Há várias práticas e abordagens que facilitam a separação de responsabilidades, como veremos nos próximos tópicos.

4.2. Gerencie suas dependências através de injeção

Mesmo com o baixo acoplamento haverá sempre uma ligação entre duas classes que precisam trabalhar em conjunto. Normalmente, há uma classe que necessita dos serviços oferecidos por outra. Qualquer mudança nesta classe que está sendo acessada pode afetar o comportamento da primeira. Há aqui uma relação natural de dependência.[5] Relações como essa indicam ainda que a semântica do cliente é incompleta sem o fornecedor.[2]

Suponha uma classe chamada `CalculadoraDeSalario`, responsável por calcular o salário de um determinado funcionário. Para estimar quais impostos serão descontados do empregado, esta classe precisa da `TabelaDeImpostos`. Existe uma relação de dependência entre essas duas classes e, por isso, dizemos que a `CalculadoraDeSalario` **depende** da `TabelaDeImpostos`. Quando o assunto é organização de classes e objetos, as dependências entre as diversas partes do sistema são um assunto delicado. Devemos nos esforçar ao máximo para diminuir o acoplamento.

Um caso recorrente em diversos projetos é a tentativa de encapsular todo o acesso a dados (persistência) de uma aplicação, centralizando-o em objetos específicos, aplicando o pattern *Data Access Object* (DAO).

```
public class ProjetoDao {
    public void salva(Projeto projeto) {
        // ...
    }

    public void remove(Projeto projeto) {
        // ...
    }

    public Projeto carrega(Long id) {
        // ...
    }
}
```

Partindo do princípio de que essa classe utiliza diretamente JDBC para o acesso ao banco, ela precisa de uma conexão, uma referência a `java.sql.Connection`,

uma referência a `java.sql.Connection` em todos os seus métodos. O uso de algum framework para acesso a dados ou de mapeamento objeto-relacional não resolveria o problema, já que ela precisaria de algo análogo a uma `Session` ou `EntityManager`.

O problema é que o DAO foi criado para encapsular os detalhes de acesso aos dados, porém, precisa de alguns recursos (neste caso, uma conexão) para realizar seu trabalho. Seria o mesmo caso com um `EnviadorDeEmails`, que necessita de uma conexão com o SMTP, por exemplo.

Uma primeira possibilidade para obter uma referência para `Connection` seria abrir conexões diretamente nos métodos que necessitam delas, deixando os detalhes por conta do próprio DAO:

```
public class ProjetoDao {
    public void salva(Projeto projeto) throws SQLException {
        String url = "jdbc:mysql://localhost/db";
        String usuario = "root";
        String senha = "password";

        Connection connection =
                DriverManager.getConnection(url, usuario, senha);

        // uso da conexão ...
        connection.close();
    }

    // outros métodos do DAO

}
```

O código anterior serve para conectar em uma base do MySQL. É inviável repeti-lo em todos os outros lugares que precisem de conexão. Além disso, ainda existem diversos outros detalhes importantes a serem considerados, como a externalização das configurações e o uso de um **pool de conexões** para gerenciamento mais inteligente dos recursos.

Repare que a classe `ProjetoDao` tem responsabilidades demais; além de saber como salvar, buscar e alterar projetos, ainda é responsável por criar uma conexão com a base de dados. Ela é um típico exemplo de uma classe com baixa coesão, devido a tantas responsabilidades.

O primeiro passo para amenizar o problema é centralizar o processo de criar conexões. Dessa forma, a classe pode depender apenas de uma interface e ignorar completamente os detalhes de criação de conexões (se a conexão é nova,

se veio de um pool, qual driver estamos usando, etc.). O uso de uma *factory* permite esconder e centralizar tais detalhes e trocar essa estratégia mais adiante, caso necessário.

```
public void salva(Projeto projeto) throws SQLException {
    Connection connection = new ConnectionFactory().getConnection();

    // uso da conexão ...

    connection.close();
}
```

A obtenção das conexões fica mais simples, mas ainda não resolve o problema. Com esse código, não é possível salvar dois `Projetos` utilizando a mesma conexão, nem executar dois métodos do DAO dentro de uma mesma transação; a cada nova invocação de método uma nova conexão é adquirida. Essa é uma má prática, conhecida como *Sessão por Operação*, *Transação por Operação*, ou ainda *Conexão por Operação* (*Session/Connection per Operation*).

Seria possível agrupar as operações que precisam da mesma conexão/transação dentro de um mesmo método do DAO:

```
public void salva(Projeto projeto, Gerente gerente) {
    Connection connection = new ConnectionFactory().getConnection();

    // ...

    connection.close();
}
```

Mas, neste caso, o DAO acabaria com uma grande quantidade de métodos, a maioria formada praticamente pela combinação de outros. Este é um claro exemplo de péssima divisão de responsabilidades, já que o DAO trabalha demais, abre conexão, cuida de transação, fecha conexão, além da tarefa que realmente lhe cabe: acessar dados. Qualquer novo tipo de transação da aplicação exige um novo método no DAO.

