Código Limpo:

Habilidades Práticas do Agile Software

Capítulo 8: Limites

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

Limites: Introdução

Ao programar, raramente utilizamos somente o que desenvolvemos; também utilizamos pacotes de outras empresas, códigos livres ou até componentes e subsistemas de nossa própria empresa.

A integração de código externo deve ser feita de modo a evitar maiores dificuldades futuras e facilitar mudanças. Nessa apresentação aprenderemos a fazer isso.

Sempre haverá uma tensão natural entre o fornecedor de uma interface e seu usuário, devido às características de pacotes e frameworks voltadas para uma maior aplicabilidade.

Já o usuário busca uma interface voltada para suas necessidades. Esse é um momento que pode causar problemas de limites no sistema.

A utilização de código de terceiros: problemas

Armazenar registros de vendas com APIs genéricas como List e Map não garantem a integridade da informação.

Métodos como clear e replace permitem remover todos os elementos e substituir um elemento, respectivamente, causando comportamentos indesejados.

O código a seguir não garante a integridade da informação, além de não especificar o tipo do objeto salvo:

```
private List records = new ArrayList();
```

A utilização de código de terceiros: solução

O uso de generics (List<Sale>) ajuda, mas não resolve todos os problemas citados.

Um exemplo de uma forma limpa de se usar o List é ilustrado a seguir:

```
public class SalesRecords {
   private List<Sale> records = new ArrayList();

   public void addRecord(Sale sale){ records.add(sale); }

   public List<Sale> getAllRecords(){ return Collections.unmodifiableList(records); }
}
```

A criação de uma classe e métodos específicos para a necessidade do negócio oculta a API genérica e evita acesso a métodos desnecessários.

A utilização de código de terceiros: outro problema

Com a solução anterior, podemos até mudar o funcionamento interno da classe SalesRecords. Se não houver mudança na assinatura, nenhum código fora da classe precisará ser refatorado.

```
public class SalesRecords {
   private Map<Integer, Sale> records = new LinkedHashMap<>();

   public void addRecord(Sale sale){ records.put(sale.id, sale); }

   public Sale findSale(int id){ return records.get(id); }

   public List<Sale> getAllRecords(){
       return Collections.unmodifiableList(new ArrayList<>(records.values()));
   }
}
```

Explorando e aprendendo sobre limites

Usar códigos de terceiros ajuda a conseguir mais funcionalidades em menos tempo. Mas, como se deve fazer a utilização desses pacotes de código?

Não é nossa função testá-los, mas fazer testes para entendermos como usar esses pacotes poderá facilitar o entendimento. Isso se chama teste de aprendizagem.

Aprendendo um novo pacote: Log4j

Digamos que vamos utilizar o pacote log4j em um projeto, como estamos animados para aprender esse novo pacote, nós já escrevemos nosso primeiro caso de teste:

```
@Test
public void logCreateTest() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.info("olá alunos do pet");
}
```

Ao rodar o teste, o logger produz um erro dizendo que precisamos de um Appender.

```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe" ...
log4j:WARN No appenders could be found for logger (MyLogger).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
```

Aprendendo um novo pacote: Log4j

Lendo um pouco, entendemos que precisamos de um ConsoleAppender, então criamos um e vamos testar novamente.

```
@Test
public void logAddAppederTest() {
   Logger logger = Logger.getLogger("MyLogger");
   ConsoleAppender appender = new ConsoleAppender();
   logger.addAppender(appender);
   logger.info("olá alunos do pet");
}
```

Novamente temos um erro, dessa vez descobrimos que o Appender não possui fluxo de saída.

```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe" ...
log4j:ERROR No output stream or file set for the appender named [null].
```

Aprendendo um novo pacote: Log4j

Depois de buscar ajuda, tentamos o seguinte:

```
@Test
public void logCreateOutputStreamTest () {
   Logger logger = Logger. getLogger("MyLogger");
   logger.removeAllAppenders() ;
   ConsoleAppender appender =  new ConsoleAppender(
      new PatternLayout("%p %t %m%n"),
       ConsoleAppender. SYSTEM_OUT );
   logger.addAppender(appender) ;
   logger.info("olá alunos do pet");
}
```

E deu certo!

```
"C:\Program Files\Java\jdk-17.0.2\bin\java.exe" ...
INFO main olá alunos do pet
```

Aprendendo um novo pacote: Log4j

Algumas coisas parecem estranhas, como ter que definir que o ConsoleAppender escreva no console.

Pesquisando descobrimos bastante coisa sobre o log4j, colocamos esse conhecimento em uma série de testes simples de unidade a seguir.

