Código Limpo:

Habilidades Práticas do Agile Software

Capítulo 11: Sistemas

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Sistemas: Introdução

“Complexidade mata. Ela drena a energia dos desenvolvedores e torna mais difícil planejar, construir e testar o produto.”

Sistemas: Construir uma cidade

O segredo para que quase todas as cidades do mundo se mantenham funcionando e não entrem em colapso é simples: divisão de tarefas.

Setores como saúde, segurança pública e energia tem cada um o seu representante, responsável por organizar a área que domina e fazê-la funcionar perfeitamente.

O que garante um funcionamento fluido é a abstração e modularidade de cada setor.

Não é preciso saber como todos os setores se encaixam para que um em específico funcione.

Sistemas: Construir um sistema

Da mesma forma que particionamos uma cidade, podemos fazer com um sistema.

Entretanto, são poucos os que aproveitam dessa estratégia.

Um código limpo garante essa modularidade e abstração no nível mais baixo do código, mas podemos ir além.

Neste capítulo, vamos entender como trazer esses benefícios para os níveis mais altos, no nível do ***sistema*** .

Separe a construção de um sistema do seu uso

Construir um sistema é bem diferente de ***usá-lo***.

Idealmente, devemos separar a inicialização, que é o momento em que os objetos são criados e suas dependências são atribuídas, da lógica de tempo de execução.

Essa é uma boa prática na escrita de códigos limpos, fundamentada no princípio da ***Separação de Preocupações*** .

Separação de Preocupações: Exemplo

Veja o exemplo de um código que amarra a iniciação com a execução do programa:

```
public Service getService() {
     if (service == null)
          service = new MyServiceImpl(...);
     return service;
 }
```

Esse código segue o modelo de ***Inicialização Tardia*** , e apresenta algumas vantagens, como uma inicialização mais rápida, visto que esse objeto só será criado caso ele seja requisitado, além da garantia que ele nunca será passado com valor null.

Separação de Preocupações: Explicação

Ainda que a Inicialização Tardia apresente vantagens, ela também vem acompanhada de problemas.

Um desses problemas é o fato de que agora temos uma dependência de `MyServiceImpl()` permanentemente no código.

Essa amarra entre a construção e a execução dificulta os testes do sistema, quebra o ***Princípio de Responsabilidade Única*** e o pior de tudo, implica que a classe deve conhecer toda a execução do sistema e garantir que tal objeto é o ideal para ser usado.

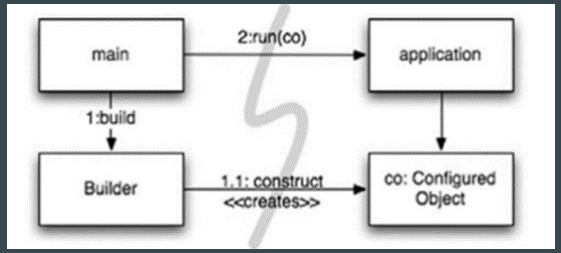
Inicialização Tardia

Uma ocorrência de Inicialização Tardia no sistema não é um grande problema, mas pode se tornar um a partir do momento que essa comodidade se torna uma constante.

Espalhar códigos de inicialização pelo sistema, na maioria dos casos, acaba levando a duplicação de código e quebra a boa prática de modularizar e centralizar esses métodos de forma concisa.

Separação do Main

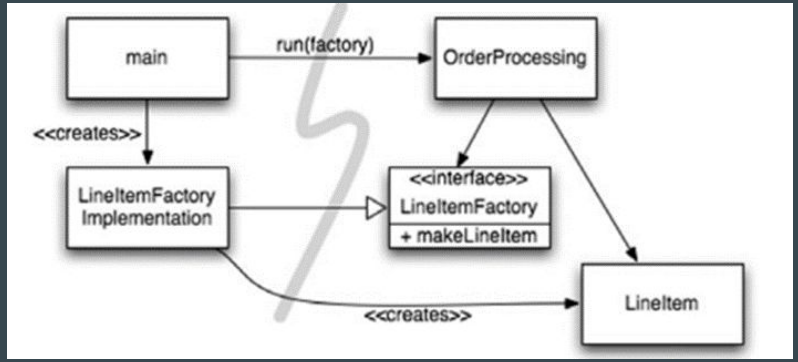
Uma forma de separar a construção do resto do código do sistema é centralizar tudo no Main, seja o código em si ou as chamadas de módulos construtores.



