Estilos arquitetônicos

Um estilo é um conjunto de **restrições** que limita o espaço de design para garantir consistência estrutural. As restrições apontam o sistema para onde se deseja ir. Elas melhoram a comunicação e encorajam uma evolução limpa. Os estilos definem características visuais e estruturais.

Um estilo estabelece:

- **Tipos de componentes:** Um conjunto de tipos de componentes (ex: processo, procedimento) que desempenham funções em runtime.
- **Topologia (Layout):** Um layout topológico dos componentes mostrando as suas relações em runtime
    - Como componentes estão fisicamente organizados
    - Relacionamentos entre componentes
    - Hierarquia ou rede de componentes
    - Diagrama estrutural do sistema
- **Restrições semânticas:** Um conjunto de restrições semânticas
    - Regras que componentes devem seguir
    - Como componentes podem/não podem interagir
    - Restrições preservam integridade conceptual
    - Garantem que sistema segue o estilo
- **Conectores:** Um conjunto de conectores (ex: data streams, sockets) que medeiam comunicação entre componentes em runtime.
    - Como componentes se comunicam
    - Mecanismos de conexão entre componentes
    - Definem protocolos de interação
    - Garantem compatibilidade

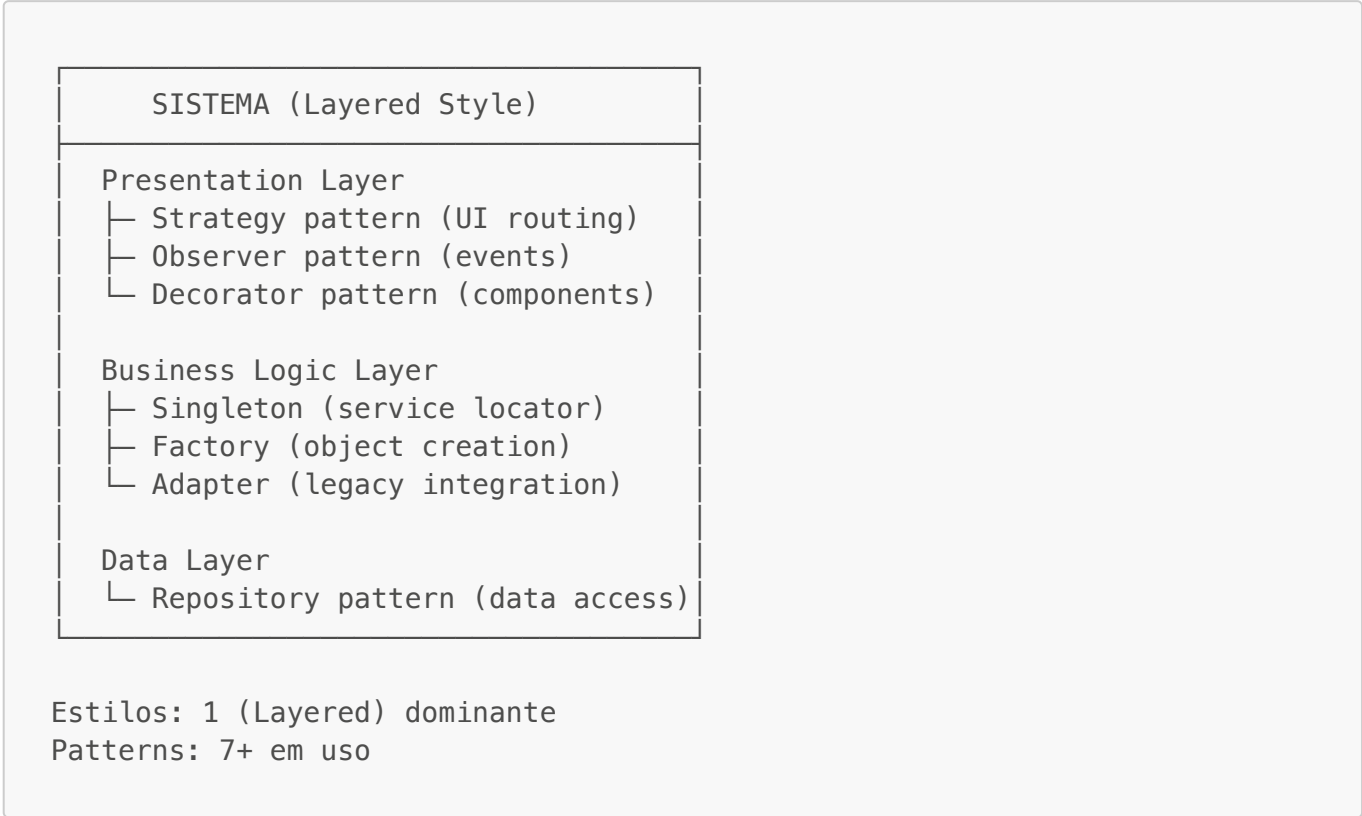
Padrões (Patterns) vs. Estilos (Styles)

