Táticas de Design para Arquitetura de Software

As táticas de design constituem a ponte estratégica entre os requisitos de qualidade / não funcionais (ASRs - Architectural Software Requirements) e as decisões técnicas concretas, garantindo robustez e previsibilidade.

As táticas fundamentais dividem-se em seis categorias críticas:

- Disponibilidade (Availability)
- Modificabilidade (Modifiability)
- Desempenho (Performance)
- Segurança (Security)
- Testabilidade (Testability)
- Usabilidade (Usability)

> Táticas são usadas pelo arquiteto para criar um design As táticas são implementadas através de padrões arquiteturais e estilos, que fornecem soluções reutilizáveis para problemas recorrentes no design de software.

Disponibilidade (Availability)

A Disponibilidade é definida como a probabilidade de um sistema estar operacional e funcional de acordo com a sua especificação num determinado momento. Fórmula:

[Disponibilidade = \frac{MTBF}{MTBF + MTTR}]

Onde:

- **MTBF (Mean Time Between Failures):** Tempo médio entre falhas, representando a confiabilidade do sistema.
- **MTTR (Mean Time To Repair):** Tempo médio para reparar, refletindo a rapidez com que o sistema pode ser restaurado após uma falha.

Falha

- Quando um componente interno não funciona
- Não é necessariamente visível externamente pelo utilizador
- Pode ser disfarçada por mecanismos de tolerância a falhas

Exemplo: Disco servidor tem bad sector → Fault (hardware falha) → Mas RAID replica dados → User não percebe

Falta

- Quando o sistema não cumpre a sua função
- Visível externamente pelo utilizador
- Pode ser causada por falhas internas ou externas
- Impacto direto

Exemplo: Página web não carrega → Failure (user percebe) → Possível causa: disco bad sector (fault)

3 pilares da Disponibilidade

Todas abordagens endereçando availability envolvem algum tipo de:

1. **Redundancy (duplicação):**
    - Múltiplas cópias de componente
    - Se um falha, outro continua
    - Invisível para utilizador

Métodos:

- Hardware redundancy (múltiplos servidores)
- Software redundancy (múltiplas instâncias)
- Data redundancy (backups)

2. **Health Monitoring (deteção):**
    - Detectar quando uma falha ocorre
    - Rápida detecção => rápida recuperação

Métodos:

- Ping/echo (está vivo?): Um componente emite um ping e espera receber um echo do componente sob escrutínio
- Heartbeat (bate o coração?): Um componente emite uma mensagem heartbeat periodicamente e outro componente escuta por ela
- Exceptions (levantou alerta?): Um método para detectar faults é encontrar uma exceção, que é levantada quando uma classe de fault é reconhecida.

3. **Recovery (recuperação):**
    - Quando falha detectada, recuperar
    - Automático (sistema reage) ou manual (admin intervém)

Métodos:

- **Failover** (mudar para backup)
- **Restart** (reiniciar componente)
- **Rollback** (voltar a estado anterior).

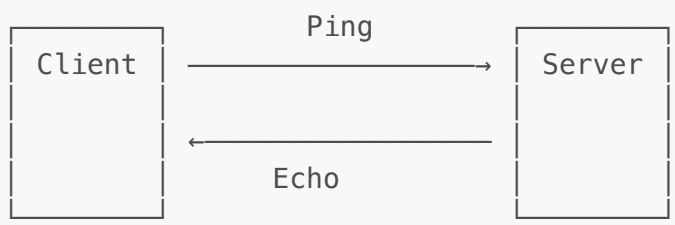
Monitorização ou recuperação é automática ou manual.

As táticas para melhorar a disponibilidade incluem:

Detecção de Faltas

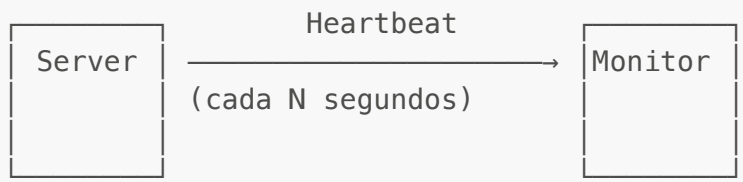
O sistema deve possuir mecanismos de introspeção para identificar anomalias precocemente:

- **Ping/Echo:** Verificação ativa onde um componente interroga outro para confirmar a vivacidade do serviço e do caminho de comunicação. Um componente emite um ping e espera receber um echo do componente sob escrutínio.




```
IF Echo recebido rapidamente → Operacional ✓
IF Echo não recebido → Falha detectada ✗
```

