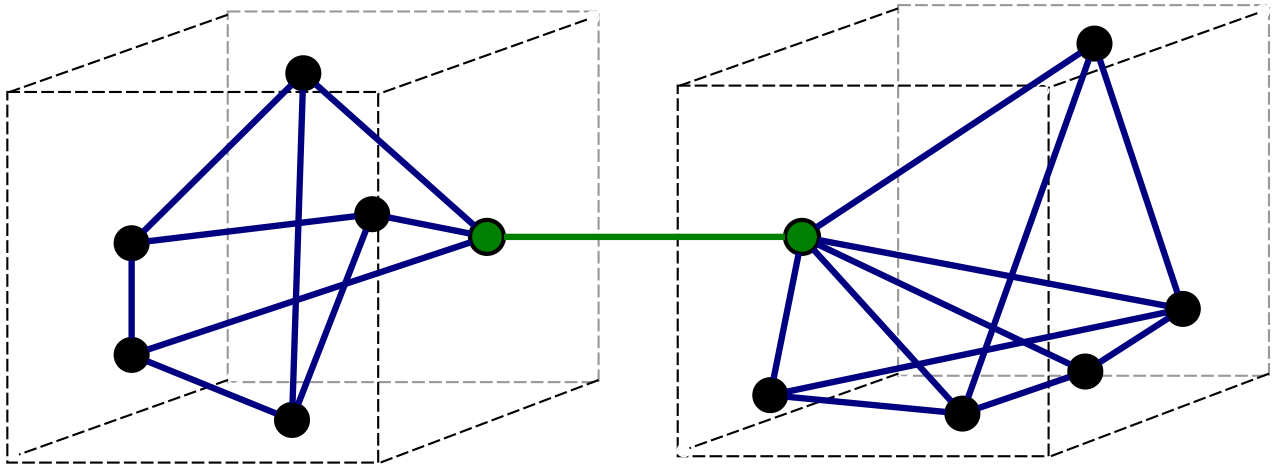
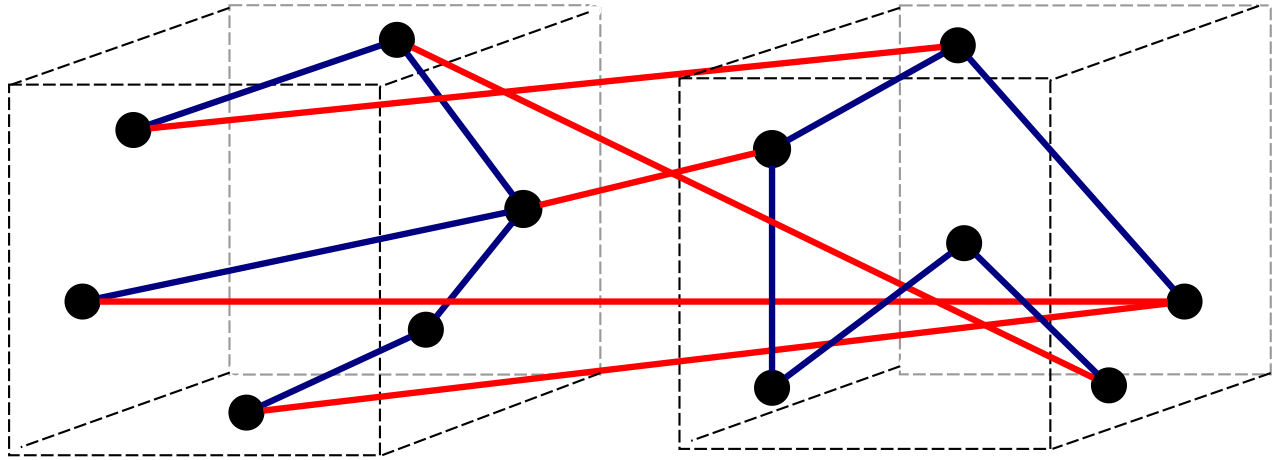
TÓPICO 11 - GERENCIAMENTO DE DEPENDÊNCIAS

Clean Code - Professor Ramon Venson - SATC 2026.1



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Acoplamento

Em engenharia de software, o acoplamento é o nível de interdependência entre dois módulos de software.

