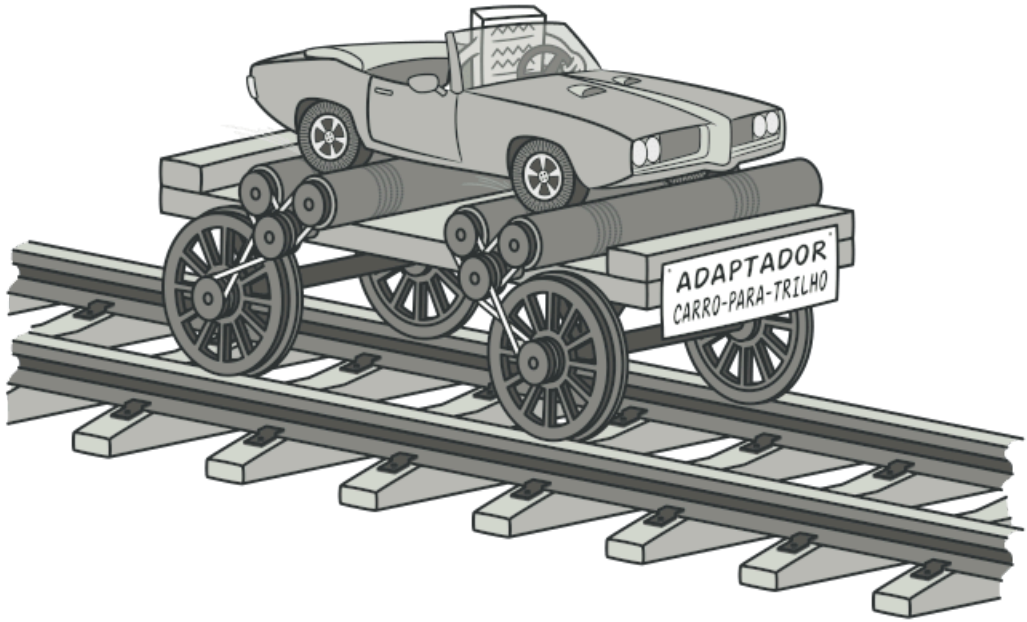
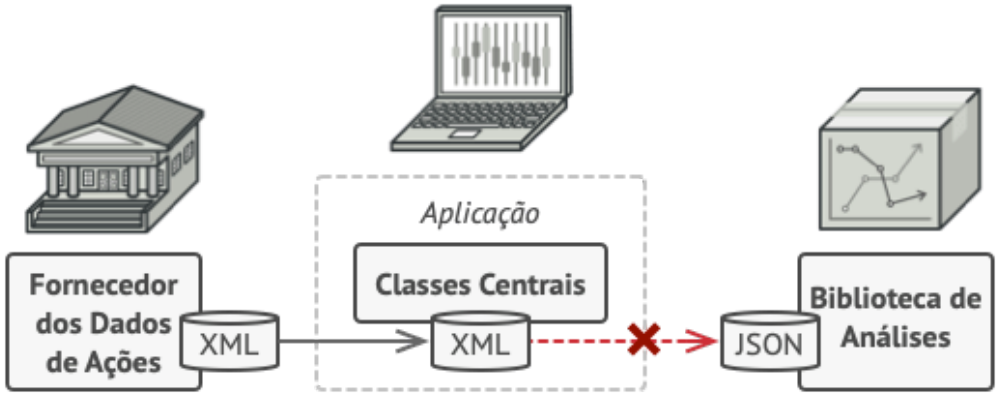
TÓPICO 09 - ADAPTER

Design Patterns - Professor Ramon Venson - SATC 2026.1

Motivação

- Sistemas evoluem e precisam integrar terceiros
- Nem toda biblioteca fala a mesma linguagem
- Reescrever tudo quase nunca é viável
- Precisamos reaproveitar código útil
- O cliente deve continuar estável





