

O Extreme Programming, chamado de XP, é um método ágil com foco no desenvolvimento de softwares. Esse método tem base em três pilares: agilidade no desenvolvimento da solução, economia de recursos e qualidade do produto final.

A metodologia ágil de Extreme Programming já foi provada com muito sucesso por diversas companhias de vários tamanhos em todo o mundo. Criada na década de 1990, o XP procura chegar à excelência nos serviços prestados.

# O que é Extreme Programming (XP)?

O sucesso do uso do XP se dá principalmente porque é baseado na satisfação do consumidor da forma mais rápida possível. Ou seja, ao invés de entregar todas as funcionalidades que você gostaria que sua aplicação tivesse em algum momento no futuro, com XP é feita a entrega apenas do necessário, no momento mais próximo.



Além disso, o Extreme Programming dá poder aos desenvolvedores para responderem com confiança para qualquer mudança nos pedidos do consumidor de forma rápida, até mesmo nas últimas etapas de desenvolvimento. O XP tem muita semelhança com SCRUM, em termos de valores e modelo de desenvolvimento de projetos.

O XP dá ênfase para o trabalho em equipe. Gestores, consumidores e desenvolvedores são todos partes iguais de um time colaborativo e precisam se reunir juntos em torno do problema que será solucionado. Os testes são uma constante dentro dos times e já começam logo no primeiro dia, entregando resultados e implementando alterações de acordo com as sugestões que vão aparecendo no uso.

# Como funciona o método de Extreme Programming?

Para que as entregas dentro do XP sejam possíveis, as equipes precisam se basear em valores, isto é, um contrato de atitudes e comportamentos que levam ao sucesso. Esses comportamentos e atitudes norteiam as ações da equipe XP em cada atividade a ser desempenhada, garantindo a integração e a sinergia necessárias para o bom desempenho. Esses valores são:

* Comunicação;
* Simplicidade;
* Feedback;
* Coragem;
* Respeito.

Trabalhando com esses conceitos, o Extreme Programming consegue desenvolver softwares de forma mais barata e em menor tempo. Com a comunicação estreita entre cliente e desenvolvedores e reuniões para saber do andamento do projeto, o produto final fica perto do que o cliente pediu a priori. O feedback do cliente fica a cargo de reavaliar o software e indicar as mudanças ou o andamento do projeto. Dessa forma, o custo exponencial cai porque não há necessidade de refazer linhas do projeto já que a simplicidade e o desejo de atender ao que foi pedido são colocados em primeiro plano.

Além dos valores, o método ágil XP também leva em consideração melhores práticas de trabalho, que têm como objetivo garantir a efetividade do trabalho da equipe XP, assim como a satisfação do cliente durante todo o processo de desenvolvimento.

São elas:

* cliente sempre à disposição;
* uso de metáforas;
* reuniões de planejamento (planning game);
* reuniões diárias, de 15 minutos, para alinhamento (stand up meeting);
* integração contínua dos módulos desenvolvidos;
* mudanças incrementais;
* entregas frequentes ao cliente (small releases);
* design simples e funcional;
* testes de aceitação;
* refatoração (refactoring) ou melhoria contínua.

# NO XP alguns itens que não pode faltar…

# - PAIR PROGRAMMING

Quando duas pessoas trabalham em uma única máquina. Um colaborador codifica e o outro faz sugestões pertinentes, trocando de funções em determinado momento. Essa troca garante uma comunicação mais fácil, garantindo maior produtividade e um projeto de alta qualidade.

Beneficio: Revisão de código contínua: muitos erros são pegos quando eles estão sendo codificados, em vez de serem descobertos por um testador, tornando a quantidade de erros consideravelmente menor;

Discussão contínua da solução: o design é aprimorado e o código fica mais sucinto;

Afinamento do par: o time resolve problemas mais rapidamente por estar coeso também em pares;

Aprendizagem: as pessoas aprendem significativamente sobre o sistema e sobre engenharia de software;

Gestão do conhecimento: o projeto acaba com várias pessoas entendendo cada pedaço do sistema;

Trabalho colaborativo: os indivíduos aprendem a trabalhar em time e a conversar mais frequentemente, dando mais fluxo à informação e à dinâmica do grupo;

Motivação: as pessoas apreciam mais o seu trabalho e o trabalho de todos.

Especificar, projetar, trabalhar, tudo em par Mesmo que essa prática do XP chame-se programação em pares, o trabalho como um todo pode ser realizado em dupla. Um programador pode parear com o cliente para tirar dúvidas; o cliente pode parear com um testador para escrever bons testes de aceitação; e um programador pode parear com um testador para discutir alguns testes unitários.

Dica: pareamento em tudo! Somos tão adeptos do trabalho em par que pareamos em atividades além do desenvolvimento de software. Lecionados cursos em par, palestramos em eventos em par etc. Sabemos que, assim, um curso ou uma palestra tem muito mais qualidade, motivação e troca de conhecimento. Então, que tal começar a parear também?

# - PLANNING GAME

Jogos de Planejamento (Planning Game): O desenvolvimento é feito em iterações semanais. No início da semana, desenvolvedores e cliente reúnem-se para priorizar as funcionalidades. Essa reunião recebe o nome de Jogo do Planejamento.

