8 - Pensamento Baseado em Componentes

Sunday, 16 March, 2025 7:42 Manhã

- Componente - como um empacotamento
- Agrupar artefato
    - Library - conjunto de classes/funções
    - Subsistema
    - Serviço - comunica via TCP/IP por exemplo

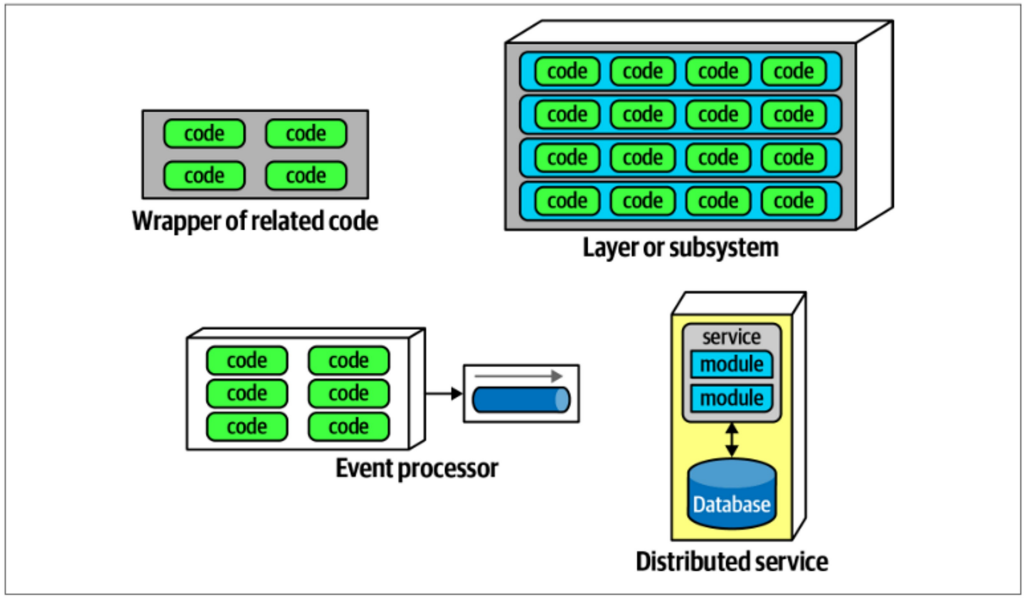


Figure 8-1. Different varieties of components

- Componente pode ser extremamente simples

- Arquitetura em geral independe do desenvolvimento
- Componente é o nível mais básico que o arquiteto lida
    - (exceto para tirar métricas de código, em que lida com código)
- A forma de juntar esses componentes é um trade-off
    - E.g. monolítico em camadas
    - Monolítico modular, etc.

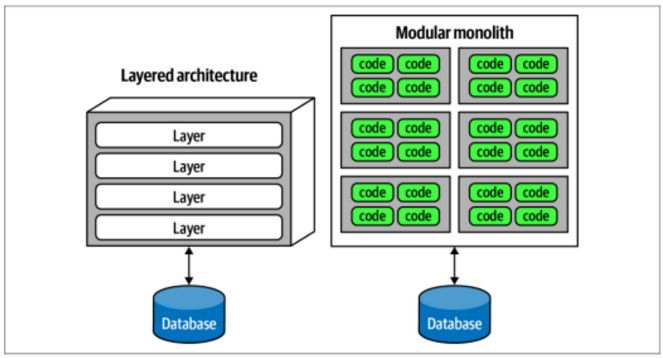


Figure 8-3. Two types of top-level architecture partitioning: layered and modular

- Cada componente pode ter outros componentes
- Particionamento a nível:
    - Domínio?
        - Organizado mas pode ter repetições
    - Técnico?
        - Pode ter operações que aparecem em todas as camadas