É geralmente um contraste ao conceito de **coesão**.

> Coesão é uma medida do quão similares são os elementos de um sistema.

Desvantagens do Acoplamento

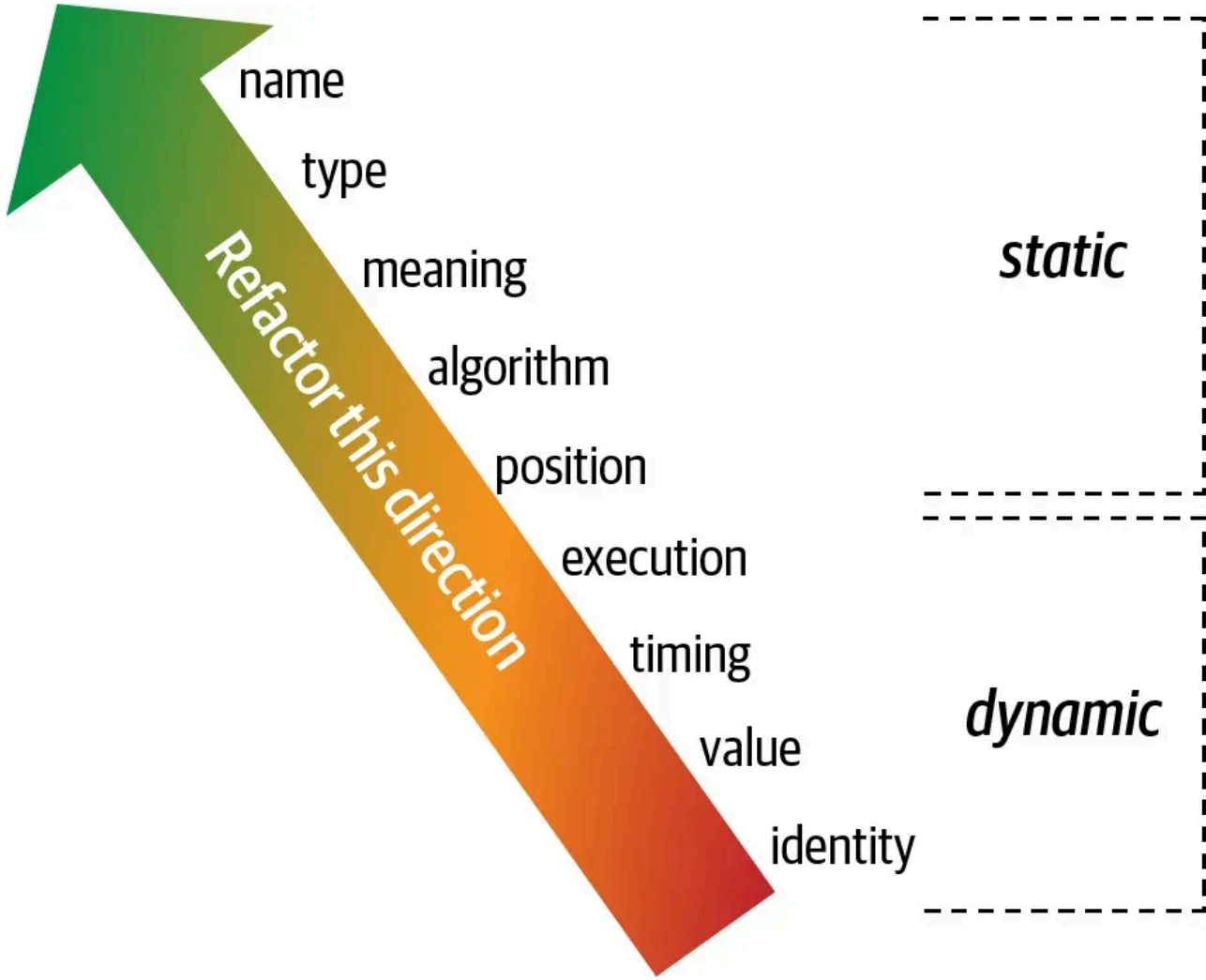
O acoplamento é inerente ao desenvolvimento de software e não pode ser evitado. É ele que mantém o software funcionando e colaborando entre si. Um software altamente acoplado, no entanto, possui características negativas, como:

1. Mudar um módulo pode afetar outros módulos, tornando-o menos estável;
2. Montagem dos módulos requer mais tempo e esforço;
3. É mais difícil testar e depurar o software;
4. É mais difícil modificar ou reutilizar o software.