Nela, o cliente identifica prioridades e os desenvolvedores as estimam. O cliente é essencial neste processo e assim ele fica sabendo o que está acontecendo e o que vai acontecer no projeto. Como o escopo é reavaliado semanalmente, o projeto é regido por um contrato de escopo negociável, que difere significativamente das formas tradicionais de contratação de projetos de software. Ao final de cada semana, o cliente recebe novas funcionalidades, completamente testadas e prontas para serem postas em produção.

# - CODE REVIEW

Utilize o Gitflow para ajudar nessa técnica, sempre que ao abrir um pull request selecione os reviewers para que assim possam realizar as revisão do seu código.

Reviewers: Integrantes do time que analisam o código e comentam se ha alguma falha ou melhores praticas a serem aplicadas no código.

Benefícios: Tanto quem revisa quanto quem esta revisando aprendendo com q implementação que o outro desenvolvedor realizou e pode comentar possíveis (melhores praticas, bad smells codes).

# - REFATORAÇÃO DE CÓDIGO PARA ALTA QUALIDADE

*“Qualquer tolo pode escrever código que o computador possa entender. Bons programadores escrevem código que humanos possam entender.” -Martin Fowler*

Pois bem, a refatoração tem o mesmo propósito na codificação. Com origem na Orientação a Objetos, é também chamada de refinamento incremental e mantém a semântica da funcionalidade, alterando apenas o design. Uma refatoração é uma mudança feita na estrutura interna do software que o faz ficar mais fácil de entender e mais barato de modificá-lo, sem mudar seu comportamento observável. Obter essa simplicidade não é algo fácil, por isso a fazemos por meio de uma série de pequenas alterações, uma a uma sem alterar esse comportamento.

Quais são seu benefícios?

* Economiza tempo e aumenta a qualidade;
* Melhora o design do software; O código torna-se mais legível e reutilizável;
* Fica mais fácil encontrar defeitos;
* Ajuda você a programar mais rápido.

A abordagem do XP é refatorar impiedosamente; ou seja, não se opta por isso. Falaremos disso a seguir, bem como sobre bad smells de código.

*Escreva o código para pessoas, não apenas para o compilador. Seu time agradecerá e alguém no futuro certamente ficará feliz em ver um bom código estruturado.*

# - Testes em XP

O XP usa uma abordagem Orientada a Objetos como seu paradigma de desenvolvimento. A figura a seguir ilustra o processo XP e mostra algumas das ideias-chave e tarefas que são associadas a cada atividade de arcabouço. O XP sugere um número de técnicas inovadoras e potentes que permitem a equipes ágeis criar frequentemente versões de software que possuem características e funcionalidades descritas e priorizadas pelo cliente.

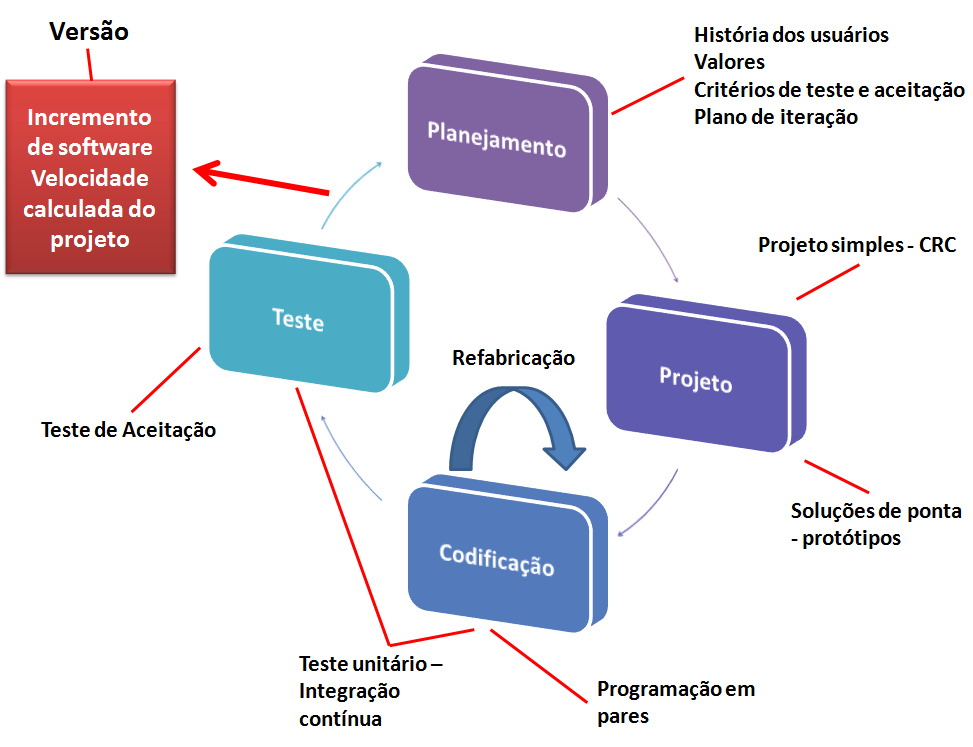
[](http://jkolb.com.br/wp-content/uploads/2013/12/xp.png)

Figura 1: Processo XP.

Fonte: PRESSMAN (2010).