A distinção entre estes conceitos reside na escala e na dominância:

- **Padrões de Design:** Operam numa escala menor e podem coexistir em múltiplos pontos de um mesmo design.
- **Padrões Arquiteturais:** Exemplos como o REST (Representational State Transfer) são aplicados a um nível mais elevado do que os padrões de desenho de baixo nível (GoF).

- **Estilos Arquiteturais:** Têm uma escala dominante e definem o sistema ao nível mais alto. Um sistema possui geralmente um único estilo dominante; se o estilo for Client-Server, espera-se que as vistas de topo do design reflitam inequivocamente componentes de cliente e servidor.

Patterns podem aparecer em qualquer lugar do design, e múltiplos patterns poderiam aparecer no mesmo design.



Big Ball of Mud

No extremo oposto da organização modular, encontramos o sistema sem estrutura evidente.

- **Características:** Caracteriza-se pela partilha promíscua de informação, ao ponto de as estruturas de dados se tornarem globalmente acessíveis.
- **Impacto:** É o conhecido "Código Esparguete", onde as intervenções assemelham-se a remendos (patches) em vez de refatorizações elegantes. Embora seja por vezes uma estratégia de engenharia "suficiente" em contextos de curto prazo, inibe severamente a **manutenibilidade** e a **extensibilidade**.

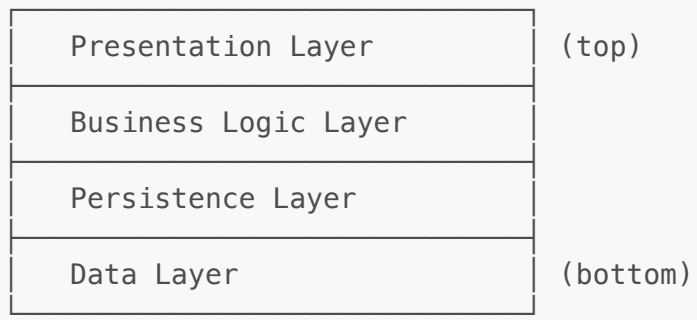
Qualidades:

- Poor maintainability
- Poor extensibility

Camadas (Layers)

O sistema é organizado como uma pilha de camadas, onde cada nível funciona como uma máquina virtual para as camadas superiores.

- **Relacionamento:** O elemento essencial é a relação de "uso" (uses relationship), que é formalmente uma especialização de uma relação de dependência.
- **Restrições:** Numa estrutura simples, uma camada só pode utilizar a camada imediatamente inferior, formando um Grafo Acíclico Dirigido (DAG).
- **Prática:** Na indústria, este estilo é frequentemente apelidado de "Código Lasanha". Embora a regra de isolamento seja estrita, mecanismos de callback permitem que camadas inferiores comuniquem com as superiores de forma segura. Este estilo promove atributos de qualidade como a modificabilidade, portabilidade e reusabilidade.
- **Use cases:** sistemas empresariais, aplicações web, sistemas operativos.
- **Quando usar:** quando há necessidade de separar responsabilidades em níveis distintos.
- **Vantagens:** separação de responsabilidades, facilidade de **manutenção**, **reutilização** e **portabilidade** de componentes, testabilidade.