Conascência

A conascência é uma métrica de software introduzida por *Meilir Page-Jones* que quantifica a dependência entre componentes de software.

Ela mede a `força` (dificuldade de mudança), `grau` (quantidade de relações) e `localidade` (proximidade na base de código)



Conascência de Nome

`Conascência Estática`

Todos os componentes precisam concordar com as mesmas nomenclaturas.

```python
def get_user(user_id: int):
    return {"user_id": user_id, "name": "Alice"}

# ❌ user_id é diferente de customerId
def create_order(customerId: int , product: str):
    return {"customerId": customerId, "product": product}

# ✅ mesma nomenclatura
def create_order(user_id: int , product: str):
    return {"user_id": user_id, "product": product}
```

Conascência de Tipo

`Conascência Estática`

Todos os componentes precisam concordar com os mesmos tipos.

```python
def get_user(user_id: int):
    return {"user_id": user_id, "name": "Alice"}

# ❌ user_id é int, mas create_order espera str
def create_order(user_id: str , product: str):
    return {"user_id": user_id, "product": product}

# ✅ mesmo tipo de dados original em user_id
def create_order(user_id: int , product: str):
    return {"user_id": user_id, "product": product}
```

Conascência de Significado

`Conascência Estática`

Todos os componentes precisam concordar com o significado de um valor.

```python
def get_user(user_id: int):
    return {"user_id": user_id, "name": "Alice"}

def create_order(user: dict, product: str):
    # Aqui o dev entendeu errado: "active = True" significa "em uso"
    if user["active"]:
        raise Exception("Usuário não pode criar pedidos (em uso).")
    return {"user_id": user["user_id"], "product": product}
```

Conascência de Posição

`Conascência Estática`

Todos os componentes precisam concordar com a ordem dos valores.

```python
def export_user():
    # Ordem: [id, nome, ativo]
    return [42, "Alice", True]

def import_user(data: list):
    # Ordem esperada: [nome, id, ativo] ❌
    return {
        "name": data[0],  # errado, vai pegar 42
        "user_id": data[1],  # errado, vai pegar "Alice"
        "active": data[2]
    }
```

Conascência de Algoritmo

`Conascência Estática`

Todos os componentes precisam concordar com o mesmo algoritmo.

```python
import hashlib

def hash_password(password: str) -> str:
    # Usa SHA-256
    return hashlib.sha256(password.encode()).hexdigest()

def validate_password(password: str, stored_hash: str) -> bool:
    # Espera que o hash tenha sido feito com MD5 ❌
    return hashlib.md5(password.encode()).hexdigest() == stored_hash
```

Conascência de Execução

`Conascência Dinâmica`

A ordem de execução é importante.

```
email = Email()
email.enviar()  # ❌ chamado antes de definir destino/mensagem
email.destino = "alice@example.com"
email.mensagem = "Olá!"
```

Conascência de Tempo

`Conascência Dinâmica`

O tempo de execução precisa ser coordenado entre os componentes.

```
# Duas threads acessando o mesmo recurso sem sincronização ❌
t1 = threading.Thread(target=sacar, args=(80,))
t2 = threading.Thread(target=sacar, args=(80,))

t1.start()
t2.start()
t1.join()
t2.join()
```

Conascência de Valores

`Conascência Dinâmica`

Valores interdependentes precisam ser modificados em conjunto.

```
conta_a = Conta(100)
conta_b = Conta(50)

# Transferência ❌ só debita, não credita a outra conta
conta_a.debitar(30)

print("Saldo A:", conta_a.saldo)  # 70
print("Saldo B:", conta_b.saldo)  # 50 (errado!)
```

Conascência de Identidade

`Conascência Dinâmica`

Todos os componentes precisam concordar com a identidade de um objeto.

```
usuario_a = Usuario(1, "Alice")
# Módulo B cria outro objeto "igual", mas não é a mesma identidade ❌
usuario_b = Usuario(1, "Alice")

print(usuario_a is usuario_b)  # False (objetos diferentes na memória)
```

Por que Conascência importa?

