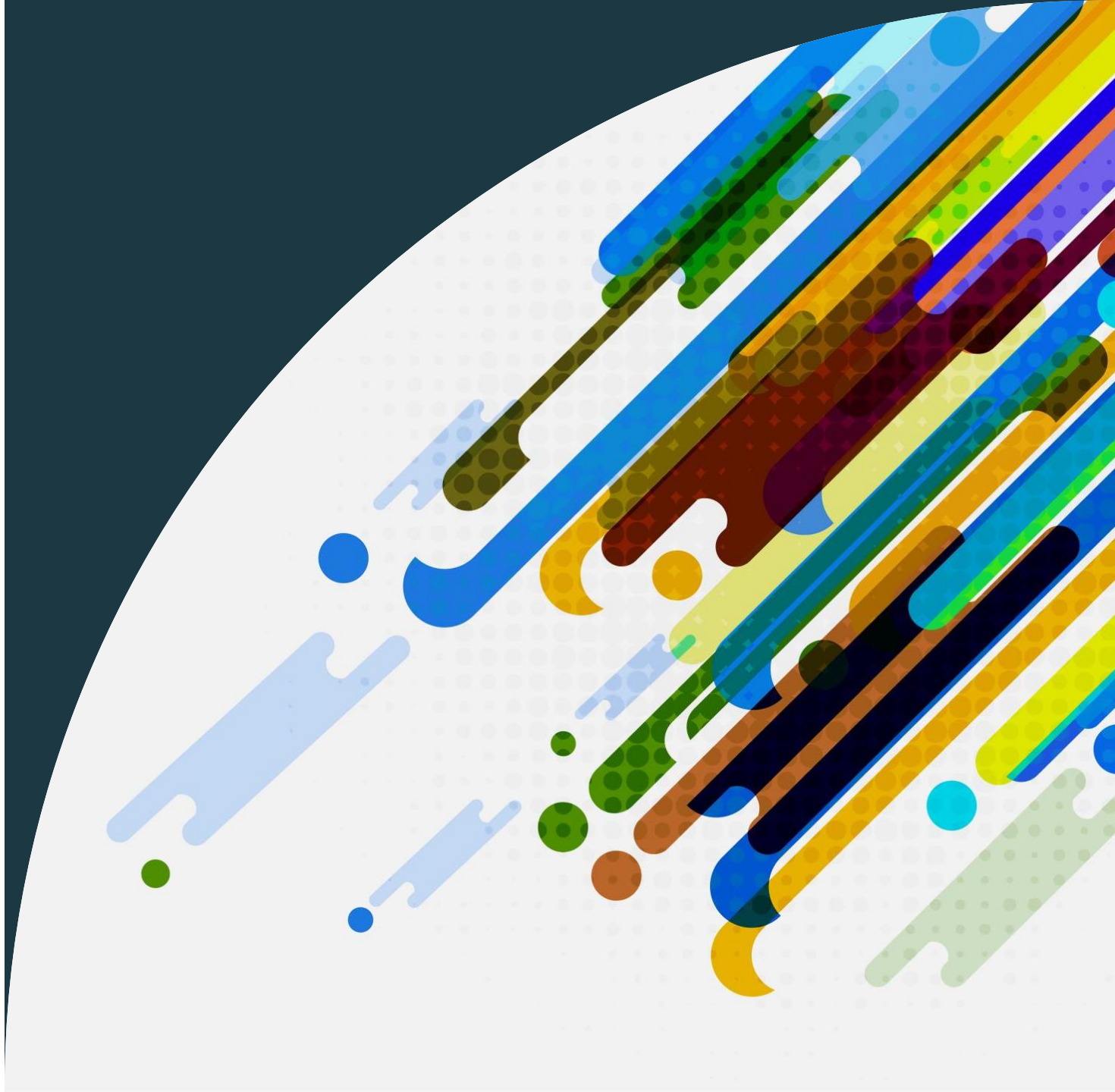
Escrevendo testes maiores

Teste de integração

Teste de sistema



Quando escrever testes maiores

- Testes unitários são os mais adequados para os testes das regras de negócio de um sistema
- Mesmo quando as classes possuem dependências é possível isolar seu comportamento usando dubles
- Existem situações, porém, que o teste unitário não é a melhor opção (lista não exclusiva):
  1. Quando um conjunto de classes deve ser testado em conjunto porque caso contrário os testes ficam muito fracos
  2. Quando uma classe depende de um resultado de outra cuja construção de um duble seria extremamente custosa
  3. Quando os componentes se comunicam com uma infraestrutura externa (tal como um banco de dados)

Exemplos

- Na sequencia serão apresentados 3 exemplos onde o uso de testes de integração é indicado:

  Exemplo 1: o cálculo do custo de um carrinho de compras depende de uma série de regras implementadas por classes distintas. O conjunto de regras é definido por um "factory". Todas estas classes podem ser testadas individualmente, mas o comportamento de grupo será melhor testado em conjunto.

