# Testes de Unidades e de Integração

Criado: Junho/2006

Últ. Atualização: Abr/2013

# Tópicos

- Testes de Unidades: modelo de falhas
- Testes de Integração: modelo de falhas
- Padrões
- Análise de Dependência

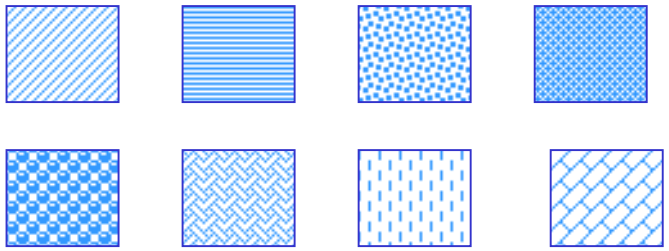
# Referências

R.Binder. Testing OO Systems. Addison Wesley, 2000, c. 12.

M.Pezzè, M.Young. Teste e Análise de Software. Artmed / Bookman Editora, 2008, c.21.

Gladys Machado Pereira Santos Lima e Guilherme Horta Travassos. *Testes de Integração Aplicados a Software Orientados a Objetos: Heurística para Ordenação de Classes,* Relatório Técnico, 2004, ES-632/04, COPPE/UFRJ.

- **Testes de Unidades**: busca de falhas em:
  - módulo ou função
  - classe
  - pequenos grupos de classes (clusters)
  - Componentes
  - Serviços



# Testes de Unidades

- Visam exercitar detalhadamente uma **unidade** do sistema.
- Sem perda de generalidade, consideraremos aqui uma unidade como sendo um **componente**, onde por componente entenda-se:
  - Entidade executável independente.
  - Seu código fonte pode ou não ser conhecido: depende se os testes estão sendo feitos pelo fornecedor ou pelo usuário do componente.
  - Pode representar:
    - Uma função.
    - Uma classe ou um tipo abstrato de dados.
    - Um grupo pequeno de classes.
    - Um *framework*.
    - Um sistema, cujo acesso se dá através de sua interface: gráfica ou API (*Application Programming Interface*).

# Modelos de falhas

- Os Testes de Unidade visam revelar a presença de falhas em:
  - interfaces: parâmetros de entrada e saída
  - estruturas de dados: integridade dos dados armazenados
  - condições de limite: a unidade opera adequadamente nos limites estabelecidos?
  - tratamento de erros

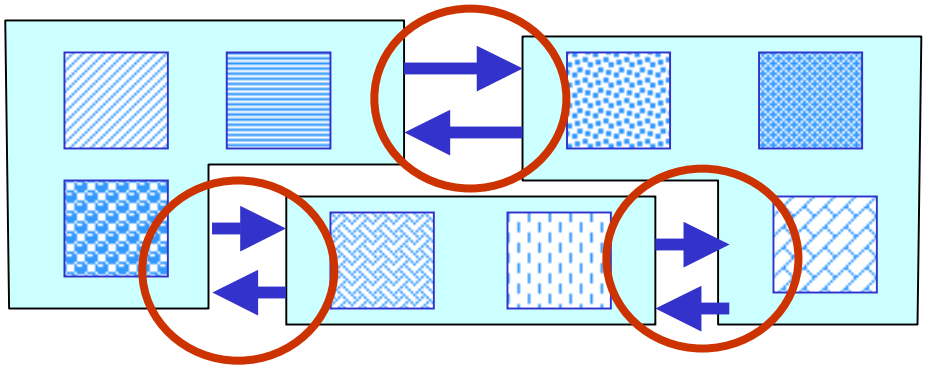
# Tipos de testes

- Caixa branca:
  - Mais comuns
  - Visam exercitar código (ou partes dele)

- Caixa preta:
  - Mais comuns: testes de interfaces
    - Visam determinar se implementação aceita dados válidos e rejeita dados inválidos
  - Menos comuns: testes baseados em modelos
    - Visam determinar se a unidade apresenta o comportamento especificado

# Testes de integração

- Integram unidades já testadas
- Visam descobrir problemas de interação e de compatibilidade entre as unidades testadas



# Falhas de integração

- **Falhas de interpretação**: ocorrem quando a funcionalidade implementada por uma unidade difere do que é esperado.
  - B implementa incorretamente um serviço requerido por A.
  - B não implementa um serviço requerido por A.
  - B implementa um serviço não requerido por A e que interfere com seu funcionamento.

- **Falhas devido a chamadas incorretas**:
  - B é chamado por A quando não deveria (chamada extra).
  - B é chamado em momento da execução indevido (chamada incorreta).
  - B não é chamado por A quando deveria (chamada ausente).

- **Falhas de interface**: ocorrem quando o padrão de interação (protocolo) entre duas unidades é violado.
  - violação da integridade de arquivos e estruturas de dados globais
  - tratamento de erros (exceções) incorreto
  - problema de configuração / versões
  - falta de recursos para atender a demanda das unidades
  - objeto incorreto é associado a mensagem (polimorfismo)

[Leung e White; Binder99]

# Mais falhas de integração

- Problemas não funcionais: ocorre quando requisitos não funcionais são violados
  - Módulo B não tem o tempo de resposta esperado por A
  - B lança exceções não esperadas por A
- Incompatibilidades dinâmicas: ocorrem quando a ligação dinâmica entre as unidades do sistema é permitida
  - Polimorfismo dinâmico entre classes
  - Conexões dinâmicas entre serviços ou componentes

[Pezzè e Young 2008]

# Importância

- Determinar se os diversos componentes de um sistema podem interoperar
  - Sistemas podem ser constituídos de componentes próprios e de terceiros (COTS)
- Desenvolvimento incremental
  - A cada novo incremento, testes de integração precisam ser realizados

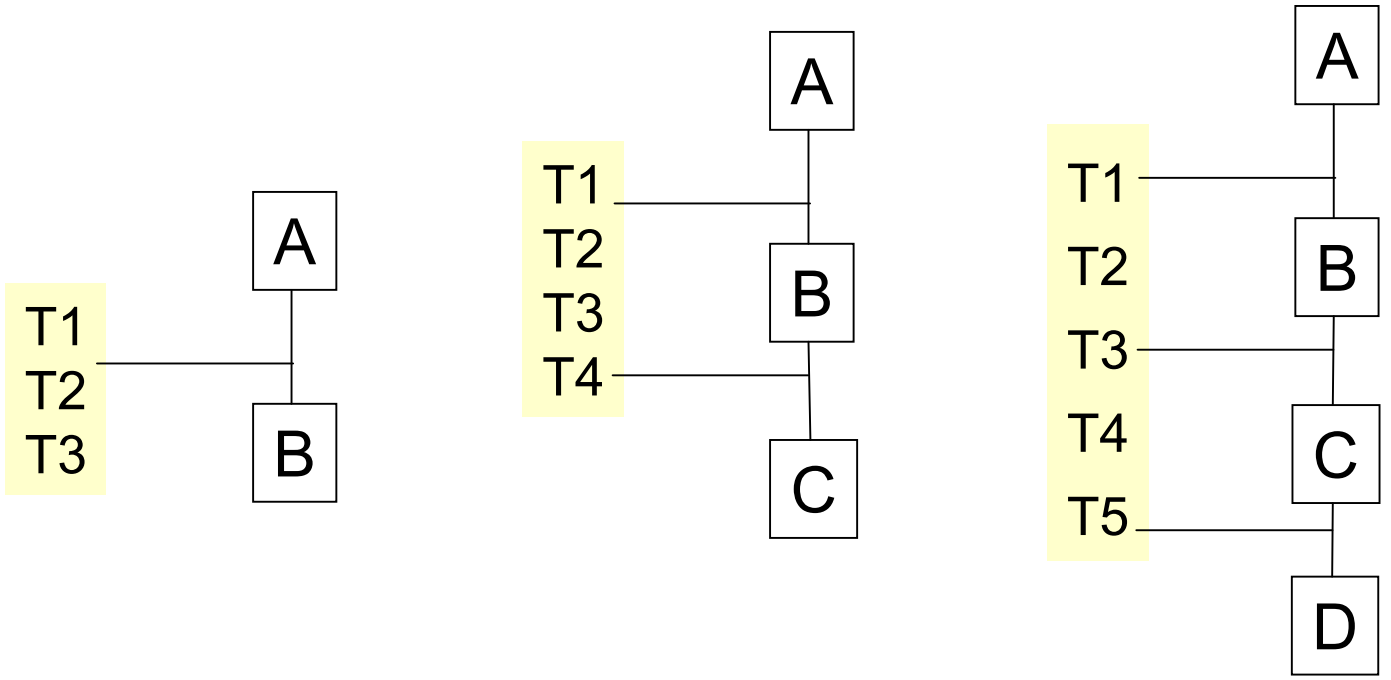
# Abordagens de integração

- Não incremental (“big-bang”):
- Incremental

# Abordagem *big-bang*

- Todas as unidades são integradas de uma só vez
  - ☺ Esforço de preparação menor
    - ☺Não necessita de drivers e stubs
  - ☹ Esforço para observação, diagnóstico e correção de falhas é maior

# Abordagem incremental



# Estratégia incremental

- Unidades são integradas aos poucos
- Segundo Pezzè e Young, as estratégias podem ser classificadas como:
  - Baseadas na estrutura do sistema
    - Uso do grafo de dependências
    - Baseadas na arquitetura
  - Orientadas por funcionalidades

# Análise de dependências

- Componentes podem depender uns dos outros de várias formas:
  - Composição e agregação (use de classes para definir atributos)
  - Herança
  - Variáveis globais
  - Chamadas a interfaces (API)
  - Objetos servidores
  - Objetos usados como parâmetros de mensagens
  - Ponteiros para objetos passados como parâmetros
  - Tipos de parâmetros usados na definição de classes genéricas

[Binder00, 13.1.3]

# Para quê serve

- Análise de dependências pode ser útil
  - Para determinar a ordem de testes
  - Para determinar impacto de modificações
  - entre outras

