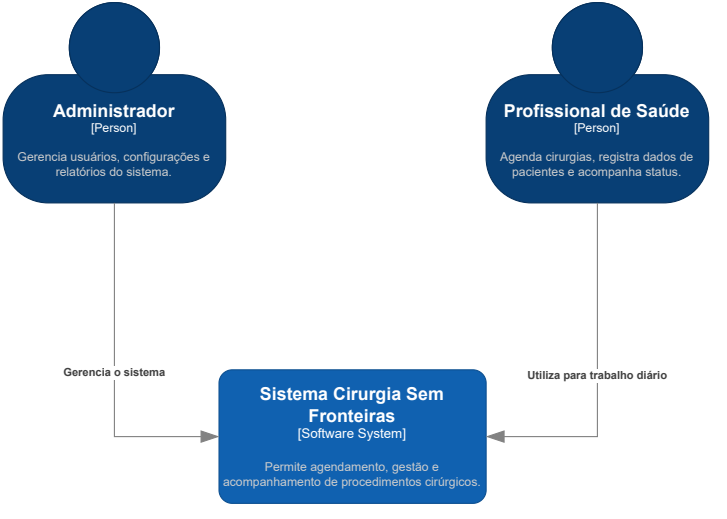


Diagrama C4 Nível 2: Contêiner - Sistema Cirurgia Sem Fronteiras

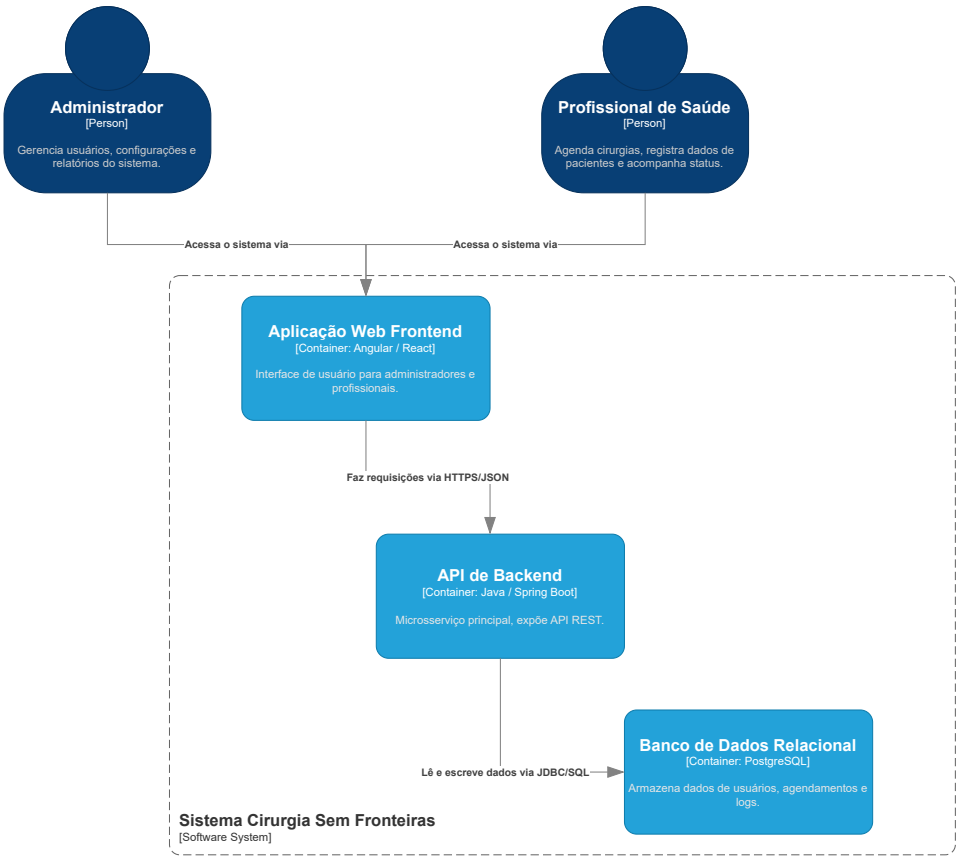
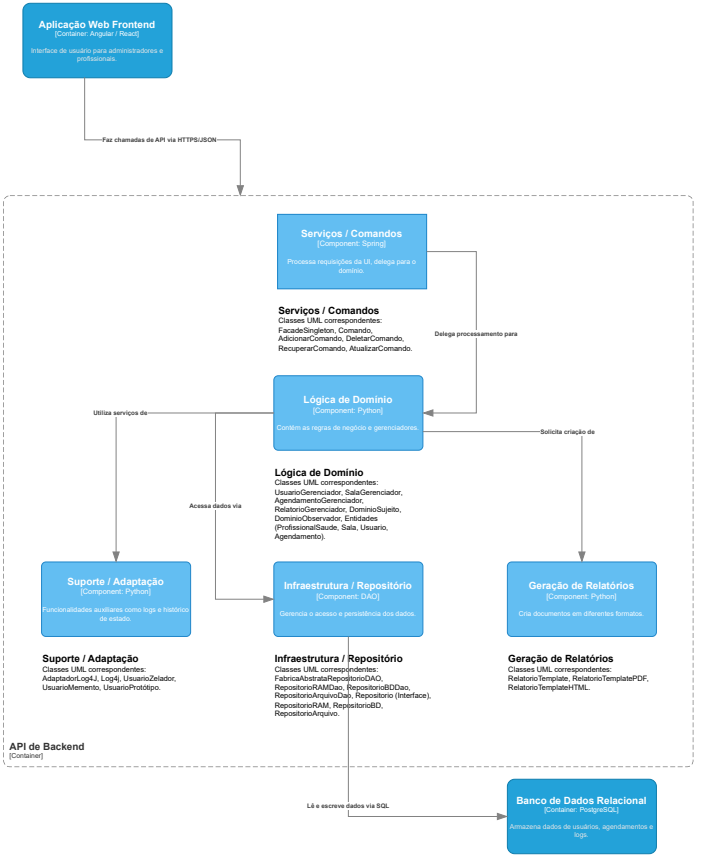