  Exemplo 2: o componente que se quer testar é um plug-in em uma arquitetura plug-and-play. O plug-in pode ser testado individualmente mas é necessário verificar se seus serviços estão sendo solicitados corretamente. É o teste de uma situação que vai além do nosso código.

  Exemplo 3: uma classe que faz acesso a um banco de dados

Exemplo 1: custo de um carrinho de compras

O custo básico de um carrinho de compras é dado pelo somatório do preço dos produtos pela quantidade. Além do custo básico, porém, uma série de outros custos podem ser adicionados e irão compor o chamado custo final. Neste exemplo são dois custos adicionais:

- Custo de frete, proporcional a quantidade de itens no carrinho

  Entre 1 e 3 itens o custo do frete é R$ 5,00

  Entre 4 e 10 itens o custo do frete é 12.50;

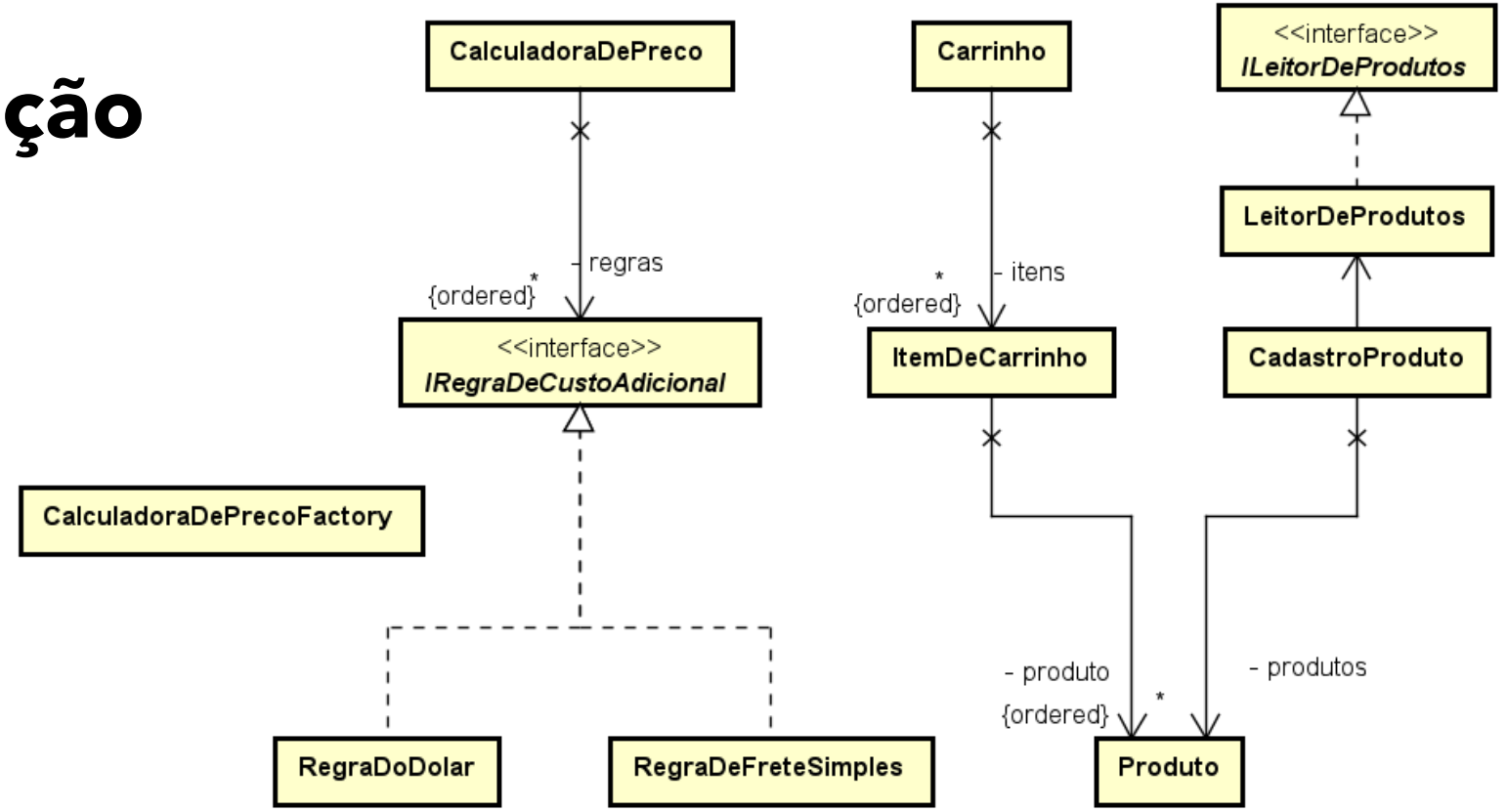
  Acima de 10 itens o custo do frete é R$ 20,00