Problema

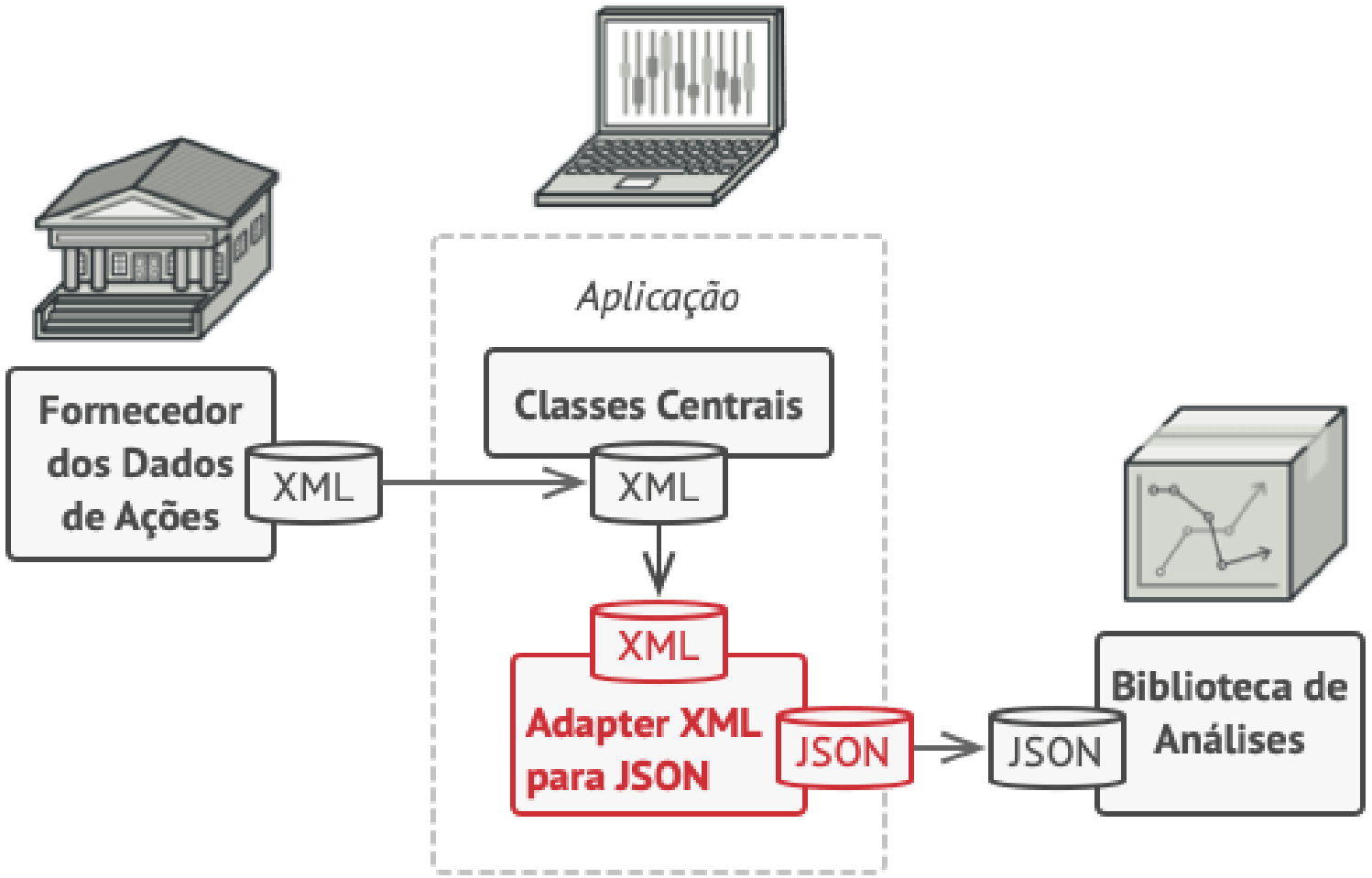
- Sistema atual recebe XML
- Biblioteca nova entende apenas JSON
- Interfaces não combinam
- Mudanças diretas espalham impacto
- Código cliente fica acoplado ao fornecedor