De alguma forma é preciso permitir que a mesma conexão seja utilizada em diversos métodos do DAO, guardando-a em um atributo. Precisamos também de um local para abrir a conexão:

```
public class ProjetoDao {
    private Connection connection;
```

```
    public void inicia() {
        this.connection = new ConnectionFactory().getConnection();
    }
}
```

Criar um método para iniciar o DAO e abrir a conexão resolve um problema, mas gera outro. Podemos nos esquecer de invocar o método `inicia`, fazendo com que `Connection` seja `null`. O objeto DAO não é um **Bom Cidadão** (*Good Citizen Pattern*),[6] já que pode ser instanciado sem que todas as suas dependências estejam resolvidas.

Para que um objeto seja um bom cidadão, ele deve sempre estar em um estado consistente. Para tanto, o uso do construtor é essencial, no qual são preenchidas todas as dependências necessárias para que o objeto possa trabalhar adequadamente, sem a subsequente necessidade de invocar *setters* ou outros métodos de inicialização e configuração. O princípio ainda vai além, definindo algumas boas práticas na manipulação de exceções, *logging* e programação defensiva, tornando esses objetos mais seguros em relação a código de terceiros.

A seguinte versão do DAO usa o construtor para resolver o problema e força a aquisição da conexão no momento de sua instanciação:

```
public class ProjetoDao {
    private final Connection connection;

    public ProjetoDao() {
        this.connection = new ConnectionFactory().getConnection();
        // poderia também abrir uma transação
    }

    public void fecha() {
        // poderíamos consolidar a transação
        this.connection.close();
    }
}
```

A conexão é aberta no construtor, porém, Java não tem destrutores. Mesmo que tivesse, ou o programador usasse o método `finalize`, não seria uma solução adequada, já que não há a garantia de quando um objeto será coletado pelo *Garbage Collector*. É fundamental ter controle sobre onde e quando as conexões são abertas e fechadas, tendo em vista que este é um recurso caro, tal como descritores de arquivos, threads, sockets e outros que usam de I/O ou outras chamadas ao sistema operacional.

Como o nosso DAO já detém a responsabilidade de criar a conexão, é seu papel fechá-la. O método `fecha` parece ser a solução, já que agora é possível instanciar o DAO e invocar várias operações dentro da mesma conexão/transação. Ainda assim, esta solução possui alguns problemas:

- O método `fecha` quebra o encapsulamento, já que expõe um detalhe específico desta implementação de DAO que estamos usando; existe uma conexão que deve ser fechada. Se a implementação do DAO for trocada por uma que persista em arquivos XML, por exemplo, não haveria mais necessidade de ter o método `fecha`, pois não existiria mais nenhuma conexão a ser fechada. Ao mesmo tempo, não se pode simplesmente remover o método `fecha`, já que ele está na interface pública dos DAOs; isso quebraria todos os seus clientes. É nesse momento que começam a surgir as implementações vazias de métodos.
- A boa prática diz: "*se abrir, feche*"! Desta forma, evita-se espalhar a responsabilidade de gerenciar o recurso que estamos manipulando. O problema é que o DAO abre a conexão, mas não sabe quando fechá-la, e delega esta tarefa a quem o utiliza. Os usuários da classe ficam responsáveis por invocar o método `fecha` e saber onde fechar algo que nem foram eles que abriram. A responsabilidade de gerenciar conexões fica assim muito espalhada pelo sistema; todos são responsáveis por saber que uma conexão existe, quando é aberta e quando deve ser fechada.
- Não há erros de compilação por não invocar o método `fecha`, portanto, ninguém é obrigado a invocá-lo. Como a responsabilidade se espalha por todo o sistema, a chance de se esquecer de fechar a conexão aumenta muito, para não dizer que é inevitável. Isso gera problemas de vazamento de conexões, que não são fechadas e podem resultar no desastroso efeito do banco rejeitando novas conexões quando já existirem muitas abertas.
- Fechar a conexão costuma ser mais complexo que uma simples invocação ao método `close`. Pode haver necessidade de controlar as exceções ou executar *rollback* no caso de uma possível transação deixada aberta. Se houver um pool, fechar pode significar devolver a conexão para lá. Para não repetir esse código em todos os DAOs, poderíamos encapsulá-lo, tal como fizemos com a criação na `ConnectionFactory`. Mas ainda precisamos nos lembrar de chamar esse novo método em todos os lugares, espalhando novamente a responsabilidade.