A conascência é uma métrica que pode ser utilizada para identificar e reduzir o acoplamento entre componentes de software.

> Radu Gheorman - Visualising Connascence to Drive Refactoring

Dependências Internas e Externas

Podemos dividir as dependências de um projeto em duas categorias:

- **Dependências Internas**: interdependências entre classes e módulos de um mesmo projeto;
- **Dependências Externas**: interdependências entre diferentes projetos;

Dependências Internas

As dependências internas são aquelas que ocorrem entre classes e módulos de um mesmo projeto.

Essas dependências podem ser de diferentes tipos, como dependências de implementação, dependências de interface, dependências de tempo de execução, etc.

Princípios para Gerenciamento de Dependências

Alguns princípios importantes para o gerenciamento de dependências incluem:

- **Separação de Responsabilidades**: Cada módulo deve ter uma única responsabilidade.
- **Aberto/Fechado**: Os módulos devem ser abertos para extensão, mas fechados para modificação.
- **Inversão de Dependência**: Os módulos devem depender de abstrações, não de implementações concretas.

Modularização

Modularizar um sistema é dividir em partes menores, com responsabilidades bem definidas.

Cada módulo deve ser coeso (fazer bem uma única coisa) e ter baixa dependência de outros módulos (baixo acoplamento).

Maneiras de Modularizar

- **Por Funcionalidade**: Dividir o sistema por capacidades do negócio (ex.: `usuarios`, `pedidos`, `pagamentos`...).
- **Por Domínio**: Dividir o sistema por subdomínios ou contextos do problema (ex.: `catalogo`, `faturamento`, `entrega`...).
- **Por Camadas**: Dividir o sistema em camadas técnicas (ex.: `presentation`, `service`, `repository`...).

Erros Comuns na Modularização

- **God Modules**: Módulos que fazem muito, como um `controller` que faz apresentação, entrada e saída, regras de negócio...
- **Código Duplicado**: Módulos que fazem a mesma coisa, mas em diferentes lugares.
- **Dependências Circulares**: Módulos que dependem uns dos outros, criando um ciclo.

Boas Práticas na Modularização

- **Responsabilidades**: Cada módulo deve ter uma responsabilidade clara;
- **Abstrações**: Usar interfaces e abstrações para reduzir dependências;
- **Composição**: Usar composição em vez de herança para criar módulos mais flexíveis;
- **Direção**: Estabeleça ordem de fluxo (ex.: `service` pode acessar `repository`, mas não o contrário).
- **Testabilidade**: Módulos devem ser testáveis de forma independente.
- **Camadas**: Separe o código de domínio, aplicação e infraestrutura.

```
def cadastrar_usuario(nome, email):
    with open("usuarios.txt", "a") as f:
        f.write(f"{nome},{email}\n")
    print("Usuário cadastrado com sucesso.")
```

```python
# repository.py
def salvar_usuario(nome, email):
    with open("usuarios.txt", "a") as f:
        f.write(f"{nome},{email}\n")

# service.py
from repository import salvar_usuario

def cadastrar_usuario(nome, email):
    salvar_usuario(nome, email)

# interface (main.py)
from service import cadastrar_usuario
nome = input("Nome: ")
email = input("Email: ")
cadastrar_usuario(nome, email)
print("Usuário cadastrado com sucesso.")
```

Injeção de Dependências

A injeção de dependências é uma técnica que permite que um objeto receba suas dependências de fora, em vez de criar as dependências dentro do objeto.

Isso permite que o objeto seja mais flexível e testável, pois as dependências podem ser facilmente substituídas ou injetadas em tempo de execução, reduzindo o acoplamento entre os objetos.

Exemplo de Injeção de Dependências

```
class ServicoEmail:
    def enviar(self, mensagem):
        print(f"Enviando: {mensagem}")

class CadastroUsuario:
    def cadastrar(self, nome):
        email = ServicoEmail()
        email.enviar(f"Bem-vindo, {nome}!")
```

Nesse exemplo, o objeto `CadastroUsuario` cria uma instância de `ServicoEmail` dentro de si mesmo, o que cria um acoplamento forte entre os dois objetos.

```python
class CadastroUsuario:
    def __init__(self, servico_email):
        self.servico_email = servico_email

    def cadastrar(self, nome):
        self.servico_email.enviar(f"Bem-vindo, {nome}!")

# Uso
servico = ServicoEmail()
cadastro = CadastroUsuario(servico)
```

Agora, o objeto `CadastroUsuario` não precisa mais conhecer os detalhes do `ServicoEmail` e pode receber o objeto `ServicoEmail` como parâmetro.