O XP é um processo ágil organizado como quatro atividades de arcabouço para:

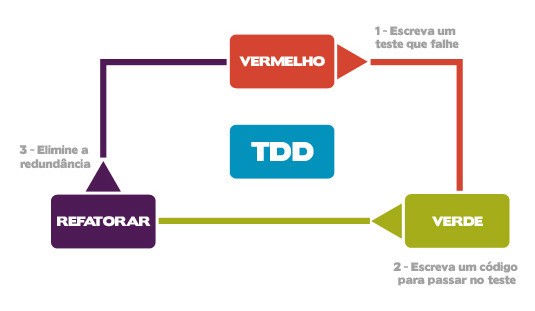
* planejamento;
* projeto;
* codificação;
* teste.

Os testes unitários que são criados devem ser implementados usando um arranjo sequencial que lhes permita ser automatizados. Isso encoraja uma estratégia de teste de regressão, sempre que o código é modificado.

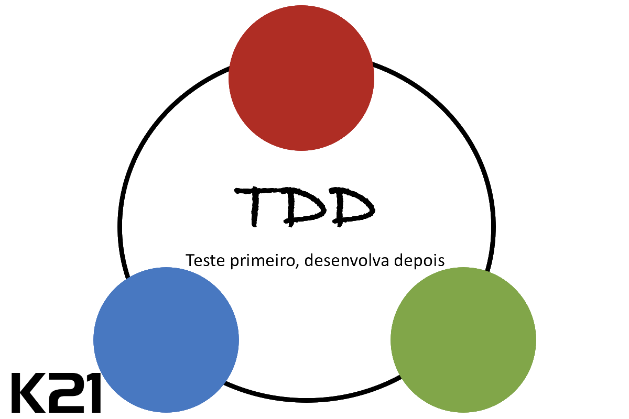
À medida que os testes unitários individuais são organizados em uma sequência universal de testes, o teste de integração e validação do sistema pode ocorrer diariamente. Isso fornece à equipe XP um indicador contínuo de progresso do desenvolvimento.

Os testes de aceitação XP, também chamados de testes do cliente, são especificados pelo cliente e focalizam as características e funcionalidades do sistema global que são visíveis e passíveis de revisão pelo cliente.

# - TDD: Desenvolvimento Guiado por Testes



O TDD (do inglês Test-driven Development ─ Desenvolvimento Guiado por Testes) é uma técnica para construção de software que guia seu desenvolvimento por meio da escrita de testes. Sua essência está em seu mantra: testar, codificar e refatorar (veja na imagem a seguir). Essa técnica surgiu a partir da refatoração, dando base para guiar a codificação. TDD gera aprendizado, porque ajuda a aprender boas práticas de programação e a criar testes automatizados.



Escrever testes antes de codificar? Sim, exatamente! No TDD, primeiramente é escrito o código de teste e depois (somente depois) o de produção. Dessa forma, a codificação será guiada por testes e o código de produção estará coberto por eles. Os testes guiarão o desenvolvimento. É uma mudança de paradigma que vale a pena conhecer.

Uma visão desse mantra como um diálogo:

Pergunta: você pergunta ao sistema por meio de um teste.

Resposta: você responde escrevendo código que passa no teste.

Refinamento: você refina a pergunta consolidando ideias, removendo itens desnecessários e clarificando ambiguidades.

Repetição: você mantém o diálogo fazendo novas perguntas.

# Padrões gerais de TDD:

* Teste seu software com testes automatizados;
* Um teste não deve afetar a execução do outro;
* Mantenha uma lista de testes;
* Escreva um teste antes de escrever código de produção;
* Comece a escrita do teste pelo método assert(…);
* Os dados de testes devem ter significados evidentes.

# Padrões de teste para TDD:

* Quando escrever casos de teste grandes, comece com sua menor parte que faça quebrar;
* Quando testar recursos caros ou complicados, utilize constantes que simulem o comportamento desses recursos (com estratégias de dummy, stub, spy, fake e mock);
* Quando estiver programando sozinho, deixe o último teste quebrado. Quando você voltar a programar, ele dirá onde você parou.;
* Quando estiver programando em time, deixe todos os testes rodando. Seu grupo precisará ter todos os testes rodando.

# Passo 1: escrever um teste que falhe

Feedback ao programador é um ponto-chave no XP e é guiado por bons testes automatizados. Aí está a importância de automatizá-los: ter um feedback, uma resposta rápida para guiar o desenvolvimento do software. Sua automatização é um investimento a curto e longo prazo, pois guia a codificação e aumenta a qualidade do software a longo prazo.

# Passo 2: escrever código funcional para rodar o teste

Agora, com um teste já falhando, é hora de escrever o código de produção para fazê-lo passar (barra verde). Dessa forma, o código de produção ficará praticamente todo coberto por testes. Neste segundo passo, o objetivo é fazer o código passar, sem se preocupar tanto com o seu design, porque isso será feito na próxima etapa: a refatoração. Ao usarmos baby steps (passos de bebê), o código é produzido até dar barra verde. Rode os testes constantemente. Os baby steps ajudarão a verificar cada passo, você terá feedback contínuo deles. Entregue frequentemente no repositório de código (algumas vezes ao dia). Todos eles devem estar rodando para enviar o código ao repositório;

* Retorne uma constante para simular o resultado e, então, transforme-a gradualmente em uma expressão com variáveis;
* Transforme os retornos constantes em expressões, após possuir dois ou três exemplos;
* Implemente diretamente as operações simples;
* Quando a implementação usar coleções de objetos, inicie sem coleções e, somente depois, utilize-as.