- **Heartbeat:** Emissão periódica de sinais de vida. A ausência do sinal dispara imediatamente um componente de correção de faltas. Um componente emite uma mensagem heartbeat periodicamente e outro componente escuta por ela.




```
IF Heartbeat recebido → Operacional ✓
IF Heartbeat FALHA → Alertar componente recovery
```

- **Exceptions:** Captura e tratamento de classes específicas de faltas reconhecidas durante a execução, evitando o crash do processo. Um método para detectar faults é encontrar uma exceção, que é levantada quando uma classe de fault é reconhecida.

```
try {
  risky_operation()
} catch (FaultException e) {
  // Fault detectado!
  notify_recovery_component()
}
```

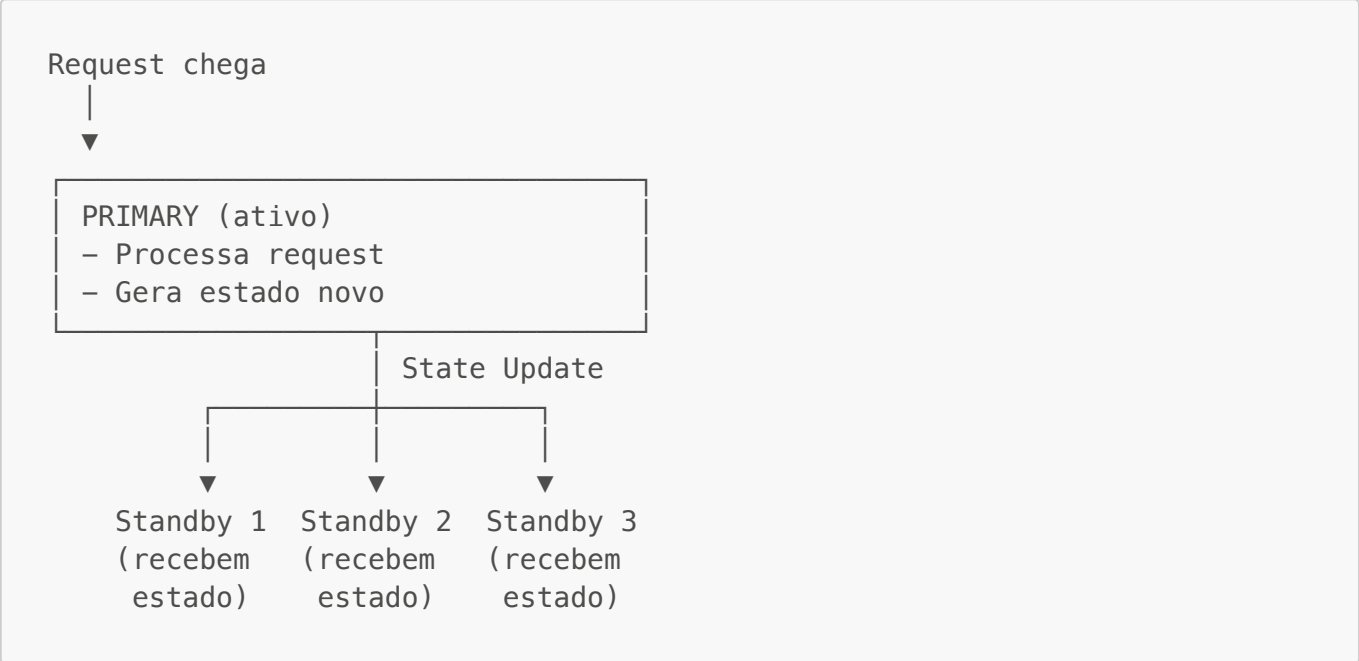
Recuperação de Faltas (Preparação e Reparação)

Objetivos:

- Corrigir o fault
- Voltar a estado consistente
- Minimizar dados perdidos

Estratégias para manter a operação mesmo na presença de componentes defeituosos:

- **Voting:** Múltiplos processadores redundantes executam a mesma tarefa. Um algoritmo de votação (ex: "maioria ganha") decide o resultado final, permitindo identificar e isolar um nó que apresente cálculos inconsistentes.
- **Active Redundancy:** Todos os componentes processam o estímulo em paralelo. O sistema utiliza a primeira resposta válida e descarta as restantes, **minimizando a latência de resposta**.
- **Passive Redundancy (Primary/Standby):** Apenas o componente primário processa pedidos e sincroniza o seu estado com os secundários. Nota de Design: Em caso de falha do primário, o sistema deve garantir que o backup está totalmente sincronizado antes de assumir a carga, o que introduz uma latência inerente à transição.



