Relatório de Arquitetura — Kanban Lite (Etapa 1)

Resumo

Este documento sumariza e justifica as decisões arquiteturais tomadas na Etapa 1 do projeto Kanban Lite (design das principais classes, interfaces e persistência). O foco foi garantir um modelo claro de domínio, baixo acoplamento entre camadas, uso de práticas modernas de C++ (smart pointers, RAII, STL, templates) e preparar a base para as próximas etapas (CLI e GUI).

---

1. Princípios gerais adotados

* Separação de responsabilidades (SoC): domínio (`domain/`), contratos (`interfaces/`) e persistência (`persistence/`) separados. Isso facilita testes, manutenção e substituição de implementações.
* Baixo acoplamento e alta coesão: componentes comunicam-se via interfaces abstratas (`IService`, `IRepository`, `IView`) em vez de dependências concretas.
* C++ moderno: C++17, `std::vector`, `std::optional`, `std::chrono`, `std::shared_ptr`/`std::unique_ptr`, RAII para recursos.

---

2. Composição vs Herança — escolha e justificativa

* Composição (prefeitura) Relações de “tem-um” (ownership lógico) foram modeladas com composição:
    - `Board` *contém* `Column`s;
    - `Column` *contém* `Card`s;
    - `Card` *contém* `Tag`s;
    - `ActivityLog` *contém* `Activity`s.

  Por que?
    - A composição reflete corretamente a semântica do domínio: uma `Column` só existe dentro de um `Board`, assim como um `Card` depende de uma `Column`. O ciclo de vida das partes é controlado pelo todo.
    - Mantém a modelagem coerente com o princípio LSP (Liskov Substitution Principle): usar herança nesses casos implicaria em uma relação “é-um” que não existe (ex.: uma `Column` não é um `Board`).
    - Garante propriedade e responsabilidade claras: cada agregado controla a criação, destruição e invariantes de seus elementos (ex.: `Board` valida nomes únicos de colunas; `Column` valida posição dos cards).

    - Facilita a persistência e serialização, pois a estrutura hierárquica em memória mapeia naturalmente para formatos como JSON: `Board → Column → Card → Tag`. Isso reduz a complexidade de salvar/carregar estado.
    - Simplifica testes unitários: cada componente pode ser testado isoladamente e, ao mesmo tempo, é fácil verificar cenários de integração (ex.: mover um card entre colunas).
    - Permite flexibilidade de implementação com smart pointers (`unique_ptr`/`shared_ptr`), deixando explícito no código quem possui cada objeto e evitando vazamentos de memória.

* Herança (restrita a contratos / polimorfismo) Usada para definir interfaces/contratos que precisam de implementação múltipla:
    - `IRepository<T>` (template) — abstrai persistência;
    - `IService` — fachada de operações de alto nível;
    - `IFilter` — filtros polimórficos aplicáveis a `Card`;
    - `IView` — contrato para apresentação (CLI/GUI).

  Por que?
    - Polimorfismo dinâmico controlado: herança permite tratar diferentes implementações de forma uniforme via ponteiros ou referências (`shared_ptr<IRepository<T>>`, `unique_ptr<IFilter>`). Isso facilita armazenar e manipular objetos heterogêneos sem conhecer a implementação concreta.
    - Troca de implementação transparente: camadas superiores (serviço, UI) dependem da interface, não da implementação. Ex.: a camada de teste pode usar `InMemoryRepository`, enquanto a versão de produção usa `FileRepository`.
    - Segurança de contratos: a interface garante que todas as implementações forneçam métodos consistentes (mesmos nomes, assinaturas e semântica), promovendo coesão e previsibilidade no uso.
    - Extensibilidade futura: novas estratégias ou filtros podem ser adicionados sem alterar código existente, respeitando o princípio Open/Closed.
    - Desacoplamento entre camadas: a UI e os serviços dependem apenas de abstrações, não de detalhes de armazenamento ou exibição, facilitando manutenção e teste unitário isolado.