# Passo 3: Refatorar Código

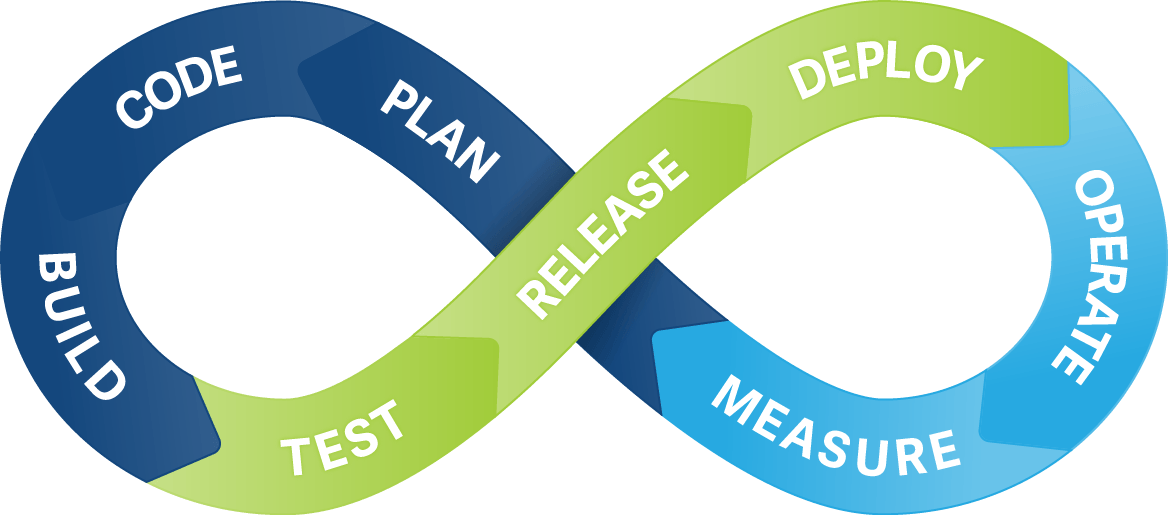
O propósito do TDD está no design do código; o teste é apenas um efeito colateral (ótimo, por sinal).

Agora com o código coberto pelo teste, é hora de melhorar seu design, refatorando-o. Um dos erros mais comuns no uso de TDD está em não refatorar constantemente. Tenha muita atenção nessa parte; se você não a fizer, terá um código mais complexo e não fará TDD. Segundo Martin Fowler, um dos gurus da refatoração, esta é uma mudança feita na estrutura interna do software para deixá-lo mais fácil de entender e barato de modificar, sem mudar seu comportamento observável. Uma das mágicas do TDD está aqui: o comportamento observável é guiado pelo feedback dos testes! Eles apontarão se a refatoração do código o alterou ou não. Cada refatoração deve ser realizada em baby steps; ou seja, não se deve fazer grandes refatorações. O feedback contínuo é essencial para evitar que o comportamento observável não seja alterado. Caso seja, os testes automatizados apontarão o problema. Um baby step deve ser simples, mas não simplório. Assim como em um passo de bebê, você deve dar o maior passo para ir em frente com total segurança. Lembre-se de refatorar também seus testes. Eles fazem parte do software, e se não forem refatorados, se tornarão complexos e dificultarão a manutenção futura.

Fazer uma sessão de Coding Dojo propicia um excelente espaço aberto e momento de aprendizado para treinar a programação em pares, TDD, projeto simples e outras práticas do XP.

# - INTEGRAÇÃO CONTINUA

*“Integração contínua não livra os bugs, mas os tornam dramaticamente mais fáceis de encontrar e de remover.” — Martin Fowler*



Integração contínua (IC) é uma prática, na qual o código que está sendo desenvolvido pelo time é integrado, versionado, construído e verificado diversas vezes ao dia em um ambiente dedicado. Os programadores XP devem integrar e fazer commit em somente uma versão no repositório de código. Cada integração é verificada por um build com testes automatizados, detectando erros o mais cedo possível. Essa prática gera sinergia com as outras do XP. Os builds das pequenas entregas são construídos por meio da IC. Ela auxilia na refatoração, pois verificará o build automaticamente a cada commit realizado no repositório. Em cada um, o padrão de codificação pode ser verificado automaticamente. A integração contínua é um processo simples que traz muito benefícios:

* Aumenta o feedback, a comunicação na equipe e a moral do time;
* Todos veem o que está acontecendo;
* Previne e descobre os problemas de integração mais cedo;
* Reduz riscos e evita a baixa qualidade;
* Não é necessário um integrador dedicado para a equipe;
* Todos têm acesso à versão mais atualizada;
* Auxilia na reutilização de código: os desenvolvedores terão sempre o código mais atualizado;