Stack de layers

Dependências:

```
Layer 3 (Presentation)
  │ uses
  ↓
Layer 2 (Business Logic)
  │ uses
  ↓
Layer 1 (Persistence)
  │ uses
  ↓
Layer 0 (Data)

Dependências só para baixo!
```

Quality Attributes:

- Modifiability (mudanças isoladas)
- Portability (trocar layers)

- Reusability (layers reutilizáveis)

Pipe-and-Filter

- Processamento de dados em série através de uma cadeia de componentes (filtros) conectados por canais (pipes)
- **Restrição:** processamento unidirecionalidade do fluxo de dados. **Filtros independentes que não partilham estado**. Filtros não sabem quem está a montante ou a jusante
- **Qualidades:**
  - Reusabilidade: filtros independentes
  - Modificabilidade: adicionar/remover filtros facilmente
  - Concorrência: filtros podem ser executados em paralelo
- **Componentes:** Filtros, Pipes, portos de leitura (read ports) e de escrita (write ports).
- Sem loops, impede recursões
- Exemplo: A linha de comandos Unix: cat “f.txt” | grep “^Braga” | cut -f 2-.

```
Input → Filter1 → Filter2 → Filter3 → Output
         (pipe)    (pipe)    (pipe)
```

Batch-Sequential vs Pipe-and-Filter

Batch-Sequential

- Processamento de dados em blocos (processamento coarse-grained)
- Alta latência: cada estágio / bloco deve terminar totalmente a tarefa e escrever o resultado (frequentemente em disco) antes do próximo começar
- Baseado em ficheiros
- Sem concorrência
- **Quando usar:** quando o processamento é pesado e a latência não é crítica

Pipe-and-Filter

- Processamento de dados em fluxo contínuo (processamento fine-grained)
- Baixa latência: o processamento começa assim que os primeiros dados chegam
- Baseado em streams (fluxos)
- Concorrência possível
- **Quando usar:** quando a latência é crítica e o processamento é leve

---

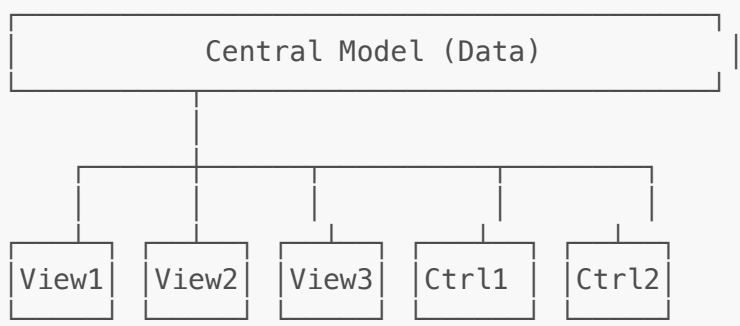
Centrado no Modelo (Model-Centered)

Este estilo centraliza o estado num repositório comum.

- **Componentes:** Modelo, Vistas (Views) e Controladores (Controllers).
- **Estratégia:** As vistas e controladores interagem exclusivamente através do modelo. Esta abordagem é particularmente valiosa quando não se conhece a configuração futura do sistema, permitindo

adicionar novas vistas ou controladores sem impactar os existentes, facilitando também a persistência e a gestão de concorrência.

- **Uso:** quando a configuração futura é desconhecida.
- **Vantagem:**
  - Desacoplamento entre views
  - Views podem ser independentes
  - Controllers não conhecem views
  - Flexibilidade extrema
- **Qualidades:**
  - Modificabilidade: adicionar/remover views/controllers facilmente porque são desacoplados
  - Extensibilidade: novo tipo de view/controller pode ser adicionado sem alterar o modelo
  - Reusabilidade: views/controllers podem ser reutilizados com diferentes modelos
  - Concorrência: múltiplas views/controllers podem operar simultaneamente no mesmo modelo



Publish-Subscribe (Pub-Sub)

Baseia-se no anúncio de eventos através de um barramento central (Event Bus).