# Tipos de dependências

- Explícitas:
  - Troca de mensagens
  - Chamada de procedimentos
  - Uso de comunicação entre processos

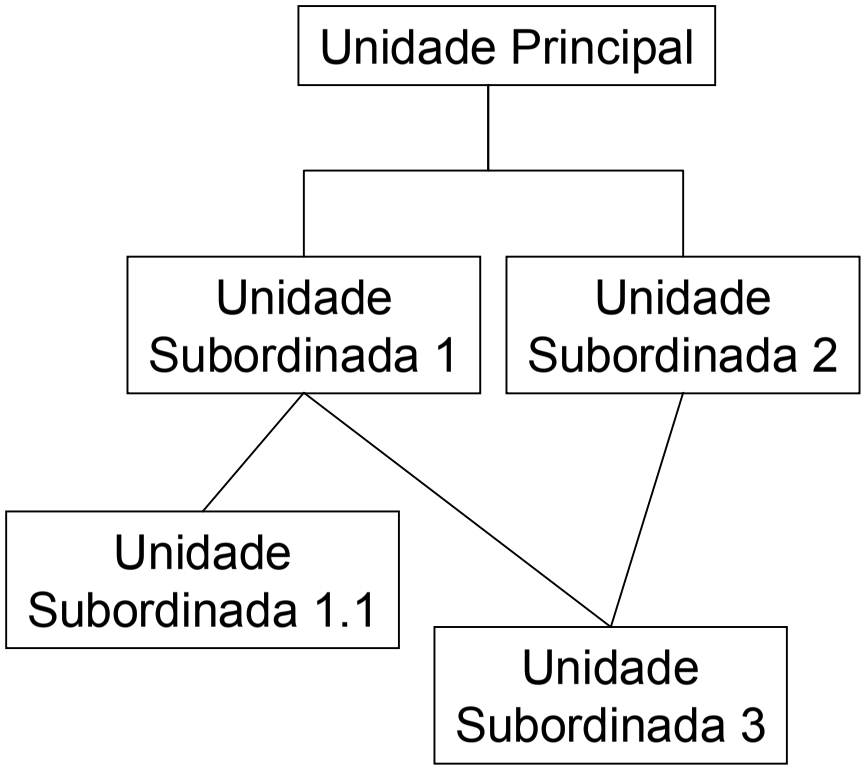
Foco das técnicas convencionais de integração

- Implícitas:
  - Comunicação através de dados persistentes
  - Restrições de seqüência
  - Restrições de tempo

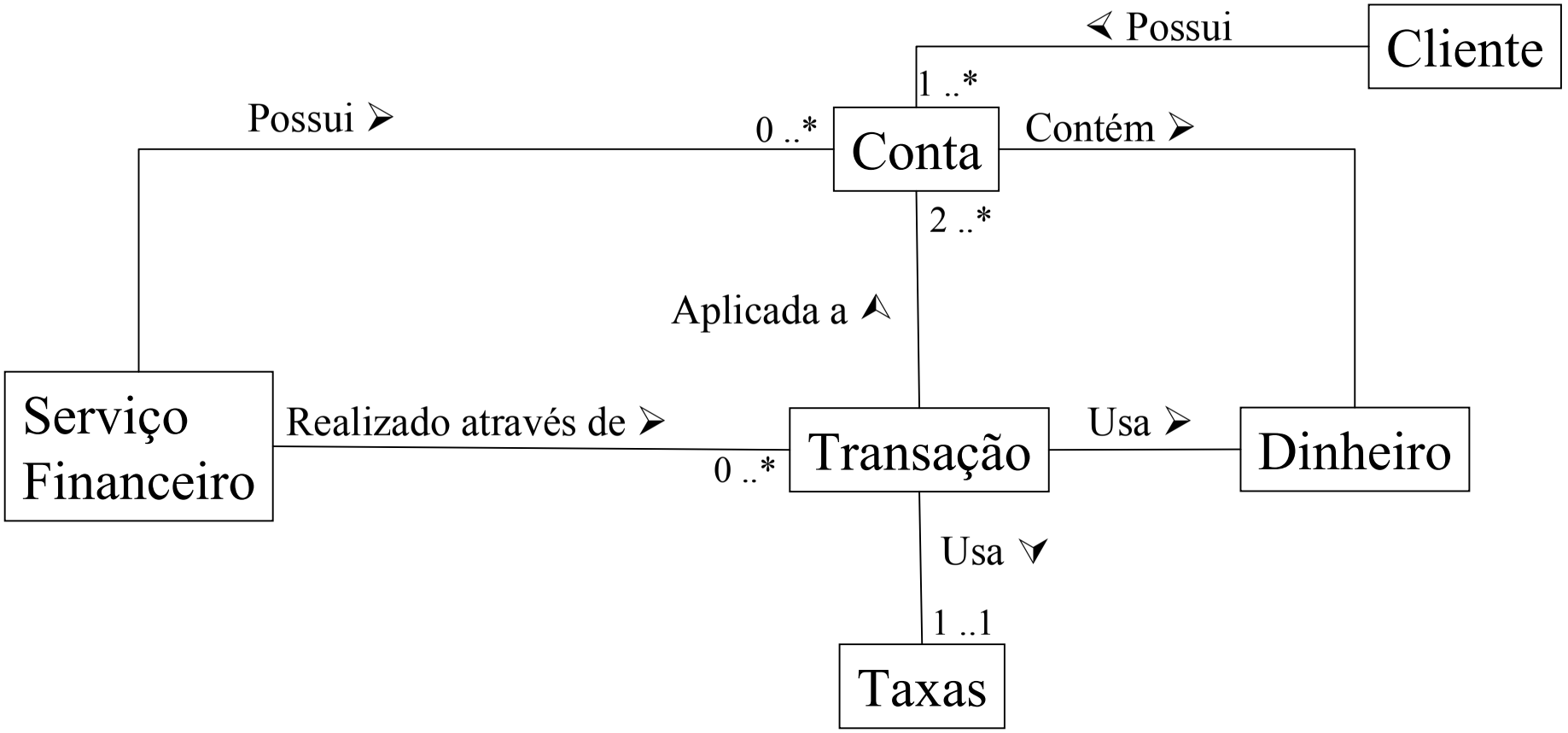
[Binder00, 13.1.3]

# Grafo de dependências

- Um modelo muito comum nos testes de integração
- Aplicável em paradigma estruturado ou OO
- **Nós:** unidades
  - Funções ou métodos
  - Objetos
  - Componentes
  - ...
- **Arcos:** dependências entre unidades:
  - A chama B
  - B é parte de A
  - A envia mensagem para B
  - ...

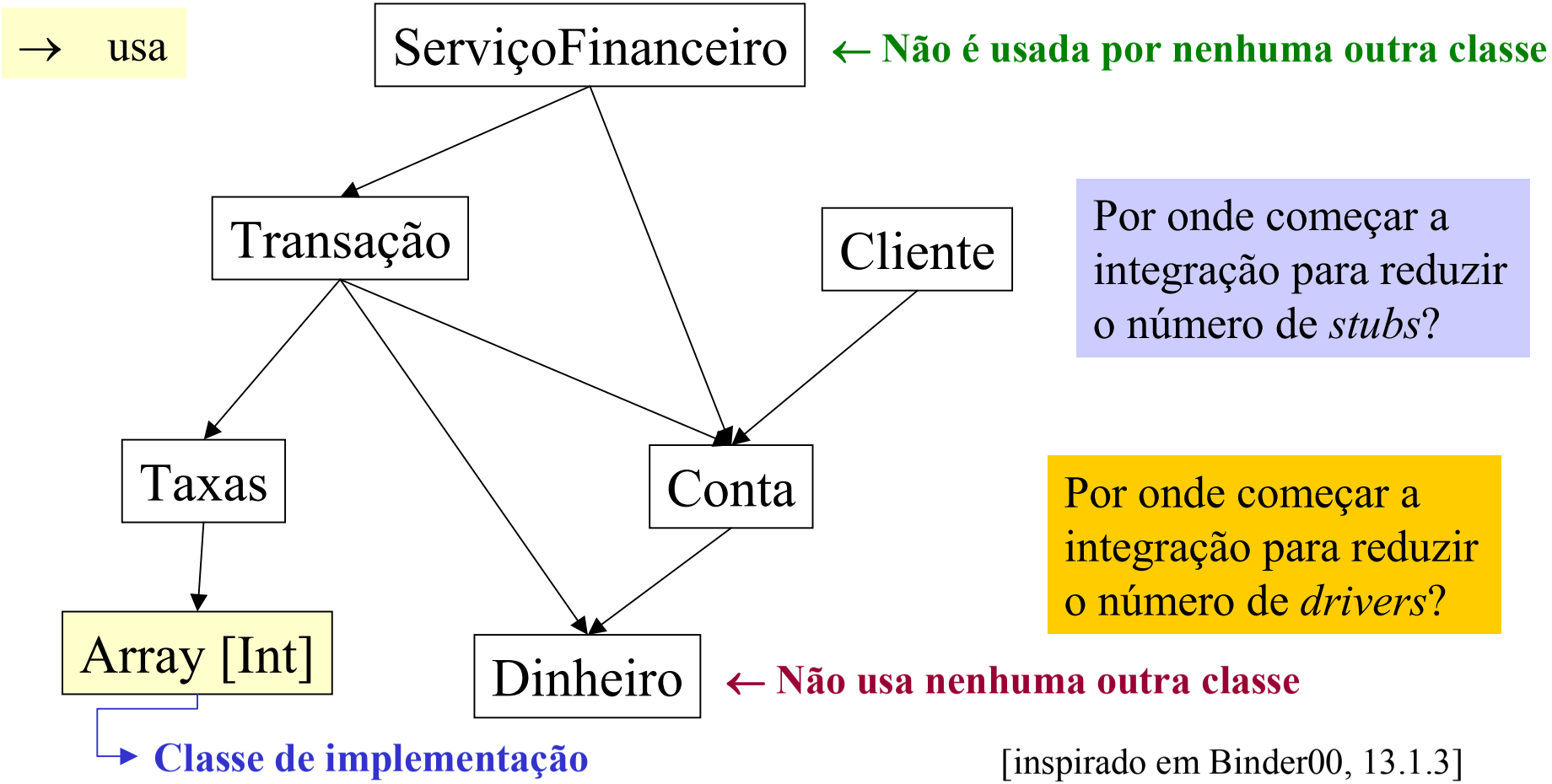


# Exemplo: Diagrama de Classes




[inspirado em Binder00, 13.1.3]

# Exemplo de dependência: *X* usa *Y*




Por onde começar a integração para reduzir o número de *stubs*?

