5. Acopladores: Smells Relacionados a Dependências Excessivas Entre Classes

Os *code smells* da categoria "Acopladores" (*Couplers*) indicam um acoplamento excessivo entre classes ou módulos, ou problemas que surgem quando o acoplamento é substituído por delegação excessiva. O acoplamento forte significa que os componentes dependem fortemente dos detalhes internos uns dos outros, tornando-os difíceis de alterar, substituir ou testar isoladamente. Isso pode levar a efeitos cascata, onde uma alteração num módulo quebra outros. O objetivo é alcançar um baixo acoplamento e alta coesão, frequentemente promovidos pelos princípios SOLID e por padrões de design.

5.1 Inveja de Funcionalidades (*Feature Envy*)

Definição: Um Método Mais Interessado em Outra Classe:
Um método exibe Feature Envy quando acede aos dados ou métodos de outra classe com mais frequência do que aos seus próprios. O método parece "invejar" as funcionalidades da outra classe e pode estar no local errado dentro da estrutura do código.
Este smell ocorre frequentemente após os dados serem movidos para uma classe de dados, mas as operações sobre esses dados são deixadas para trás.21 Pode também surgir de um mal-entendido sobre onde a responsabilidade deve residir. Os problemas incluem a violação do encapsulamento 24, o aumento do acoplamento entre classes 22, a redução da coesão da classe de origem 24 e a complicação dos testes.24 É um indicador de um design de classe pobre.

Técnica de Refatoração: Mover Método (e potencialmente Extrair Método primeiro):
A solução primária é Mover Método (Move Method): relocar o método "invejoso" para a classe da qual ele está "invejando" as funcionalidades. Isso coloca o comportamento mais próximo dos dados que ele opera. Se apenas parte de um método exibe Feature Envy, a técnica Extrair Método (Extract Method) deve ser usada primeiro para isolar a parte invejosa, e então esse método extraído pode ser movido. Ferramentas como JDeodorant podem detectar Feature Envy e sugerir refatorações de Mover Método.

Exemplo em Java: Detectando Feature Envy e Aplicando Mover Método:

- **Antes:**

```Java
class Customer {
    private String name;
```

```
    private Address address; // Address é outra classe

    public Customer(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public String getName() { return name; }
    public Address getAddress() { return address; }
}

class Address {
    private String street;
    private String city;
    private String zipCode;

    public Address(String street, String city, String zipCode) {
        this.street = street;
        this.city = city;
        this.zipCode = zipCode;
    }

    public String getStreet() { return street; }
    public String getCity() { return city; }
    public String getZipCode() { return zipCode; }

    // Este método pode pertencer aqui se for uma utilidade geral de endereço
    public String getFullAddressLine() {
        return street + ", " + city + " " + zipCode;
    }
}

// Classe exibindo Feature Envy
class OrderPrinter {
    public void printOrderShippingLabel(Order order) {
        Customer customer = order.getCustomer();
        Address custAddress = customer.getAddress(); // Obtendo objeto Address
```

```
        // Este método é "invejoso" dos dados de Address
        // Faz múltiplas chamadas aos getters de Address
        String label = customer.getName() + "\n" +
                       custAddress.getStreet() + "\n" +
                       custAddress.getCity() + ", " + custAddress.getZipCode();
        System.out.println(label);
    }
}

class Order { // Classe auxiliar para o exemplo
    private Customer customer;
    public Order(Customer customer) { this.customer = customer; }
    public Customer getCustomer() { return customer; }
}
```

