```
public Periodo adiaUmaSemana() {
   Calendar novoFim = (Calendar) this.fim.clone();
   novoFim.add(Calendar.DAY_OF_MONTH, 7);
   return new Periodo(inicio, novoFim);
}
```

E, com uma pequena modificação, podemos implementar o design pattern *flyweight* em nossa classe, compartilhando a instância do Calendar de início do, período entre o objeto original e o novo, com uma semana adiada. Para tanto, precisaríamos de um outro construtor privado para ser chamado no adiaUmaSemana que não fizesse o clone.

Utilizar classes imutáveis traz um trabalho a mais junto com os diversos benefícios descritos. Você deve considerar fortemente criar sua classe como imutável.

3.5. CUIDADO COM O MODELO ANÊMICO

Um dos conceitos fundamentais da orientação a objetos é o de que você não deve expor seus detalhes de implementação. Encapsulando a implementação, podemos trocá-la com facilidade, já que não existe outro código dependendo desses detalhes, e o usuário só pode acessar seu objeto através do contrato definido pela sua interface pública.¹⁹

Costumeiramente, aprendemos que o primeiro passo nessa direção é declarar todos seus atributos como private:

```
public class Conta {
    private double limite;
    private double saldo;
}
```

Para acessar esses atributos, um desenvolvedor que ainda esteja aprendendo vai rapidamente cair em algum tutorial, que sugere a criação de *getters* e *setters* para poder trabalhar com esses atributos:

```
public class Conta {
   private double limite;
   private double saldo;

   public double getSaldo() {
      return saldo;
}
```





```
(
```

```
public void setSaldo(double saldo) {
    this.saldo = saldo;
}

public double getLimite() {
    return limite;
}

public void setLimite(double limite) {
    this.limite = limite;
}
```

Essa classe contém alguns métodos que acabam por ferir as ideias do encapsulamento. O método setSaldo é um bom exemplo disso, já que dificilmente o saldo em uma entidade Conta será simplesmente "substituído" por outro. Para alterar o saldo de uma conta, é necessário alguma operação que faça mais sentido para o domínio, como *saques* ou *depósitos*.

Nunca crie um getter ou setter sem uma necessidade real; lembre-se de que precisamos que essa necessidade seja clara para criar qualquer método que colocamos em uma classe. Particularmente, os getters e setters são campeões quando falamos em métodos que acabam nunca sendo invocados e, além disso, grande parte dos utilizados poderia ser substituída por métodos de negócio.²⁰

Essa prática foi incentivada nos primórdios do AWT, para o qual era recomendado criar *getters* e *setters* para serem invocados no preenchimento de cada campo visual da sua interface gráfica com o usuário, cunhando o termo *JavaBean*. Os EJBs também contribuíram para esta prática, como será visto adiante.

Criando classes desta forma, isto é, adicionando getters e setters sem ser criterioso, códigos como conta.setSaldo(conta.getSaldo() + 100) estarão espalhados por toda a aplicação. Se for preciso, por exemplo, que uma taxa seja debitada toda vez que um depósito é realizado, será necessário percorrer todo o código e modificar essas diversas invocações. Um search/replace ocorreria aqui; péssimo sinal. Podemos tentar contornar isso e pensar em criar uma classe responsável por esta lógica:

```
public class Banco {
   public void deposita(Conta conta, double valor) {
      conta.setSaldo(conta.getSaldo() + valor);
   }
   public void saca(Conta conta, double valor) {
      if (conta.getSaldo() >= valor) {
```







```
conta.setSaldo(conta.getSaldo() - valor);
} else {
    throw new SaldoInsuficienteException();
}
}
```

Esse tipo de classe tem uma característica bem procedural, fortemente sinalizada pela ausência de atributos e excesso do uso de métodos como funções (deposita e saca poderiam ser estáticos). Além disso, pode-se dizer que esta classe tem uma *intimidade inapropriada* com a classe Conta, pois conhece demais sua implementação interna. Repare que o método saca verifica primeiro se o saldo é maior que o valor a ser sacado, para, então, retirar o dinheiro. Esse tipo de lógica deveria estar dentro da própria classe Conta.

O princípio do *Tell, Don't Ask* prega exatamente isso: você deve dizer aos objetos o que fazer (como sacar dinheiro), evitando perguntar em excesso o estado ao objeto, como getSaldo, e, a partir desse estado, tomar uma decisão.²¹

Esse tipo de classe é comumente encontrada e é classificada como o pattern *Business Object* por concentrar a lógica de negócios. Já a classe Conta, por ter apenas os dados, recebia o nome *Value Object* (hoje, este pattern tem outro significado). Da forma como está, temos separados nossos dados na classe Conta e a lógica de negócio na classe Banco, rompendo o princípio básico de manter comportamento e estado relacionados em uma única classe.