Por onde começar a integração para reduzir o número de *drivers*?

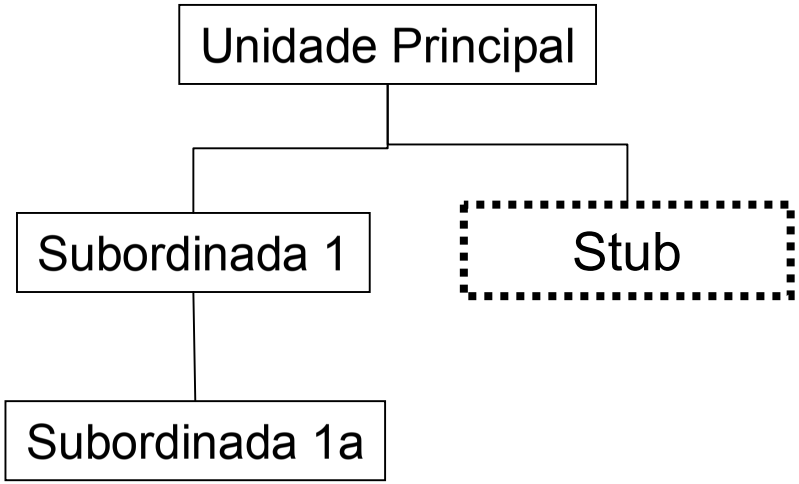
[inspirado em Binder00, 13.1.3]

# Critérios baseados no grafo de dependências

- Cobertura de nós:
  - Exercitar cada unidade pelo menos uma vez
  - Descendente (“top-down”)
  - Ascendente (“bottom-up”)
  - Mista (*backbone*)
- Cobertura de arcos:
  - Executar cada interação pelo menos uma vez
  - Integração por colaboração

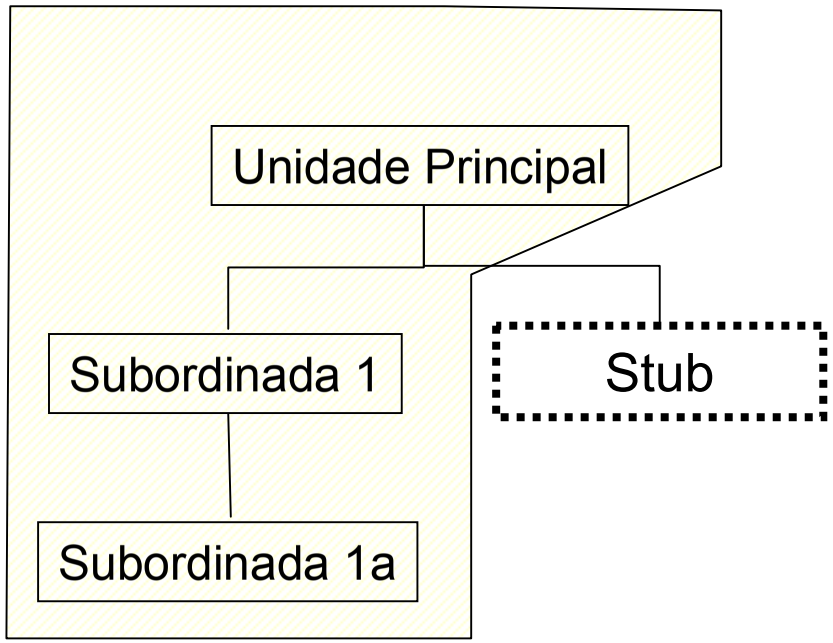
# Integração descendente (“top-down”)

- Começa com a unidade principal e vai aos poucos integrando as unidades subordinadas
- Em OO: classes de controle primeiro
- Utiliza ***stubs*** em lugar das unidades subordinadas

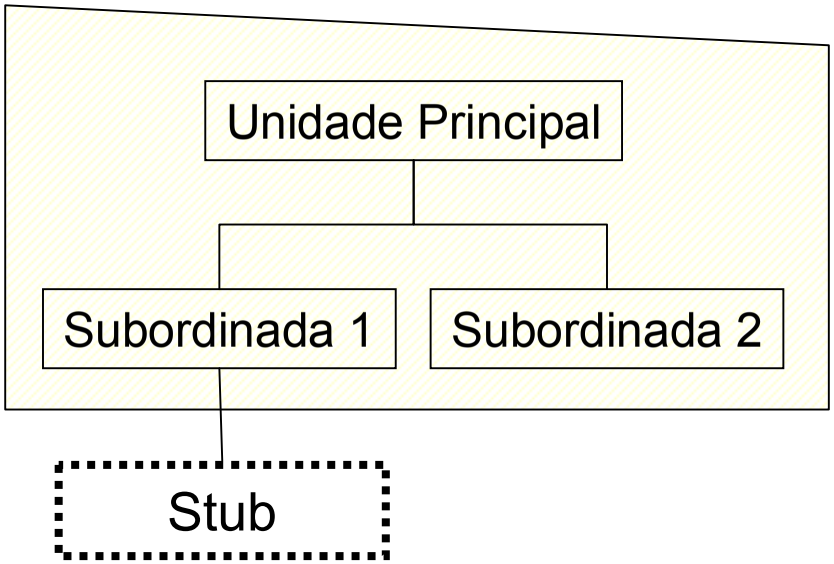


# Integração descendente (“top-down”)

- Pode ser feita em profundidade ou em largura
- interfaces de mais alto nível são testadas mais cedo



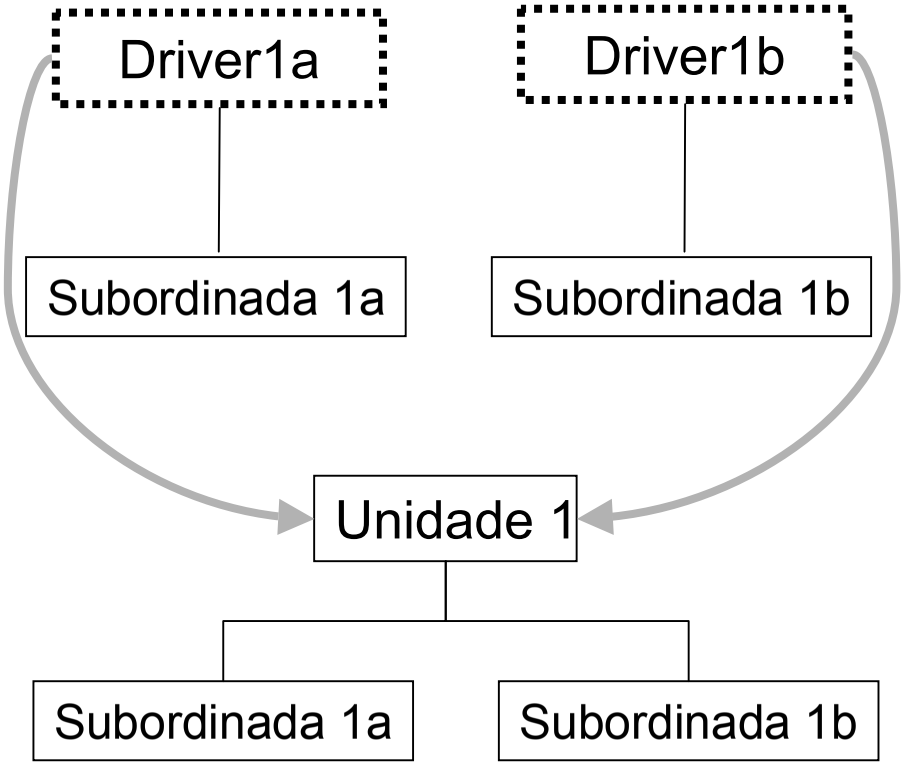

em profundidade




em largura

# Integração ascendente (“bottom-up”)

- Começa a integração pelas unidades subordinadas
- Em OO: começar pelas classes independentes ou que usam poucas servidoras
- Utiliza ***drivers*** em lugar das unidades de controle
- Os algoritmos de mais baixo nível são testados antes de serem integrados ao resto do sistema