Assim, o fluxo se torna claro: o Main constrói todos os objetos necessários para o sistema e os passa para a aplicação, que simplesmente espera que eles já estejam disponíveis e prontos para uso, escondendo assim a lógica da construção.

Fábricas: Exemplo

Existem casos em que não é possível criar todos os objetos de uma vez, como é o caso do exemplo abaixo, simulando um sistema de processamento de pedidos:



Fábricas: Explicação

Nesse caso, uma nova instância de LineItem precisa ser criada cada vez que um novo pedido é processado.

Para que isso seja feito sem sujar o código com a lógica da criação, podemos usar ***Fábricas Abstratas*** , interfaces que dão autonomia para a aplicação escolher ***quando*** criar uma instância, enquanto os detalhes de ***como*** ela será criada permanecem centralizados sob o domínio do Main.

Essa estratégia possibilita até mesmo que a aplicação crie diferentes instâncias com base nas suas necessidades, por meio da passagem de diferentes parâmetros.

Injeção de Dependência

Uma outra forma de isolar a construção da execução de um sistema é por meio da ***Injeção de Dependência*** , onde se torna possível aplicar a ***Inversão de Controle*** (IoC, sigla em inglês)

A estratégia aqui é repassar responsabilidades secundárias de um objeto, como a de instanciar dependências, para outros objetos criados especificamente para tais funções.

Na maioria dos casos, o responsável por essas ações secundárias serão o Main ou um container de propósito específico.

Injeção de Dependência: Exemplo

A seguir, temos uma exemplificação de uma ***Injeção de Dependência*** usando a API Java JNDI:

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

Nesse caso, a classe principal apenas faz uma chamada esperando o retorno da dependência que satisfaça suas condições.

Injeção de Dependência Verdadeira

No exemplo anterior, apesar de ocorrer a ***Inversão de Controle*** , o fato da classe ainda ser responsável pela chamada torna a ***Injeção de Dependência*** apenas “parcial”.

Numa ***Injeção de Dependência Verdadeira*** , a classe é completamente passiva.

Ela deve oferecer métodos e/ou construtores que irão receber a injeção, que então serão usados pelos containers responsáveis por criar as dependências.

Dessa forma, é possível aproveitar tanto os benefícios da ***Inicialização Tardia*** quanto da ***Inversão de Controle*** .

Aumentando o Escopo: Introdução

É impossível que a primeira versão de um sistema seja perfeita.

Com isso em mente, o ideal é focar nas partes críticas, o que é preciso para o funcionamento no momento atual, e adicionar funcionalidades aos poucos.

TDD, refatoração e um código limpo garantem que essa ideia funcione no nível da aplicação. Mas e quando falamos do sistema como um todo ?

A mesma estratégia pode ser aplicada, sem considerar um planejamento prévio ?

Aumentando o Escopo: Explicação

A resposta é sim. É possível, desde que a ***Separação de Preocupações*** discutida anteriormente seja feita corretamente.

Veremos um exemplo de uma arquitetura que não faz essa separação corretamente, o Enterprise JavaBeans 2 (EJB2).

O código a seguir é um ***Entity Bean,*** uma representação na memória de uma entidade do banco de dados.