- **Depois (Mover Método - conceitualmente, a lógica de formatação da etiqueta move-se para Address ou um método é adicionado a Address):**

```
Java
class Customer { /*... mesmo que antes... */
    private String name;
    private Address address;

    public Customer(String name, Address address) {
        this.name = name;
        this.address = address;
    }
    public String getName() { return name; }
    public Address getAddress() { return address; }
}

class Address {
    private String street;
    private String city;
    private String zipCode;

    public Address(String street, String city, String zipCode) {
```

```
        this.street = street;
        this.city = city;
        this.zipCode = zipCode;
    }

    public String getStreet() { return street; }
    public String getCity() { return city; }
    public String getZipCode() { return zipCode; }

    // Método movido para aqui (ou um novo criado)
    public String getShippingLabelLine() {
        return street + "\n" + city + ", " + zipCode;
    }
}

class OrderPrinter {
    public void printOrderShippingLabel(Order order) {
        Customer customer = order.getCustomer();
        // Agora chama o método em Address, não acedendo mais diretamente aos seus campos
        String label = customer.getName() + "\n" +
                       customer.getAddress().getShippingLabelLine();
        System.out.println(label);
    }
}
class Order { /*... mesmo que antes... */
    private Customer customer;
    public Order(Customer customer) { this.customer = customer; }
    public Customer getCustomer() { return customer; }
}
```

Benefícios de Resolver Feature Envy:
A resolução deste smell leva a uma menor duplicação de código (se o tratamento de dados for centralizado), melhor organização do código (métodos para tratar dados ficam próximos aos dados reais) e acoplamento reduzido.
Quando Ignorar:
Por vezes, o comportamento é intencionalmente mantido separado dos dados, por exemplo, nos padrões Strategy ou Visitor, onde a flexibilidade de trocar algoritmos ou adicionar

operações a uma hierarquia de classes sem modificá-la é um objetivo de design.

5.2. Intimidade Inapropriada (*Inappropriate Intimacy*)

Definição: Classes que Sabem Demais Sobre os Detalhes Internos Umas das Outras:
Este smell ocorre quando uma classe utiliza excessivamente os campos ou métodos internos de outra classe, ou quando duas classes estão fortemente acopladas através de comunicação bidirecional. Boas classes devem saber o mínimo possível umas sobre as outras para facilitar a manutenção e reutilização. O termo "Negociação Interna" (Insider Trading) é por vezes usado como sinónimo.

Razões para Emergência e Problemas Causados:
Pode surgir de uma falta de limites claros ou de responsabilidades bem definidas entre as classes. Também pode ser um efeito colateral da tentativa de partilhar dados ou funcionalidades de forma demasiado direta. Os problemas incluem alto acoplamento, tornando o sistema frágil (alterações numa classe podem facilmente quebrar a outra), difícil de manter e mais complicado para reutilizar componentes independentemente.

Técnicas de Refatoração:

- **Mover Método (*Move Method*) / Mover Campo (*Move Field*):** Mover as partes (campos ou métodos) que estão a ser acedidas de forma inapropriada para a classe que as utiliza primariamente, ou para uma terceira classe comum, se isso fizer sentido. Isto aplica-se se a classe original não precisar verdadeiramente dessas partes.
- **Extrair Classe (*Extract Class*) e Esconder Delegado (*Hide Delegate*):** Se a intimidade se deve a uma necessidade legítima de colaboração sobre um subconjunto de responsabilidades, Extrair Classe pode criar uma nova classe para essas responsabilidades partilhadas, e Esconder Delegado pode gerir a interação, tornando a relação mais "oficial" e controlada.
- **Mudar Associação Bidirecional para Unidirecional (*Change Bidirectional Association to Unidirectional*):** Se as classes são mutuamente interdependentes, tentar quebrar o ciclo tornando a associação unidirecional.
- **Substituir Delegação por Herança (*Replace Delegation with Inheritance*) (ou vice-versa):** Se a intimidade é entre uma subclasse e uma superclasse, estas podem ser opções dependendo da natureza da relação.
- Mover métodos para evitar intimidade inapropriada é uma abordagem geral sugerida.