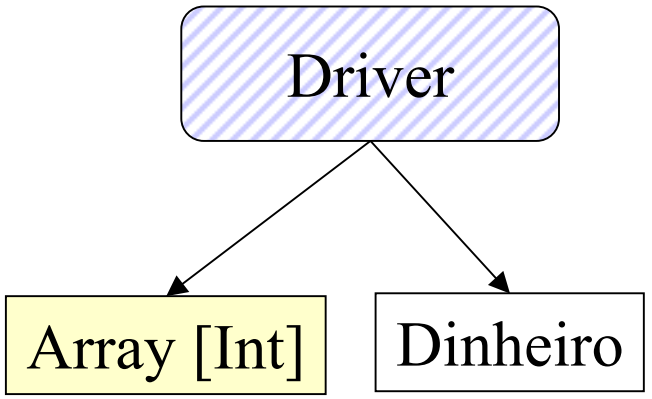


# Determinação da ordem de testes

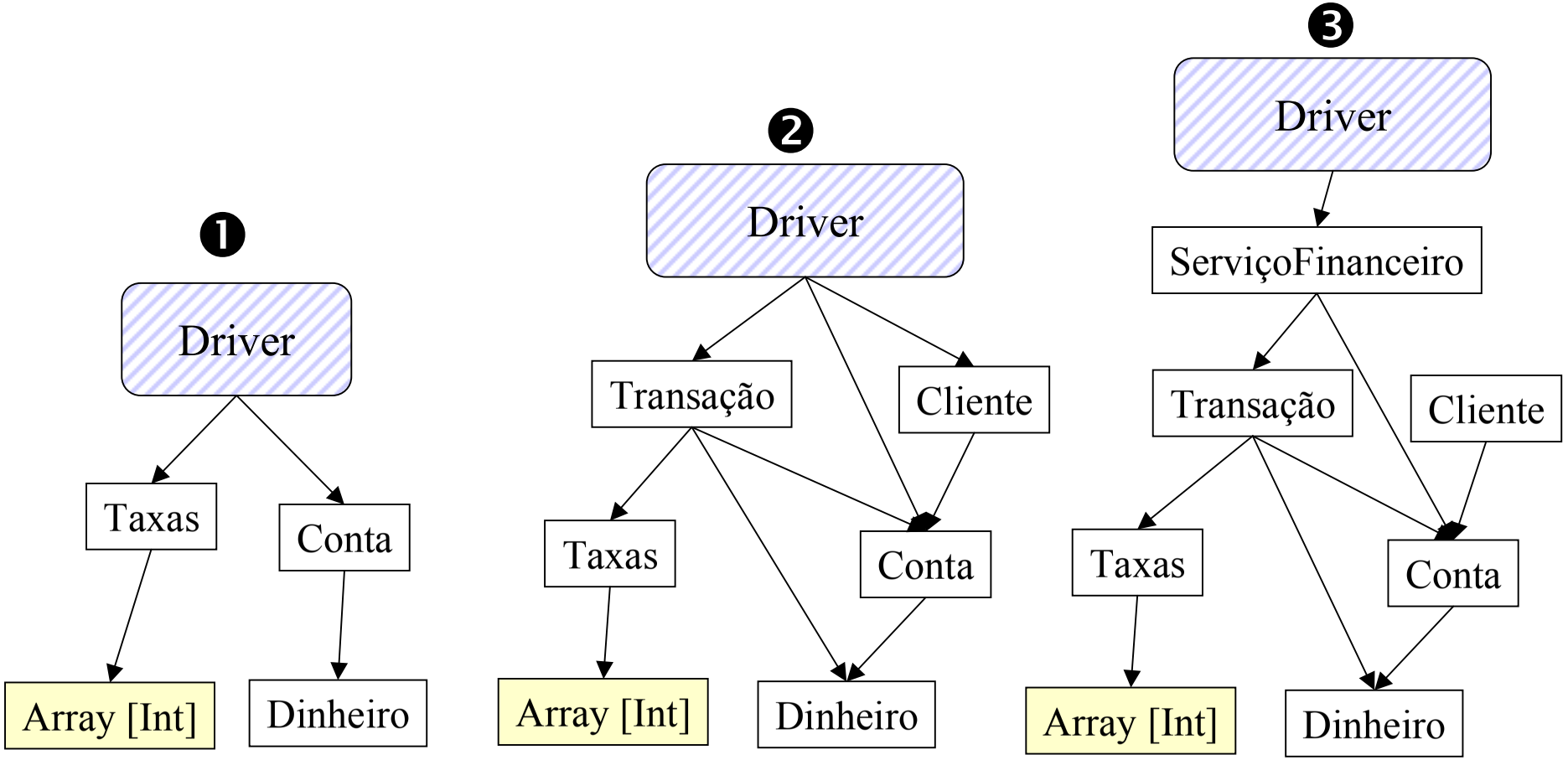
- Existem várias propostas com base no grafo de dependências:
  - Caso não existam ciclos: uso de algoritmos de ordenação (ex.: ordenação topológica)
  - Começar pelas classes que não dependem de nenhuma outra

    ⇨Uso de *drivers*
  - Começar pelas classes que não prestam serviços a nenhuma outra

    ⇨Uso de *stubs*



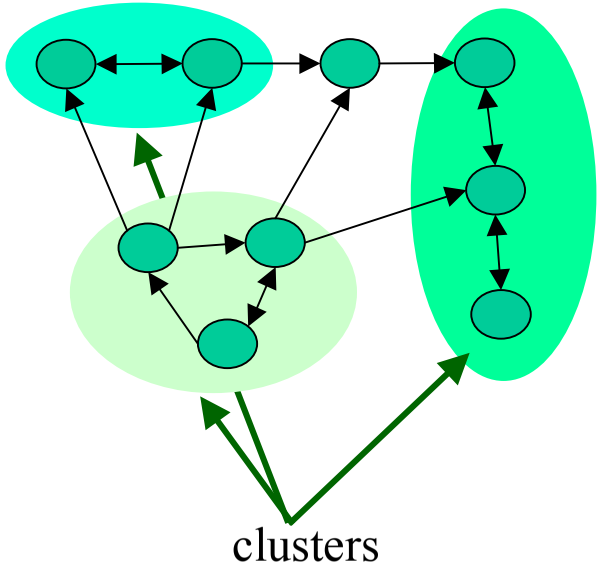
# Determinação da ordem de testes



# Detecção de Ciclos

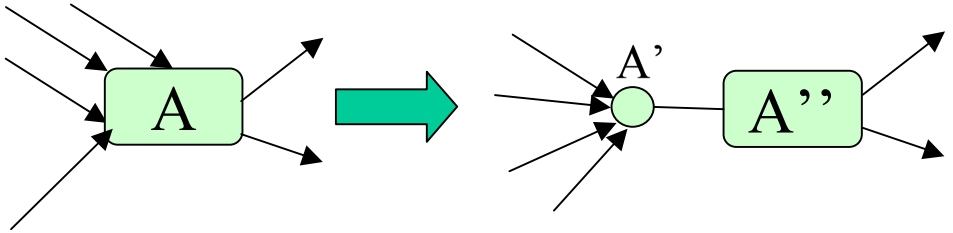
- Um aspecto importante do Grafo de Dependências é a possibilidade de detectar ciclos, isto é, elementos que estão fortemente acoplados