É o que chamamos de **modelo anêmico** (*anemic domain model*),²² no qual nossa classe Conta parece não ter responsabilidade alguma no sistema, nenhuma ação relacionada. É necessário uma classe externa, Banco, para dar alguma ação para nossa Conta, tratando-a quase como um fantoche.²³ Com isso, a classe Banco conhece detalhes da implementação da classe Conta e, se esta mudar, Banco muito provavelmente mudará junto.

Podemos unir a lógica de negócio aos dados de uma maneira simples, inserindo métodos na classe Conta e removendo os que apenas acessam e modificam diretamente seus atributos:

```
public class Conta {
   private double saldo;
   private double limite;

public Conta(double limite) {
     this.limite = limite;
}
```





```
(
```

```
public void deposita(double valor) {
    this.saldo += valor;
}

public void saca(double valor) {
    if (this.saldo + this.limite >= valor) {
        this.saldo -= valor;
    } else {
        throw new SaldoInsuficienteException();
    }
}

public double getSaldo() {
    return this.saldo;
}
```

Aqui mantivemos o getSaldo, pois faz parte do domínio. Também adicionamos algumas manipulações ao método saca, e poderíamos debitar algum imposto em cima de qualquer movimentação financeira no método deposita. Enriqueça suas classes com métodos de negócio, para que não se tornem apenas estruturas de dados. Para isso, cuidado ao colocar getters e setters indiscriminadamente. Devemos encapsular os dados em atributos de objetos e, ainda, lembrar que a orientação a objetos prega a troca de mensagens (invocação de métodos) de maneira a concentrar as responsabilidades a quem pertence os dados. O próprio Alan Key, que cunhou o termo "programação orientada a objetos", ressalta que "o termo foi uma má escolha, pois diminui a ênfase da ideia mais importante, a troca de mensagens".²⁴

É possível seguir a mesma linha para entidades do JPA/Hibernate, verificando a real necessidade dos getters e setters. Por exemplo, a necessidade de um método setId para a chave primária torna-se discutível no momento em que um framework utiliza reflection ou manipulação de bytecode para ler atributos privados.

Algumas vezes, os *getters* e *setters* são, sim, necessários, e alguns patterns até mesmo precisam de uma separação de lógica de negócios dos respectivos dados.²⁵ Práticas como o *Test Driven Development* podem ajudar a não criar métodos sem necessidade.

Porém, frequentemente, entidades sem lógica de negócio, com comportamentos codificados isoladamente nos *business objects*, caracterizam um modelo de domínio anêmico. É muito fácil terminar colocando a lógica de negócio, que poderia estar em em nossas entidades, diretamente em Actions do Struts Actions do Struts, ActionListeners do Swing e *managed beans* do JSF,





transformando-os em transaction scripts. Este modelo acaba ficando com um forte apelo procedural e vai diretamente na contramão das boas práticas de orientação a objetos e do Domain-Driven Design. 19,26

3.6. Considere Domain-Driven Design

Todo software é desenvolvido com um propósito concreto, para resolver problemas reais que acontecem com pessoas reais. Todos os conceitos ao redor do problema a ser resolvido são o que denominamos domínio. O objetivo de toda aplicação é resolver as questões de um determinado domínio.

Domain-Driven Design (DDD) significa guiar o processo de design da sua aplicação pelo domínio. Parece óbvio, mas muitos softwares não são projetados de acordo com o domínio em que atuam. Podemos perceber essa realidade analisando o código de diversos sistemas atuais, nos quais as entidades não condizem com a realidade dos usuários e são de difícil entendimento. 26,22

Segundo o DDD, é impossível resolver o problema no domínio do cliente sem entendê-lo profundamente. É claro que o desenvolvedor não quer se tornar um completo especialista na área do cliente, mas deve compreendê-la o suficiente para desenvolver guiado pelo domínio.

Para isto acontecer, o ponto-chave é a **conversa**. Conversa constante e profunda entre os especialistas de domínio e os desenvolvedores. Aqueles que conhecem o domínio em detalhes devem transmitir conhecimento aos desenvolvedores. Juntos, chegarão a termos e vocábulos em comum, uma *língua comum*, que todos utilizem. È a chamada **Língua Ubíqua** (*Ubiquitous Language*).

Esta língua é baseada nos termos do domínio, não totalmente aprofundada neste, mas o suficiente para descrever os problemas de maneira sucinta e completa.

Durante a conversa constante, cria-se um modelo do domínio (ou Domain *Model*). É uma abstração do problema real, que envolve os aspectos do domínio que devem ser expressados no sistema, desenvolvida em parceria pelos especialistas do domínio e desenvolvedores. É este modelo que os desenvolvedores implementam em código, que deve ocorrer literalmente, item por item, como foi acordado por todos. Isto possibilita o desenvolvimento de um código mais claro, e, principalmente, que utiliza metáforas próprias do domínio em questão.

Seu programa deve expressar a riqueza do domain model. Qualquer mudança no modelo deve ser refletida no código. Caso o modelo se torne inviável para se implementar tecnicamente, ele deve ser mudado para se tornar implementável.