- **Spare:** Mantém recursos de reserva prontos a serem provisionados para substituir componentes falhados.

Recuperação de Faltas (Reação)

Trazer um componente de volta ao cluster exige cautela técnica:

- **Shadow:** O componente reintroduzido opera em paralelo com o sistema ativo, processando dados reais mas sem entregar a resposta ao utilizador. Isto permite verificar a consistência e o comportamento do componente antes de o tornar oficial.
- **State Resynchronization:** Essencial para sistemas stateful. Envolve a transferência de estado incremental ou total para o componente que regressa, garantindo que ele possui a versão mais recente dos dados do sistema.
- **Rollback:** Em sistemas onde a corrupção de dados é detectada, o arquiteto deve permitir que o sistema regresse a um "ponto de restauro" (checkpoint) estável conhecido.

Prevenção de Faltas

Evitar falha = melhor que recuperar falha

- **Removal from Service:** Manutenção preventiva, como reinícios agendados para mitigar memory leaks ou fragmentação de recursos antes que causem uma falha.

```
1. Detector: Memory > Threshold?
2. SIM → Iniciar graceful shutdown
3. Redireciona requests para outro servidor
4. Aguarda requests atuais terminarem
5. Shutdown servidor antigo
6. Restart servidor (limpo memory)
7. Re-adiciona ao pool
```

- **Transactions:** Implementação das propriedades ACID para evitar estados inconsistentes e colisões entre threads simultâneas:
    1. *Atomicidade:* Garante que operações complexas sejam "tudo ou nada".
    2. *Consistência:* Previne a gravação de dados que violem regras de integridade, evitando que threads paralelas corrompam a lógica de negócio.
    3. *Isolamento:* Garante que transações simultâneas não interfiram umas com as outras, crucial para evitar race conditions.
    4. *Durabilidade:* Assegura que, uma vez confirmada a transação, os dados resistam a falhas de energia ou sistema.
- **Process Monitor:** Um watchdog externo que monitoriza a saúde dos processos e tem autoridade para terminar instâncias zumbis e instanciar novas.

```
1. MONITOR deteta: Worker não responde (heartbeat falta)
2. MONITOR deleta: Kill process (liberta recursos)
3. MONITOR cria: Nova instância do processo
4. Nova instância: Começa fresco (estado limpo)
```

Desempenho (Performance)

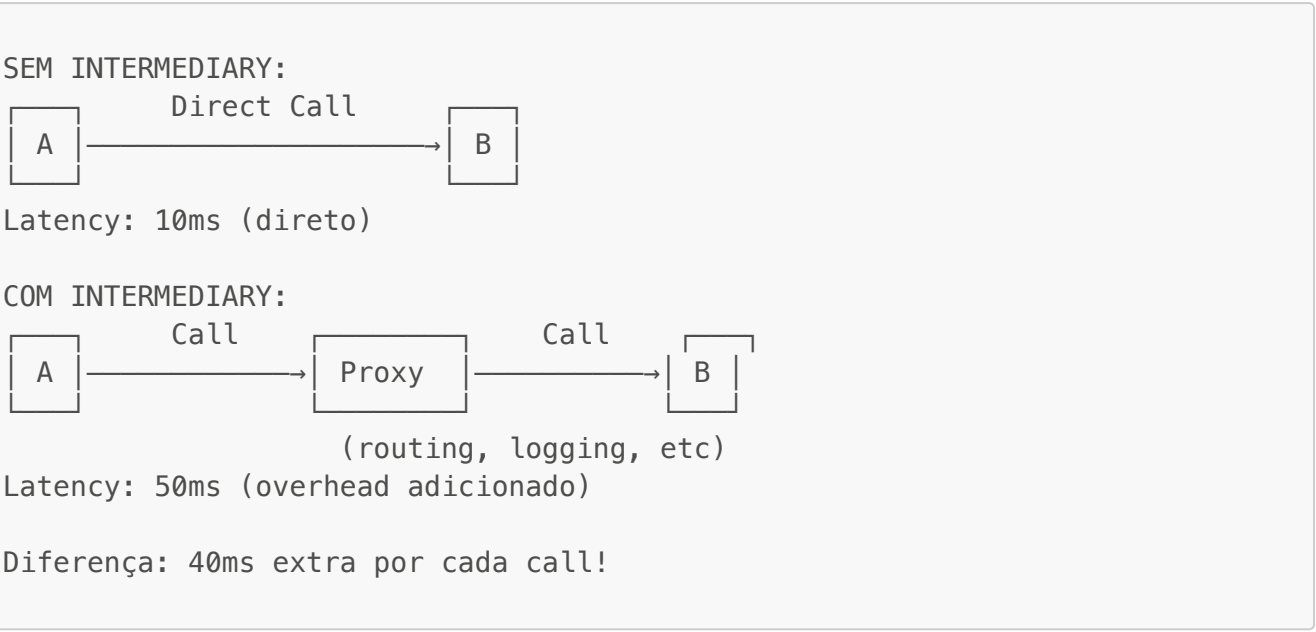
O desempenho refere-se à capacidade do sistema de responder a estímulos externos dentro de limites temporais aceitáveis. As táticas de desempenho focam-se em otimizar a latência, throughput e utilização de recursos.