- Elementos que precisam ser refatorados

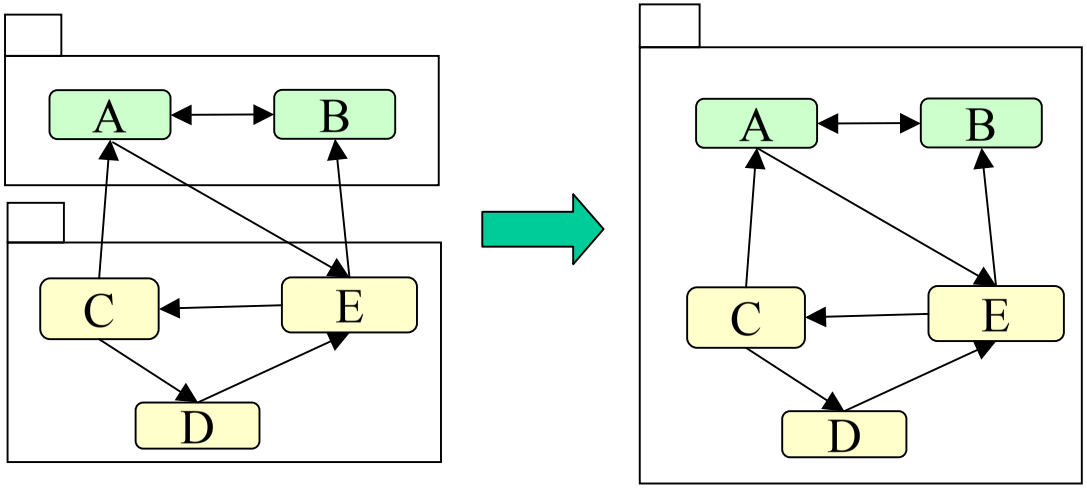


# Eliminação de Ciclos

- Dependências entre classes:
  - Dividir a classe, criando uma interface

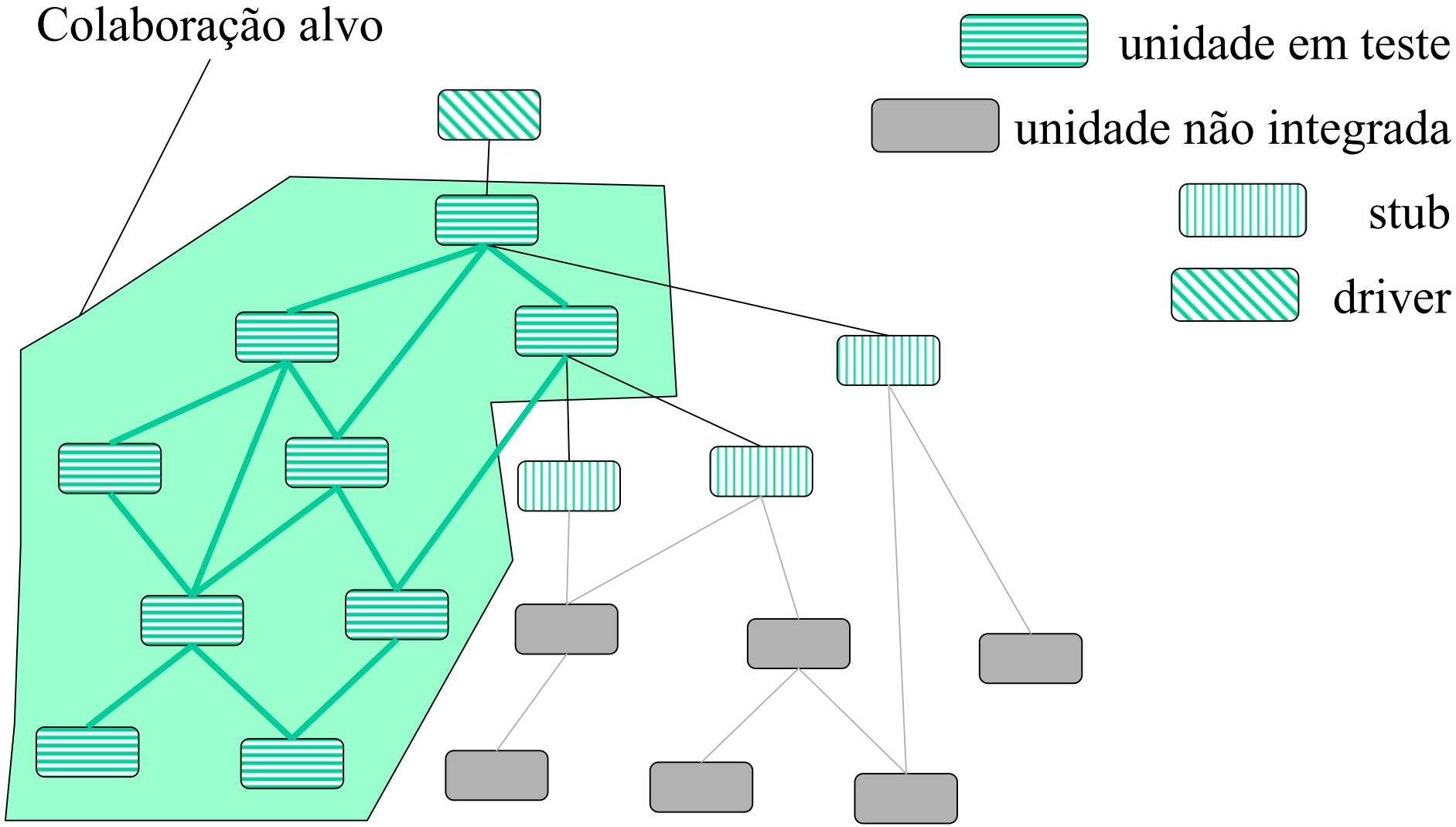



- Dependências entre pacotes:
  - Juntar pacotes
  - Dividir as classes fortemente acopladas em outro pacote

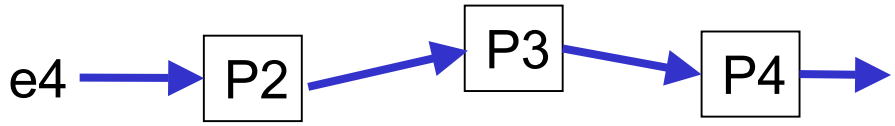


# Integração por colaboração

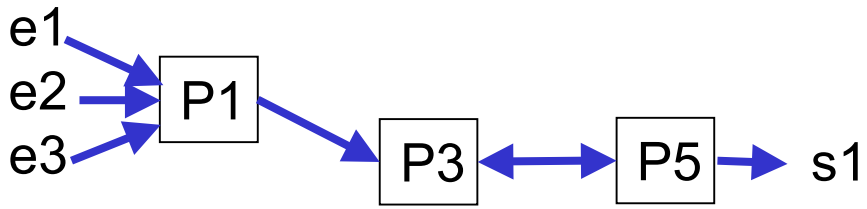
- Objetivo: cobertura de interações entre unidades
- Integrar unidades necessárias para realizar uma determinada colaboração
- As colaborações devem ser explicitamente especificadas
  - Diagramas de colaboração ou de atividades podem servir de modelo de base
- A ordem de integração das colaborações também pode ser obtida com o uso de um grafo de dependências
  - Critério: cobertura de arcos do grafo


Colaboração alvo
unidade em teste
unidade não integrada
stub
driver

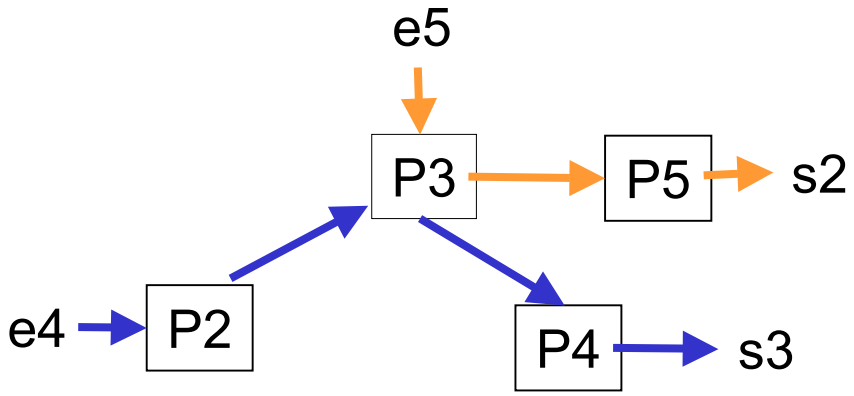
# Mais exemplo de integração por colaborações




teste de uma colaboração




teste de uma colaboração com várias entradas

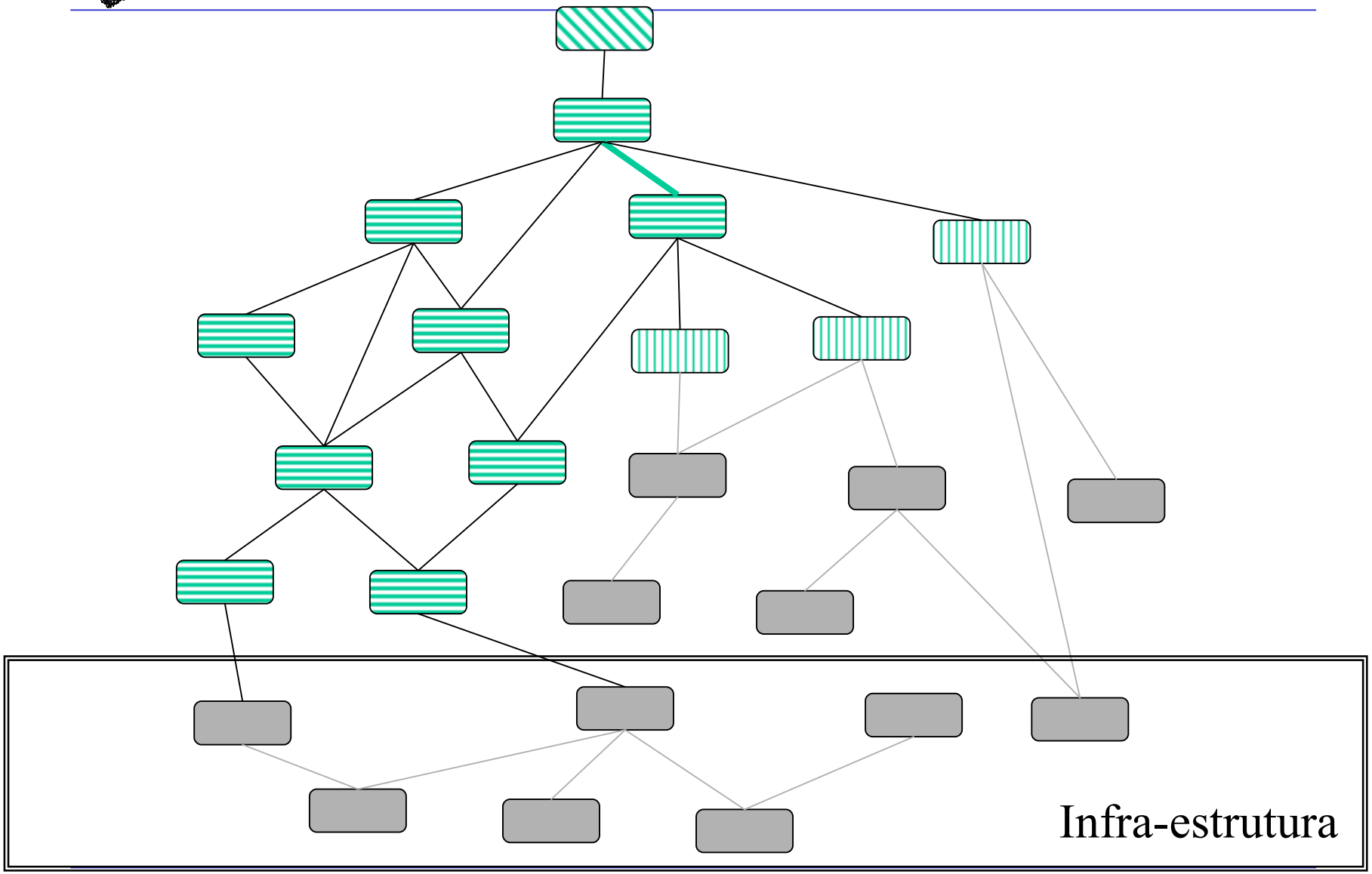



teste de várias colaborações

Sommerville96, c.22.3]

# Integração mista

- Combina várias das técnicas anteriores
  - Usar as técnicas mais adequadas de acordo com a parte do sistema que se quer integrar
- Útil quando sistema a ser integrado depende de uma infra-estrutura (*backbone*) que também está em desenvolvimento
  - Não vale a pena criar stubs para substituir a infra-estrutura de execução
  - Começar os testes integrando as unidades que compõem a infra-estrutura
  - Depois da infra-estrutura ter sido devidamente testada, integrar as demais unidades do sistema a ela, da maneira mais conveniente


Infra-estrutura

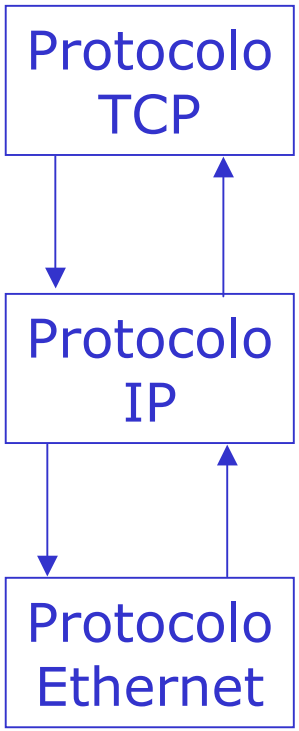
# Baseados na arquitetura

- Integração por camadas
  - Exercitar incrementalmente as interfaces e componentes em uma arquitetura em camadas
- Integração cliente/servidor
  - Exercitar redes de componentes fracamente acoplados que usam um servidor comum
- Integração de serviços distribuídos
  - Exercitar redes de componentes fracamente acoplados par a par
- Integração freqüente
  - Executar testes de integração periodicamente (por hora, por dia, por semana)