Aprendendo um novo pacote: Log4j

Abaixo estão os testes de aprendizagem para a compreensão do Log4j

```
class Log4jTest {

   private Logger logger;

   @BeforeEach
   public void initialize() {
       logger = Logger.getLogger("logger");
       logger.removeAllAppenders() ;
       Logger.getRootLogger().removeAllAppenders() ;
   }
   @Test
   public void basicLogger() {
       BasicConfigurator. configure();
       logger.info("basicLogger");
   }
```

Aprendendo um novo pacote: Log4j

```
    @Test
    public void addAppenderWithStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n"),
            ConsoleAppender.SYSTEM_OUT));
        logger.info("addAppenderWithStream");
    }

    @Test
    public void addAppenderWithoutStream() {
        logger.addAppender(new ConsoleAppender(
            new PatternLayout("%p %t %m%n")));
        logger.info("addAppenderWithoutStream");
    }
}
```

Agora que você sabe como obter um console simples e configurado, encapsule esse conhecimento conforme aprendido anteriormente de forma que isole o resto da aplicação da interface do log4j

Testes de aprendizagem são melhores que de graça

Os testes acabam não custando nada, já que teríamos que aprender sobre a API de qualquer maneira.

Testes de aprendizagem também facilitam a verificação de novas versões dos pacotes externos, garantindo o comportamento e a compatibilidade com o nosso código.

Qualquer incompatibilidade será descoberta imediatamente.

Uso de código que ainda não existe

Há outro tipo de limite, o que separa o conhecido do desconhecido.

Nem todas as partes do código serão conhecidas, seja por opção de não olhar para além do limite, pelo código ainda não estar pronto ou por outra razão.

As vezes precisamos trabalhar com algo que depende de uma parte desconhecida para o seu funcionamento.

Para contornar isso, podemos criar uma interface própria, se preocupando somente em definir os limites, sem pensar no funcionamento interno.

Uso de código que ainda não existe

O código a seguir ilustra a interface de um repositório.

```
public interface Repository <K, T> {
   void create(T item);
   void update(T newItem);
   T findOne(K key);
   List<T> findAll();
   void delete(K key);
}
```

Ao se criar um objeto do tipo Repository, não há preocupação com o comportamento concreto do repositório, apenas com a API que será utilizada.

Uso de código que ainda não existe

Com a interface definida, podemos implementar uma classe repository falsa para testar outras classes e continuar o desenvolvimento.

```
public class FakeSalesRepository implements Repository<Integer, Sales> {
    private Map<Integer, Sales> records= new LinkedHashMap<>();
    @Override
    public void create(Sales item) { records.put(item.id, item); }
    @Override
    public void update(Sales newItem) { records.replace(newItem.getCode(), newItem); }
    @Override
    public Sales findOne(Integer key) { return records.get(key); }
    @Override
    public List<Sales> findAll() { return new ArrayList<>(records.values()); }
    @Override
    public void delete(Integer key) { records.remove(key); }
}
```

Adapter

O design pattern Adapter é um padrão estrutural criado para possibilitar que objetos com interfaces incompatíveis trabalhem juntos.

Esse padrão possibilita reutilizar classes existentes sem modificar o código fonte, convertendo a interface de um objeto para a interface do outro objeto, como no exemplo do próximo slide.

Adapter

A classe a seguir possui implementação útil, mas uma API incompatível com as necessidades da aplicação.

```
public class TomadaTresPinos { // interface existente
   public void ligarTomadaTresPinos () {
       //implementação do método
   }
}
```

Para ser adotada na aplicação, a classe acima deveria estar de acordo com a interface a seguir:

```
public interface TomadaDoisPinos { //interface necessária
   void ligarTomadaDoisPinos ();
}
```

Adapter

A criação da uma classe adaptadora permite utilizar as funcionalidades úteis de acordo com a API desejada pelas regras de negócio da aplicação.

```
public class AdaptadorDeTomada implements TomadaDoisPinos{

    private TomadaTresPinos tomadaTresPinos;

    public AdaptadorDeTomada(TomadaTresPinos tomadaTresPinos) {
        this.tomadaTresPinos = tomadaTresPinos;
    }

    @Override
    public void ligarTomadaDoisPinos() {
        tomadaTresPinos.ligarTomadaTresPinos();
        System.out.println("Usando o adapter para conectar a tomada");
    }
}
```

Conclusão

Os interesses no projeto de uma API de terceiros nem sempre são os mesmos da nossa aplicação.

Sempre que utilizar código de terceiros, tente isolá-lo da aplicação.

Crie classes para ocultar detalhes de APIs quando elas forem mais abrangentes que o necessário.

Use testes de aprendizagem para fortalecer a compreensão e a segurança sobre APIs externas.

Crie interfaces para especificar API de funcionalidades ainda não existentes ou muito voláteis.

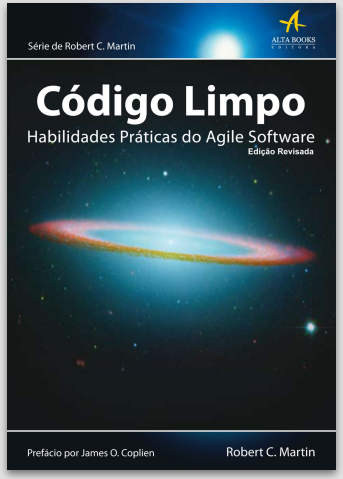
Utilize um Adapter para adotar funcionalidades úteis fornecidas por interfaces incompatíveis.

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

Luiz Chiquetano, Autor

Aluno e bolsista do PET/ADS desde abril de 2023
LinkedIn

Lucas Oliveira, Revisor

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