Código com Forte Acoplamento: Entity Bean

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

 public interface BankLocal extends java.ejb.EJBLocalObject {
      String getStreetAddr() throws EJBException;
      String getCity() throws EJBException;
      String getState() throws EJBException;
      String getZipCode() throws EJBException;

      void setStreetAddr(String streetAddr) throws EJBException;
      void setCity(String city) throws EJBException;
      void setState(String state) throws EJBException;
      void setZipCode(String zip) throws EJBException;
      Collection getAccounts() throws EJBException;
      void setAccounts(Collection accounts) throws EJBException;
      void addAccount(AccountDTO accountDTO) throws EJBException;
 }
```

Código com Forte Acoplamento: Implementação

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
      // Business logic...
      public abstract String getStreetAddr();
      public abstract String getCity();
      public abstract String getState();
      public abstract String getZipCode();
      public abstract void setStreetAddr(String streetAddr);
      public abstract void setCity(String city);
      public abstract void setState(String state);
      public abstract void setZipCode(String zip);
      public abstract Collection getAccounts();
      public abstract void setAccounts(Collection accounts);

...
```

Código com Forte Acoplamento: Implementação

```
...
   public void addAccount(AccountDTO accountDTO) {
      InitialContext context = new InitialContext();
      AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
      AccountLocal account = accountHome.create(accountDTO);
      Collection accounts = getAccounts();
      accounts.add(account);
   }

   // EJB container logic
   public abstract void setId(Integer id);
   public abstract Integer getId();
   public Integer ejbCreate(Integer id) { ... }
   public void ejbPostCreate(Integer id) { ... }
   public void setEntityContext(EntityContext ctx) {}
   public void unsetEntityContext() {}
   public void ejbLoad() {}
   public void ejbStore() {}
   public void ejbRemove() {}
}
```

Código com Forte Acoplamento: Explicação

No código mostrado, muitos métodos referentes ao ciclo de vida do container são incorporados diretamente na classe.

Essa prática torna algumas ações, como testes unitários ou até mesmo reúso em outras arquiteturas extremamente complexos.

Outro problema é a redundância, uma vez que é comum o uso de Data Transfer Objects (DTOs) na arquitetura EJB2.

Preocupações Transversais

Existem ocasiões em que algumas preocupações ultrapassam os limites dos objetos, e são chamadas de ***Preocupações Transversais*** .

Esse é o nome usado quando partes diferentes do código precisam seguir uma mesma estratégia, indo contra o ideal de modularização.

Um exemplo é a persistência de dados, onde normalmente queremos persistir todos os dados usando uma única estratégia ou banco de dados.

Programação Orientada a Aspectos

Como foi dito anteriormente, as estratégias para lidar com as ***Preocupações Transversais*** acabam ferindo a modularidade do sistema.

Para recuperar essa modularidade, podemos usar a Programação Orientada a Aspectos (AOP), onde os aspectos determinam partes do sistema que devem substituídas.

Essa declaração organiza como o código deve funcionar de forma não-invasiva.

Veremos a seguir três mecanismos que usam aspectos: Proxies Java, Frameworks AOP usando Java puro, e a extensão AspectJ.

Proxies Java

O uso de Proxies é recomendado para problemas mais simples, como encapsular chamadas de métodos de classes ou objetos individuais.

Por padrão, Proxies oferecidos pela JDK só funcionam em interfaces.

Para usar em classes, é preciso usar uma biblioteca de manipulação de bytecode, como Code Generation Library (CGLIB), ASM ou Javassist.

A seguir, temos um exemplo apresenta o esqueleto de um Proxy JDK para a aplicação Bank de antes, apenas com os Getters e Setters da lista de contas:

Proxies Java: Exemplo – Interface

```
// The abstraction of a bank.
 public interface Bank {
     Collection<Account> getAccounts();
     void setAccounts(Collection<Account> accounts);
 }
```

Proxies Java: Exemplo – Implementação

```
// BankImpl.java
 public class BankImpl implements Bank { //POJO
      private List<Account> accounts;

      public Collection<Account> getAccounts() {
         return accounts;
      }

      public void setAccounts(Collection<Account> accounts) {
         this.accounts = new ArrayList<Account>();
         for (Account account: accounts) {
            this.accounts.add(account);
         }
      }
 }
```

Proxies Java: Exemplo - Handler

```
// "InvocationHandler" required by the proxy API.
 public class BankProxyHandler implements InvocationHandler {
      private Bank bank;
      public BankHandler (Bank bank) {
         this.bank = bank;
      }
      // Method defined in InvocationHandler
      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
         String methodName = method.getName();
         if (methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
         } else if (methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
         } else {
  ...
```

Proxies Java: Exemplo – Instância Concreta

Por fim, em algum lugar pertinente do código, a instanciação concreta do Proxy:

```
// Somewhere else...
 Bank bank = (Bank) Proxy.newProxyInstance(
      Bank.class.getClassLoader(),
      new Class[] { Bank.class },
      new BankProxyHandler(new BankImpl()));
```

Proxies Java: Considerações

Com isso, temos uma forma de lidar com a transversalidade do código de forma menos invasiva.