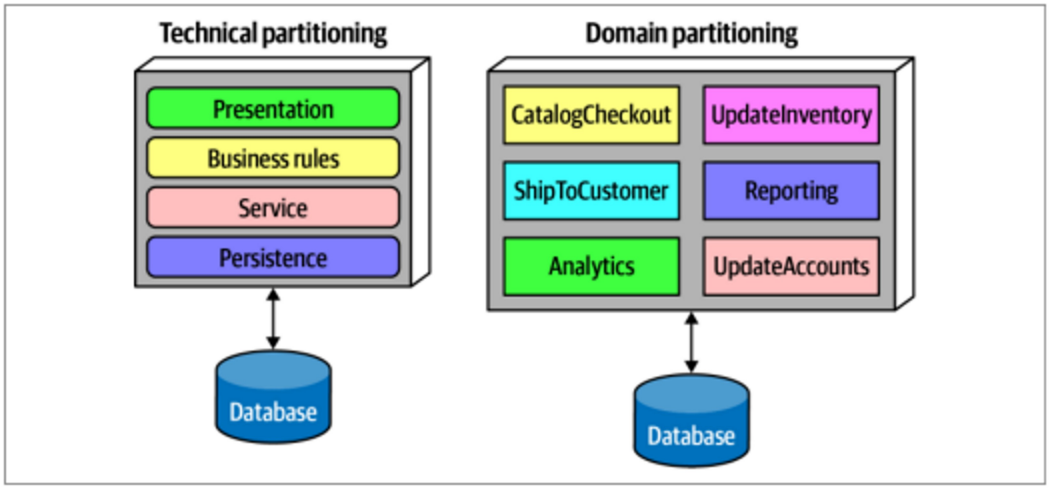


Figure 8-4. Two types of top-level partitioning in architecture

- Arquitetura em camadas é bem comum
- **Lei de Conway** - Arquitetura imita estrutura de comunicação da própria empresa
- Por domínio parece ser tendência, mas é um trade-off

Estudo de Caso: Silicon Sandwiches

- Duas opções, técnicamente e por domínio:

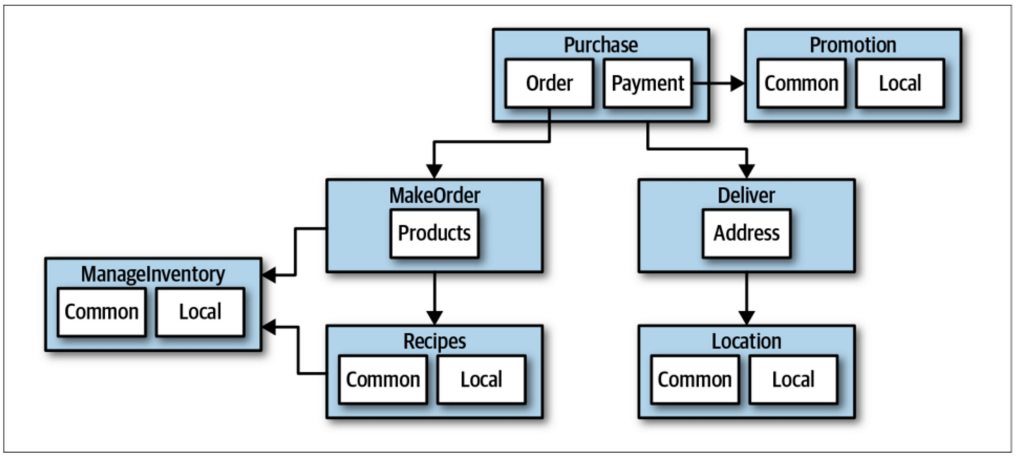


Figure 8-6. A domain-partitioned design for Silicon Sandwiches

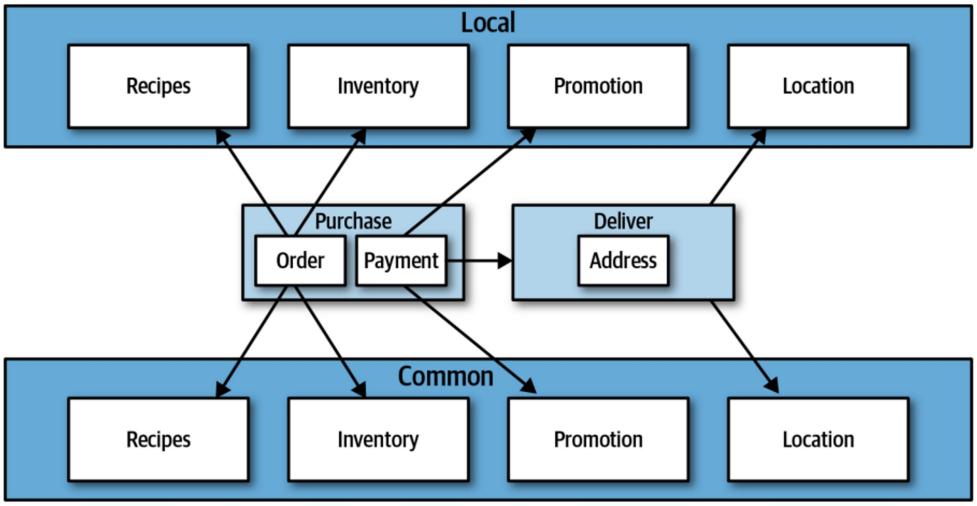


Figure 8-7. A technically partitioned design for Silicon Sandwiches

- Particionamento por domínio:
  - Vantagens:
    - Representa melhor funcionamento do negócio
    - Facilita criar equipes multifuncionais
    - Combina bem com arquitetura monolítica modular e de microsserviços
    - Facilita migrar dados e componentes para a arquitetura distribuída
  - Desvantagens:
    - Mas código pode ficar repetido em vários domínios
- Particionamento técnico:
  - Vantagens:
    - Separa bem o código
    - Combina bem com arquitetura em camada
  - Desvantagens:
    - Alto grau de acoplamento, toda alteração afeta tudo.
    - Pode haver duplicação de "local" e "common" mesmo assim
    - Alto acoplamento de dados, difícil migração.

- Desenvolvedores dividem cada componente definido pelo arquiteto em classes e funções.
  - Essa divisão é responsabilidade de ambos arquitetos e desenvolvedores.
  - Devem interagir e iterar sobre o design projetado.
  - Três etapas em Loop:
    - Analisar funções e responsabilidades
    - Analisar as características da arquitetura
      - Cada componente tem uma especificação
        - E.g. se dois componentes lidam com login de usuários mas em quantidades diferentes, suas especificações devem ser diferentes
    - Reestruturar os componentes
      - Incorporar feedback dos desenvolvedores
- Dificilmente o design inicial será bom

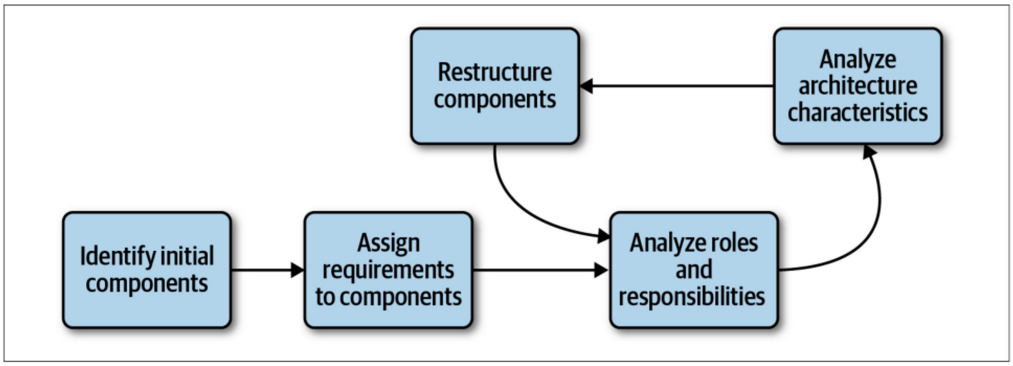
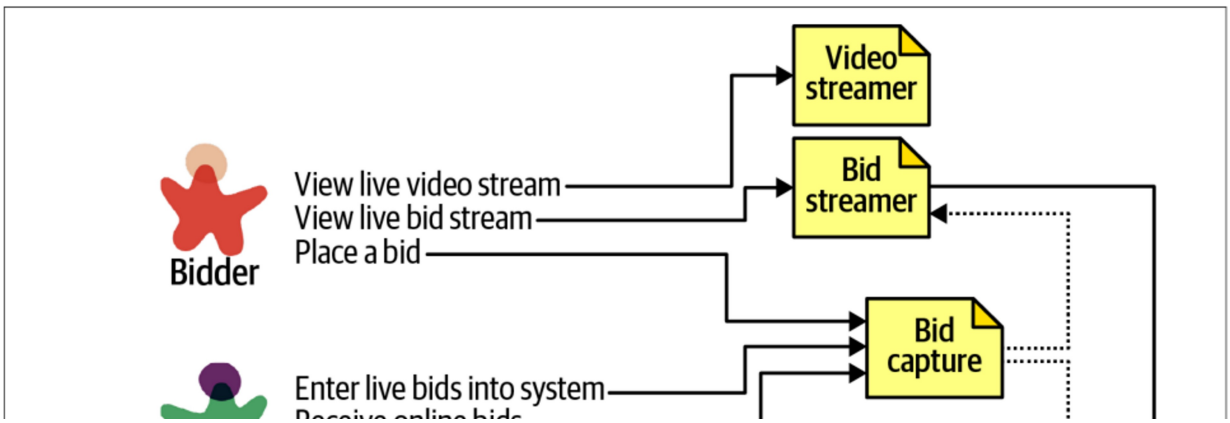


