PERCURSO DE APRENDIZAGEM 2

Universidade de Fortaleza | EAD

AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE

PROGRAMAÇÃO DEFENSIVA

Este trabalho está licenciado com uma Licença Creative Commons
Atribuição-NãoComercial-SemDerivações 4.0 Internacional.

Sumário

1. Legibilidade de código

2. Planejamento e escrita de testes unitários

3. Manipulação e tratamento de erros

4. Técnicas de fatoração: Extração e movimentação de métodos, renomeação de variáveis, classes e métodos e extração de classes

Sumário clicável

Voltar ao sumário

No segundo percurso de aprendizado do componente curricular Ambiente de Desenvolvimento de Software, visitaremos conceitos importantes de programação, ou, de uma forma mais específica, iremos visitar os principais conceitos da programação defensiva. Primeiramente, veremos a importância da legibilidade de código. Em um primeiro momento, é possível considerar a escrita de código-fonte como a parte mais importante no processo de programação. Todavia, veremos que, no dia a dia de um desenvolvedor, a tendência é que a leitura seja realizada com maior frequência do que a escrita. Logo, realizar uma programação pensando na legibilidade pode melhorar a qualidade do código sendo produzido e trazer outros benefícios.

Outro aspecto de qualidade de código que aprenderemos é a redução da quantidade de erros (ou bugs) que existem dentro dele. Mesmo que de forma não intencional, é comum introduzirmos erros enquanto programamos. Uma tarefa importante é o planejamento de testes. Veremos que um código pode ser dividido em unidades e, consequentemente, podemos testá-las a fim de verificar seu correto funcionamento enquanto procuramos por erros. Damos a esses testes o nome de testes unitários.

Em seguida, iremos aprender como manipular e tratar erros, bem como aplicar técnicas de fatoração que nos ajudam a modificar o código existente para melhorar sua qualidade. Por fim, veremos como realizar essas técnicas dentro dos mais populares ambientes de desenvolvimento de software.

Olá

Voltar ao sumário

1. Legibilidade de código

No percurso anterior, estudamos ambientes de desenvolvimento e seus componentes, desde a instalação e configuração de uma IDE até a aplicação de seus principais utilitários. Além disso, vimos diferentes tarefas que fazem parte do conjunto de responsabilidades de um desenvolvedor. Neste percurso, faremos uma análise mais aprofundada na programação. Ou, mais especificamente, na programação defensiva.

Quando falamos de programação, geralmente nos referimos à tarefa de implementar uma sequência de passos executáveis que visam realizar uma computação ou resolver um problema. Todavia, não existe apenas uma solução para um problema. Mesmo em um grupo com muitos desenvolvedores, é muito difícil que dois deles proponham soluções iguais, pois existem inúmeras formas de se implementar um código. Note que, mesmo que existam múltiplos códigos-fonte para resolver um mesmo problema, ainda que todos funcionem por completo, existirão diferenças na qualidade dos códigos produzidos.

EXEMPLO: Para entendermos essas diferenças, considere o seguinte cenário: João produz seu código-fonte seguindo boas práticas, tendo certeza de que seu código será de fácil leitura, agradável e catalisador de mudanças produtivas. O código de Pedro também funciona perfeitamente, mas é inferior em termos de legibilidade. Imagine que seu chefe lhe dá a escolhe de realizar manutenção no código de um dos desenvolvedores. Você preferiria trabalhar no código de João ou no código de Pedro? É nesse contexto que estudaremos a importância da legibilidade de código.

Muitos acreditam que a escrita de código-fonte é a parte mais comum no processo de programação. Todavia, ao observar o dia a dia de um desenvolvedor, vemos que na verdade a leitura é realizada com maior frequência do que a escrita. Além disso, um ambiente de desenvolvimento moderno conta com utilitários que produzem código-fonte de forma automática, diminuindo ainda mais a frequência com a qual o desenvolvedor escreve linhas de código. Atualmente, desenvolvedores também podem contar com ferramentas que transformam diagramas em código-fonte com grande facilidade. Isto faz com que os desenvolvedores se preocupem em maior grau com a leitura e compreensão de código e outros artefatos em um projeto de software.

Voltar ao sumário

Importante!

Ambientes de desenvolvimento especializados em um conjunto específico de linguagens de programação são capazes de disponibilizar em alguns instantes um espaço de trabalho que abstrai uma série de boas práticas, alcançando altos níveis de abstração.

Dessa forma, é necessário que o desenvolvedor faça adesão aos novos padrões de desenvolvimento que melhoram o entendimento e aumentam a qualidade de um software (LAMOUNIER, 2021). Logo, a pergunta que devemos fazer neste momento é: o que faz de um software um software de qualidade?

Segundo Martin (2020), "Software de excelente qualidade significa código com um design simples e testado. É um código que não temos receio de modificar e que possibilita que os negócios respondam de modo rápido. É um código flexível e robusto".

Podemos concluir que devemos buscar simplificar nossos códigos, tentando ser o mais direto possível e, consequentemente, melhorando a legibilidade. Podemos concluir também que se faz necessária a aplicação de testes, mas essa tarefa veremos no próximo circuito. Por ora, devemos buscar um código limpo, sem trechos ambíguos ou confusos.

É possível considerar então que o código de um desenvolvedor é um código limpo quando este permite que outros desenvolvedores leiam seu conteúdo com facilidade e entendam suas especificações, diminuindo custos de manutenção, facilitando a correção de erros e assim por diante (LEWIS, 2020).

A prática de desenvolvimento de código limpo, no inglês *Clean Code*, procura remediar o fato de que muitos programadores acabam desenvolvendo péssimas práticas de trabalho, tanto individualmente, como dentro de suas equipes de desenvolvimento. Em outras palavras, desenvolver um código limpo é uma ótima forma de incentivar um trabalho eficiente e preciso. Uma miríade de fatores pode definir um código como limpo (MARTIN, 2019):

- Ter nomes descritivos;
- Ser legível;
- Mostrar boas escolhas;
- Ser de fácil manutenção;
- Possuir abstrações claras e linhas de controles diretas;
- Ser eficiente e claro;

Voltar ao sumário

- Ter um número reduzido de bugs;
- Executar manipulação e tratamento de erros;
- Atender aos objetivos propostos;
- Possuir testes unitários;
- Apresentar dependências limitadas;
- Ser preciso e direto; e
- Conter uma API mínima.

Como podemos ver, muitos pontos podem ajudar a definir um código limpo. Em geral, uma escrita acompanhada de boas práticas leva a um código com maior legibilidade.

Importante!

É importante ressaltar que um código de péssima legibilidade pode funcionar por completo, mas sua manutenibilidade é reduzida quando comparado a um código limpo. Ao longo dos anos, engenheiros se reuniram em grupos para pensar nas práticas que machucam a manutenibilidade em um software e tentar definir um conjunto delas e suas respectivas características.

O termo *code smells*, ou "mau cheiro" de código, foi criado por Kent Beck e popularizado por Martin Fowler. Hoje, este termo é utilizado para descrever as práticas que tornam a manutenção de código um trabalho árduo. Imagine o seguinte: você está iniciando seu dia de trabalho, você abre seu ambiente de desenvolvimento e tudo está funcionando perfeitamente. Só que algo cheira mal – não literalmente, mas existe um incômodo enquanto você trabalha no código.

Em seu livro, Fowler (2004) menciona que Kent se inspirou no termo enquanto trocava as fraldas de seu bebê. Nesta analogia, podemos dizer que existem inúmeras fragrâncias para códigos-fonte, ou, em outras palavras, dizemos que alguns códigos cheiram bem enquanto outros códigos cheiram mau. **Na prática, alguns projetos são abandonados por completo pelo único motivo de possuir um "cheiro ruim".**

Devemos concordar que a busca por um código limpo e evitar "maus cheiros" são aspectos importantes no desenvolvimento de software. Para isso, vamos estudar os principais pontos que um desenvolvedor deve estar ciente ao produzir seu código-fonte, a começar pelos nomes a serem utilizados.

Voltar ao sumário

Por vezes, podemos pensar que nomes pequenos ocupam menos espaço, são mais rápidos de digitar, poluem menos a visão do desenvolvedor e assim por diante. Inclusive, quando escrevemos algoritmos (por exemplo, utilizando pseudocódigo), aplicamos práticas de nomeação provenientes de base matemática como, por exemplo, teoria dos conjuntos, na qual um conjunto é representado por uma única letra maiúscula.

Muitos algoritmos, mesmo que simples, se implementados ao pé da letra, terão atributos, variáveis e parâmetros representados por apenas uma única letra minúscula. Considere então um cenário complexo, de difícil implementação. A leitura desse código pode ser inviável.

Nomes pequenos podem ser confusos ou até mesmo ambíguos e/ou difíceis de entender, mas fazer uso de algumas técnicas simples podem fazer toda a diferença. Utilizamos nomes em declarações de variáveis, métodos, procedimentos, funções, parâmetros, classes, pacotes etc. Ao criar qualquer um destes elementos de código, devemos pensar na responsabilidade e no propósito dele. Nosso objetivo é ser o mais coeso e descritivo possível, para que outro desenvolvedor, ou até mesmo o próprio criador do código, possa se beneficiar no futuro.

Segundo Mayer (2015), “Um nome deve poder ser lido na sua língua nativa, ou seja, nada de abreviaturas que criarão palavras inexistentes e sem nexo, não é necessário estabelecer abreviaturas de tipagem junto ao nome definido, pois o código deve ser simplificado, que não permitirão que seja encontrada com facilidade dentro do código, causando problemas nos momentos da manutenção dos programas”.

Ao nomear um elemento de código, se não nos esforçarmos agregar significado ao nome, deixamos para o futuro leitor a responsabilidade de descobrir seu significado. O leitor deve então estudar o contexto, visitando as linhas em volta do código e dentro do bloco em consideração, quando existir. Note que um tempo maior será necessário para entender ou confirmar o significado de um nome. Como um exemplo, veja o código Java abaixo:

```
public int calcular(int numero) {
    int resultado = numero;

    for(int i = numero - 1; i > 1; i--) {
        resultado = resultado * i;
    }

    return resultado;

}
```

Voltar ao sumário

Em uma classe Java, temos um método calcular que possui um único parâmetro chamado valor. Em um primeiro momento, não sabemos o que este método faz. Precisamos utilizar o contexto de código para averiguar sua funcionalidade. Dentro do bloco do método, em sua primeira linha, temos a declaração de uma variável chamada resultado. Depois de ser utilizada em um loop for, a variável resultado é retornada como resposta da computação do método. Neste momento, ainda é difícil saber o que o método calcular faz. Em uma nova investigação mais detalhada do loop, talvez você consiga entender o significado do método. Mas, em vez disso, veremos, na imagem abaixo, o mesmo método, só que utilizando com um nome mais descritivo.

```
public int fatorial(int numero) {
    int resultado = numero;

    for(int i = numero - 1; i > 1; i--) {
        resultado = resultado * i;
    }

    return resultado;

}
```

Agora, ao ler a declaração do método com o novo nome, percebemos que o método busca computar o valor fatorial de um número. Ao utilizar este novo nome, podemos investigar com mais facilidade o corpo do método para confirmar que este é realmente o caso.

Quando um método possui um nome não descritivo, a tendência é que, outros métodos implementados pelo mesmo desenvolvedor também tenham nomes não descritivos. Já tivemos trabalho com um método, imagine se este acontece com nomes de classes, atributos e funções em um projeto inteiro?