[Binder00, 13.1.3]

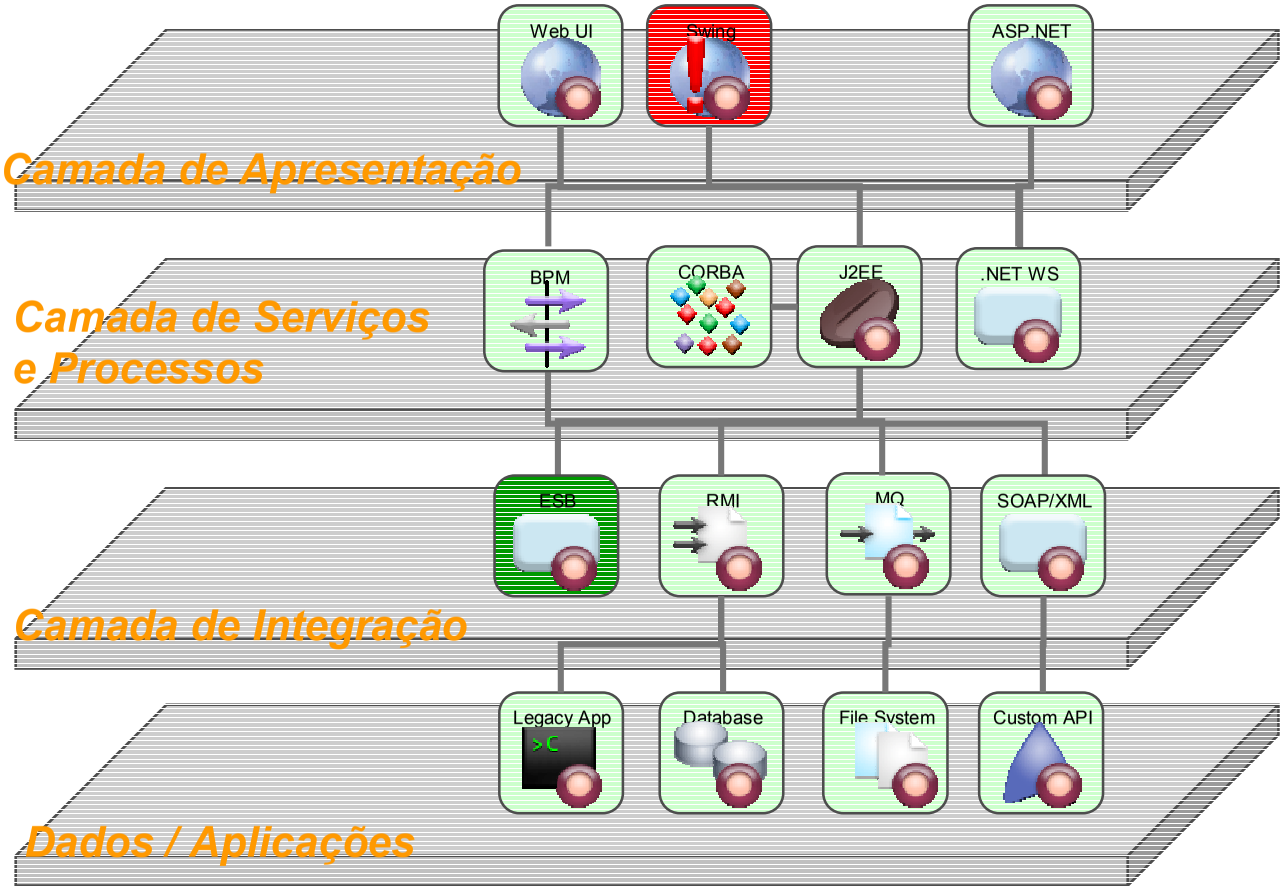
# Integração por camadas

- Útil quando o sistema é modelado como uma hierarquia que permite interfaces somente entre camadas adjacentes
- Também combina diversas estratégias:
  - Quaisquer das estratégias já vistas para integrar unidades internas a cada camada
  - Cada camada é testada isoladamente
  - Usar estratégia descendente ou ascendente para integrar as camadas