Ferramentas

Antes de falarmos de ferramentas, é preciso estar claro que integração contínua é muito mais uma atitude do que apenas o uso de ferramentas. Ferramentas servem para dar suporte a um processo; ou seja, uma por si só não faz nada. Algumas das mais utilizadas para essa integração atualmente são: o Jenkins (fork do Hudson), o Bamboo e o CruiseControl. Essas ferramentas trabalham em conjunto com as de versionamento, de build, de qualidade de código e de testes automatizados. Há uma gama delas para cada tecnologia. As ferramentas de versionamento controlam o código-fonte, gerenciando as versões do software. Existem dois modelos principais:

Modelo cliente-servidor: todas as versões do código ficam centralizadas no servidor. Os clientes possuirão apenas a versão de trabalho e executarão comandos sobre o repositório central. Exemplos dessas ferramentas são o Subversion (SVN) e o Concurrent Version System (CVS);

Modelo distribuído: uma cópia integral de todo o repositório de código fica em cada cliente, ou seja, cada um também é um servidor, não sendo necessário ter um central. Exemplos famosos desse modelo de ferramentas são o Git e o Mercurial.

As ferramentas de build empacotam os executáveis do software de acordo com as configurações e uma série de tarefas, também gerenciando dependências de bibliotecas. Normalmente, as ferramentas de desenvolvimento (IDEs) já têm integração a essas de build, sendo um processo transparente para o desenvolvedor.

Exemplos desse tipo são o Ant, o Maven e o MSBuild. As ferramentas de qualidade de código preocupam-se com o seu design. Muitos problemas podem ser automaticamente detectados por elas. São os principais tipos:

* Analisadores de código (por exemplo, o FindBugs e o PMD);
* Verificadores de cobertura de código (por exemplo, o JaCoCo e o Code Coverage);
* Verificador de padrão de codificação (CheckStyle);
* Frameworks de testes de código (JUnit e PHPUnit);
* Gerenciador de qualidade de código (Sonar).

Sobre ferramentas de teste, existem diversas para cada tipo. Para testes funcionais na web, existe o Selenium;

# - Behaviour-Driven Development (BDD)

O Behaviour-Driven Development (BDD ─ Desenvolvimento Guiado por Comportamento) possui o mesmo propósito do XP: unir as pessoas de desenvolvimento com as de negócio. O BDD baseou-se no TDD, e serve também para análise ágil e teste de aceitação automatizado. Ele utiliza a especificação por exemplos, uma abordagem colaborativa para definir requisitos a partir de exemplos realistas, em vez de enunciados abstratos. Cada exemplo é entendido por meio de um cenário com dados reais. Uma história de usuário pode possuir diversos exemplos reais de utilização. Um de cenário poderia ser:

Cenário: pesquisando extreme programming no Google.

Dado que estou na pesquisa do Google

E digito extreme programming no campo de busca

Quando pressiono a tecla Enter

Então é exibida a primeira página de uma lista dos websites com conteúdo relevante sobre o assunto.

Quando o software possuir essas especificações executáveis, ele terá a documentação viva; ou seja, ela estará conectada diretamente ao código por meio dos testes de aceitação. Ferramentas de frameworks auxiliam a escrita e a execução dos cenários JBehave (Java), Cucumber (Ruby), RSpec (Ruby), SpecFlow (.Net), e Jasmine (JavaScript).

# - Código limpo

Ok, já falamos de código limpo (clean code) no começo do artigo, mas a importância é tão grande no XP que vale a pena reforçar essa prática. O livro Código limpo do Uncle Bob (Robert C. Martin) apresenta a disciplina de escrever código limpo, trazendo diversas dicas, como: nomes significativos, funções, comentários, formatação, objetos, estruturas de dados, tratamento de exceções, testes de unidade, classes, concorrência e odores de código. E lembre-se sempre: código limpo não é o que mais se refatora, mas o que menos se suja.

# - Entrega contínua e DevOps

Entrega contínua é um conjunto de princípios e práticas com o objetivo de compilar, testar e liberar software ao cliente de forma mais rápida e frequente. Essa abordagem encaixa-se muito bem com as práticas do XP de liberação constante de pequenas versões e de integração contínua. Podemos dizer que entrega contínua também é uma prática extrema, pois está bastante alinhada com o primeiro princípio do Manifesto Ágil: nossa maior prioridade é satisfazer o cliente por meio da entrega contínua e adiantada de software com valor agregado. Jez Humble e David Farley descrevem os princípios da entrega de software no livro Entrega contínua:

* Criar um processo de confiabilidade e repetitividade de entrega de versão;
* Automatize quase tudo;
* Mantenha tudo sob o controle de versão;
* Se é difícil, faça com mais frequência e amenize o sofrimento;
* A qualidade deve estar presente desde o início;
* “Pronto” quer dizer “entregue”;
* Todos são responsáveis pelo processo de entrega;
* Melhoria contínua.

Seguindo tais princípios, você terá um processo aprimorado de entrega contínua.

Usando TDD, quando acabamos, realmente acabamos. Ou seja, dificilmente temos que retornar ao código futuramente para corrigir falhas, pois possíveis falhas já foram detectadas e corrigidas durante a confecção dos testes.