Os mais diversos elementos de código podem ser significativamente melhorados ao seguir um conjunto de simples recomendações, como (LAMOUNIER, 2021):

- Métodos: utilizar verbos, minimizando o número de caracteres tanto quanto possível;
- Funções: utilizar poucos caracteres e ser objetivo, focando o entendimento de seu papel principal;
- Variáveis: utilizar nomes significativos e posicioná-las próxima ao local principal de uso;
- Classes: utilizar substantivos simples e evitar verbos.

Estas técnicas simples podem ter um impacto significativo, mas não precisamos parar por aí. É muito comum, na engenharia de software, a recomendação de uso de padrões de

Voltar ao sumário

projeto. Estes padrões foram desenvolvidos depois de anos de aprendizado no mercado de trabalho e ajudam desenvolvedores a obter um código limpo e mais legível através da adesão de seus princípios.

Importante!

Um conjunto de princípios muito conhecido para o paradigma orientado a objetos é o SOLID, um acrônimo de acrônimos, onde cada letra se refere a um princípio distinto.

Engenheiros vem utilizando os princípios SOLID em projetos orientados a objeto para melhorar a legibilidade e facilitar a manutenção de código (MARTIN, 2019). Este conjunto de princípios se tornou tão popular que hoje é utilizado como um atestado de qualidade na escrita de código. Logo, é de nosso interesse estudar cada um dos princípios.

Single-responsibility principle

A maneira mais fácil de definir o *Single-responsibility principle*, ou princípio de responsabilidade única, em português, é que toda classe deve ter apenas uma responsabilidade. Se um desenvolvedor modificar uma classe por mais de um motivo, muito provavelmente essa classe tem mais de uma responsabilidade.

Ao utilizar este princípio, devemos exercitar o raciocínio de nos questionar qual é a responsabilidade de uma classe. Caso um desenvolvedor note que uma segunda responsabilidade está sendo considerada dentro de uma única classe, quebre-as em duas classes, transferindo a segunda responsabilidade para a nova classe. Ao nomeá-la, lembrar de seguir as práticas de código limpo.

Open-closed principle

O *open–closed principle*, ou princípio aberto-fechado, encoraja o uso de extensões de código e bloqueia as modificações de código. Utilizando as palavras do nome do princípio, podemos dizer que um código deve estar aberto para extensões e fechado para modificações. Isto significa que, quando novas funcionalidades precisam ser adicionadas ao código-fonte, o desenvolvedor deve fazer uso de extensões ao invés de modificar diretamente o código original.

Note que, se um código está funcionando, adicionar linhas implica em um risco de introduzir erros inadvertidamente. Seguindo este princípio, um desenvolvedor pode criar uma nova classe que estende a classe original (ou utilizar interfaces), introduzindo novas

Voltar ao sumário

linhas de código apenas na classe nova, que se referem à nova funcionalidade. Dessa forma, a classe original não sofre modificações e não corre riscos.

Liskov substitution principle

No terceiro princípio SOLID, temos o *Liskov substitution principle*, ou princípio da substituição de Liskov. Neste princípio, que leva o nome de sua criadora, operações que funcionam através de uma referência a uma superclasse devem funcionar através de referências de suas respectivas subclasses.

Considere as seguintes classes: Veículo, Avião e Carro. Considere ainda que a primeira é superclasse das outras duas e todas as três possuem um método chamado ligar. Com o princípio da substituição de Liskov, se uma chamada ao método ligar para uma instância de Veículo funciona, essa chamada deve funcionar para Avião e Carro também.

Interface segregation principle

No *interface segregation principle,* ou princípio de segregação de interface, classes não devem ser obrigadas a implementar métodos que não utilizam. Suponha que uma interface Veículo que tem três métodos: ligar, desligar e recolherTremDePouso. Agora, imagine que as classes Avião e Carro implementam esta interface. Note que Avião poderá implementar os três métodos sem problemas. Todavia, não faz sentido para a classe Carro implementar o método recolherTremDePouso. Seguindo este princípio, podemos criar uma interface a parte, chamada VeículoAéreo, que estende da interface Veículo, e move o método recolherTremDePouso para a nova interface. Dessa forma, a classe Carro pode implementar a interface Veículo, que não tem mais o método recolherTremDePouso, enquanto a classe Avião implementa a interface VeículoAéreo.

Dependency inversion principle

Por fim, o último princípio é chamado de *dependency inversion principle*, ou princípio da inversão de dependência. Neste princípio, um desenvolvedor deve buscar criar dependências a abstrações, e não a implementações.

Em outras palavras, quando desenvolvemos um código, devemos procurar utilizar interfaces ou classes abstratas ao invés de classes concretas, para evitar que nosso código dependa de detalhes de implementação.

Considere que uma classe Pessoa utiliza uma classe Carro dentro de seu código. Dessa forma, podemos dizer que a classe Pessoa depende da implementação da classe Carro. Com isso, é possível que modificações realizadas na classe Carro impactem a classe Pessoa. Uma forma de aderir ao princípio de inversão de dependência é modificar a classe Pessoa para fazer uso da interface Veículo ao invés da classe Carro. Dessa forma, a classe Carro pode ser modificada, mas Veículo e Pessoa não sofrem modificações.

Voltar ao sumário

Mesmo que os princípios SOLID se apliquem apenas ao software do paradigma orientado a objetos, eles são parte integrante de importantes metodologias, como a metodologia ágil de desenvolvimento de software. Além disso, muito padrões de projeto implementam os seus princípios. Dependendo do software que está sendo produzido, um grande número de padrões pode ser aplicado, mas não estudaremos todos eles.

É importante ressaltar que a legibilidade do código deve ser uma prioridade dentre as tarefas de um desenvolvedor. Nos próximos circuitos deste percurso de aprendizagem, vamos aprender a importância de testes e aprenderemos a aplicar as principais técnicas de fatoração de código.

2. Planejamento e escrita de testes unitários

Dentre essenciais critérios para avaliarmos a qualidade de um código-fonte, podemos utilizar a quantidade de erros (ou *bugs*) existentes. Neste caso, queremos diminuir a quantidade de *bugs*, ou, em um cenário ideal, gostaríamos de um software livre de *bugs*. Agora, como sabemos se um software possui bugs?

Mesmo de forma não intencional, é comum introduzirmos erros enquanto programamos. Suponha que você é o responsável por verificar se um software possui *bugs*. Como você faria para verificar se este software realmente possui bugs? E se este for mesmo o caso, onde encontram-se esses bugs?

Segundo Gonçalvez (2019), “A verificação dos softwares ocorre por meio de testes realizados antes da entrega do produto aos consumidores. O teste é a última etapa do desenvolvimento de um software. Como você deve imaginar, é uma fase de suma importância. Por meio dela, é possível verificar e resolver problemas e bugs — caso ambos sejam encontrados. O teste de software tem como objetivo encontrar erros e produzir softwares com a maior qualidade possível, tornando-os seguros e confiáveis para utilização”.

Voltar ao sumário

Importante!

Uma boa prática é assumir que bugs sempre existem e, diante dessa informação, precisamos realizar a importante tarefa de planejamento de testes. Estes testes visam reduzir a quantidade de erros presentes em um programa e melhorar sua qualidade. Todavia, para realizar um bom planejamento de testes, precisamos entender quais tipos de testes existem e quando utilizá-los.

Vimos no percurso de aprendizagem anterior que um desenvolvedor com um ambiente de desenvolvimento de software está pronto para programar. Entretanto, existem várias etapas para que o código produzido chegue ao cliente, sendo o primeiro deles a identificação das necessidades de um cliente. Uma vez que entendemos suas necessidades, precisamos documentá-las para que exista uma comunicação efetiva entre cliente e equipe de desenvolvimento. Subsequentemente, estes documentos se tornam insumos na atividade de análise e projeto de um sistema de software. Isto é, a partir dos requisitos de um cliente, uma estrutura de projeto de software é construída, definindo seus principais componentes e suas interações.

Em seguida, a partir do que foi desenhado anteriormente, um desenvolvedor pode iniciar o processo de construção de código. Neste momento, um desenvolvedor pode, de forma inadvertida, introduzir um erro. Entretanto, erros podem ter sido introduzidos no momento de análise ou de projeto do sistema. Até mesmo durante o processo de definição de necessidades do cliente, pode ocorrer uma falha na comunicação por parte do cliente, que não sabe exatamente o que quer, ou por parte da equipe, que não compreende os requisitos do cliente. De toda forma, erros podem ser introduzidos em qualquer etapa da construção de um software.

EXEMPLO: Digamos que um gerente de projeto é responsável pelo desenvolvimento de um plano de testes. Podemos dizer que o objetivo deste gerente é elaborar um plano de testes apropriado ao projeto. Em outras palavras, um plano simples demais pode ser danoso, bem como um plano complicado demais.

Note que, em um plano de testes muito casual, erros podem passar despercebidos e, em um plano de testes muito formal, esforços desnecessários podem ser empregados sem que haja um retorno proporcional. Um planejamento de testes eficiente leva o gerente a uma construção ordenada do software e à descoberta de erros em cada estágio do processo de construção (PRESSMAN, 2021).

Voltar ao sumário

Como erros podem ser introduzidos em qualquer etapa da construção de um software, é importante entendermos como abordar o código produzido. Veremos em breve que um código pode ser dividido em unidades e, consequentemente, podemos testá-las a fim de verificar seu correto funcionamento enquanto procuramos por erros. Damos a esses testes o nome de testes unitários. Por ora, vamos pensar em um software com base em seus componentes. No caso do Google Mail, vamos tomar como exemplo o componente que realiza o envio de novos e-mails. Como podemos planejar os testes desse componente para verificar seu funcionamento correto?

Importante!

Um planejamento de testes eficiente deve levar em consideração múltiplos cenários que podem ocorrer no domínio de negócio. Isto é, cenários que buscam ser próximos à sequência de passos que o cliente executaria. Estes passos podem ser documentos no que chamamos de casos de teste.

Um caso de teste é um documento que contém os passos a serem executados para verificar o funcionamento de um componente de software. Além disso, este documento contém uma descrição dos resultados esperados durante cada passo de execução. Um teste bem-sucedido é indicado pelas condições definidas no caso de teste. Se todas as condições são atendidas e o resultado obtido corresponde com o resultado esperado, dizemos que o componente está funcionando corretamente. Caso contrário, documenta-se a diferença entre o resultado obtido e o resultado esperado, para que esta diferença seja utilizada a fim de encontrar a origem da discrepância, isto é, a localização de um erro ou bug.

No exemplo do Google Mail, note que clientes distintos podem enviar e-mails de maneiras distintas. Dessa forma, o planejamento de testes deve ser flexível o suficiente para atender diferentes tipos de partes interessadas. Um bom planejamento de testes não atende apenas parte de seus usuários. Um bom planejamento de testes busca atender à maior quantidade de tipos distintos dentre seus usuários. Ainda nesse exemplo, é possível considerar que o recebimento de um e-mail seja uma parte importante do teste de envio de um e-mail. Em outras palavras, um e-mail só é enviado corretamente se este chega com sucesso ao destinatário apropriado.

O componente de recebimento de e-mails eventualmente será testado e assim por diante, até que todos os componentes do software de e-mail sejam testados. Outras estratégias de testes podem ser empregadas em conjunto com testes de componente, como por

Voltar ao sumário

exemplo testes de integração (veremos mais detalhes sobre esta tarefa mais para a frente no componente curricular). As estratégias de testes já estão bem definidas na área de engenharia de software e a maior parte delas segue um mesmo modelo de teste que incluem as seguintes características genéricas (PRESSMAN, 2021):

- Inspeções eficazes e frequentes fazem um teste ser mais eficaz. Isto é, muitos erros serão eliminados durante as inspeções, ou seja, antes do começo do teste.
- Um plano de testes inicia no nível de componente e progride em direção à integração do sistema de software como um todo.
- O desenvolvedor do software e um grupo de teste independente são responsáveis pela realização do teste.
- A depuração deve ser realizada, estando associada a alguma estratégia de teste.