- Para piorar, nesta abordagem ainda não é possível salvar um `Projeto`, atualizar um `Usuario` e remover uma `Historia` usando a mesma conexão, ou dentro da mesma transação. Como cada DAO possui sua própria conexão, não há como compartilhar a mesma entre vários deles; o escopo de existência de uma conexão é interno ao de um DAO.

A solução para todos esses problemas é tirar do DAO a responsabilidade de gerenciar abertura e fechamento de conexões. Sua única responsabilidade (*Single Responsibility Principle*)[3] será tratar das operações de manipulação do banco, mas sem a preocupação de gerenciar conexões. Aumentamos sua coesão e diminuímos seu acoplamento com outras partes do sistema.

Mas não podemos simplesmente remover todo o código relacionado a `Connection`. Para executar seus comandos SQL, ele ainda precisa de uma conexão. Em vez, então, de criar uma nova conexão no DAO, ele passa a receber uma conexão já criada por alguém. E, com esta conexão, poderá executar sua responsabilidade de manipular o banco de dados sem outras preocupações. Receber a conexão em vez de criá-la é um exemplo do princípio da **Injeção de dependências** (*Dependency Injection – DI)*. Uma forma fácil de codificar seria recebendo no construtor da classe:

```
public class ProjetoDao {
    private final Connection connection;

    public ProjetoDao(Connection connection) {
        this.connection = connection;
    }

    public void adiciona(Projeto p) {
        // usa a connection
    }

    // não precisamos de um método fecha()
}
```

Ao dizer que os objetos não vão mais atrás de suas dependências, mas que agora devem apenas recebê-las de alguém, invertemos quem está no controle de gerenciar a dependência. Em vez de fazer os objetos irem atrás daquilo que precisam, o programador faz com que eles recebam os objetos já inicializados e preparados para uso. É o que se chama de **Inversão de Controle** (*Inversion of Control – IoC*). Injeção de dependências é uma das formas mais comuns de IoC.[7,8]

Este conceito teve origem há um bom tempo, no paper de Richard Sweet em 1988,[9] que introduziu o **Hollywood Principle**: "*Don't call, us we will call you*!" (não nos chame, nós o chamaremos!). O objetivo é, ao invés de decidir o momento certo de invocar cada componente, deixar-se ser invocado quando adequado. Ao invés de decidir o melhor momento para invocar certo componente, deixa-se ser invocado por alguém que vai julgar o momento correto. Em DI, vemos esta prática ao deixar de invocar o código de gerenciamento da conexão e passar a ser invocado, por exemplo, no construtor, com tudo já pronto para uso (Figura 4.1).

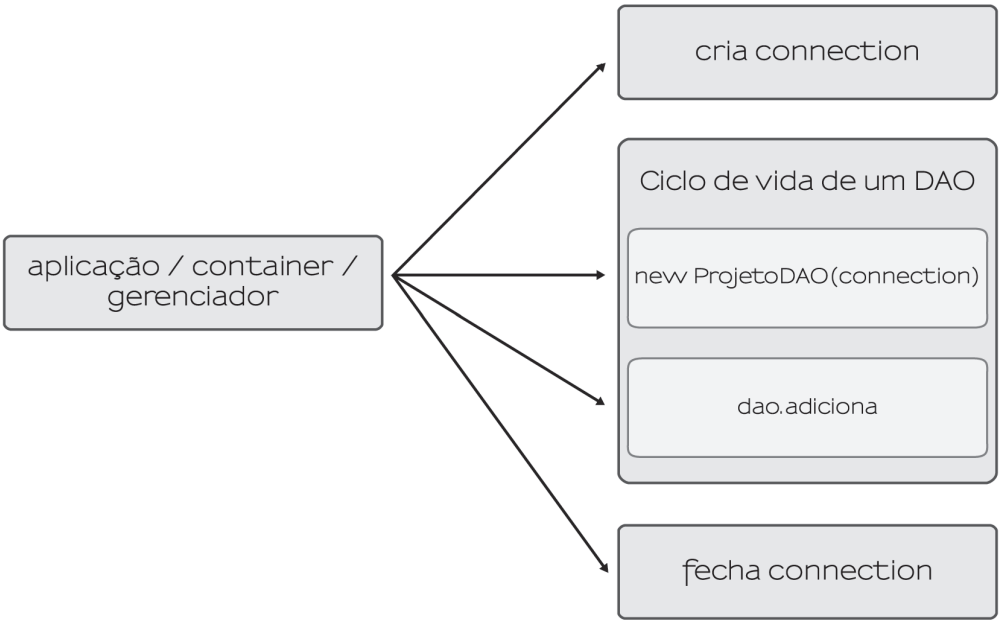

Figura 4.1 – "Não nos chame, nós o chamaremos: Hollywood Principle".