Uma [**pesquisa**](http://www.nist.gov/public_affairs/releases/n02-10.htm) do Departamento de Comércio dos EUA, publicada em 2002, revelou que [**falhas de software**](http://en.wikipedia.org/wiki/Software_bug) são tão comuns e tão danosas que se estima que causem um prejuízo anual de mais de 60 bilhões de dólares para a economia americana. O estudo também alega que, embora não seja possível remover todos os erros, mais de um terço destes custos poderia ser eliminado caso se utilizasse uma infra-estrutura de testes melhor, que permitisse identificar e remover defeitos mais cedo e de forma mais eficaz. Atualmente, calcula-se que cerca de 50% dos defeitos são encontrados apenas nas fases finais dos projetos, ou após os sistemas começarem a ser utilizados em produção.

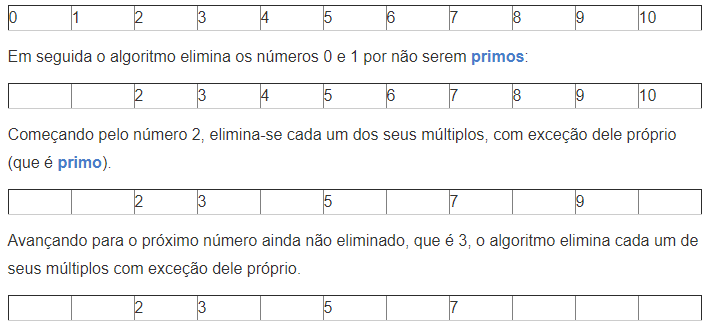
Esse artigo apresenta a técnica de desenvolvimento orientado a testes, que tem como um de seus objetivos antecipar a identificação e correção de [**falhas**](http://en.wikipedia.org/wiki/Software_bug) durante o desenvolvimento. Será utilizado um pequeno exemplo para demonstrar o uso dessa técnica, conhecida em inglês como Test-driven development ou TDD.

## Um exemplo com JUnit

O exemplo é um programa para gerar [**números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), utilizando um conhecido algoritmo criado na antigüidade, descrito no quadro "Como funciona o [**Crivo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php). O leitor sem grandes inclinações matemáticas não precisa se intimidar com este exemplo, pois não é necessário compreender o algoritmo para aplicar os princípios do TDD, basta entender que o resultado esperado é uma seqüência de números primos até um determinado valor N, por exemplo 2, 3, 5, 7 e 11 para N=11.

## Como funciona o [Crivo de Eratóstenes](http://www.numaboa.com.br/criptologia/matematica/testRapid.php)?

[**Números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) têm papel importante no uso de computadores. Destacam-se em particular as abordagens de [**criptografia**](http://pt.wikipedia.org/wiki/Criptografia) utilizando [**chaves públicas**](http://pt.wikipedia.org/wiki/Chave_p%C3%BAblica), que são fortemente baseadas no uso de [**números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) grandes. Gerar tais números de forma determinística é um desafio que vem sendo estudado há muito tempo e um dos algoritmos mais conhecidos para esse fim foi criado pelo matemático grego **[Eratóstenes](http://pt.wikipedia.org/wiki/Erat%C3%B3stenes)** (que viveu no século III AC), por isso chama-se [**Crivo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php). O algoritmo possui um funcionamento simples. Ele começa criando uma lista de números que vai de zero até o número máximo solicitado. Por exemplo, se estivéssemos buscando [**números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até 10, o algoritmo começaria produzindo a lista a seguir:



 processo é repetido até se alcançar o número máximo informado. No caso do exemplo acima, os passos executados já foram suficientes para identificar todos os [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até 10.

Para a construção dos testes sobre o algoritmo de geração de números [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), será utilizado o popular framework **[JUnit](http://www.junit.org/)**, que já vem integrado em vários IDEs [**Java**](http://java.sun.com/), entre eles [**Eclipse**](http://www.eclipse.org/), **[IntelliJ](http://www.jetbrains.com/)**, [**NetBeans**](http://www.netbeans.org/) e **[JBuilder](http://www.borland.com/jbuilder)**. Neste artigo, utilizamos o [**Eclipse**](http://www.eclipse.org/), por exemplo.

Para utilizar o **[JUnit](http://www.junit.org/)**, você precisará colocar o arquivo junit.jar no classpath do seu ambiente de desenvolvimento preferido. Ele pode ser obtido fazendo-se o download do arquivo [**Junit3.8.1.zip**](http://prdownloads.sourceforge.net/junit/junit3.8.1.zip?download), e extraindo-se o arquivo junit.jar.

Nosso ponto de partida será a classe de teste apresentada na [**Listagem 1**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#1), que faz uso de uma outra classe que não criamos ainda, chamada GeradorPrimos (que implementará o [**Crivo de Eratóstenes**](http://pt.wikipedia.org/wiki/Erat%C3%B3stenes)). Desta forma o teste se torna uma "especificação" de como a classe GeradorPrimos deverá funcionar quando estiver pronta.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. import junit.framework.TestCase;
3. public class GeradorPrimosTeste extends TestCase {
4. public void testePrimosGeradosAteNumeroDois() throws Exception {
5. GeradorPrimos geradorPrimos = new GeradorPrimos();
6. assertEquals("2", geradorPrimos.gerarPrimosAte(2));
7. }
8. }

**Listagem 1**: verifica se é capaz de gerar primos até o valor máximo 2.

Espera-se que a classe gere um string, com uma lista de [**números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), separados por vírgula, menores ou iguais ao valor passado como argumento. Por exemplo, se buscássemos números primos até dez, teríamos como resultado a string "2, 3, 5, 7".