Um modelo de teste eficaz espera que os desenvolvedores façam uso do depurador em suas estratégias de teste. No percurso de aprendizagem anterior, vimos que a depuração na IDE Eclipse, realizada através do modo Debug, é representada pelo símbolo de um inseto (em inglês, “bug”). Podemos dizer que o utilitário mais usado para corrigir erros em código é o *debugger*, ou depurador.

Importante!

Quando não sabemos o que há de errado com nosso código, podemos executá-lo em modo *debug* com o auxílio de pontos de depuração, ou *breakpoints*.

A maioria das IDEs utilizam o mesmo símbolo para depuração e a mesma forma de adicionar breakpoints (clicar duas vezes ao lado esquerdo da linha). Em geral, uma execução em modo depuração será pausada ao chegar em uma linha com breakpoint, para que o desenvolvedor veja diversos detalhes antes que a linha seja executada, na esperança de corrigir um possível erro na linha. No próximo circuito de aprendizagem, veremos como realizar a manipulação e tratamento de erros quando estes são identificados.

De um modo geral, um bom planejamento de testes se preocupa em verificar se todos os componentes estão funcionando apropriadamente, de forma individual e em conjunto. Uma vez que o sistema de software tenha passado por uma de aprimoramento, este será validado pelo cliente e pode entrar em novas iterações com mais fases de aprimoramento. Neste momento, é importante diferenciar verificação e validação.

Voltar ao sumário

Segundo Pressman (2021),

> "Verificação refere-se ao conjunto de tarefas que garantem que o software implemente corretamente uma função específica. Validação refere-se ao conjunto de tarefas que asseguram que o software foi criado e pode ser rastreado segundo os requisitos do cliente".

Em outras palavras, o processo de verificação se preocupa em averiguar se uma funcionalidade foi implementada em conformidade com uma função previamente definida. Um exemplo seria verificar se uma função produz as saídas esperadas diante de um respectivo conjunto arbitrário de entradas, como verificar que um programa de ordenação que imprime "1, 2, 3" ao receber como entrada "3, 2, 1".

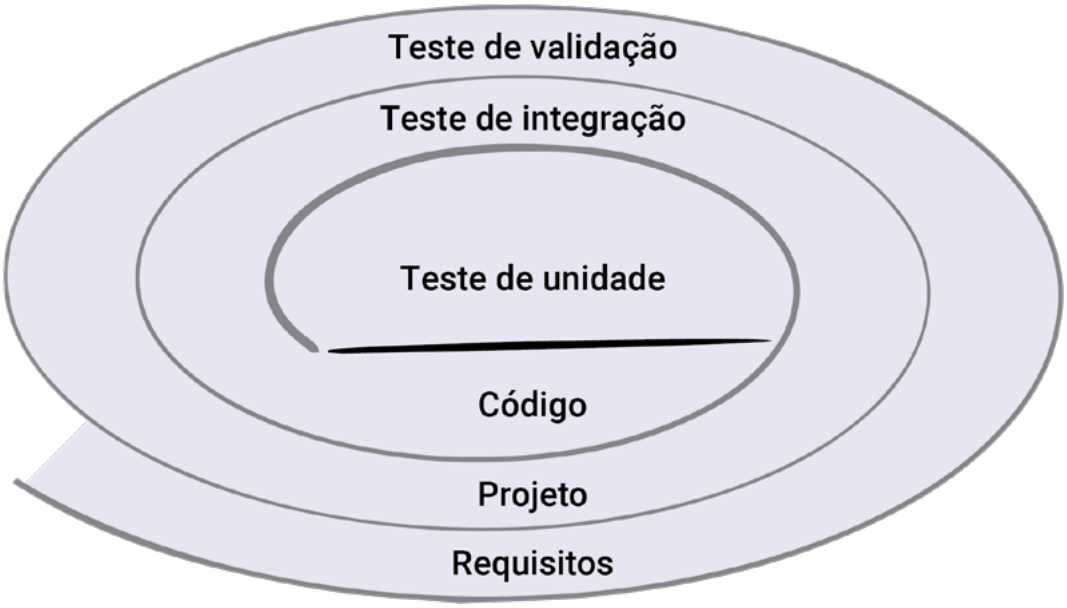
O processo de validação se preocupa em averiguar se uma funcionalidade atende aos requisitos do cliente. Um exemplo seria pedir que um cliente valide que uma função realmente atende uma (ou mais) de suas necessidades. Neste caso, nossa atenção não está no fato de uma solução ter sido implementada de forma correta, mas sim se a solução está relacionada a uma necessidade do cliente.

Imagine que o dono de um restaurante contrata uma empresa de desenvolvimento de software para implementar algumas funcionalidades a fim de melhorar a eficiência de algumas tarefas do dia a dia. Com um mês de trabalho, a empresa constrói duas funções: *Realizar pedido* e *Mandar correspondência*. Em um processo de verificação, a equipe averigua que ambas as funções foram implementadas corretamente. Entretanto, no processo de validação, o cliente valida a função *Realizar pedido*, mas indica que a função *Mandar correspondência* não ajuda a resolver nenhum de seus problemas.

Note que estratégias de testes podem estar presentes durante todo o processo de desenvolvimento de um software e de diversas formas. Faz-se necessário então entender quais tipos de testes são realizados ao longo de um projeto de construção de software. Para isso, vamos avaliar um exemplo de estratégia de teste, conforme mostra a figura abaixo.

Voltar ao sumário

Figura 1 - estratégia de teste



Fonte: Adaptado de PRESSMAN (2021).

Teste de unidade

Na figura acima, vemos que uma estratégia de testes válida pode ser representada por uma espiral. Em cada loop da espiral, podemos notar a presença de um tipo de teste e um artefato. É possível notar também que existe nessa estratégia de testes uma noção de ordem, na qual o loop mais interno é executado primeiramente e assim por diante, até que a espiral seja executada por completo. Para entendermos em detalhes esta estratégia, vamos investigar cada loop individualmente.

O primeiro tipo de teste executado neste cenário é o Teste de unidade ou Teste unitário. Este teste é acompanhado do artefato denominado código, que nada mais é do que o código-fonte a ser testado. Um teste unitário recebe esse nome pois realiza uma verificação de uma unidade de código por vez. Dessa forma, é importante definirmos o que é uma unidade de código. Segundo Delamaro (2016), o teste unitário:

> "[...] tem como foco as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. Nesse contexto, espera-se que sejam identificados erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas ou simples erros de programação. Como cada unidade é testada separadamente, o teste de unidade pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio desenvolvedor, sem a necessidade de dispor-se do sistema totalmente finalizado".

Voltar ao sumário

Em geral, criamos uma classe de testes específica para cada classe presente no código-fonte. Nesta classe de testes, será criado um método de teste para cada método respectivo presente na classe original, a fim de verificar o correto funcionamento da classe original e de seus métodos como um todo. No próximo circuito, veremos um exemplo concreto dessa abordagem de testes.

Teste de integração

Uma vez que as unidades de código estejam verificadas, ou seja, sabemos que elas funcionam independentemente, faz-se necessária a verificação se elas funcionam de uma forma conjunta. Dessa forma, organizamos em um repositório local ou remoto uma versão que contém todas as unidades de código. Esta primeira versão completa tende a associar as unidades, que antes eram independentes, em múltiplos pontos. Em outras palavras, uma unidade de código potencialmente faz uso de uma ou mais unidades de código distintas, o que pode impactar em seu funcionamento.

Considere um cenário em que uma classe Biblioteca possui testes unitários verificados. Da mesma forma, uma classe Livro funciona apropriadamente e não existem problemas em seus testes unitários. Todavia, quando combinadas em uma única versão, essas classes podem influenciar o funcionamento uma da outra. Note que os testes unitários de ambas não encontram um problema, mas é possível que um problema venha a ser aparente depois que essas duas classes são integradas. Para isso, realizamos um teste de integração.

Em um teste de integração, não consideramos um código como artefato, mas sim um projeto como um todo. **O objetivo principal de um teste de integração é desvendar se existem erros ao executar um projeto por completo.** Ou seja, quando todas as unidades de código são executadas em um único conjunto e de forma integrada. Nesta estratégia de testes em espiral, o teste de integração só pode ser feito depois que todos os testes unitários são realizados. Por fim, após a execução do teste de integração, executamos o último teste – validação.

Teste de validação

Conforme mencionado anteriormente, o processo de validação se preocupa em averiguar se uma funcionalidade atende aos requisitos do cliente. Ou seja, nossa atenção não está no fato de uma solução ter sido implementada de forma correta, mas sim se a solução está relacionada a uma necessidade do cliente. Consequentemente, o artefato que utilizamos para realizar um teste de validação são os requisitos de um projeto. Estes requisitos podem ser funcionais ou não funcionais, mas ambos podem ser rastreados a uma necessidade do cliente e ao domínio de problemas.

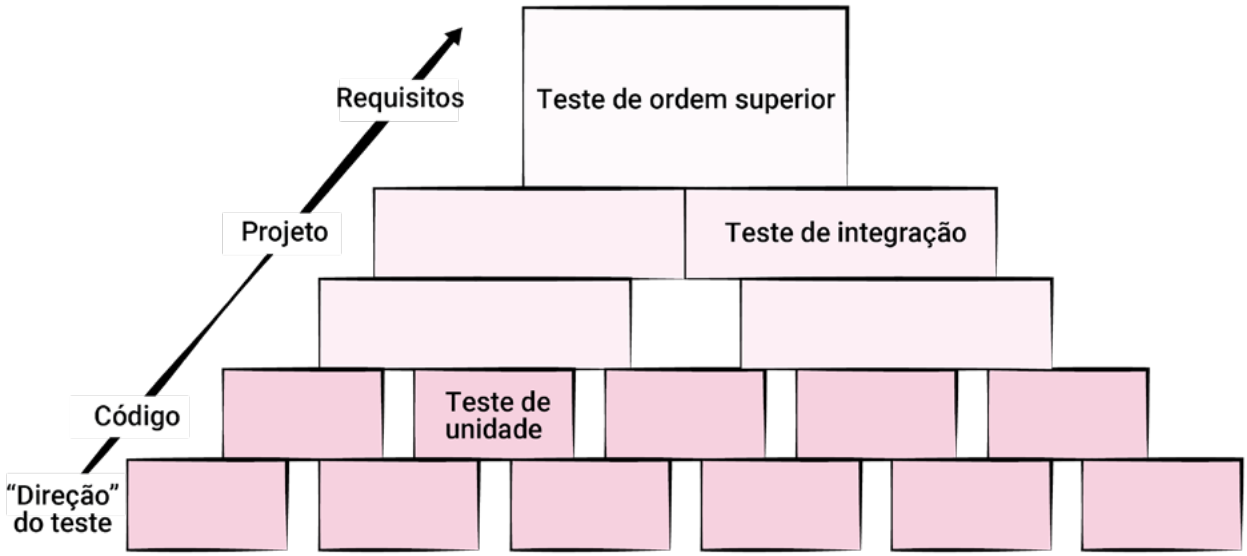
Voltar ao sumário

Como um cliente tende a desconhecer os detalhes técnicos de uma solução em desenvolvimento, é comum que este cliente não entenda e/ou não saiba da existência dos componentes da solução e suas interações. O maior interesse do cliente está na utilização da solução. Na estratégia de testes em espiral, um teste de validação só acontece depois que os testes unitários são executados e os testes de integração são executados.