- **Mecanismo de Confiança:** Existe um contrato de confiança mútua implícito; os publicadores confiam que o barramento entrega os eventos, e os subscritores confiam que receberão apenas aquilo que subscreveram.
- **Regras de Ignorância:** Produtores e consumidores são mutuamente ignorantes (oblivious). O publicador não sabe quem consome o evento, o que torna o sistema altamente evoluível e fácil de manter.
- **Qualidades:**
  - Manutenibilidade: Mudanças isoladas porque os componentes são independentes
  - Escalabilidade: Adição de novos componentes sem impacto
  - Flexibilidade: Reconfigurações dinâmicas

- **Quando usar:** quando há necessidade de alta escalabilidade e flexibilidade

---

Client-Server & N-Tier

Define uma relação assimétrica onde os Clientes iniciam pedidos a Servidores. No modelo multicamada (N-Tier), o sistema é organizado em níveis com responsabilidades funcionais exclusivas:

1. **Apresentação (Presentation tier):** Exclusiva para interação com o utilizador.
2. **Lógica (Logic tier):** Coordena a aplicação e processa comandos.
3. **Dados (Data tier):** Responsável exclusiva pela persistência. A independência do servidor é crucial: este não deve conhecer a identidade do cliente até ser contactado.

- **Vantagens:**
    - Separação de responsabilidades: Cada tier tem responsabilidade clara
    - Escalabilidade: Tiers podem rodar em máquinas diferentes
    - Manutenibilidade: Mudanças isoladas por tier
    - Testabilidade: Cada tier testável independentemente
    - Reusabilidade: Lógica e dados podem ser reutilizados por diferentes clientes
- **Quando usar:** quando há necessidade de separar responsabilidades e suportar múltiplos tipos de clientes

---

Peer-to-Peer (P2P)

Ao contrário do modelo hierárquico, este é um estilo igualitário (egalitarian). Cada nó tem a capacidade (embora não a obrigação) de atuar simultaneamente como cliente e servidor. Esta simetria elimina a dependência de um nó central, promovendo a disponibilidade e a resiliência.

- Sem client/server
- Sem master/slave
- Sem controller/controlled
- Igualdade total
- Sem nó central
- Sem coordenador
- Descentralizado
- Distribuído
- **Qualidades:**
    - Disponibilidade: Sem ponto único de falha
    - Escalabilidade: Adição de nós sem impacto significativo
    - Resiliência: Rede adapta-se a falhas de nós individuais

- **Quando usar:** quando há necessidade de alta disponibilidade e resiliência

---

Map-Reduce

Estilo híbrido (Runtime & Allocation) desenhado para o processamento de grandes volumes de dados (Big Data).

- **Regra de Ouro:** O conjunto de dados deve ser divisível e passível de ser processado por funções de mapeamento (map) e redução (reduce).
- **Robustez:** O estilo é intrinsecamente tolerante a falhas; se um computador num aglomerado falhar, o sistema recupera a tarefa, distribuindo o cálculo por outras máquinas.
- **Qualidades:**
  - Scalability: Processamento distribuído em múltiplos nós
  - Fault-tolerance: Recuperação automática de falhas
  - Performance: Processamento paralelo eficiente
- **Quando usar:** quando há necessidade de processar grandes volumes de dados de forma eficiente e tolerante a falhas

---

Micro-services

Três Características Principais da Arquitetura de Micro-serviços (além da manutenção) **1. Escalabilidade Independente**

A arquitetura de micro-serviços permite escalar cada serviço conforme a procura específica. Por exemplo, o serviço de "disponibilidade de quartos" pode receber mais requisições durante picos de reservas, podendo escalar horizontalmente enquanto o serviço de "processamento de pagamentos" continua com menos instâncias. Isto contrasta com monolitos onde a aplicação inteira escala como um bloco.

2. Independência de Implantação e Desenvolvimento Cada serviço pode ser desenvolvido, testado e implementado independentemente. Equipas podem trabalhar em paralelo sem dependencies bloqueantes. Um serviço pode ser atualizado ou rollback sem afectar outros, reduzindo risco de downtime. A cadência de release pode variar por serviço.

3. Resiliência e Isolamento de Falhas

Se um micro-serviço falha (ex.: serviço de email), outros serviços continuam funcionais. A falha fica isolada, evitando cascata de falhas como num monolito. Padrões como circuit-breaker, retry e timeout permitem graceful degradation em vez de falha total do sistema.