O método assertEquals("2", geradorPrimos.gerarPrimosAte(2)), do nosso primeiro método de teste, é usado para verificar se a classe que estamos testando é capaz de gerar [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) corretamente até o valor máximo 2. É um caso muito simples, afinal, estamos só no início. O único [**número primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até 2 é o próprio 2, que é a resposta esperada para esse caso.

O código do teste ainda não é compilável, pois a classe GeradorPrimos ainda não existe: o compilador vai considerar inválida a chamada ao método geradorPrimos.gerarPrimosAte(). Isso é normal quando escrevemos testes primeiro (antes do próprio código da aplicação). Estamos utilizando uma técnica conhecida como [**programação por intenção**](http://www.xprogramming.com/xpmag/acsIntention.htm), na qual escrevemos linhas de código usando classes e métodos que ainda não existem e serão criados para atender às necessidades do teste.

Neste ponto, já temos a interface básica da classe GeradorPrimos definida, pelo seu uso nos testes. Estamos chamando de "interface" as assinaturas, ou seja, o nome dos métodos, os tipos de retorno e os tipos dos parâmetros.

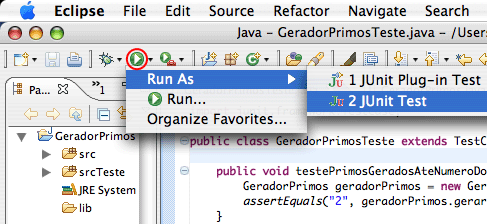
Vamos criar uma primeira versão da classe GeradorPrimos, escrita da forma mais simples possível (ao menos por enquanto). O código da [**Listagem 2**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#2) é suficiente para o teste compilar, mas isso ainda não significa que a classe GeradorPrimos passará no teste. Quando o teste funcionar o **[JUnit](http://www.junit.org/)** vai apresentar uma barra verde indicando que tudo correu bem. Se falhar apresentará uma barra vermelha e uma mensagem indicando qual dos testes quebrou.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

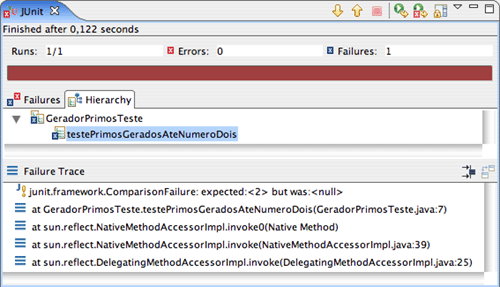
1. public class GeradorPrimos {
3. public String gerarPrimosAte(int i) {
4. return null;
5. }
7. }

**Listagem 2**: estrutura mínima do método gerarPrimosAte().

Podemos ver isso executando o **[JUnit](http://www.junit.org/)** com o que temos até o momento. Para executar o teste no [**Eclipse**](http://www.eclipse.org/), você deve escolher a opção Run|Run As>JUnit Test. Você também pode usar o atalho apresentado na [**Figura 1**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#figura1).

**  
Figura 1**. Ativando o JUnit no [**Eclipse**](http://www.eclipse.org/).

Ao executarmos esse teste, o **[JUnit](http://www.junit.org/)** mostra a barra vermelha apresentada na [**Figura 2**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#figura2).

**  
Figura 2**. Barra vermelha do **[JUnit](http://www.junit.org/)**.

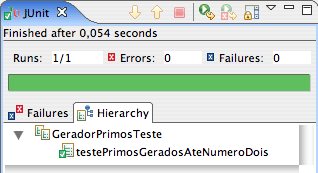
Isso era de se esperar, afinal ainda não escrevemos a implementação correta do método que gera os números primos. Quando se programa utilizando TDD, é importante ter certeza de que o teste realmente será capaz de capturar um erro. Por isso sempre começamos inserindo um erro no código e conferindo se o teste falhou.

Em seguida, para verificar se o teste detecta o funcionamento correto da classe, fazemos outra implementação simples (e temporária) do método gerarPrimosAte(), retornando a string "2":

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public String gerarPrimosAte(int i) {
2. return "2";
3. }

Executando o teste novamente, o **[JUnit](http://www.junit.org/)** mostra que tudo correu bem, como pode ser observado na [**Figura 3**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#figura3).

**  
Figura 3**. Barra verde do **[JUnit](http://www.junit.org/)**.

No desenvolvimento orientado a testes, a primeira preocupação é escrever o teste e assegurar que ele funcione corretamente. Para fazer isso com segurança, é necessário certificar-se de que ele falha, quando temos absoluta certeza de que deveria falhar - e que passa quando temos total confiança de que deveria passar. Descobrimos isso começando por soluções obviamente simples que quebrem ou façam o teste funcionar. Depois disso, com a segurança de que o teste está correto, podemos cuidar da implementação real da classe, com a tranqüilidade de que o teste irá acusar erros ou sucessos de forma coerente.

Esse é um princípio muito utilizado em [**Extreme Programming**](https://www.desenvolvimentoagil.com.br/xp), chamado [**passos de bebê**](https://www.desenvolvimentoagil.com.br/xp/principios/passos_bebe): avançar cuidadosamente dando um pequeno passo seguro de cada vez e só passando à atividade seguinte quando há certeza de que a atividade atual está 100% em ordem.