No exemplo do restaurante que contrata uma empresa para o desenvolvimento de um software, podemos dizer que o cliente, dono do restaurante, está interessado em responder dois principais questionamentos: o que foi implementado resolve ou alivia meus problemas? O que foi implementado atende a todas as minhas necessidades? Caso a resposta dessas perguntas seja positiva, o teste de validação revela que o projeto foi verificado e validado.

Existem outras estratégias de testes além da estratégia em espiral, mas a maior parte delas utilizar estes três testes de alguma maneira. Por exemplo, é possível considerar uma estratégia de testes de modo procedimental, representado, por exemplo, por quatro etapas (PRESSMAN, 2021). A figura abaixo mostra esta estratégia de testes.

Figura 2 - Estratégia de testes



Fonte: Pressman, 2021.

Voltar ao sumário

3. Manipulação e tratamento de erros

Acabamos de aprender a importância de um bom planejamento de testes. Uma estratégia de testes bem executada pode ajudar um desenvolvedor a achar erros dentro de seu código. Aplicar uma correção ou realizar ajustes pode ser fácil quando sabemos que existe um erro, mas procurar por erros pode ser difícil. **O que podemos fazer para encontrá-los? E se não encontrarmos, quando devemos parar de procurar? Se sim, nosso código está livre de erros?**

Nosso código sempre está passível a erros, então podemos responder a última pergunta com certa facilidade, pois devemos sempre considerar que existem erros em nossos códigos. Em algum momento, devemos parar de procurar por erros, pois o custo não irá compensar o ganho em qualidade, que diminui ao longo dos testes. Todavia, devemos tentar o nosso melhor para garantir que erros "escondidos" sejam identificados.

Testes estão presentes no desenvolvimento de software em vários momentos e de diversas formas. Existem erros de diversos tipos: erro de digitação em uma classe, erro de compatibilidade entre componentes da sua pilha de software, erro de integração entre o software e pacotes externos e assim por diante. Vamos começar pelo erro mais singular, uma unidade. Considere um software que possua algumas classes, incluindo uma classe livros, representada no exemplo a seguir:

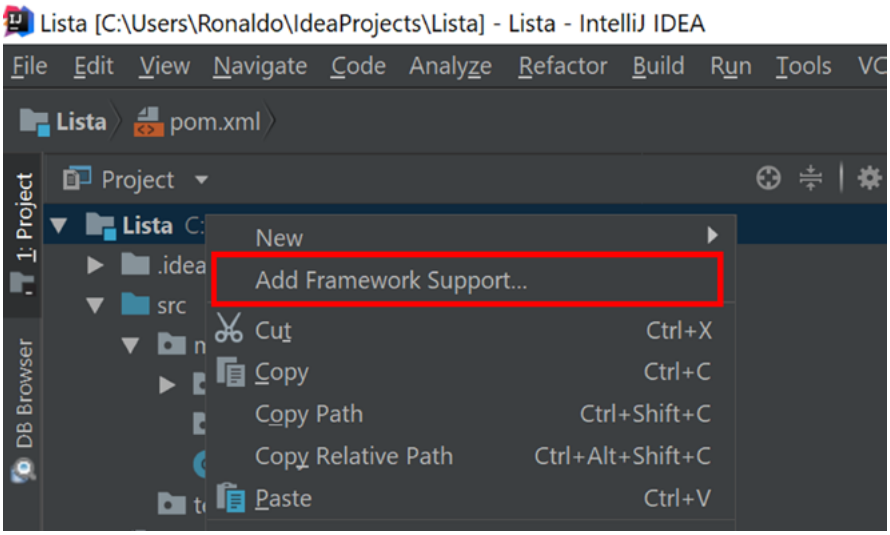
Livro.java

```java
package entidades;

public class Livro {
    private String titulo;
    private String nomeAutor;
    private int anoPublicacao;

    public Livro(String titulo, String nomeAutor, int anoPublicacao) {
        this.titulo = titulo;
        this.nomeAutor = nomeAutor;
        this.anoPublicacao = anoPublicacao;
    }

    public String getTitulo() {
        return nomeAutor;
    }

    public String getNomeAutor() {
        return titulo;
    }

    public int getAnoPublicacao() {
        return anoPublicacao;
    }

}
```

Voltar ao sumário

Conforme definido no circuito anterior, podemos considerar métodos ou classes como unidades de código a serem testadas, a fim de localizar erros. Geralmente, refere-se ao teste de uma classe específica e seus métodos. Neste exemplo, queremos testar a classe Livro. Para acompanhar o exemplo prático, você pode instalar o IntelliJ (uma instrução de instalação pode ser encontrada no próximo circuito) ou replicar os passos equivalentes em outra IDE.

No Intellij, crie um projeto Java e copie o conteúdo da classe Livro. Após isso, acesse as opções do projeto e escolha a opção *Add Framework Support...* e inclua o framework *Maven,* que fará o controle de dependências. Veja o exemplo:

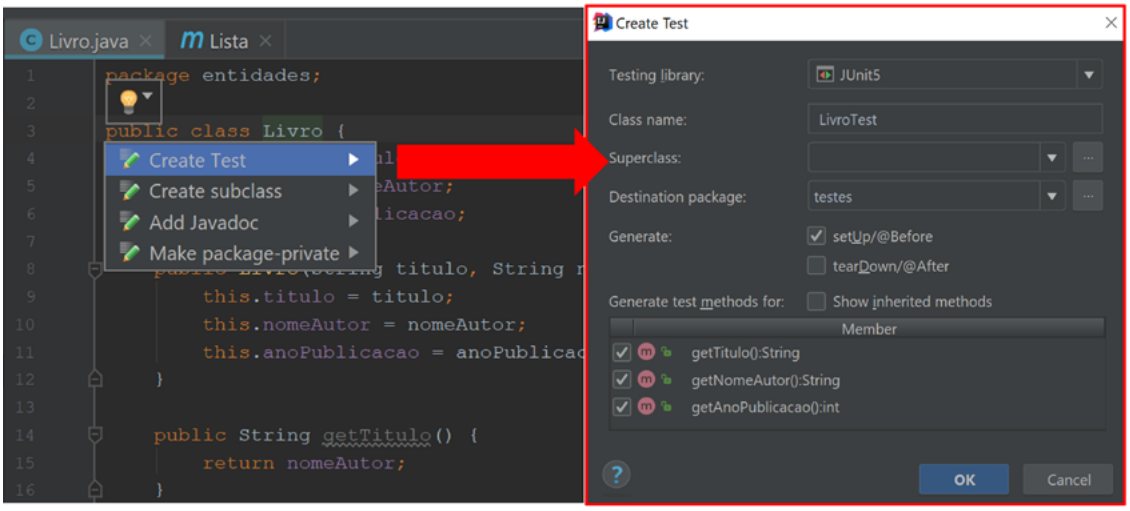


Vamos adicionar ao arquivo *pom.xml*, automaticamente gerado em nosso projeto, a dependência do JUnit. Dentre os frameworks mais populares para a realização de testes em Java, JUnit é fácil de utilizar e contém todos os elementos para a realização dos mais variados testes unitários (GONÇALVEZ, 2019). Copie o código XML abaixo ao final do arquivo *pom.xml*, logo antes da linha que contém </project> (marcação do final), Veja o exemplo:

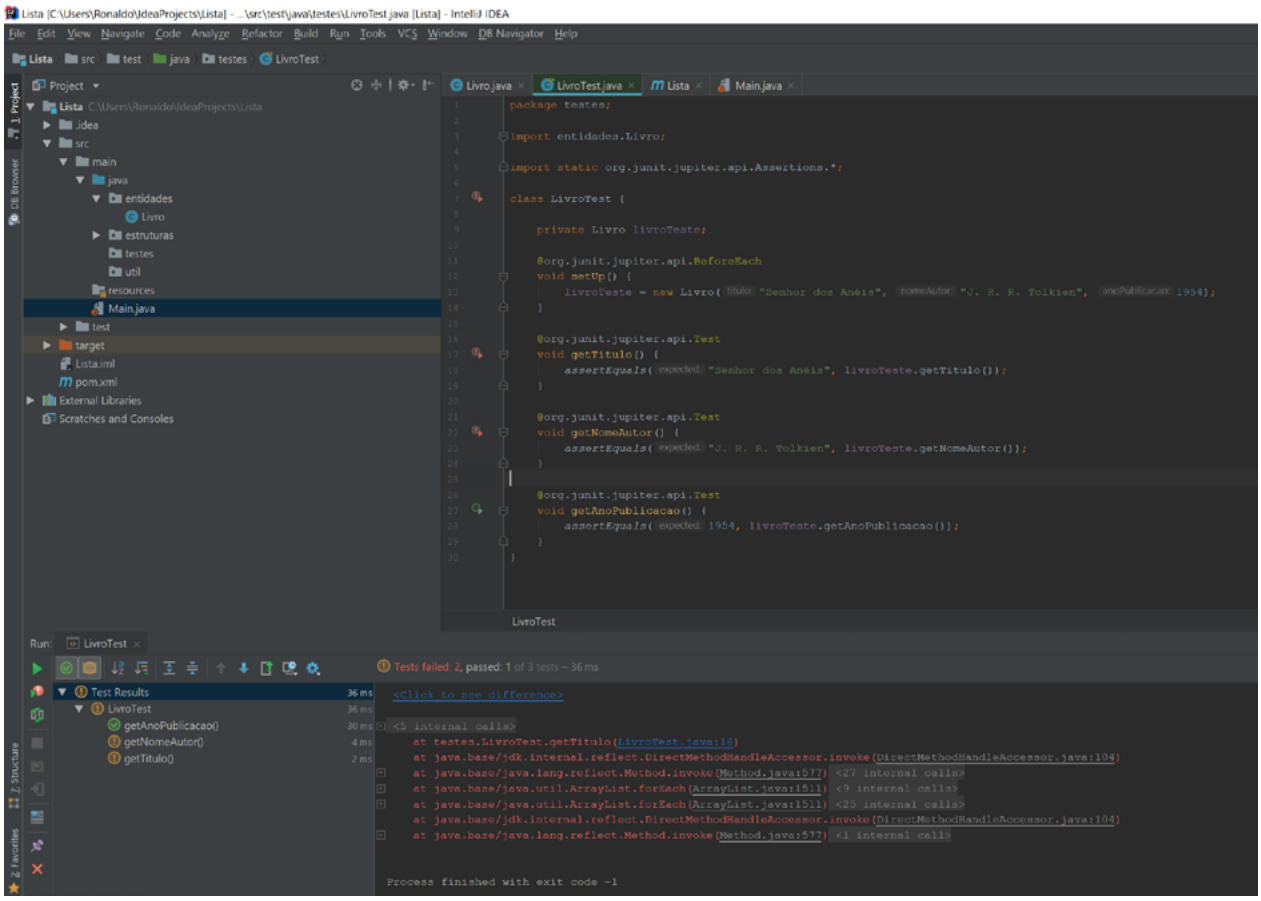
```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>RELEASE</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Voltar ao sumário

Em seguida, abra a classe Livro, clique com o cursor do mouse no nome da classe e um ícone de lâmpada irá aparecer. Clique em criar teste e preencha o formulário de criação de teste unitário tal qual exibido no exemplo abaixo:



É possível que você precise clicar para instalar o JUnit, caso seja a primeira vez que você cria um teste unitário. De qualquer modo, depois deste processo, nosso primeiro teste unitário foi criado com sucesso. Vamos conferir o que está acontecendo. Copie o código do teste abaixo e veja qual é o resultado. Para executar a classe de teste, clique no balão de executar teste na linha 7, veja no exemplo:



Voltar ao sumário

Importante!

O método chave para a execução de um teste unitário é o *Assert*. No português, significa afirmar, este método possui várias versões. A utilizada aqui é o *assertEquals*, que recebe dois parâmetros: o primeiro é o valor esperado, criado no setUp do teste; o segundo é a chamada do método que queremos testar.

Se o método foi implementado corretamente, ele irá retornar o mesmo nome que consta no primeiro parâmetro, e o *assertEquals* retornará que eles são iguais, indicando um teste de sucesso. Caso o método retorne qualquer valor diferente do esperado, o *assertEquals* indicará que o teste falhou. Considerando os resultados apresentados, tente corrigir os métodos que estão falhando nos testes unitários e execute um teste no qual todos os métodos da classe Livro obtêm sucesso. Outras variações do assert incluem (mas não está limitado a):

- assertTrue: afirma se a condição booleana passada como parâmetro é verdadeira;
- assertFalse: afirma se a condição booleana passada como parâmetro é falsa;
- assertNull: afirma se o objeto passado como parâmetro é nulo;
- assertNotNull: afirma se o objeto passado como parâmetro não é nulo;
- assertEquals: afirma se os objetos passados como parâmetro são iguais;
- assertNotEquals: afirma se os objetos passados como parâmetro não são iguais;
- assertSame: afirma se os objetos passados como parâmetro referem ao mesmo objeto;
- assertNotSame: afirma se os objetos passados como parâmetro não referem ao mesmo objeto.

Muitos não gostam de admitir, mas todo software pode incluir bugs (geralmente este é o caso, para não dizer sempre). Agora, bugs são erros? A origem da palavra bug sinaliza a existência de um agente que desvia uma tarefa de sua execução esperada. No circuito anterior, falamos de planejamento de testes com a preocupação de ter um código correto. Erros e exceções são sinônimos?

Segundo Schildt (2015),

> "Uma exceção é um erro que ocorre no tempo de execução. Usando o subsistema Java de tratamento de exceções, você pode tratar erros de tempo de execução de maneira estruturada e controlada. Embora a maioria das linguagens de programação moderna ofereça algum tipo de tratamento de exceções, o suporte Java é fácil de usar e flexível".

Voltar ao sumário

Neste circuito, vamos visitar as três principais classificações de erro em um software, também chamadas de exceções, e vamos aprender a fazer o tratamento apropriado desses erros.

Erros padrões

Quando o programador projeta o código para acessar um método ou atributo de um objeto, mas, no momento de execução, este objeto não é inicializado ou por algum outro motivo acaba com o valor *null* (ou nulo) atribuído a ele. Tentar acessar um atributo ou método de um valor nulo gera um dos erros mais comuns que encontramos, o *null pointer exception*.

Outro erro muito comum é o *array out of bounds*, quando o programador implementa uma lógica para acessar posições de um vetor (ou outro tipo abstrato de coleção), mas o índice desejado em tempo de execução não está se referindo a algo que esteja contido no vetor.

EXEMPLO: acessar a posição 10 em um vetor que possui 5 posições, de 0 a 4.

Chamamos estes erros de erros padrões e eles são caracterizados por terem uma origem na implementação. Isto é, são erros gerados por código indevido. Note que, se não checarmos se a variável "b" possui o valor 0, isso pode gerar um erro padrão chamado *division by zero*. Veja o exemplo.