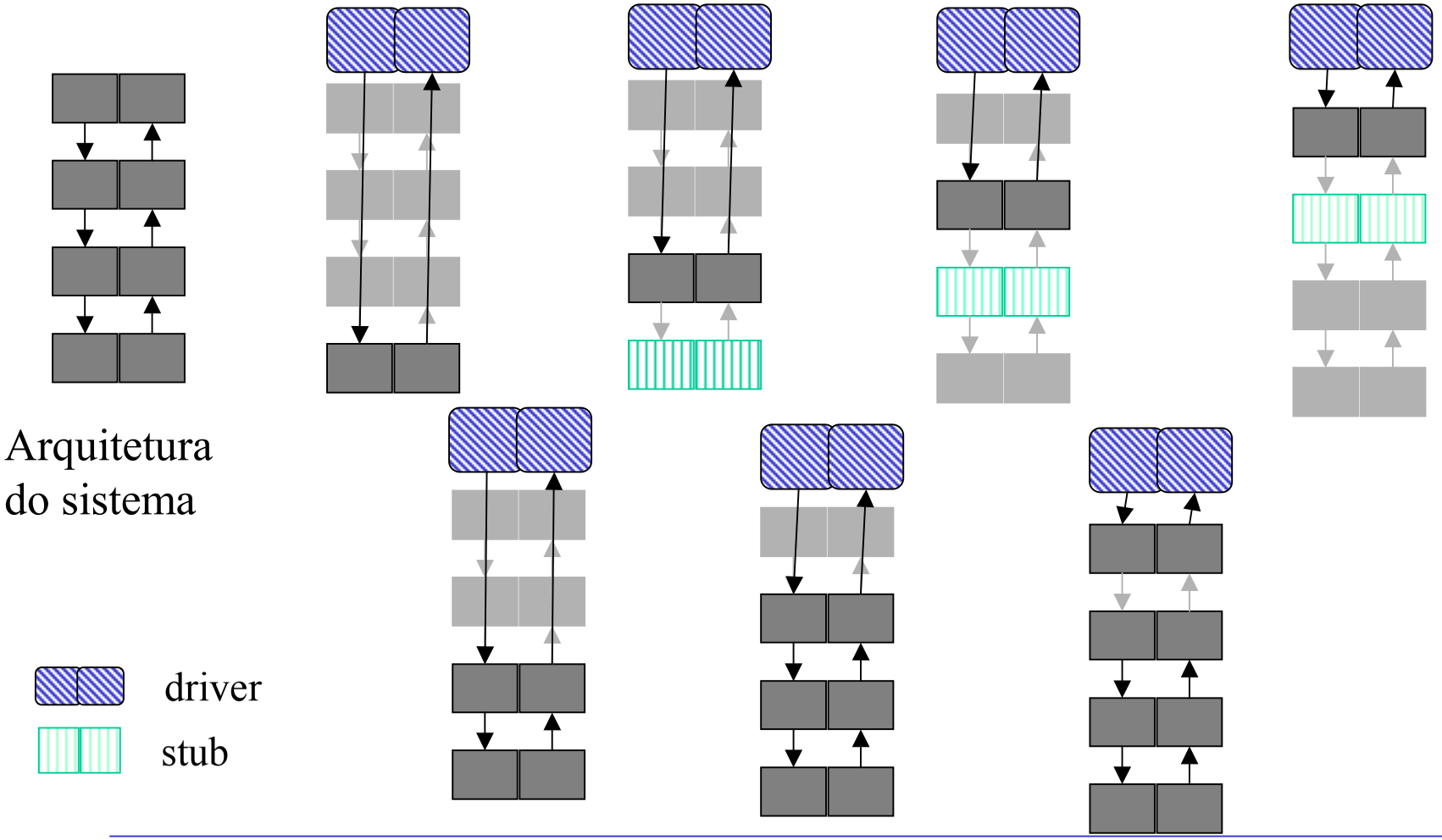


# Mais exemplo de arquiteturas em camadas

Arquitetura orientada a serviços



# Uso da estratégia de integração ascendente

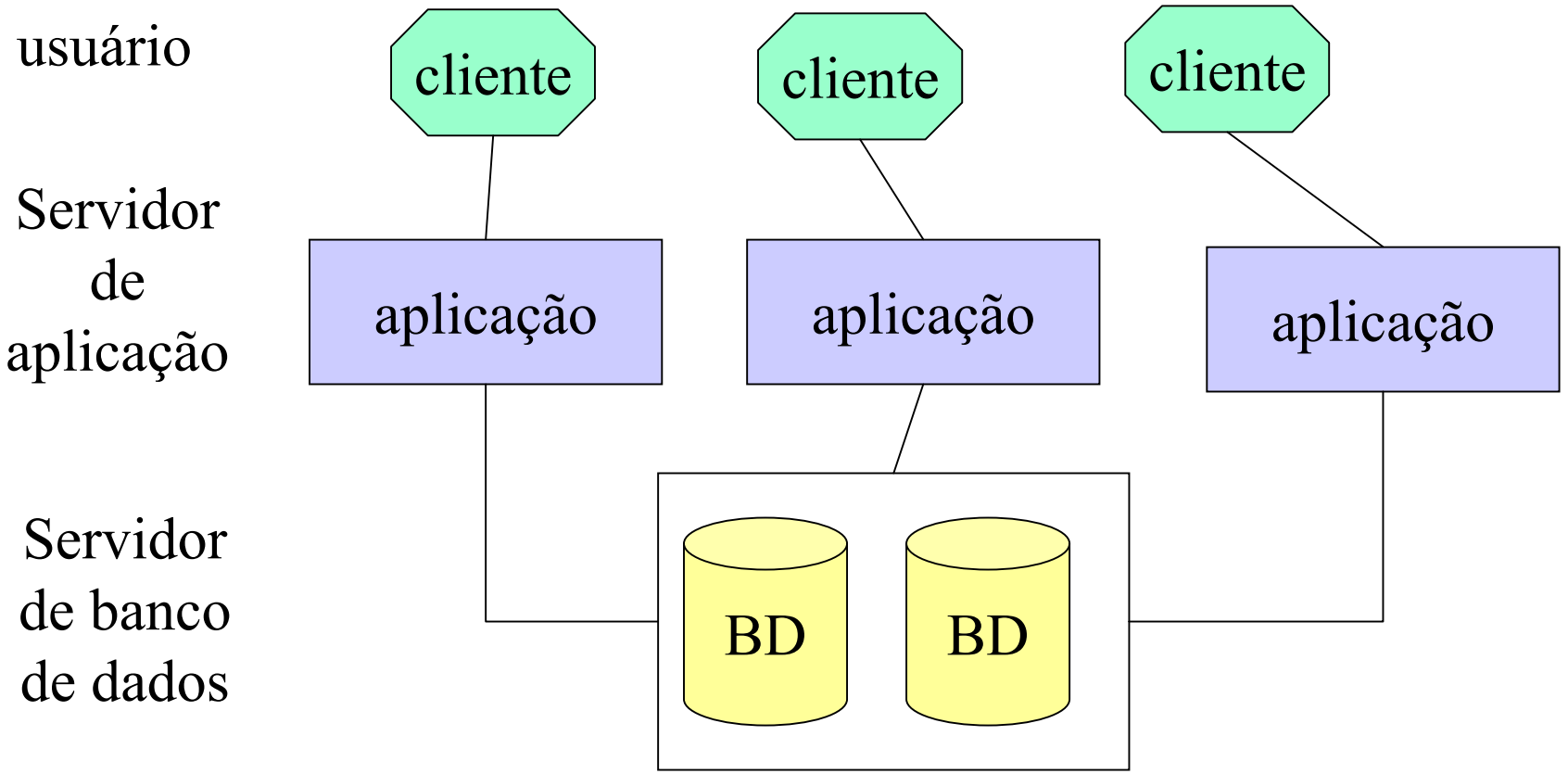


# Integração cliente/servidor

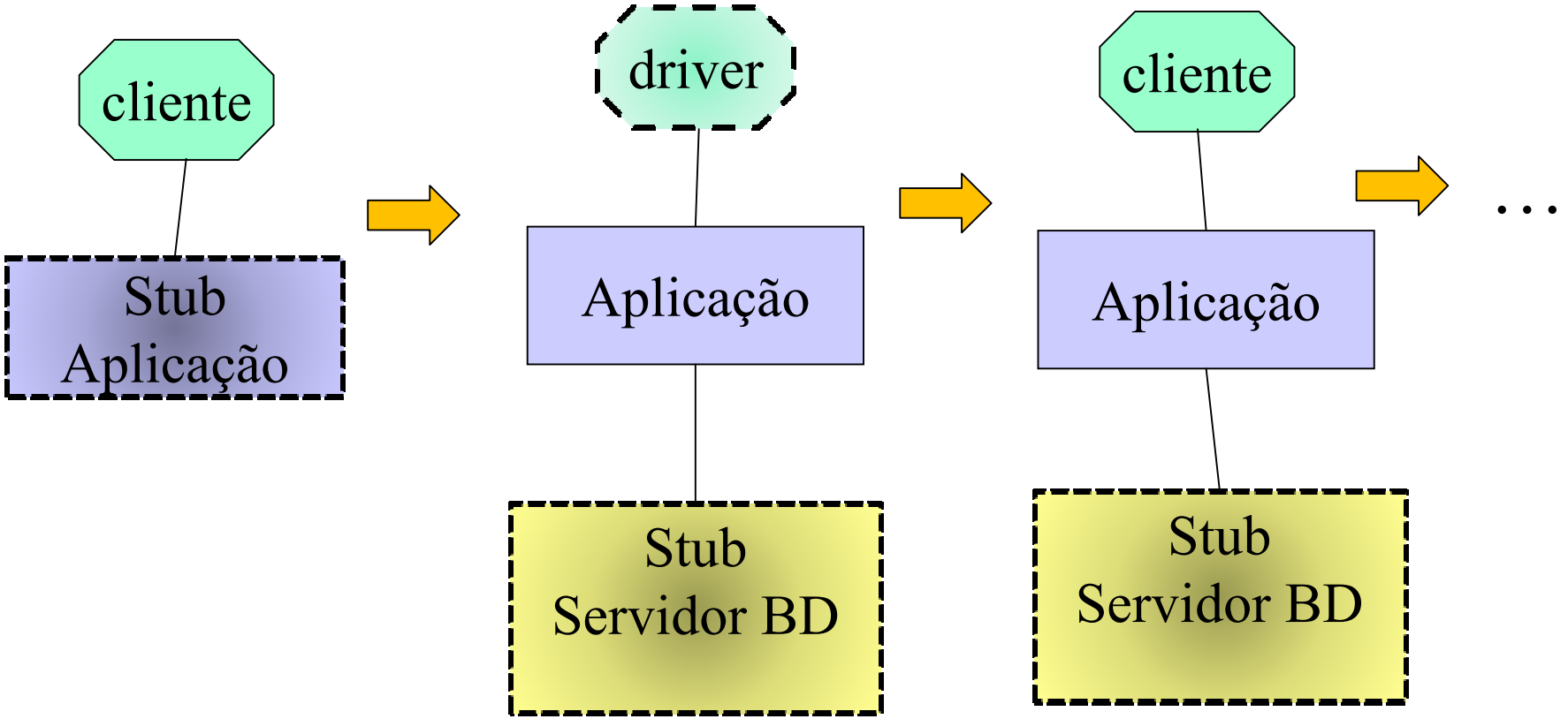
- Objetivo: exercitar interfaces entre unidades clientes fracamente acopladas a um único servidor
  - Na verdade, é um tipo especial de arquitetura cliente/servidor
- Útil quando a arquitetura é distribuída, sem um ponto único de controle:
  - Servidores reagem a mensagens dos clientes
  - Clientes respondem a estímulos do ambiente
- Estratégia:
  1. Testar cada cliente com um stub do servidor
  2. Testar o servidor com stub do cliente
  3. Integrar clientes ao servidor

[Binder2000, c13]

# Exemplo de arquitetura (*three-tier*)



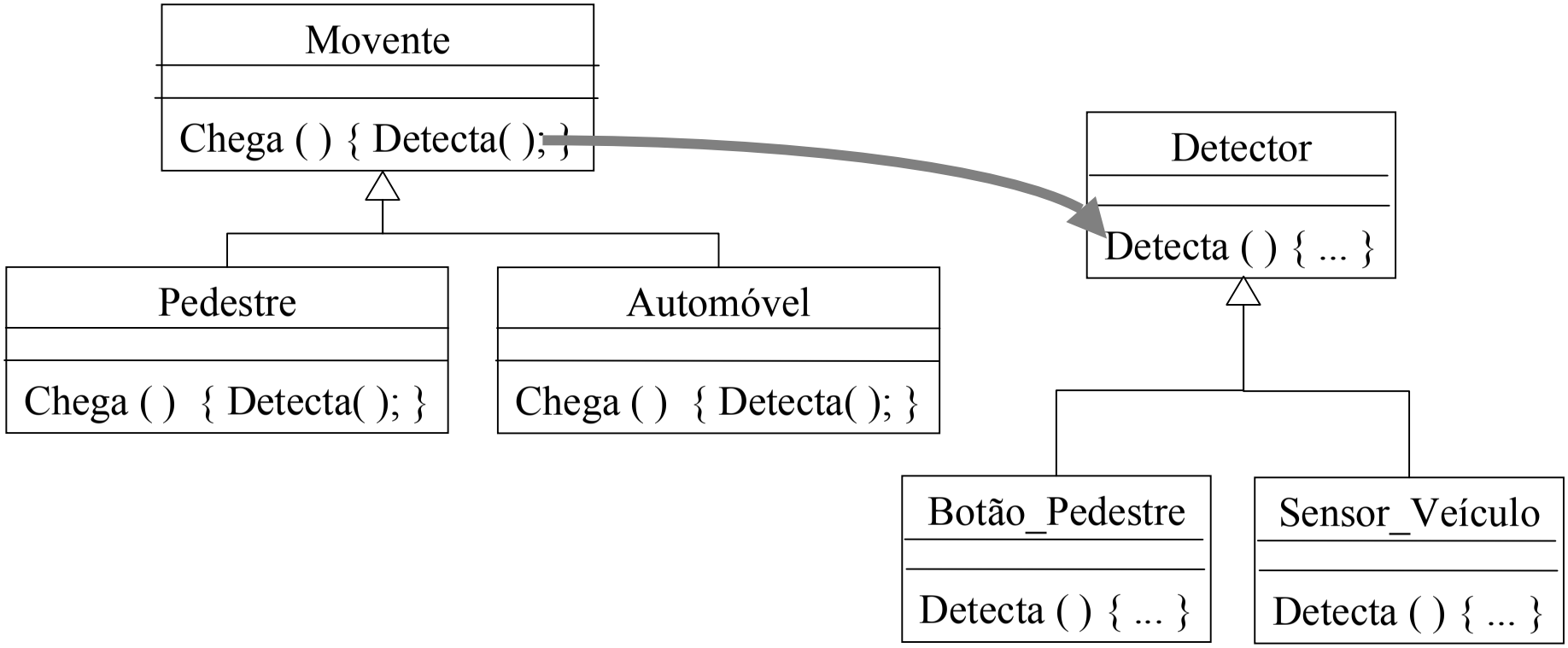
# Exemplo de estratégia de integração



# Testes de integração OO

- Em OO os Testes de Integração devem levar em conta outros relacionamentos além da dependência:
  - Generalização – especialização
    - Polimorfismo e ligação dinâmica
  - Associação
  - Agregação/Composição :
    - Interação inter-classes considera as possíveis combinações de mensagens, métodos e objetos

# Exemplo: polimorfismo



# Teste de interações polimórficas

- Identificar o conjunto de possíveis classes polimórficas
- Se o conjunto de polimórficas for pequeno, teste todas
- Senão proceda por amostragem (escolha aleatória)