Mas a inversão pode acontecer em outros cenários, que não no gerenciamento de dependências. Uma aplicação visual pode precisar saber quando certo botão é pressionado. Em vez de verificar o estado do botão de tempos em tempos, pode-se apenas registrar um listener que será notificado quando o evento ocorrer. O código de tratamento do evento passa a ser invocado pelo botão, em vez de invocar métodos para descobrir seu estado. O controle foi invertido. Outro exemplo são os containers Web e EJB do Java EE, que invocam no momento certo os métodos os métodos de uma Servlet ou os callbacks de um EJB. **DI é uma das formas de implementar IoC, mas não a única.**

Voltando ao exemplo de DI, resta a questão de quem gerenciará a conexão. Aquele que deseja usar o DAO, precisará se preocupar com isso. Imagine um sistema Web com centenas de Actions diferentes, todas envolvendo o uso de algum DAO. Antes, podia-se simplesmente criá-lo sem se preocupar com conexão; agora passa-se a se preocupar com a conexão:

```
public class AdicionaProjetoAction {
    public void execute() {
        Connection connection;
        connection = new ConnectionFactory().getConnection();
        ProjetoDao dao = new ProjetoDao(connection);
        dao.adiciona(...);
    }
}
```

Usar a `ConnectionFactory` alivia um pouco do problema, mas não é uma boa solução. A action, além de ter responsabilidades normais ligadas a Web, como manipular *request* e *response*, agora também está acoplada a `Connection`, `ConnectionFactory` e DAO. Como diminuir esse acoplamento?

Injeção de dependências novamente. A única responsabilidade da action é adicionar um projeto no DAO, e, para isso, só necessita de um DAO, não precisa se preocupar com os detalhes de sua criação ou de abertura de conexão:

```
public class AdicionaProjetoAction {
    private final ProjetoDao dao;

    public AdicionaProjetoAction(ProjetoDao dao) {
        this.dao = dao;
    }

    public void execute() {
        this.dao.adiciona(...);
    }
}
```

Pode parecer que estamos apenas empurrando o problema, já que alguém terá que criar essa Action. E, agora, instanciá-la exige a existência de um DAO, que, por sua vez, exigirá uma conexão. O problema parece aumentar. Mas o segredo de DI é pensar em cada componente isoladamente, de uma maneira quase egoísta com relação ao restante do sistema. É justamente o princípio de Hollywood na

prática, pensando em cada classe como o ponto mais importante da aplicação e empurrando responsabilidades secundárias para outra classe lidar.

Mas, claro, alguém terá que resolver esse emaranhado de dependências em alguma hora. Será algum componente do sistema cuja única responsabilidade seja exatamente a ligação (*wiring*, ou amarrar, injetar) das dependências. É o componente que estará no controle da aplicação, para que todo o resto possa usar DI e IoC. Na injeção de dependências, existe um dependente (o DAO) e uma dependência (a conexão); e quem amarra tudo isso é um provedor (*provider*). Por ser algo tão importante e bastante comum em diversas aplicações, normalmente usa-se um framework pronto como provider. Spring, Pico Container, Google Guice e o CDI do Java EE 6 são exemplos de container de DI, como veremos no tópico seguinte.

É importante ressaltar, porém, a diferença entre a prática de Injeção de Dependências e o uso de algum framework específico. Inverter o controle e declarar suas dependências para ser injetadas por alguém é considerado uma boa prática de design, e é até mesmo um design pattern bem aceito.[10] Devemos programar nossas classes com DI sempre que possível.

Mas isso não implica que sejamos obrigados a usar algum framework como os citados anteriormente. Há quem defenda evitar o uso de frameworks de DI e que o próprio código de gerenciamento das dependências seja implementado, já que é algo que costuma ser fácil de ser escrito pelo programador, além de ficar mais explícito como os objetos se relacionam.[11,12] É uma opinião que gera bastante polêmica, já que os frameworks modernos são simples de usar e livram o programador de mais uma dor de cabeça, por menor que ela possa ser. Há ainda uma discussão: se, em linguagens mais flexíveis que Java, como Scala ou Ruby, a própria linguagem já não traz recursos equivalentes aos frameworks de DI.[13,14]

Independentemente da forma de se implementar, é praticamente consenso que o uso de Injeção de Dependências é uma prática de design quase obrigatória. Com as vantagens de diminuir o acoplamento entre os componentes, aumentar a coesão das partes do sistema e promover uma melhor separação de responsabilidades, o uso de DI é altamente recomendado.

4.3. Considere usar um framework de Injeção de Dependências

Discutimos no tópico anterior a importância do uso da inversão de controle e da injeção de dependências. Mesmo sendo considerada uma boa prática de design,