```
private double calcular(double a, double b) {
    return a/b;
}
```

Erros externos

Alguns erros fogem ao nosso controle, pois não são gerados por código indevido. Um software gera um erro pois houve uma falha no gerenciamento de memória por parte da Java Virtual Machine (JVM). A JVM é um dos componentes que criou destaque para a linguagem Java, por ajudar no controle de ponteiros e tornar Java uma linguagem multiplataforma, entre outros. Podemos chamar estes erros de erros externos ou erros gerados por componentes fora do nosso controle.

Voltar ao sumário

Diferente dos erros padrões, estes são mais difíceis para se planejar, mas veremos que a manipulação e o tratamento deles podem ser (e acabam sendo) iguais aos dos erros padrões, nos quais faremos uso de estruturas apropriadas para que um software possa se comportar conforme desejado, mesmo que haja um erro ou exceção.

Erros lançados

Antes de encerrarmos as classificações, vale mencionar também a classe de erros lançados. Por vezes, a execução normal de um código não geraria um erro padrão.

Todavia, pode ser do interesse de um desenvolvedor que um erro seja lançado a depender do estado de um objeto ou outro elemento do código.

EXEMPLO: um software pede que o usuário informe sua data de nascimento. Este usuário em consideração possui data de nascimento 01/01/1980, mas informa como seu nascimento a data 12/12/2090. Por se tratar de uma data no futuro, um desenvolvedor pode verificar que é impossível esta data ser uma data apropriada para o nascimento do usuário, e lança um erro, muito semelhante a um erro padrão, mas este é um erro fabricado chamado *invalid date*. Estes erros serão manipulados e tratados da mesma forma que erros padrões.

Agora que sabemos que todos esses erros são manipulados e tratados da mesma forma, como podemos realizar essas tarefas? Um comportamento padrão seria fazer uso do console, no qual essas informações de erro são apresentadas. Entretanto, pensando em uma pilha de execução (ou Stack Trace), um erro irá desencadear vários outros erros, o que, por vezes, torna a leitura do console uma atividade desafiadora. Dessa forma, qual é a melhor forma de tratar um erro?

Importante!

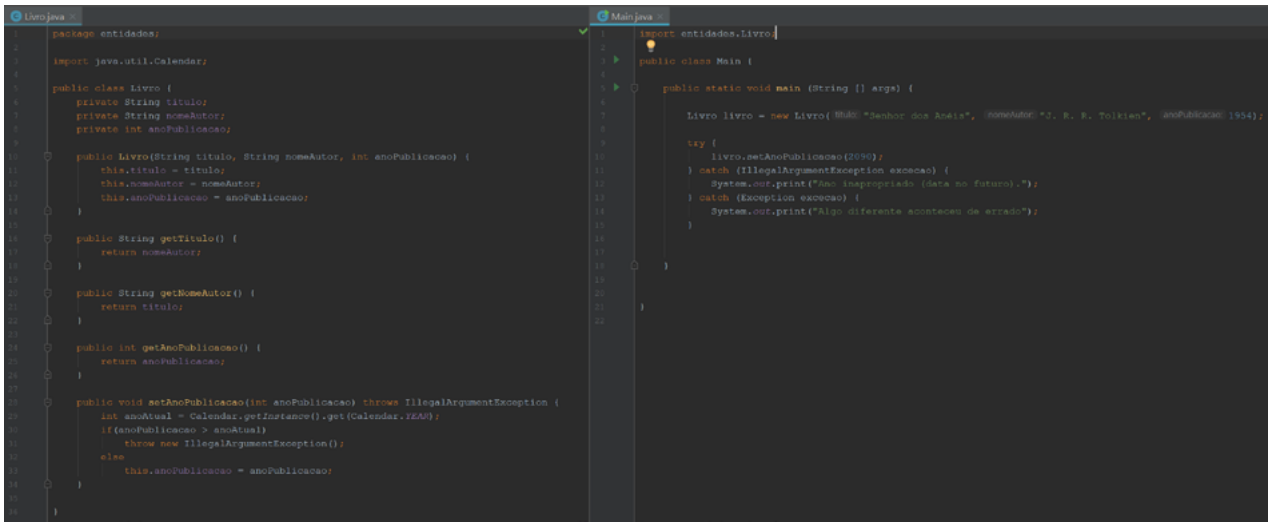
Podemos fazer uso de recursos como *System.out.print* ou equivalente para construir uma saída de console formatada apropriadamente, mas como capturamos a execução antes que o comportamento padrão mostre a *Stack Trace*? Um desenvolvedor deve entender como funciona o tratamento de exceções, senão não pode ser plenamente bem-sucedido (SCHILDT, 2015).

Voltar ao sumário

Uma forma de tratar exceções em Java é através da estrutura *try-catch*. O primeiro passo é saber onde esse erro pode acontecer, algo para o qual planejamos nossos testes no circuito de aprendizado anterior. Sabendo onde uma exceção pode acontecer, podemos abarcar a respectiva porção de código em um bloco *try*, e, caso venha a ocorrer um erro, será acionado o tratamento pela cláusula *catch*. Veja o exemplo abaixo:

```
int vetor = new int[4];