Exemplo em Java: Intimidade Inapropriada e Refatoração para Melhor

Encapsulamento :

- **Antes:**

```
Java
import java.util.Date;
import java.util.List;

// Assumir que OrderRepository está definido noutro local
interface OrderRepository { List<Order> getAllOrders(); }

class Order {
    public Date deliveryDate; // Campo público - acesso direto é uma forma de intimidade
    private boolean processed = false;

    public Order(Date deliveryDate) {
        this.deliveryDate = deliveryDate;
    }

    public Date getDeliveryDate() { // Getter é aceitável
        return deliveryDate;
    }

    // Método que idealmente deveria ser usado por OrderService,
    // mas OrderService pode estar a fazê-lo diretamente
    public void markAsProcessed() {
        this.processed = true;
    }
    public boolean isProcessed() { return processed; }
}

class OrderService {
    private OrderRepository orderRepository;

    public OrderService(OrderRepository repo) {
        this.orderRepository = repo;
    }

    // Intimidade Inapropriada: OrderService sabe demais sobre a lógica de deliveryDate de Order
```

```
    public int calculateLateOrders() {
        List<Order> orders = orderRepository.getAllOrders();
        int lateOrders = 0;
        for (Order order : orders) {
            // Comparação direta de datas, OrderService é íntimo com a lógica de datas de Order
            if (order.getDeliveryDate().before(new Date())) {
                lateOrders++;
            }
        }
        return lateOrders;
    }

    // Outro exemplo de intimidade: OrderService manipula diretamente o estado de Order
    public void processOrder(Order order) {
        //... alguma lógica de processamento...
        // Alterar diretamente o estado interno de Order - demasiado íntimo
        // order.processed = true; // Se 'processed' fosse público, ou usando um setter demasiado baixo nível
        order.markAsProcessed(); // Isto é melhor, mas a decisão de o chamar pode ser demasiado íntima
                                 // se OrderService ditar demasiado do ciclo de vida de Order.
        System.out.println("Pedido processado.");
    }
}
```

- **Depois:**

```
Java
import java.util.Date;
import java.util.List;

interface OrderRepository { List<Order> getAllOrders(); }

class Order {
    private Date deliveryDate; // Encapsulado
    private boolean processed = false;

    public Order(Date deliveryDate) {
        this.deliveryDate = deliveryDate;
```

```
    }

    // A responsabilidade de determinar o atraso está agora dentro de Order
    public boolean isLate() {
        return deliveryDate.before(new Date());
    }

    // Order gere as suas próprias alterações de estado
    public void process() {
        //... lógica interna para o que significa um Pedido ser processado...
        this.processed = true;
        System.out.println("Pedido foi processado internamente.");
    }
    public boolean isProcessed() { return processed; }
    public Date getDeliveryDate() { return deliveryDate; } // Getter é aceitável para acesso de leitura
}

class OrderService {
    private OrderRepository orderRepository;

    public OrderService(OrderRepository repo) {
        this.orderRepository = repo;
    }

    public int countAllLateOrders() { // Renomeado para clareza
        List<Order> orders = orderRepository.getAllOrders();
        int lateOrders = 0;
        for (Order order : orders) {
            // OrderService agora pergunta a Order se está atrasado
            if (order.isLate()) {
                lateOrders++;
            }
        }
        return lateOrders;
    }
```

```
    public void processAnOrder(Order order) {
        // OrderService diz ao Pedido para se processar
        order.process();
        // OrderService pode fazer outras coisas relacionadas com o processamento de um pedido
        // que são da sua própria responsabilidade, e.g., notificar outros sistemas.
        System.out.println("OrderService completou a sua parte do processamento do pedido.");
    }
}
```

Benefícios de Reduzir a Intimidade Inapropriada:
A redução deste smell facilita a manutenção e reutilização, pois as classes tornam-se mais independentes. Melhora também o encapsulamento.

5.3. Cadeias de Mensagens (*Message Chains*)

Definição: Longas Sequências de Chamadas de Métodos (Violando a Lei de Deméter):
Uma Cadeia de Mensagens ocorre quando o código envolve uma sequência de chamadas como objetoA.getObjetoB().getObjetoC().fazerAlgumaCoisa(). O cliente solicita um objeto, que por sua vez solicita outro, e assim por diante. Isso frequentemente viola a Lei de Deméter (Princípio do Menor Conhecimento), que afirma que um objeto deve chamar apenas métodos dos seus "amigos" diretos.
Razões para Emergência e Problemas Causados (Fragilidade, Acoplamento Forte à Navegação):
Podem surgir da exposição excessiva de estruturas internas de objetos. O código do cliente torna-se dependente do caminho de navegação ao longo da estrutura de classes. Os problemas incluem tornar o código frágil; qualquer alteração nas relações intermediárias da cadeia exige a modificação do cliente. Acopla fortemente o cliente à estrutura interna de múltiplos objetos e torna o código difícil de ler, entender e manter.