Entretanto, a quantidade de código e a complexidade atrelada deixam mais difícil de escrever um código limpo, tornando essa abordagem menos atrativa.

Além disso, Proxies não oferecem uma forma de substituir trechos de código espalhados por todo o sistema, um requisito importante para uma solução orientada a aspectos.

Frameworks AOP com Java puro

Como forma de fugir da complexidade de trabalhar manualmente com Proxies, podemos usar alguns frameworks, como o Spring AOP ou o JBoss AOP.

Aplicando a lógica em objetos Java puros, que não tem nenhuma dependência externa, esses frameworks tornam o código mais simples de testar e expandir, se necessário.

A principal diferença está no uso dos Proxies, que são criados e instanciados automaticamente pelo framework, de forma transparente para o usuário.

A seguir, um exemplo usando Spring:

Frameworks AOP com Java puro - Exemplo

```
<beans>
      ...
      <bean id="appDataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
            destroy-method="close"
            p:driverClassName="com.mysql.jdbc.Driver"
            p:url="jdbc:mysql://localhost:3306/mydb"
            p:username="me"/>
      <bean id="bankDataAccessObject"
      class="com.example.banking.persistence.BankDataAccessObject"
            p:dataSource-ref="appDataSource"/>
      <bean id="bank"
      class="com.example.banking.model.Bank"
            p:dataAccessObject-ref="bankDataAccessObject"/>
      ...
 </beans>
```

Frameworks AOP com Java puro – Explicação

Cada “bean” no código anterior funciona como uma camada de encapsulamento, tal qual uma boneca russa:

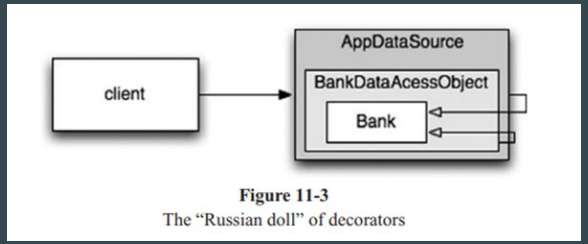


Figure 11-3
The “Russian doll” of decorators

Para o cliente, ele está simplesmente invocando um método da classe Bank, quando na verdade o método está sendo invocado na camada mais externa.

AspectJ

Por fim, a terceira ferramenta e mais completa delas é a linguagem AspectJ.

O AspectJ basicamente estende as ferramentas nativas do Spring e do JBoss, trazendo formas mais refinadas de separar as preocupações do sistema.

Apesar de ser considerado um pouco mais complexo, por demandar uma certa familiaridade com as ferramentas para ser usado no seu potencial máximo, algumas estratégias incorporadas, como as Annotations, facilitam a adaptação.

Considerações Finais

Por fim, a regra de ouro é ***Simplicidade.*** Um sistema bem estruturado e com uma lógica clara torna o desenvolvimento muito mais fluido e os testes mais simples.

Trabalhar em um sistema modularizado e pouco dependente de frameworks externos garante uma maior agilidade, e consegue entregar valor para o cliente de forma efetiva.

Seja em nível de código ou de sistema, o ideal é trabalhar com a estratégia que melhor se adapta ao projeto, pois exagerar na complexidade mata projetos e drena as energias dos desenvolvedores.

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:**
Habilidades Práticas do Agile Software. 1. ed.
Rio de Janeiro: Alta Books, 2009.

A equipe

Gabriel Gatti, Autor

Aluno e voluntário do PET/ADS desde abril de 2023.

João Melão, Revisor

Aluno do terceiro período, é voluntário do PET/ADS desde fevereiro de 2023.

Lucas Oliveira, Tutor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.