try {
    vetor[10] = 1000;
} catch (ArrayOutOfBoundsException excecao) {
    System.out.print("Este erro seria um
ArrayOutOfBoundsException, pois:")
    System.out.print("tentamos acessar a posição 10, mas temos
apenas 4 posições, de 0 a 3.")
}
```

Note que apenas erros do tipo *array out of bounds* podem ser capturados no exemplo acima. Todavia, podemos usar mais de uma cláusula *catch* para um único bloco *try*. Vamos revisitar nosso exemplo da classe Livro. Podemos fabricar a ocorrência de um erro em Java através do comando *throw*. Um *illegal argument exception* será fabricado se a data informada no parâmetro for no futuro. Note que temos duas cláusulas *catch* para a chamada do método setAnoPublicacao. Caso algum outro erro aconteça, a cláusula *Exception* irá capturar, pois é a mais elevada na hierarquia. Veja o exemplo:



Manipulação e tratamento de exceções se tornam essenciais para verificar se um código está executando apropriadamente. No específico caso de Java, cada exceção é representada por uma classe do tipo *Throwable*. Isto é, toda exceção em Java estende a classe *Throwable*. A tabela abaixo mostra os principais erros e seus significados.

Voltar ao sumário

Tabela 1 - Erros e seus significados

Exceção	Significado
ArithmeticException	Erro aritmético, como a divisão de inteiros por zero.
ArrayIndexOutOfBoundsException	O índice do array está fora dos limites.
ArrayStoreException	Atribuição de um tipo incompatível a um elemento do array.
ClassCastException	Coerção inválida.
EnumConstantNotPresentException	É feita a tentativa de usar um valor de enumeração não definido.
IllegalArgumentException	Argumento inválido usado para chamar um método.
IlegalMonitorStateException	Operação de monitor inválida, como esperar em uma thread não bloqueada.
IllegalStateException	O ambiente ou o aplicativo está no estado incorreto.
IllegalThreadStateException	Operação solicitada não compatível com o estado atual da thread.
IndexOutOfBoundsException	Um índice de algum tipo está fora dos limites.
NegativeArraySizeException	Array criado com um tamanho negativo.
NullPointerException	Uso inválido de uma referência nula.
NumberFormatException	Conversão inválida de um string para um formato numérico.
SecurityException	Tentativa de violar a segurança.
StringIndexOutOfBoundsException	Tentativa de indexar fora dos limites de um string.
TypeNotPresentException	Tipo não encontrado.
UnsupportedOperationException	Uma operação sem suporte foi encontrada.

Exceção	Significado
ClassNotFoundException	Classe não encontrada.
CloneNotSupportedException	Tentativa de clonar um objeto que não implementa a interface **Cloneable**.
IllegalAccessException	O acesso a uma classe é negado.
InstatiationException	Tentativa de criar um objeto de uma interface ou classe abstrata.
InterrupledException	Uma thread foi interrompida por outra thread.
NoSuchFieldException	Um campo solicitado não existe.
NoSuchMethodException	Um método solicitado não existe.
ReflectiveOperationException	Superclasse de exceções relacionadas à reflexão.

Fonte: Adaptado de SCHILDT, 2015.

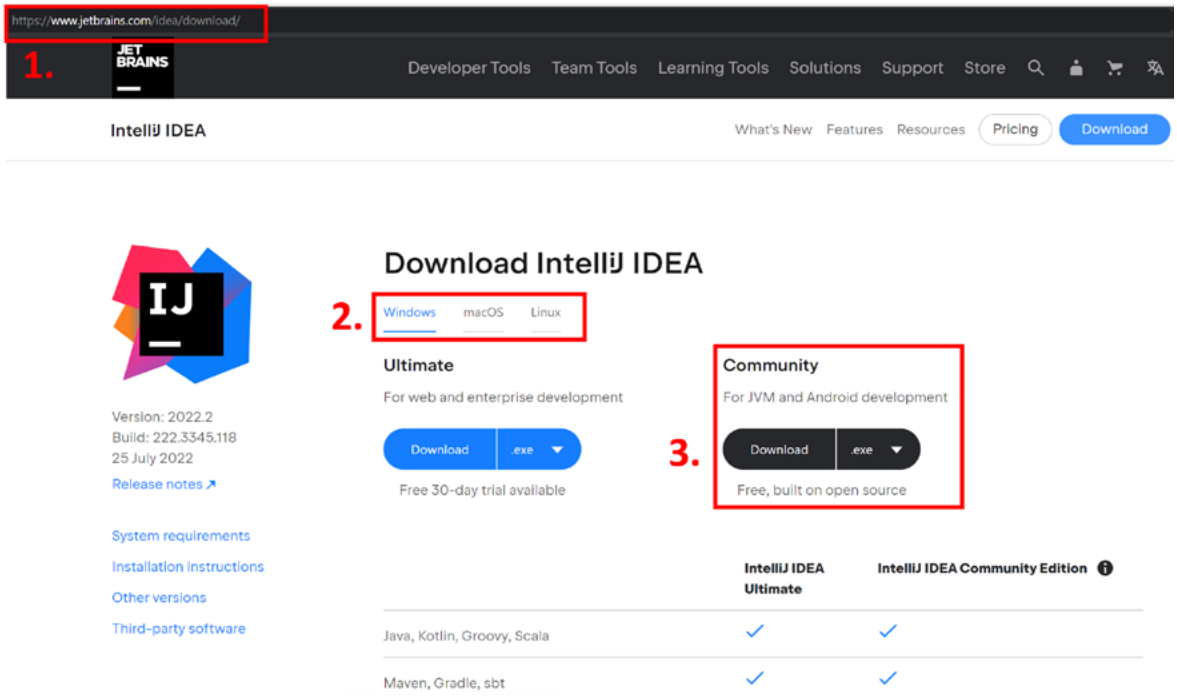
Voltar ao sumário

4. Técnicas de fatoração: Extração e movimentação de métodos, renomeação de variáveis, classes e métodos e extração de classes

No percurso de aprendizado anterior, realizamos a instalação e a configuração do nosso ambiente de desenvolvimento Eclipse. Falamos sobre os pontos principais que usamos como critério na escolha de um ambiente de desenvolvimento e chegamos a discutir sobre as ferramentas utilitárias que usamos para a produção de código-fonte, o que nos ajuda no nosso dia a dia como desenvolvedor. Através de exemplos de aplicativos que estamos habituados a utilizar, como Instagram, WhatsApp, Facebook etc., chegamos a conversar sobre as suas respectivas versões para as plataformas mais populares: desktop, web, mobile e nuvem.

Como existem várias plataformas e os próprios ambientes de desenvolvimento que vamos utilizar são diversos, queremos visitar as principais operações disponíveis em um ambiente sem se limitar a apenas uma IDE. Fazemos isso para evitar se prender a uma IDE em específico, pois, atualmente, é recomendado que desenvolvedores sejam familiarizados com múltiplos ambientes de desenvolvimento. Quando temos o poder de escolha e optamos em utilizar sempre a mesma IDE, isto pode acabar se tornando um vício - comportamento que devemos impedir que aconteça. Esta prática está alinhada também com o objetivo de conquistar um código limpo, livre de “maus cheiros”.

Figura 3 - Baixar IntelliJ IDEA



Fonte: IntelliJ IDEA, 2022.

Voltar ao sumário

Neste circuito, daremos início ao uso do IntelliJ IDEA. Para acompanhar o material de estudos, faça a instalação do ambiente através do site oficial. A imagem acima mostra como baixar o IntelliJ e o passo a passo a seguir deve ser seguido para garantir que você adquira a versão correta do ambiente de desenvolvimento:

1. Acesse o site oficial: https://www.jetbrains.com/idea/download/
2. Selecione sistema operacional apropriado. No meu caso, Windows.
3. Clique para fazer o download da versão *Community*.

Atualmente, a versão *Ultimate* é paga, enquanto a versão *Community* é gratuita para a produção de código *open source*. Neste mesmo endereço eletrônico, existe uma tabela com as diferenças entre estas versões, mas utilizaremos o IntelliJ IDEA Community por ser mais acessível. Caso você tenha alguma versão diferente do IntelliJ já instalada, é possível utilizá-la sem que haja problemas.

Quando falamos em desenvolvimento, a primeira atividade que vem em mente é programação. Quando pensamos em programação, pensamos em construir código. Vários desenvolvedores, principalmente os que estão iniciando, focam a construção de código voltava para o seu funcionamento. Todavia, ainda que seja essencial que um programa funcione corretamente, vimos anteriormente que a legibilidade de código é de extrema importância.

Importante!

Em muitos cenários, você pode se deparar com um software que possui funcionalidades que apresentam uma execução conforme o esperado pelo cliente, mas uma investigação de seu código-fonte revela que foi este foi construído sem o amparo de boas práticas.

A manutenção de projetos de software que se encaixam neste cenário podem ser tão difíceis que alguns acabam abandonados. Neste percurso, vamos aprender uma tarefa importante do dia a dia de um desenvolvedor para que cenários como esse possam ser evitados, a fatoração.

Na matemática, o processo de fatoração consiste em representar uma expressão através de um produto de fatores.

Voltar ao sumário

EXEMPLO: um exemplo de uso de fatores seria colocar os comuns em evidência: 2x + 2 = 2(x + 1).

E no contexto de desenvolvimento? É possível extrair porções de código comum e colocar em "evidência"? A resposta é sim. Podemos entender então que o processo de fatoração de código significa modificar a forma como este é apresentado para alcançar algum benefício. Nos circuitos anteriores, mostramos que a legibilidade de um código e o uso de boas práticas são caraterísticas importantes para a manutenção de um software.

Quando produzimos um código e percebemos que existe uma solução mais elegante, podemos voltar atrás e ajustá-lo. Chamamos esse processo de refatoração de código, o ato de revisitar um código já construído para aplicar técnicas de fatoração, sem que novas funcionalidades sejam construídas. Em outras palavras, quando revisitamos um código que está funcionando a fim de alcançar um dos seguintes objetivos:

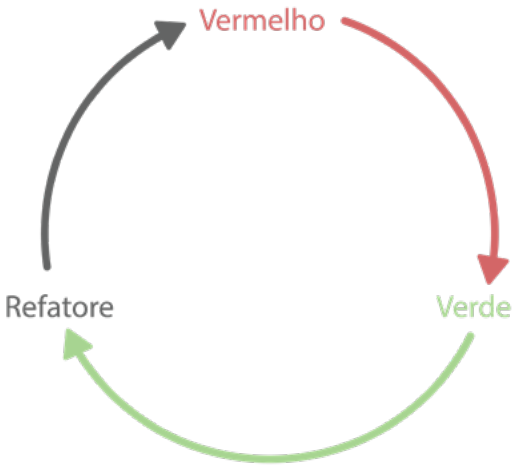
1. Adesão a boas práticas, como código limpo;
2. Remoção de "maus cheiros" em código; e
3. Melhoria da manutenibilidade de uma forma geral.

Em uma refatoração, não existe uma situação ou cenário novo de uso em um software. Estamos apenas revisitando algo que já foi criado para melhorar uma porção do código. Segundo Martin (2020), a refatoração não deve ser incluída em um planejamento como uma atividade organizada. Isto é, não devemos alocar ao processo de refatoração um tempo apenas para essa atividade dentro do planejamento de projeto. Essa atividade deve ocorrer com frequência ao longo do desenvolvimento, ou seja, em outras palavras, cabe ao desenvolvedor aplicar as técnicas de fatoração diariamente em seu código.

Nos circuitos anteriores, vimos a importância de uma estratégia de testes e que qualidade de código está ligada a um número reduzido de *bugs*. Consequentemente, a refatoração também faz parte do processo de testes. Observe, na figura abaixo, um exemplo popular de processo que relaciona refatoração a testes, conhecido como Vermelho/Verde/Refatore (MARTIN, 2020).

Voltar ao sumário

Figura 4 - Vermelho/Verde/Refatore



Fonte: Adaptado de Martin, 2020.

O primeiro passo que o desenvolvedor deve executar nesse processo é escrever um teste que irá falhar, seguido de um teste que irá ser bem-sucedido. Por fim, o desenvolvedor deve modificar o código a fim de deixá-lo mais "limpo" e o ciclo retorna para o primeiro passo.

Neste processo, o desenvolvedor tem a liberdade de focar na sua ideia original de desenvolvimento, sem se preocupar com legibilidade, testes ou bugs. Após construir sua ideia, que provavelmente irá falhar em termos de código limpo e testes, o desenvolvedor passa a se preocupar em fazer o código funcionar. Ou seja, ele deve trabalhar até que todos os testes sejam bem-sucedidos.

Uma vez que todas as funcionalidades estejam executando apropriadamente, o desenvolvedor se preocupa agora em tornar o código mais limpo e o ciclo se repete até que o software seja implementado por completo.

Importante!

Vale ressaltar que o desenvolvedor não fará "bagunça" propositalmente. Este processo visa resolver "bagunças", mas tem como base a premissa de que o desenvolvedor aplicará técnicas de fatoração logo após a realização das demais atividades.

Para isso, vejamos o exemplo abaixo de classe Java com "mau cheiro" que foi recentemente implementada e vamos tentar entender o que esta classe Java se propõe a fazer. Confira o exemplo

Voltar ao sumário

```java
P.java
package e;

public class P {
    int [] e;
    int q;

    public P(){
        e = new int[10];
        q = 0;
    }

    public void e(int e) {
        this.e[q] = e;
        q++;
    }

    public int d() {
        int q = this.q;
        this.q--;
        return e[q];
    }
}
```

O que essa classe Java faz? Você conseguiu entender de imediato? É possível que mesmo separando alguns minutos você ainda não consiga entender o que está acontecendo.

Ao escolher nomes de variáveis, evitamos usar nomes muito curtos ou muito longos, tentando ser o mais descritivos o possível enquanto atentamos à legibilidade. Não existe consenso, mas escrever menos do que 3 caracteres pode dificultar a compreensão.

De maneira semelhante, nomes muito longos tendem a trocar o problema. Antes, enquanto era difícil de se entender o que uma letra ou duas significavam, agora temos o problema de uso excessivo de caracteres para um único nome.

EXEMPLO: caso uma variável que tenha 30 caracteres em seu nome apareça várias vezes eu um trecho de código, grande parte da tela será dedicada apenas ao nome da variável, dificultando a leitura do código. Podemos considerar uma boa prática o uso de menos que 17 caracteres por nome.

Veja no exemplo abaixo como ficaria nosso código caso o desenvolvedor siga essa simples recomendação (estamos fazendo uso de Javadocs – texto em verde – para melhorar ainda mais a legibilidade), Confira no exemplo:

Voltar ao sumário

```java
Pilha.java
package estruturas;

/**
 * Implementação básica da estrutura de dados Pilha
 */
public class Pilha {
    int [] elementos;
    int quantidade;

    public Pilha(){
        elementos = new int[10];
        quantidade = 0;
    }

    /**
     * Método para empilhar um novo elemento na pilha
     * @param elemento
     */
    public void empilhar(int elemento) {
        elementos[quantidade] = elemento;
        quantidade++;
    }