Latência: Tempo decorrido entre o envio de um pedido e a receção da resposta.

Táticas de Gestão da Demanda de Recursos

- **Eficiência Computacional:** Refinamento de algoritmos para reduzir o custo de processamento por evento.
    - Algoritmo mais rápido => menos CPU usado
    - Menos CPU => menor latency

- **Redução de Overhead:** O arquiteto deve identificar onde a remoção de intermediários (como wrappers ou brokers) pode beneficiar a latência.



- **Controle da Taxa de Eventos:** Em sistemas sobredimensionados, a redução da frequência de amostragem (sampling) para monitorização diminui a carga. Aviso Crítico: O arquiteto deve estar ciente de que reduzir a frequência de amostragem em fluxos de eventos externos pode resultar na perda definitiva de pedidos/dados.
  - Monitorar mais frequente => mais trabalho
  - Reduzir frequência => menos trabalho
- **Concorrência e Balanceamento:** Introduzir threads paralelas e distribuir a carga para reduzir o tempo de bloqueio.
- **Múltiplas Cópias e Réplicas:** Reduzir a contenção em servidores centrais distribuindo a computação (ex: arquitetura cliente-servidor).
- **Caching:** Réplica de dados em repositórios de alta velocidade. Trade-off: O uso de cache introduz a complexidade de **manter as cópias consistentes e sincronizadas**, o que **pode gerar overhead de rede e processamento**.
- **Aumento de Recursos:** Escalabilidade vertical ou horizontal.
- **Arbitragem de Recursos**
- **Scheduling Policy:** É o coração da arbitragem em cenários de contenção. O arquiteto deve definir políticas de escalonamento (ex: Prioridade Fixa vs. Prioridade Dinâmica) para gerir como e quando um recurso é alocado a tarefas concorrentes, priorizando o que é crítico para o negócio.

MNEMÔNICAS E PALAVRAS-CHAVE

- **Disponibilidade: DER**

```
D = Detection (Ping, Heartbeat, Exception)
E = Emergency Recovery (Voting, Active, Passive)
R = Repair (Removal from service, Monitor, Transaction)

Dica: "DER = Detect, Emergência Recovery, Repair"
```

- **Desempenho: DMA**

```
D = Demand reduction (efficiency, tradeoff, rate)
M = Management (concurrency, copies, cache)
A = Arbitration (allocate resources)

Dica: "DMA = Demand, Management, Allocation"
```

Resumo Final

```
╔═══════════════════════════════════════════════════════╗
║          DESIGN TACTICS - RESUMO EXECUTIVO            ║
╠═══════════════════════════════════════════════════════╣
║                                                       ║
║ TACTIC = Design decision impacting quality            ║
║                                                       ║
║ AVAILABILITY:                                         ║
║ ├─ Detection: Ping, Heartbeat, Exception              ║
║ ├─ Recovery: Voting, Active, Passive Redundancy       ║
║ └─ Prevention: Removal, Transactions, Monitor         ║
║                                                       ║
║ PERFORMANCE:                                          ║
║ ├─ Demand: Efficiency, Tradeoff, Rate                 ║
║ ├─ Management: Concurrency, Copies, Caching           ║
║ └─ Arbitration: Resource allocation                   ║
║                                                       ║
║ KEY METRICS:                                          ║
║ ├─ Availability = MTBF/(MTBF+MTTR)                    ║
║ └─ Latency & Throughput                               ║
║                                                       ║
║ TRADE-OFFS:                                           ║
║ ├─ Performance vs Modifiability                       ║
║ ├─ Availability vs Cost                               ║
║ └─ Consistency vs Availability                        ║
║                                                       ║
╚═══════════════════════════════════════════════════════╝
```