Figure 8-8. Component identification cycle

- Definir a granularidade ideal do componente é importante e difícil.
  - Pouco granular --> Alto acoplamento interno, difícil implementar e testar
  - Muito granular --> Muita comunicação entre os componentes
- Como definir os componentes?
  - **Armadilha da entidade**:
    - Antipadrão. Confundir arquitetura com entidades de um banco de dados relacional.
    - Confundir relações do banco de dados como fluxo de trabalho na aplicação.
      - (Naked Objects - família de frameworks para criar CRUDs simples)
  - Abordagem *ator/ações*:
    - Identificar os quem e os quês que são feitos.
  - Event Storming:
    - Pressupõe-se o uso de mensagems/eventos.
    - Tenta-se descobrir quais eventos ocorrem e cria componentes em torno disso.
  - Abordagem do fluxo de trabalho:
    - Como um event storming genérico.
    - Imaginam-se os fluxos de trabalho e criam-se componentes baseados nisso.

- Exemplo: Caso do Leilão
  - Abordam ator/ações é genérica e funciona bem
  - Ações:
    - Proponente (bidder)
      - Exibe vídeo ao vivo, faz um lance, etc.
    - Leiloeiro
      - Insere os lances, recebe os lances etc
    - Sistema
      - Inicia o leilão, faz o pagamento, etc.
  - Com isso, podemos imaginar a seguinte arquitetura com os componentes:



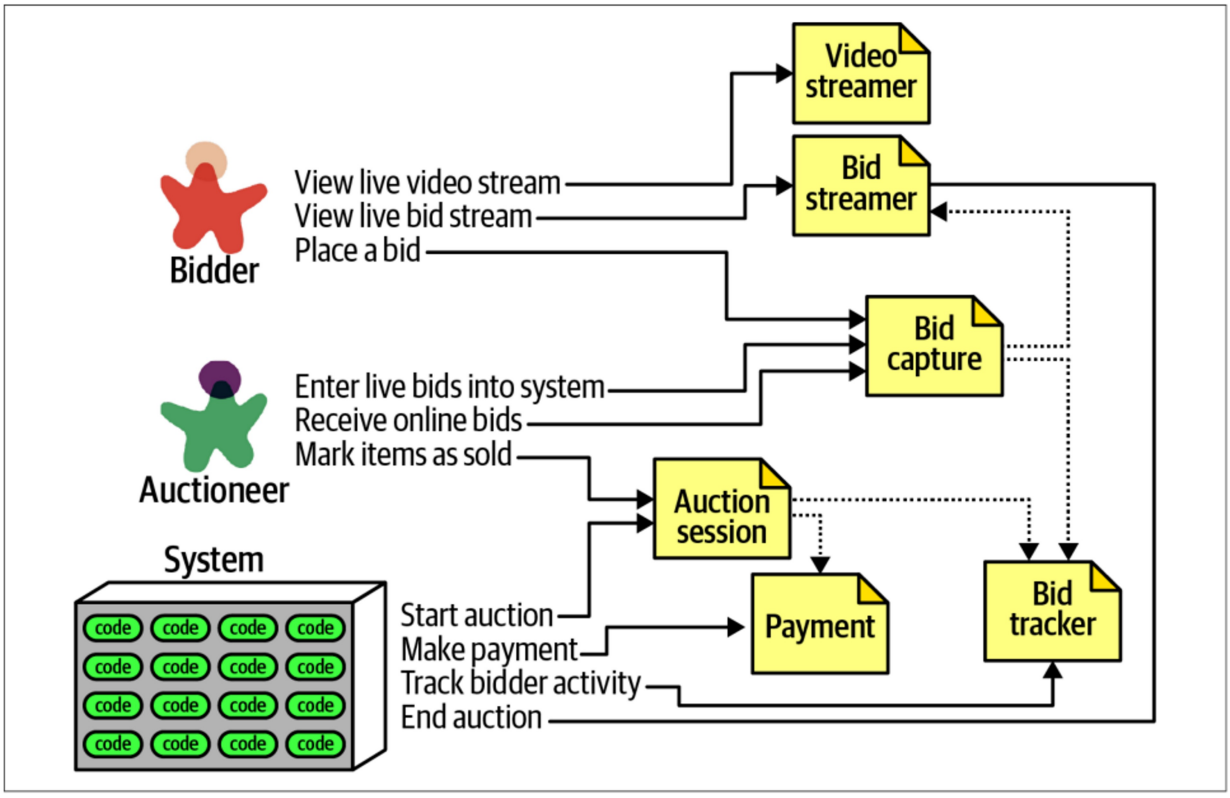


Figure 8-10. Initial set of components for Going, Going, Gone

- A partir disso podemos pensar em como essa estrutura funcionaria, analisando as características.
- Elasticidade do sistema para leiloreiros precisa ser muito menor do que para proponentes.
- Confiabilidade dos sistemas para o leiloeiro é mais importante do que dos proponentes.
  - Têm características diferentes.
  - Sendo assim, podemos dividir o item "Bid Capture" em um componente para o leiloeiro e um para o proponente.
  - O componente "Bid Tracker" agregará ambos.

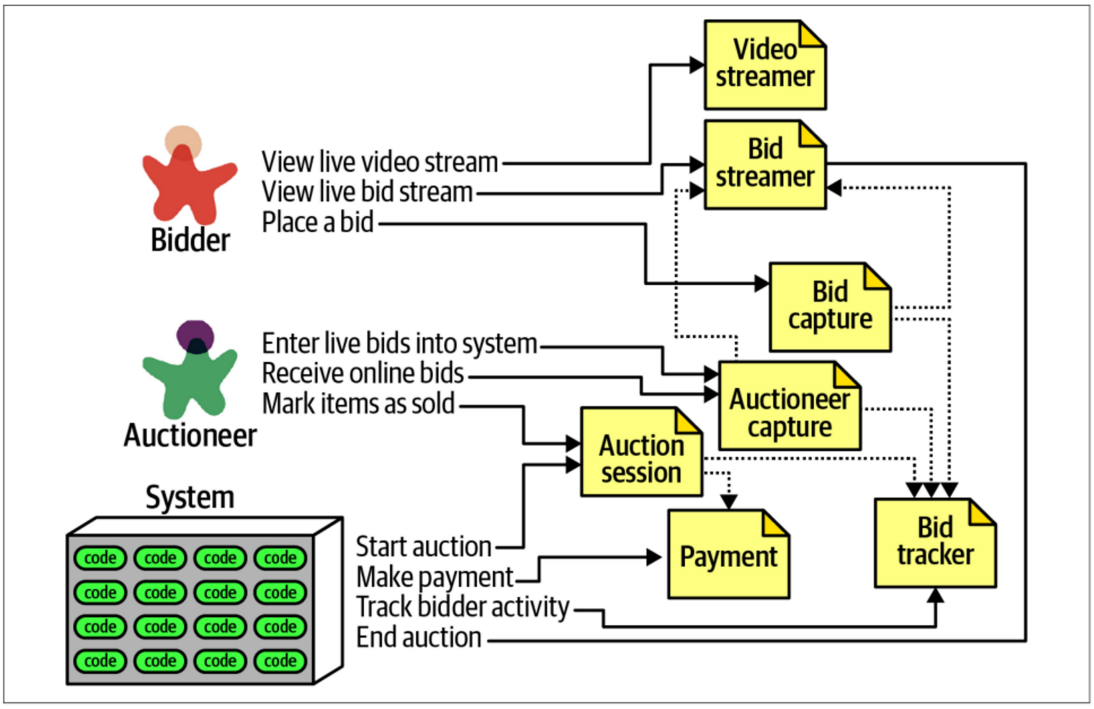
- 

Figure 8-11. Incorporating architecture characteristics into GGG component design

- Existem muitas formas de resolver este problema, nunca há uma solução única.

- A arquitetura deve ser monolítica ou distribuída?
  - Monolítica:
    - Uma única unidades de implementação
    - Um único banco de dados, em geral
  - Distribuída:
    - Vários serviços comunicando-se.
    - Mais independentes.
  - Um critério importante pra escolher é a quantidade de "Quantum de arquitetura" (i.e. conjunto de critérios de arquitetura).
    - Se um quantum bastar, estrutura monolítica costuma ser uma boa escolha.
    - Se requerer muitas quantums, estrutura distribuída pode fazer mais sentido.
    - No exemplo do leilão:
      - Características diferentes de componentes de *Capture* apontam para arquitetura distribuída.
      - Os dois streamers com requisitos diferentes (read-only intenso vs read/write constante) também apontam para distribuído.