Técnicas de Refatoração:

- **Esconder Delegado (*Hide Delegate*):** A técnica primária. Cria-se um novo método no objeto no início da cadeia (ou num objeto intermediário) que encapsula a cadeia, escondendo a delegação do cliente. O cliente então chama este único método.
- **Extrair Método (*Extract Method*) e Mover Método (*Move Method*):** Considerar por que o objeto final está a ser usado. Pode fazer sentido extrair a funcionalidade realizada pelo final da cadeia para o seu próprio método e, em seguida, mover este método para uma classe anterior na cadeia, ou mesmo para

a primeira classe.

- **Introduzir Variáveis Intermediárias (*Introduce Intermediate Variables*):** Pode decompor a cadeia para melhorar a legibilidade, embora isso não resolva fundamentalmente o problema do acoplamento.

Exemplo em Java: Uma Cadeia de Mensagens e Sua Refatoração usando Esconder Delegado:

- **Antes:**

```
Java
// Assumir que estas classes existem e têm os respetivos métodos getter
class SaveItem {
    private boolean enabled;
    public void setEnabled(boolean enabled) { this.enabled = enabled;
System.out.println("SaveItem enabled: " + enabled); }
    public boolean isEnabled() { return enabled; }
}
class Customizer {
    private SaveItem saveItem = new SaveItem();
    public SaveItem getSaveItem() { return saveItem; }
}
class DockablePanel {
    private Customizer customizer = new Customizer();
    public Customizer getCustomizer() { return customizer; }
}
class Modelisable {
    private DockablePanel dockablePanel = new DockablePanel();
    public DockablePanel getDockablePanel() { return dockablePanel; }
}
class MasterControl {
    private Modelisable modelisable = new Modelisable();
    public Modelisable getModelisable() { return modelisable; }

    public void disableSaving() {
        // Cadeia de Mensagens: Longa sequência de getters

this.getModelisable().getDockablePanel().getCustomizer().getSaveItem().setEnabl
ed(false);
```

```
    }
     public void enableSaving() {

this.getModelisable().getDockablePanel().getCustomizer().getSaveItem().setEnabl
ed(true);
    }
}
```

- **Depois (Esconder Delegado em MasterControl ou Modelisable):**

```
Java
// SaveItem, Customizer, DockablePanel permanecem os mesmos para este exemplo específico de
refatoração.
// Modelisable pode ser um bom local para esconder parte da cadeia.

class Modelisable {
    private DockablePanel dockablePanel = new DockablePanel();
    // getDockablePanel() pode tornar-se privado ou package-private
    // public DockablePanel getDockablePanel() { return dockablePanel; }

    // Novo método para esconder a delegação para ativar/desativar a funcionalidade de gravação
    public void setSaveFunctionalityEnabled(boolean enabled) {
        this.dockablePanel.getCustomizer().getSaveItem().setEnabled(enabled);
    }
}

class MasterControl {
    private Modelisable modelisable = new Modelisable();
    // getModelisable() pode não ser mais necessário publicamente se todas as interações
    // passarem por novos métodos.

    public void disableSaving() {
        // O cliente (MasterControl) agora chama um único método em Modelisable
        modelisable.setSaveFunctionalityEnabled(false);
    }

    public void enableSaving() {
        modelisable.setSaveFunctionalityEnabled(true);
    }
```

```
}
```

Benefícios e Considerações (Evitando o smell de Middle Man):
Os benefícios incluem a redução de dependências entre classes na cadeia e a diminuição de código cliente inchado, além de melhorar a manutenibilidade. No entanto, uma ocultação de delegação excessivamente agressiva pode levar ao smell de Middle Man, onde uma classe acaba com muitos métodos que apenas delegam, tornando difícil ver onde a funcionalidade realmente ocorre. É necessário um equilíbrio.