Testabilidade

Uma das principais vantagens da injeção de dependências é que ela facilita a criação de testes unitários.

No exemplo anterior, se quisermos testar o método `cadastrar` sem enviar um e-mail, podemos criar um objeto `ServicoEmail` fictício (*mock*) que não envia e-mails.

Estratégias de Redução de Acoplamento

- **Arquiteturas Orientadas à Eventos**: Componentes não se comunicam diretamente;
- **Observer Pattern**: Um componente notifica outros componentes quando algo acontece;
- **Strategy Pattern**: Encapsulamento de diferentes comportamentos em classes distintas, permitindo a escolha dinâmica do comportamento em tempo de execução;
- **Command Pattern**: Encapsula uma solicitação como um objeto, permitindo a separação entre o pedido de execução e o objeto que a executa;

> Mais estratégias e padrões podem ser encontrados em RefactoringGuru

Dicas e Boas Práticas

- **Funções puras**: Evite efeitos de borda e mantenha a função livre de efeitos colaterais;
- **Composição**: Use composição em vez de herança sempre que possível;
- **Abstrações**: Use interfaces e abstrações para reduzir dependências;
- **Evite dependências circulares**: Evite ciclos de dependência entre módulos;

Dependências Externas

As dependências externas são bibliotecas, frameworks ou outros componentes que o código depende para funcionar corretamente.

Essas dependências são críticas, pois afetam a estabilidade e a manutenibilidade do sistema, mesmo que não sejam parte do código-fonte.

Riscos

Entre os maiores riscos de depender de um código que você não controla:

- **Disponibilidade**: A biblioteca pode ficar indisponível durante o desenvolvimento do projeto;
- **Atualizações**: A biblioteca pode ser atualizada, o que pode quebrar o código;
- **Segurança**: bibliotecas podem conter vulnerabilidades de segurança;
- **Dependência Transitiva**: Ao instalar uma biblioteca, você também instala todas as suas dependências;
- **Licenças**: Dependendo da licença, você pode estar sujeito a restrições de uso;

Alternativas ao acoplamento de bibliotecas

- **Escrever código**: Implementar funcionalidades do zero;
- **Wrappers**: Encapsular o uso de uma biblioteca em uma interface;
- **Bridge/Adapter Pattern**: Para bibliotecas com APIs instáveis;
- **Fallback**: "Se a biblioteca falhar, use uma função alternativa";

Boas práticas para dependências externas

- Use bibliotecas bem estabelecidas e com manutenção ativa.
- Sempre fixe a versão da dependência quando a previsibilidade for crítica (ex.: `==1.2.3`); se isso não for viável, use um operador como `^4.5.0` para garantir compatibilidade a partir de uma versão mínima conhecida.
- Avaliar bibliotecas antes de usar (maturidade, manutenibilidade, comunidade)
- Faça auditoria de segurança regularmente.
- Quando possível, encapsule bibliotecas com uma abstração interna.

Boas práticas em equipe

- **Documentar**: documente dependências importantes;
- **Padronizar**: padronize o uso de bibliotecas por time/projeto;
- **Automatizar**: automatize a atualização de dependências para garantir segurança;
- **Testar**: testar aplicações que dependem de bibliotecas externas;

Gerenciadores de Dependências

Ferramentas que ajudam a gerenciar as dependências de um projeto. Exemplos de gerenciadores incluem:

- **npm** e **yarn** (javascript);
- **pip** (python)
- **composer** (php)
- **maven** (java)
- **cargo** (rust)

Alguns gerenciadores garantem também o isolamento de ambiente e controle de versão das bibliotecas.

Material de Apoio

- Clean Architecture
- RefactoringGuru
- Wikipedia - Coupling
- Wikipedia - Cohesion
- Connascence
- Medium - Conascência em Arquitetura de Software: O que é e por que importa