- Custo adicional para produtos em dólar

  Se houver pelo menos um produto no carrinho cujo custo é em dólar, será acrescentada uma taxa fixa de R$ 7,50

Diagrama de classes da solução

- Observe como fica fácil de criar novas regras e/ou conjuntos de regras
- Observe como fica fácil de testar cada uma das regras individualmente



A implementação

- Para garantir a extensibilidade do código, definiu-se a interface padrão das regras dos custos adicionais (IRegraDeCustoAdicional)
- Cada custo em particular foi implementado como uma classe distinta que implementa a interface IRegraDeCustoAdicional (RegraDoFreteSimples, RegraDoDolar).
- A classe CalculadoraDeCustos tem a responsabilidade de calcular os custos de um carrinho de compras. O cálculo do custo básico é fixo e a lista de custos adicionais é informada por injeção de dependência,
- Os métodos de cálculo de custos recebem por parâmetro o carrinho sobre o qual os custos devem ser computados
- Essa implementação facilita o acréscimo de novos tipos de custos (ou descontos) conforme a necessidade

Testando os cálculos adicionais

- As classe RegraDoFreteSimples depende apenas da quantidade de itens no carrinho. Então é fácil gerar mocks que respondam a quantidade de acordo com a necessidade

  0 itens, 1 item, 3 itens, 4 itens, 10 itens, 15 itens

- A classe RegraDoDolar já depende do tipo de moeda em que o produto é negociado. Neste caso a elaboração de um mock fica complicada e é mais fácil criar instancias de carrinho conforme a necessidade ← Teste de integração

  Nenhum item em dólar, pelo menos um item em dólar, mais de um item em dólar

Testando a factory e a calculadora de precos

- A classe CalculadoraDePrecoFactory é fácil de testar

  Basta verificar as características da lista de custos adicionais gerada

- Já a classe CalculadoraDePrecos é mais complicada de testar

  O cálculo do custo básico irá necessitar de um carrinho qualquer para testar a soma dos preços unitários dos produtos multiplicados pela quantidade ← teste de integração

  Já o cálculo dos custos adicionais pode ser feito criando-se mocks para os custos adicionais que retornem valores conhecidos. Pode-se usar o mesmo carrinho que para o teste do custo básico porque este não irá influenciar nos testes e irá retornar um custo básico conhecido

  Finalmente precisamos testar a calculadora de preços usando um carrinho "real" e custos adicionais "reais" de maneira a verificar se funcionam todos em conjunto ← teste de integração → ver prox. slide

Teste de integração final

- Utilizando as técnicas baseadas em análise do domínio identifique um conjunto de casos de teste que procure testar o conjunto de classes de maneira integrada. Note que o particionamento é feito agora em um nível mais alto de abstração.

  Carrinho:

  Vazio, com um elemento, com vários elementos

  Produtos:

  Uma unidade, várias unidades

  Preço de entrega:

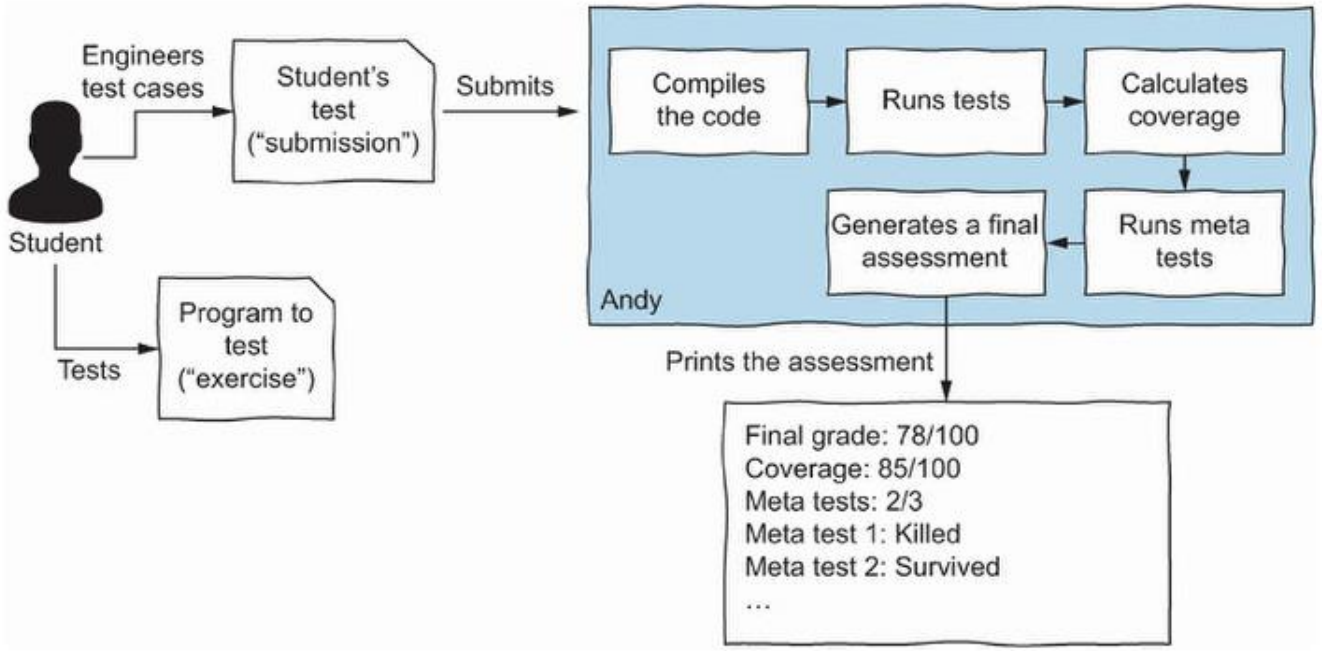
  1 a 3 itens, 4 a 10 itens, mais de 10 itens

  Tipo de moeda:

  Com produtos em dólar, sem produtos em dólar

Exemplo 2: teste de sistemas que vão além do nosso código

- Imagine um sistema que verifica se os testes que os alunos projetaram usando JUnit tem a cobertura adequada (“testVerifier”)
- Além disso analisa outras métricas do código fazendo meta análises
- O fluxo deste sistema pode ser visto na figura ao lado

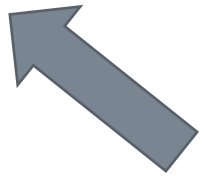


Testando o “testVerifier”

- O teste do “testVerifier” só é possível integrando-se todos os módulos

  A saída de um módulo é a entrada do próximo e, gerar “dubles” para essas entradas é extremamente complicado

  Os diferentes módulos são chamados em sequencia a partir da ferramenta de geração de plugins do VSCode. Então não basta saber se os módulos funcionam individualmente. É necessário saber se a forma como eles são chamados – no caso pelo VSCode – é a forma pela qual o desenvolvedor entendeu que eles seriam chamados.



Exemplo 3: teste de classes que fazem acesso a dados

- Testar sistemas que fazem acesso a dados, se foram corretamente projetados, é fácil

  Basta criar mocks ou fakes para os DAOs

- O desafio esta em testar os DAOs propriamente ditos. Em princípio não há muita diferença em relação aos testes das outras classes:

  Usam-se as mesmas técnicas de geração de casos de teste analisando-se o que deve ser retornado em cada consulta

  Usa-se JUnit para implementar os casos de teste

  Para cada teste

    Garantir que o banco está no esta inicial correto

    Conectar com o banco de dados

    Aplicar os casos de teste

    Fechar a conexão com o banco

    Verificar os resultados

Como lidar com os testes de BD na prática

- Usar bancos em memória para acelerar os testes
- Usar um banco de dados específico para testes independente do de produção
- Usar ferramentas de apoio tais como flyway https://flywaydb.org ou liquibase https://www.liquibase.org que ajudam a colocar o banco no estado inicial ou usar o esquema correto.
- Usar "entity builders": muitas vezes os testes necessitam que sejam criadas entidades. Nem sempre essas entidades são simples de criar. O uso de "builders" para a criação de entidades pode facilitar a criação de casos de teste