Diagrama C4 Nível 3: Componentes - API de Backend



Registro dos Padrões de Projeto Utilizados

1. Singleton / Facade

Classes: `FacadeSingleton`
Objetivo: Este padrão combina a simplicidade de acesso do *Facade* com o controle de instância única do *Singleton*. A classe `FacadeSingleton` centraliza o acesso aos serviços do sistema, fornecendo um ponto único de interação com as funcionalidades principais, ao mesmo tempo em que garante que apenas uma instância da fachada seja criada. Isso reduz a complexidade do sistema para o cliente e controla o ciclo de vida da fachada.
Cor de marcação: Azul Claro

---

2. Abstract Factory

Classes: `FabricaAbstrataRepositorioDAO`
Objetivo: Este padrão é utilizado para a criação de famílias de objetos relacionados (repositórios de dados), sem acoplar o código às classes concretas. A fábrica abstrata define a interface para criação de objetos, enquanto as classes concretas implementam os diferentes tipos de persistência (RAM, arquivo e banco de dados). Isso permite flexibilidade e substituição dinâmica da forma de armazenamento.
Cor de marcação: Verde

---

3. Template Method

Classes: `RelatorioTemplate`, `RelatorioTemplatePDF`, `RelatorioTemplateHTML`
Objetivo: O padrão *Template Method* define o esqueleto da geração de relatórios em uma classe abstrata (`RelatorioTemplate`), permitindo que subclasses (`RelatorioTemplatePDF` e `RelatorioTemplateHTML`) implementem as variações específicas de formato. Isso assegura consistência no processo de geração de relatórios e promove reutilização de código.
Cor de marcação: Vermelho

---

4. Adapter

Classes: `AdaptadorLog4J`, `AbstratoLogs`, `FabricaLogs`
Objetivo: O padrão *Adapter* é empregado para integrar a biblioteca externa `Log4j` ao sistema. A classe `AdaptadorLog4J` atua como tradutor entre a interface de logs utilizada no projeto (`AbstratoLogs`) e a API externa. Dessa forma, o sistema permanece desacoplado da implementação concreta de logging, permitindo substituições futuras sem impactos no restante do código.
Cor de marcação: Cinza

---

5. Command

Classes: `Comando`, `AdicionarComando`, `AtualizarComando`, `DeletarComando`, `RecuperarComando`
Objetivo: O padrão *Command* encapsula cada ação do sistema em um objeto separado, permitindo tratar operações como entidades independentes. Isso facilita o gerenciamento de histórico de ações, filas de execução e a possibilidade de desfazer/refazer operações. A interface `Comando` define a estrutura, enquanto as classes concretas implementam os comandos específicos.
Cor de marcação: Laranja

---

6. Memento

Classes: `UsuarioMemento`, `UsuarioZelador`, `UsuarioGerenciador`
Objetivo: O padrão *Memento* permite salvar e restaurar o estado interno de um objeto sem violar o encapsulamento. No projeto, ele é utilizado no gerenciamento de usuários, armazenando estados anteriores e possibilitando a restauração quando necessário. O `UsuarioMemento` guarda o estado, o `UsuarioZelador` administra os mementos, e o `UsuarioGerenciador` coordena o processo.
Cor de marcação: Roxo

---

7. Prototype

Classes: `UsuarioProtótipo`, `Usuario` (abstrato), `Administrador`, `ProfissionalSaude`, `Paciente`
Objetivo: O padrão *Prototype* é utilizado para criar novos objetos a partir da clonagem de instâncias já existentes, evitando a necessidade de instanciar classes concretas diretamente. Isso permite a criação flexível de diferentes tipos de usuários, mantendo baixo acoplamento e facilitando a expansão futura da hierarquia de usuários.
Cor de marcação: Amarelo

---

8. Observer

Classes: `DominioSujeito`, `DominioObservador`, `Agendamento`, `AgendamentoGerenciador`
Objetivo: O padrão *Observer* estabelece uma relação de dependência de um-para-muitos, garantindo que alterações no estado de um objeto sujeito (`DominioSujeito`) sejam notificadas automaticamente a todos os observadores (`DominioObservador`). No projeto, é aplicado ao contexto de agendamentos, assegurando atualização consistente em múltiplos componentes interessados.
Cor de marcação: Azul Escuro

9. DAO (Data Access Object)

Classes: `Repositorio` (interface), `RepositorioRAM`, `RepositorioBD`, `RepositorioArquivo`, `RepositorioRAMDao`, `RepositorioArquivoDao`, `RepositorioBDDao`
Objetivo: O padrão *DAO* é utilizado para abstrair e encapsular o acesso aos dados da camada de persistência. A interface `Repositorio` define as operações básicas de CRUD, enquanto as implementações concretas (`RepositorioRAM`, `RepositorioBD`, `RepositorioArquivo`) realizam essas operações em diferentes mecanismos de armazenamento. Esse padrão isola a lógica de negócio dos detalhes de persistência, permitindo manutenção mais simples e a substituição transparente da tecnologia de armazenamento.
Cor de marcação: Marrom