    /**
     * Método para desempilhar o elemento do topo da pilha
     * @return
     */
    public int desempilhar() {
        int posicaoRemovida = quantidade;
        quantidade--;
        return elementos[posicaoRemovida];
    }
}
```

Note que modificar os nomes fez com que a classe ficasse mais fácil de ser entendida. Percebemos então que antes de aprendermos as técnicas de fatoração, precisamos reconhecer quais são os principais pontos em um código que precisam de melhoria. Para isso, vamos estudar em mais detalhes os “maus cheiros” de código que mencionamos anteriormente neste percurso.

Código repetido

Imagine que está implementando um software de controle de um sensor de temperatura e que, em múltiplas partes do seu código, você coloca três linhas de código para atualizar o valor de temperatura na interface gráfica do sensor. Agora, imagine que você descobriu um pequeno erro nessas três linhas de código.

Você terá que visitar cada local em que essas linhas são repetidas e fazer os ajustes, tendo um grande retrabalho.

Voltar ao sumário

Entretanto, podemos aplicar uma técnica de fatoração para extrair essas três linhas de código para um método especializado, que faz a atualização do valor de temperatura na interface gráfica do sensor. Dessa forma, removemos o código repetido e caso um ajuste seja necessário, faremos ele apenas uma vez.

Métodos muito grandes

Procedimentos ou funções podem realizar desde tarefas pequenas até tarefas enormes. Até aí tudo bem, mas note que realizar manutenção em um método pequeno é mais simples do que em um método grande. Suponha que dois métodos precisam de um ajuste, o primeiro tem duas linhas e o segundo duzentas. No primeiro, não existe muito espaço para se perder, será fácil saber onde o ajuste deve ser feito. Já no segundo método, existe a possibilidade de que tenhamos que ler todas as duzentas linhas de código múltiplas vezes até localizar onde o ajuste deve ser feito.

Importante!

Para evitar que isso aconteça, podemos aplicar uma técnica de fatoração para remover o código repetido de dentro do método ou quebrar a funcionalidade em partes menores.

Classes muito grandes

Podemos utilizar boas práticas para delimitar responsabilidades à uma classe. Muitas vezes este trabalho é próximo ao intuitivo quando utilizamos um paradigma orientado a objetos. Todavia, caso exista uma classe com, por exemplo, duas mil linhas, por mais que faça parte de sua proposta, navegar em seu conteúdo pode ser trabalhoso. Quando classes ficam muito grandes, verificamos se podemos reduzir os métodos dentro da classe. Caso isso já tenha sido feito, podemos extrair alguns métodos para outras classes, quebrando a complexidade e melhorando a legibilidade do código. Em breve, aprenderemos a técnica de extração de método.

Escrita em múltiplas línguas

Quando desenvolvemos um software, podemos fazer o uso de várias linguagens de programação. Em nossos exemplos, estamos usando a linguagem Java. Todavia, nossa linguagem natural é o português. Isto significa que, ao fazer a escolha de nomes de variáveis, métodos, entre outros, utilizamos o português para descrever sua funcionalidade ou utilidade. Infelizmente, não é difícil encontrar exemplos de código que possuem inglês como linguagem natural além da língua portuguesa.

Voltar ao sumário

Consideramos uma boa prática utilizar um único idioma ou linguagem natural em um único projeto. Diante de necessidades específicas, podemos implementar componentes de um software baseados em uma língua, enquanto outros componentes são baseados em outra. A mistura de idiomas em um único componente é desencorajada, mas podemos ter múltiplos idiomas configurados em uma única IDE para projetos distintos.

Importante!

Para manter um conjunto consistente de boas práticas, é importante aderir às práticas de código limpo, que não tem "mau cheiro". Para isso, precisamos aplicar técnicas de fatoração e ajustar nosso código, sem modificar sua lógica, mas melhorando seu entendimento.

Agora, veremos as principais técnicas de fatoração no ambiente de desenvolvimento Intellij IDEA, mas a grande maioria de IDEs possuem as mesmas técnicas ou técnicas semelhantes.

Extração e movimentação de métodos

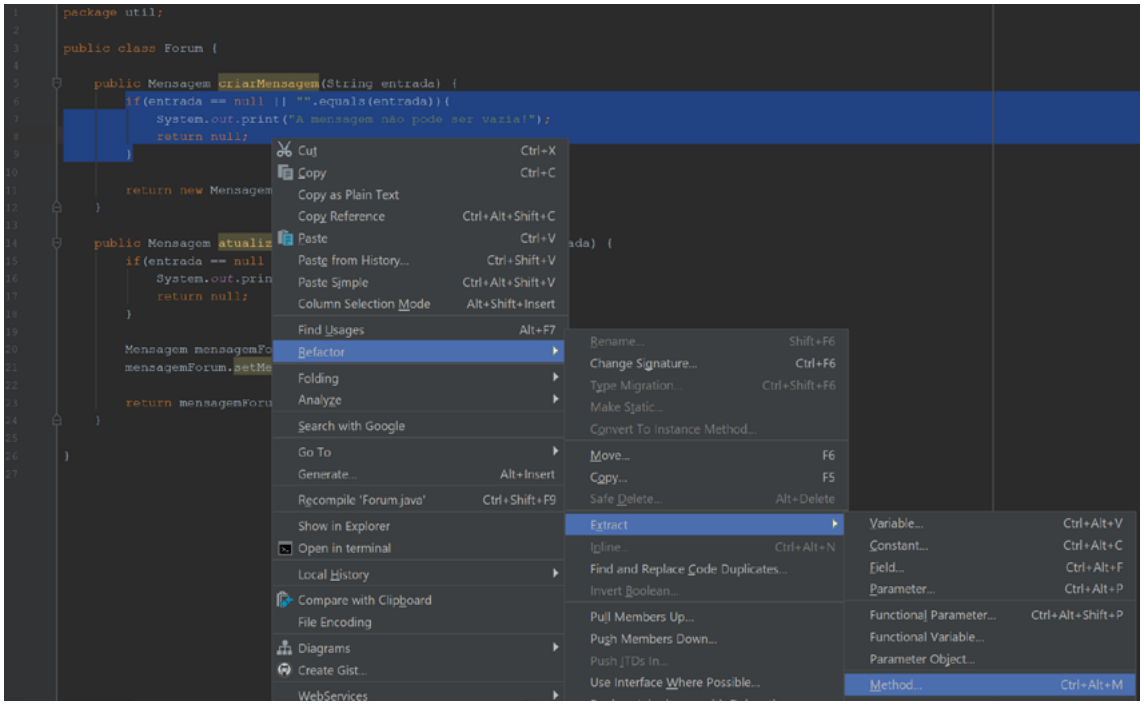
Em um cenário no qual existe código repetido, podemos enxergar uma porção de código em comum como fator que pode ser colocado em "evidência", como em nossa analogia com a definição matemática. Chamamos esta técnica de fatoração de extração de métodos e ela já vem inclusa na maior parte das IDEs.

Observe a classe Fórum abaixo e note que dentro dos métodos criarMensagem e atualizarMensagem existem quatro linhas iguais. Este é um ótimo exemplo para aplicar uma extração de método, confira no exemplo:

Voltar ao sumário

```
package util;

public class Forum {

    public Mensagem criarMensagem(String entrada) {
        if(entrada == null || "".equals(entrada)){
            System.out.print("A mensagem não pode ser vazia!");
            return null;
        }

        return new Mensagem(entrada);
    }

    public Mensagem atualizarMensagem(int idMensagem, String entrada) {
        if(entrada == null || "".equals(entrada)){
            System.out.print("A mensagem não pode ser vazia!");
            return null;
        }

        Mensagem mensagemForum = Mensagem.acharPorID(idMensagem);
        mensagemForum.setMensagem(entrada);

        return mensagemForum;
    }

}
```

Ao selecionar as linhas de código para serem extraídas, o Intellij IDEA detecta automaticamente o uso das mesmas linhas de código em outros métodos da classe Fórum, confira no exemplo:



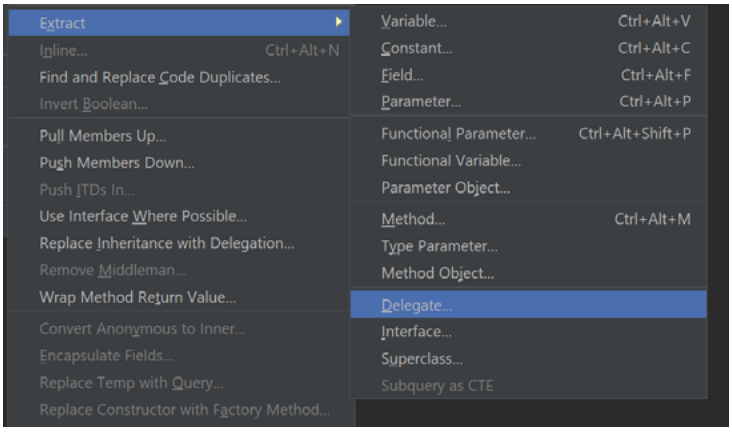
É possível acessar essa funcionalidade por meio do atalho Ctrl + Alt + M, ou navegando pelo botão direito conforme imagem acima. Uma vez selecionada e confirmando a extração para ambos os métodos, a classe resulta conforme o exemplo abaixo:

Voltar ao sumário

```java
package util;

public class Forum {

    public Mensagem criarMensagem(String entrada) {
        if (mensagemVazia(entrada)) return null;

        return new Mensagem(entrada);
    }

    public Mensagem atualizarMensagem(int idMensagem, String entrada) {
        if (mensagemVazia(entrada)) return null;

        Mensagem mensagemForum = Mensagem.acharPorID(idMensagem);
        mensagemForum.setMensagem(entrada);

        return mensagemForum;
    }

    private boolean mensagemVazia(String entrada) {
        if(entrada == null || "".equals(entrada)){
            System.out.print("A mensagem não pode ser vazia!");
            return true;
        }
        return false;
    }

}
```

Extração de classes

A extração de classes ocorre de forma semelhante à extração de métodos. Uma vez que parte do código de uma classe esteja selecionado (por exemplo, um ou mais métodos), podemos delegá-los a uma classe nova. No Intellij, esta funcionalidade fica logo abaixo do Extract Method, conforme mostra o exemplo a seguir:



Renomeação de classes, métodos e variáveis

Suponha que queremos mudar o nome de um método. Note que, caso este método esteja sendo utilizado em outro local, precisaremos mudar o nome lá também. Caso este método seja utilizado em múltiplos locais de um projeto, cada instância de uso deve ser ajustada. Em vez de procurar cada chamada manualmente, podemos fazer uso da técnica de refatoração presente em uma IDE.

Voltar ao sumário

Essa funcionalidade irá localizar e alterar automaticamente todas as instâncias de uso do método, de acordo com a modificação feita em apenas uma delas. Veja o exemplo:

```
1   package util;
2
3   public class Forum {
4
5       public Mensagem criarMensagem(String entrada) {
6           if (estaVaz(entrada)) return null;
7
8           return new Mensagem(entrada);
9       }
10
11      public Mensagem atualizarMensagem(int idMensagem, String entrada) {
12          if (estaVaz(entrada)) return null;
13
14          Mensagem mensagemForum = Mensagem.acharPorID(idMensagem);
15          mensagemForum.setMensagem(entrada);
16
17          return mensagemForum;
18      }
19
20      private boolean estaVaz(String entrada) {
21          if(entrada == null || "".equals(entrada)){
22              System.out.print("A mensagem não pode ser vazia!");
23              return true;
24          }
25          return false;
26      }
27
28  }
29
```

No exemplo acima, estamos atualizando o nome do método mensagemVazia da classe Fórum para um novo nome: estaVazia. Note que enquanto estamos digitando a modificação em apenas um local, outros locais estão sendo atualizados simultaneamente.

O mesmo processo pode ser utilizado para renomear classes, variáveis e demais elementos em um código (parâmetros, pacotes, etc.). Experimente fazer uso dessa técnica de formatação no Intellij, através do botão direito ou do atalho padrão Shift+F6. Em IDEs diferentes, como no Eclipse, os atalhos provavelmente não são os mesmos, mas a maioria das técnicas estarão presentes.

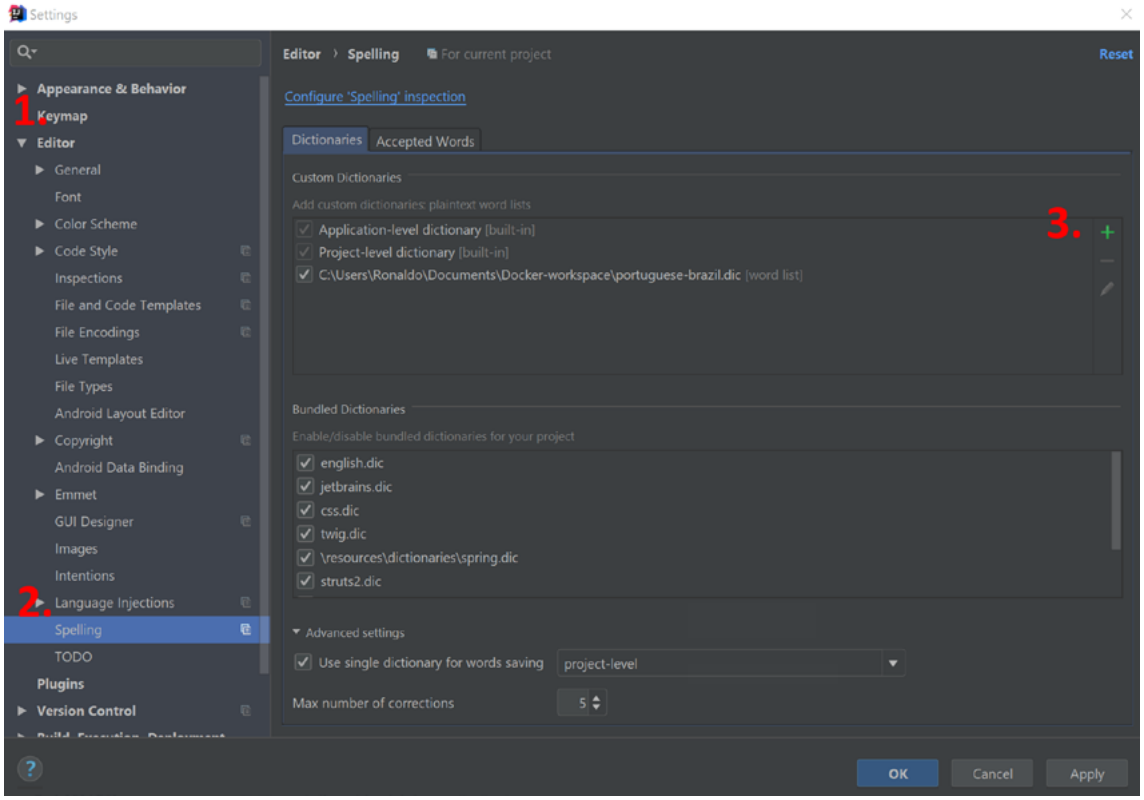
Checagem gramatical de nomes e comentários

Ao nomear variáveis, sua IDE pode indicar erros gramaticais e fazer sugestões de alteração de nome, caso configurada para tal. O comportamento do Extract Method e demais técnicas de fatoração continuará da mesma maneira, mas, conforme discutimos ao longo deste percurso de aprendizagem, é possível que você queira configurar sua IDE para considerar mais de um idioma.

O idioma inglês vem por padrão na grande parte das IDEs, mas podemos adicionar outros idiomas, como por exemplo, o português. Isso pode ser feito através do uso de um dicionário. No Intellij, aperte Ctrl + Alt + S para abrir as *settings*, ou navegue até lá através de File >> Settings.

Voltar ao sumário

Veja o exemplo:



Uma vez que a janela de configurações está aberta, acesse (1.) Editor, depois (2.) Spelling e por último, clique no (3.) + para adicionar o dicionário de português do Brasil. Agora seu código irá reconhecer a linguagem portuguesa. Se você fez todos esses exercícios, bom trabalho. No próximo percurso, vamos estudar gerência de configuração e mudanças.

Voltar ao sumário

Resumo:

Neste segundo percurso de aprendizado do componente curricular Ambiente de Desenvolvimento de Software, visitamos conceitos importantes de programação defensiva. Primeiramente, estudamos a importância da legibilidade de código, pois, mesmo que alguns considerem a escrita de código-fonte como a parte mais importante no processo de programação, vemos que na verdade a leitura tende a ser realizada com maior frequência do que a escrita no dia a dia de um desenvolvedor. Estudamos elementos que deixam um código limpo e de fácil entendimento. Além disso, visitamos os principais motivos que deixam um código com "mau cheiro", práticas que indicam um distanciamento do código limpo e ideal. Vimos que realizar uma programação pensando na legibilidade pode melhorar a qualidade do código em produção e trazer muitos outros benefícios.

Outro aspecto de qualidade que aprendemos foi a redução da quantidade de erros (ou *bugs*) que existem dentro de um código-fonte. Mesmo que de forma não intencional, é comum introduzirmos erros enquanto programamos, o que torna o planejamento de testes uma tarefa muito importante. Vimos que um código pode ser dividido em unidades a fim de serem testadas - a que damos o nome de testes unitários - para verificar seu correto funcionamento enquanto procuramos por erros. Esclarecemos as diferenças entre verificação e validação e vimos alguns tipos distintos de estratégias para testes, incluindo a estratégia espiral, com testes unitários, testes de integração e testes de validação.

Em seguida, realizamos um estudo de como manipular e tratar erros e aprendemos a modificar um código-fonte preexistente e a melhorar sua qualidade através da aplicação de técnicas de fatoração. Aprendemos também que a refatoração deve ser realizada em conjunto com testes e ser uma tarefa natural de qualquer desenvolvedor. Por fim, identificamos os mais comuns "maus cheiros" em código e vimos como aplicar as principais técnicas de fatoração dentro de um dos mais populares ambientes de desenvolvimento de software da atualidade, o IntelliJ IDEA.

Voltar ao sumário

Referências

DELAMARO, Marcio. **Introdução ao Teste de Software**. Grupo GEN, 2016. 9788595155732. E-book. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595155732

FERREIRA, Arthur G. **Interface de programação de aplicações (API) e web services**. Editora Saraiva, 2021. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9786553560338/.

FOWLER, Martin. **Refatoração**. Grupo A, 2004. 9788577804153. E-book. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788577804153/

GONÇALVEZ, Priscila de F. et al. **Testes de software e gerência de configuração**. Grupo A, 2019. 9788595029361. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788595029361/

LAMOUNIER, Stella Marys D. **Qualidade de software com Clean Code e técnicas de usabilidade**. Editora Saraiva, 2021. 9786589965565. E-book. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9786589965565/

LEWIS, Elijah. **Clean Code**: Advanced and Effective Strategies To Use Clean Code Methods. São Paulo, 2020.

MARTIN, Robert C. **Arquitetura limpa**: O guia do artesão para estrutura e design de software. Editora Alta Books, 2019. 8550804606.

MARTIN, Robert C**. Código limpo**: habilidades práticas do Agile software. Rio de Janeiro: Alta Books, 2019.

MARTIN, Robert C. **Desenvolvimento Ágil Limpo**. Editora Alta Books, 2020. 9788550816890. E-book. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788550816890/

MASCHIETTO, Luis G.et al. **Desenvolvimento de Software com Metodologias Ágeis**. Grupo A, 2021. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9786556901824/.

MAYER, Carlos R.; MAZER, Ademir Jr. **Visão Introdutória sobre os conceitos do código limpo**. 2015. Disponível em: https://semanaacademica.org.br/system/files/

artigos/visao_introdutoria_sobre_os_conceitos_de_codigo_limpo.pdf

PRESSMAN, Roger S.; MAXIM, Bruce R**. Engenharia de software**. Grupo A, 2021. 9786558040118. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9786558040118/

SCHILDT, Herbert. **Java para iniciantes**. Grupo A, 2015. 9788582603376. E-book. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788582603376/

SEBESTA, Robert. **Conceitos de Linguagens de Programação**. Grupo A, 2018. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/9788582604694/

SOMMERVILLE, Ian. **Engenharia de software**. Tradução Maurício de Andrade. 9. ed. São Paulo: Pearson Education do Brasil; Addison-Wesley, 2011.

FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA
ENSINANDO E APRENDENDO

UNIVERSIDADE DE FORTALEZA (UNIFOR)

Presidência
Lenise Queiroz Rocha

Vice-Presidência
Manoela Queiroz Bacelar

Reitoria
Fátima Maria Fernandes Veras

Vice-Reitoria de Ensino de Graduação e Pós-Graduação
Maria Clara Cavalcante Bugarim

Vice-Reitoria de Pesquisa
José Milton de Sousa Filho

Vice-Reitoria de Extensão
Randal Martins Pompeu

Vice-Reitoria de Administração
José Maria Gondim Felismino Júnior

Diretoria de Comunicação e Marketing
Ana Leopoldina M. Quezado V. Vale

Diretoria de Planejamento
Marcelo Nogueira Magalhães

Diretoria de Tecnologia
José Eurico de Vasconcelos Filho

Diretoria do Centro de Ciências da Comunicação e Gestão
Danielle Batista Coimbra

Diretoria do Centro de Ciências da Saúde
Lia Maria Brasil de Souza Barroso

Diretoria do Centro de Ciências Jurídicas
Katherinne de Macêdo Maciel Mihaliuc

Diretoria do Centro de Ciências Tecnológicas
Jackson Sávio de Vasconcelos Silva

AUTOR
RONALDO PINHEIRO GONÇALVES JUNIOR

Graduado em Ciência da Computação (2011) e Mestre em Informática Aplicada (2015), ambos pela Universidade de Fortaleza, e Doutorando em Engenharia de Software pela University of Texas at Dallas. Possui experiência em Desenvolvimento Full Stack de aplicativos para Web e Nuvem. Docente do Centro de Ciências Tecnológicas da Universidade de Fortaleza e participante de projetos com foco nas áreas de Computação em Nuvem, Engenharia de Requisitos e Sistemas de Recomendação.

RESPONSABILIDADE TÉCNICA

UNIVERSIDADE DE FORTALEZA ENSINANDO E APRENDENDO — VRE Vice-Reitoria de Ensino de Graduação e Pós-Graduação — Universidade de Fortaleza EAD

COORDENAÇÃO DA EDUCAÇÃO A DISTÂNCIA

Coordenação de Ensino e Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Planejamento Educacional
Ana Flávia Beviláqua Melo

Supervisão de Recursos EAD
Francisco Weslley Lima

Supervisão de Ensino e Recurso
Liliane Rios Cajazeiras Mendes

Analista Educacional
Lara Meneses Saldanha Nepomuceno

Projeto Instrucional
Maria Mirislene Vasconcelos

Revisão Gramatical
Janaína de Mesquita Bezerra

Identidade Visual / Arte
Francisco Cristiano Lopes de Sousa

Editoração / Diagramação
Régis da Silva Pereira

Produção de Áudio e Vídeo
José Moreira de Sousa
Pedro Henrique de Moura Mendes

Programação / Implementação
Renan Alves Diniz
Thaís Rozas Teixeira