---

3. Smart pointers, RAII e gerenciamento de recursos

* `std::shared_ptr` Utilizado sempre que múltiplas entidades precisam compartilhar a posse lógica de um objeto:
    - Ex.: `Board` mantém `shared_ptr<Column>`, e colunas podem ser referenciadas por views ou logs de atividades.

    - Benefício: elimina riscos de dangling pointers e facilita o compartilhamento seguro sem duplicar objetos.
    - Permite que diferentes camadas (domínio, UI, histórico) tenham referências consistentes a um mesmo objeto, sem se preocupar com desalocação manual.
* `std::unique_ptr` Utilizado para posse exclusiva e objetos temporários:
    - Ex.: `clone()` de `IFilter` retorna `unique_ptr<IFilter>`, garantindo que o objeto seja destruído automaticamente quando sair de escopo.
    - Benefício: reforça ownership única, deixando claro quem é responsável pelo ciclo de vida.
    - Evita ambiguidades de gerenciamento de memória, mantendo código mais seguro e legível.
* RAII (Resource Acquisition Is Initialization) Aplicado especialmente para recursos não gerenciados pelo C++ (ex.: arquivos, sockets):
    - Ex.: `FileRepository` encapsula `std::fstream` e outros recursos I/O.
    - Benefício: garante liberação automática de recursos mesmo em caso de exceções, prevenindo leaks e mantendo consistência.
    - Permite escrever código mais robusto, sem necessidade de `try/catch` explícito só para liberar recursos.
* Por que evitar raw pointers?
    - Raw pointers não indicam ownership; quem deve desalocar o objeto pode ficar ambíguo, aumentando risco de memory leaks ou double delete.
    - Smart pointers tornam o ownership explícito, melhoram a legibilidade do código e permitem integração segura com STL e APIs modernas.
    - Facilitam a implementação de padrões de projeto e abstrações (ex.: polimorfismo com `shared_ptr` e `unique_ptr`) sem comprometer segurança de memória.

---

4. Padrões arquiteturais adotados

* Repository (IRepository<t>)
    - Abstrai persistência; facilita testes (in-memory) e troca por implementações (file/DB). `IRepository` é um template para reaproveitamento.
* Service / Facade (IService)
    - Orquestra operações de alto nível (criar board, mover card, etc.). Interface única para CLI/GUI; reduz acoplamento direto aos repositórios.
* Strategy / Filter (IFilter)

    - Filtros polimórficos para consultas/filtragens de `Card`. Permite composição de critérios e extensibilidade.

* Observer (implícito, futuro)
    - Desejável para GUI reativa (notificar views sobre mudanças). Por ora, responsabilidade é do `IService` atualizar `IView`.

Padrões Arquiteturais adotados — Por que?

1. **Repository (`IRepository<T>`)**

* Problema resolvido: o domínio não deve conhecer detalhes de persistência (arquivos, DB, memória).
* Por que adotar:
    - Garante baixo acoplamento: posso trocar de `InMemoryRepository` para `FileRepository` ou até `DatabaseRepository` sem mudar nada no domínio ou no serviço.
    - Facilita testes unitários: em vez de depender de I/O real, posso injetar um repositório em memória rápido e determinístico.
    - Centraliza a lógica de acesso a dados (CRUD), evitando duplicação em várias partes do código.
* Alternativa rejeitada: acessar arquivos diretamente dentro de `Board`/`Service`. Isso quebraria separação de responsabilidades e dificultaria testes.