3.5. Homem do Meio (*Middle Man*)

Definição: Uma Classe que Delega Primariamente o Trabalho:
Uma classe Middle Man é aquela onde a maioria dos seus métodos simplesmente delega chamadas para métodos em outra classe. Se uma classe realiza apenas uma ação – delegar – a sua existência é questionável.
Razões para Emergência (ex: Remoção Excessivamente Agressiva de Cadeias de Mensagens) e Problemas Causados:
Pode ser o resultado de uma aplicação excessivamente zelosa de Esconder Delegado ao remover Cadeias de Mensagens. Pode também ocorrer se o trabalho útil de uma classe for gradualmente movido para outras classes, deixando uma "casca vazia". Os problemas incluem a adição de complexidade desnecessária e uma camada extra de indireção, aumentando o tamanho do código sem adicionar valor significativo. Pode ser cansativo se cada nova funcionalidade no delegado exigir um novo método de delegação no servidor.

Técnica de Refatoração: Remover Homem do Meio (Remove Middle Man):
O cliente deve chamar o objeto delegado diretamente. Isso envolve:

1. Criar um método *getter* na classe servidora (*Middle Man*) para aceder ao objeto delegado.
2. Substituir as chamadas do cliente aos métodos de delegação do *Middle Man* por chamadas diretas aos métodos do delegado, obtidos através do novo *getter*. Se a classe não fizer mais nada, deve-se considerar se ela é realmente necessária.

Exemplo em Java: Identificando um Middle Man e Aplicando Remover Middle Man:

- **Antes:**

```
Java
class RealWorker {
    public void performTask() {
        System.out.println("RealWorker: Executando a tarefa real.");
```

```
    }
}

// Classe MiddleMan
class Manager {
    private RealWorker worker = new RealWorker();

    // Método de delegação
    public void assignTask() {
        worker.performTask(); // Simplesmente delega para RealWorker
    }

    // Potencialmente muitos outros métodos que apenas delegam para RealWorker
    // public String getWorkerStatus() { return worker.getStatus(); }
}

class Client {
    public void initiateWork() {
        Manager manager = new Manager();
        manager.assignTask(); // Cliente fala com o Manager
    }
}
```

- **Depois (Remover Middle Man):**

```
Java
class RealWorker {
    public void performTask() {
        System.out.println("RealWorker: Executando a tarefa real.");
    }
}

class Manager { // Manager ainda pode existir se tiver outras responsabilidades não delegatórias
    private RealWorker worker = new RealWorker();

    // Getter para o delegado
    public RealWorker getDelegateWorker() {
        return worker;
```

```
    }
    // assignTask() é removido, ou o papel do Manager muda.
    // Se Manager tivesse outros papéis, eles permaneceriam.
    // Se assignTask fosse o seu único papel, e agora o Cliente obtém o worker,
    // Manager poderia tornar-se uma Classe Preguiçosa ou ser removido.
}

class Client {
    public void initiateWork() {
        Manager manager = new Manager(); // Cliente ainda pode precisar do Manager por outras razões
        RealWorker actualPerformer = manager.getDelegateWorker(); // Cliente obtém o trabalhador real
        actualPerformer.performTask(); // Cliente fala diretamente com RealWorker
    }
}

// Alternativa se o único propósito do Manager fosse a delegação:
// class Client {
//     public void initiateWork() {
//         RealWorker worker = new RealWorker(); // Cliente cria/obtém RealWorker diretamente
//         worker.performTask();
//     }
// }
// Neste caso, a classe Manager poderia ser removida completamente.
```

Benefícios e Quando Ignorar (Middle Men Intencionais em Padrões de Design):
Os benefícios incluem código menos volumoso e complexidade reduzida.34 No entanto, não se deve remover Middle Men criados por razões específicas e válidas 34:

- Para evitar dependências entre classes (atuando como uma camada de desacoplamento deliberada).
- Quando fazem parte de padrões de design como Proxy, Decorator ou Facade, que utilizam intencionalmente uma estrutura de "homem do meio".