[McGregor e Copeland 98]

# Teste das interações interclasses

- Testes exaustivos das interações entre emissores e receptores de mensagens ⇒ todas as formas de cada mensagem serão testadas (i.e., definições nas superclasses e em todas as subclasses)
- Os testes devem exercitar :
  - todos os emissores e seus estados possíveis
  - todos os receptores e seus estados possíveis
  - todos os parâmetros e seus estados possíveis

☞ o número de testes pode ser muito grande

# Técnica OATS

- OATS (Orthogonal Array Testing), proposta por McGregor et al, visa ajudar na seleção das combinações apropriadas
- OATS considera os itens independentes como sendo **fatores**, aos quais podem ser atribuídos um número finito de valores, designados de **níveis**
  - ex.:

    fator = todas as classes da hierarquia de classes

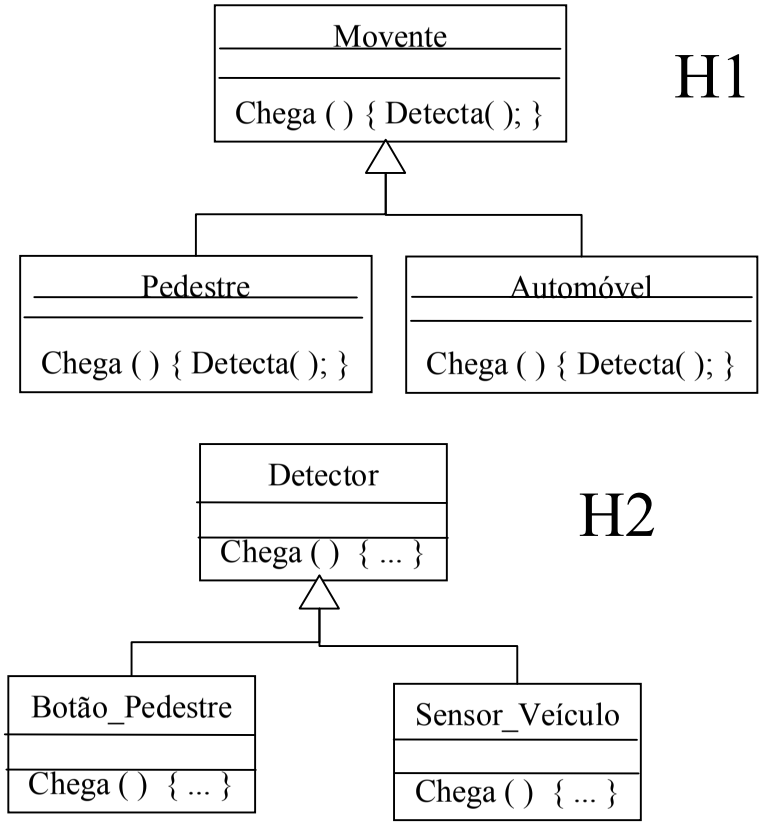
    nível = uma classe da hierarquia
- construir tabela:
  - coluna = fator
  - linha = caso de teste
- cada linha deve exercitar combinações dos níveis aos pares

# Exemplo de aplicação da OATS

- Supor que existem 3 fatores(A, B e C), cada qual com 3 níveis:
  - 1: raiz
  - 2 e 3: subclasses
- Testar todas as combinações possíveis requer 27 casos de teste
- Testar pares de combinações segundo OATS requer somente 9 casos de teste

| | **A** | **B** | **C** |
|---|---|---|---|
| CT1 | 1 | 1 | 3 |
| CT2 | 1 | 2 | 2 |
| CT3 | 1 | 3 | 1 |
| CT4 | 2 | 1 | 2 |
| CT5 | 2 | 2 | 1 |
| CT6 | 2 | 3 | 3 |
| CT7 | 3 | 1 | 1 |
| CT8 | 3 | 2 | 2 |
| CT9 | 3 | 3 | 2 |

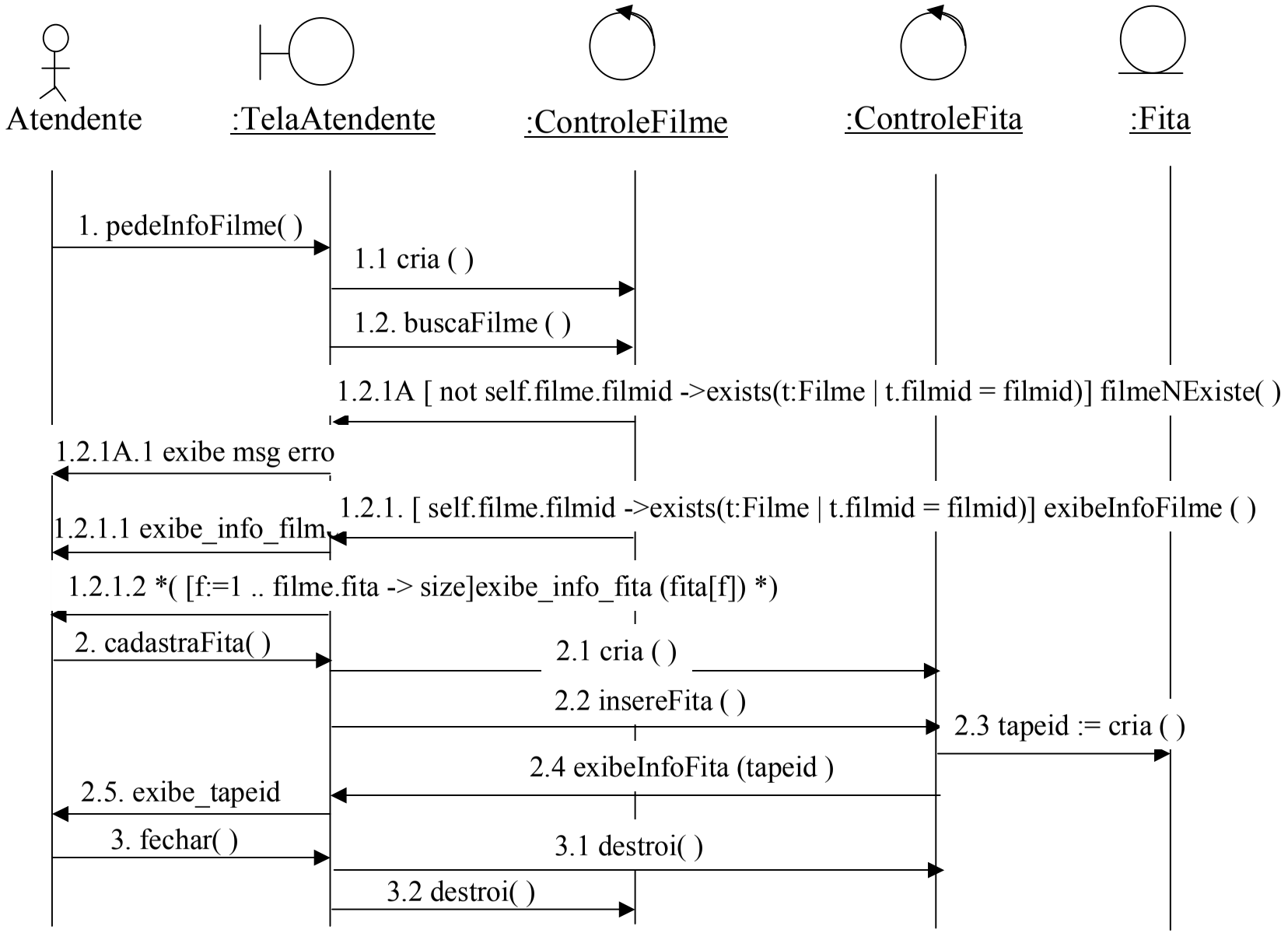
# Exemplo de aplicação da OATS




| | **H1** | **H2** |
|---|---|---|
| CT1 | Movente | Detector |
| CT2 | Movente | Botão_Pedestre |
| CT3 | Movente | Sensor_veículo |
| CT4 | Pedestre | Detector |
| | ... | |
| CT9 | Automóvel | Sensor_Veículo |

# Testes baseados em cenários

- Baseiam-se nos diferentes cenários de uso do sistema (obtidos através dos casos de uso)
- Pode-se testar cenários mais comuns primeiro. Cenários alternativos ou de exceção podem ser considerados depois
- Diagramas de interação (seqüência ou colaboração) podem servir de base para os testes, bem como o Diagrama de Atividades
- Analogia: integração por colaboração


Atendente
:TelaAtendente
:ControleFilme
:ControleFita
:Fita
1. pedeInfoFilme( )
1.1 cria ( )
1.2. buscaFilme ( )
1.2.1A [ not self.filme.filmid ->exists(t:Filme | t.filmid = filmid)] filmeNExiste( )
1.2.1A.1 exibe msg erro
1.2.1. [ self.filme.filmid ->exists(t:Filme | t.filmid = filmid)] exibeInfoFilme ( )
1.2.1.1 exibe_info_film
1.2.1.2 *( [f:=1 .. filme.fita -> size]exibe_info_fita (fita[f]) *)
2. cadastraFita( )
2.1 cria ( )
2.2 insereFita ( )
2.3 tapeid := cria ( )
2.4 exibeInfoFita (tapeid )
2.5. exibe_tapeid
3. fechar( )
3.1 destroi( )
3.2 destroi( )

# Considerações

- Cada método de uma classe deve ser executado pelo menos uma vez
  - criar matriz assinalando, para cada cenário, os métodos e respectivas classes
- O diagrama de seqüência permite também identificar as entradas e saídas

# Sumário

- Testes de integração são úteis mesmo que as unidades tenham sido testadas
- Abordagens de integração incremental são “mais baratas” do que abordagem não-incremental
- Testes de desempenho, estresse e tolerância a falhas devem ser realizados cedo na fase de testes pois podem implicar em grandes alterações (talvez de projeto)
- Modificações no sw são inevitáveis e introduzem falhas $\Rightarrow$ testes aplicados precisam ser armazenados para uso em testes de regressão