---

2. **Service / Facade (`IService`)**

* Problema resolvido: expor ao usuário/CLI/GUI uma API simples que orquestra várias operações complexas.
* Por que adotar:
    - Encapsula regras de negócio de alto nível (ex.: validar se coluna existe antes de mover um card).
    - Evita espalhar lógica em `main()` ou em views (mantendo separação de responsabilidades).
    - Fornece um único ponto de entrada para a aplicação (parecido com *Application Service* em DDD).
* Benefício extra: quando trocar de interface (CLI → GUI), a mesma `IService` pode ser usada, sem duplicar lógica.
* Alternativa rejeitada: deixar `Board`/`Column` fazerem operações de aplicação diretamente. Isso acoplaria entidades a casos de uso e dificultaria evolução.

---

3. **Strategy (`IFilter`)**

* Problema resolvido: permitir filtragem de `Card`s com múltiplos critérios configuráveis (por tag, prioridade, usuário, etc.).
* Por que adotar:
    - Usa polimorfismo dinâmico: posso aplicar diferentes filtros (`TagFilter`, `PriorityFilter`, `CompositeFilter`) sem alterar o código que executa a busca.
    - Favorece extensibilidade: se amanhã surgir um novo critério de filtragem, basta criar outra classe que implementa `IFilter`.
    - Evita condicionais gigantes e repetitivas dentro do serviço.
* Alternativa rejeitada: usar apenas `if/else` ou `switch` para cada tipo de filtro. Isso seria rígido, não aberto a extensões (violaria OCP — *Open/Closed Principle*).

---

4. **Observer (planejado, Etapa 3)**

* Problema resolvido: manter GUI sincronizada com mudanças no modelo.
* Por que adotar:
    - Permite que `View`s se inscrevam no `Service`/`Board` e sejam notificadas automaticamente quando dados mudarem.
    - Evita polling (ficar checando manualmente se algo mudou), que é ineficiente e incorreto.
    - No futuro, pode ser integrado facilmente com frameworks GUI como Qt (signals/slots).
* Alternativa rejeitada: atualizar a GUI manualmente em cada operação. Isso cria código frágil, duplicado e difícil de manter.

---

5. **Padrão de Camadas (implícito)**

* Por que adotar:
    - Organizar código em camadas claras:
        - Domínio = entidades e lógica local.
        - Interfaces = contratos e abstrações.
        - Persistência = infraestrutura de armazenamento.
        - Aplicação (Service) = orquestração de casos de uso.
    - Essa separação segue boas práticas de arquitetura hexagonal/clean architecture e facilita manutenção, testes e substituição de partes.
* Alternativa rejeitada: código monolítico (GUI chamando direto entidades e persistência), que gera forte acoplamento e baixa testabilidade.

---

5. Templates e STL

* `IRepository<T, Id = std::string>` usa templates para tipar repositórios por entidade.
* `std::vector` para coleções; `std::optional` para buscas que podem falhar; `std::chrono::system_clock::time_point` para timestamps.
* Motivo: STL provê containers e utilitários testados e eficientes, reduzindo implementação ad hoc.

---

6. Tratamento de erros

* Exceções específicas para persistência: `FileRepositoryException : std::runtime_error`.
* Regras:
    - Erros de uso (ex.: argumento inválido) → `std::invalid_argument`.
    - Erros críticos / I/O → `std::runtime_error` ou exceção customizada.
* CLI/GUI capturam e exibem mensagens amigáveis; camadas internas documentam contratos.

---

7. Mapeamento rápido (onde aplicar conceitos)

* Composição: `Board.h`, `Column.h`, `Card.h`.
* Interfaces/polimorfismo: `interfaces/IRepository.h`, `interfaces/IFilter.h`, `interfaces/IService.h`, `interfaces/IView.h`.
* Persistência RAII: `persistence/FileRepository.h`.
* Smart pointers: presentes em headers do domínio e interfaces.

---

8. Conclusão & próximos passos

A Etapa 1 fornece uma base sólida e justificada para implementar a Etapa 2 (CLI) e Etapa 3 (GUI). As principais decisões — composição para modelagem de domínio, herança apenas para contratos, uso de smart pointers e padrões Repository/Service — foram tomadas para maximizar clareza, testabilidade e extensibilidade do sistema.