Conceito principal

- **Adapter** é um padrão estrutural
- Ele converte uma interface em outra
- O cliente usa a interface esperada
- O serviço real continua intocado
- A tradução fica isolada no adaptador

Como funciona

- O cliente chama uma interface conhecida
- O adaptador implementa essa interface
- Internamente, ele traduz a chamada
- Depois delega ao serviço incompatível
- Resultado volta no formato esperado


Aplicação
Fornecedor dos Dados de Ações
XML
Classes Centrais
XML
XML
Adapter XML para JSON
JSON
JSON
Biblioteca de Análises

Papéis no padrão

- **Client** usa uma interface esperada
- **Target** define o contrato desejado
- **Adaptee** é o serviço incompatível
- **Adapter** traduz e delega
- Foco: compatibilidade sem quebrar o cliente

Exemplo conceitual

- Portal financeiro espera JSON
- Sistema legado fornece XML
- Não podemos alterar o legado
- Criamos um tradutor entre os dois
- O cliente continua consumindo JSON

Código: contrato esperado

```
interface ProvedorRelatorio {
    String obterRelatorioEmJson();
}
```

- O cliente depende do contrato `Target`
- Nada de acoplamento ao legado

Código: serviço incompatível

```
class SistemaLegadoXml {
    public String obterRelatorioEmXml() {
        return "<relatorio><total>150</total></relatorio>";
    }
}
```

- Classe útil, mas incompatível
- Queremos reutilizar sem modificar

Código: adaptador

```
class XmlParaJsonAdapter implements ProvedorRelatorio {
    private SistemaLegadoXml sistemaLegado;

    public String obterRelatorioEmJson() {
        String xml = sistemaLegado.obterRelatorioEmXml();
        return "{ \"total\": 150 }";
    }
}
```

- O adaptador traduz formatos
- O cliente recebe o que espera

Analogia

- Plugue brasileiro e tomada europeia
- Equipamentos continuam os mesmos
- O adaptador faz o encaixe funcionar
- Ele traduz a interface física

Quando usar

- Integrar código legado
- Reaproveitar bibliotecas externas
- Compatibilizar APIs diferentes
- Isolar regras de conversão
- Evitar mudanças no cliente principal

Quando não usar

- Quando a interface pode ser refatorada facilmente
- Quando não há incompatibilidade real
- Quando o adaptador vira lógica de negócio
- Quando a camada extra só complica
- Quando composição simples já resolve

Vantagens

- Reaproveita código existente
- Reduz acoplamento com terceiros
- Centraliza conversões
- Facilita evolução do sistema
- Protege o cliente de mudanças externas

Desvantagens

- Adiciona novas classes
- Pode aumentar a complexidade geral
- Não corrige design ruim
- Adaptadores grandes viram problema
- Uso excessivo cria camadas artificiais

Relações com outros padrões

- **Facade** simplifica subsistemas inteiros
- **Decorator** mantém a interface base
- **Proxy** controla acesso, sem traduzir
- **Bridge** separa abstração e implementação
- **Adapter** traduz interfaces incompatíveis

Exemplo prático

```
Notificador n = new SmsAdapter(new ServicoLegadoSms());
NotificadorTurma app = new NotificadorTurma(n);
app.avisarInicioDasAulas();
```

- O cliente usa `Notificador`
- O legado continua intacto
- A tradução fica no `SmsAdapter`

Resumo final

- Adapter traduz uma interface em outra
- Resolve integração sem reescrever tudo
- Muito útil com legado e terceiros
- Deve focar em compatibilidade
- Menos acoplamento, mais reaproveitamento