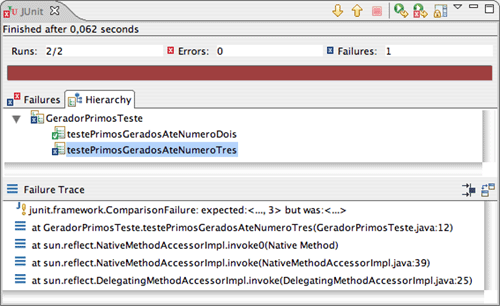
## Incrementando os testes

Para continuar, poderíamos simplesmente escrever o restante do algoritmo, mas a verdade é que a classe já produz respostas certas para os testes que temos até o momento. Seria melhor começar criando novos cenários que levassem à necessidade de estender a implementação da classe GeradorPrimos. Por exemplo (seguindo o princípio dos [**passos de bebê**](https://www.desenvolvimentoagil.com.br/xp/principios/passos_bebe)) será que a classe conseguirá gerar números primos até 3? Eis um teste para descobrirmos isso:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testePrimosGeradosAteNumeroTres() throws Exception {
2. GeradorPrimos geradorPrimos = new GeradorPrimos();
3. assertEquals("2, 3", geradorPrimos.gerarPrimosAte(3));
4. }

Executando o teste no **[JUnit](http://www.junit.org/)**, descobrimos (naturalmente) que ele falha, como podemos ver na [**Figura 4**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#figura4).

**  
Figura 4**. Barra vermelha com vários testes no **[JUnit](http://www.junit.org/)**.

Agora vamos forçar o teste a funcionar com a implementação mais simples possível. Veja a [**Listagem 3**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#3). Com essa modificação, o teste passa. Mas, algo começa a incomodar. Primeiro, o código ainda simplifica demais o problema, pois quando outros números forem informados será difícil gerar a resposta com essa linha de raciocínio. Além disso, a variável i não expressa bem sua intenção e o código de teste começou a apresentar uma incômoda [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode). Para percebê-la, veja o código completo da classe de teste na [**Listagem 4**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#4).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public String gerarPrimosAte(int i) {
2. if (i == 2)
3. return "2";
4. else
5. return "2, 3";
6. }

**Listagem 3**: código suficiente para gerar [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até o valor máximo 3.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. import junit.framework.TestCase;
3. public class GeradorPrimosTeste extends TestCase {
4. public void testePrimosGeradosAteNumeroDois() throws Exception {
5. GeradorPrimos geradorPrimos = new GeradorPrimos();
6. assertEquals("2", geradorPrimos.gerarPrimosAte(2));
7. }
9. public void testePrimosGeradosAteNumeroTres() throws Exception {
10. GeradorPrimos geradorPrimos = new GeradorPrimos();
11. assertEquals("2, 3", geradorPrimos.gerarPrimosAte(3));
12. }
13. }

**Listagem 4**: dois testes até o momento e [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode) de código.

## Simplificando seqüências de testes

A variável geradorPrimos é instanciada da mesma forma nos dois métodos de teste e a estrutura do assertEquals() é basicamente a mesma, mudando apenas os parâmetros. Isso fere um importante princípio, conhecido pela sigla [**DRY**](http://xp.c2.com/OnceAndOnlyOnce.html), do inglês "Don't Repeat Yourself" (não se repita). É importante eliminar [**duplicações**](http://c2.com/cgi/wiki?DuplicatedCode) para tornar nosso código mais claro e mais fácil de manter. Antes de prosseguirmos com o desenvolvimento, faremos algumas **[refatorações](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** simples, começando por eliminar essa [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode).

[**Refatorar**](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao) é uma prática comum em [**Extreme Programming**](https://www.desenvolvimentoagil.com.br/xp) e significa alterar o código, sem alterar o que ele faz. Trata-se de uma mudança efetuada apenas para melhorar a estrutura do código, tornando-o mais simples, mais legível e, portanto, mais fácil de manter. A razão mais comum para **[refatorar](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** é a identificação de [**duplicações**](http://c2.com/cgi/wiki?DuplicatedCode). Elas são danosas porque quando temos que alterar algo que está duplicado, nosso trabalho é maior, pois a alteração tem que ser feita em vários lugares, ao invés de um só. Além disso, o potencial de erros se eleva. Por exemplo, quando alteramos um trecho de código que se repete em dez lugares diferentes, existe a chance de alterarmos em quase todos os lugares e esquecermos um, o que normalmente gera um erro.

Para retirarmos a [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode) identificada, vamos [**extrair um método**](http://www.refactoring.com/catalog/extractMethod.html). Trata-se de uma **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** na qual isolamos um trecho do código, que se repete em vários lugares, em um método que possa ser chamado em cada um dos lugares nos quais o trecho de código era utilizado. Assim, qualquer alteração nesse trecho de código pode ser feita em um único lugar e afetará todos os pontos da aplicação que o utilizam.

Para solucionar a [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode), extraímos o método verificaPrimosGerados() que você encontra na [**Listagem 5**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#5). Ele recebe como parâmetro a lista de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) que espera-se que seja produzida e o número máximo até o qual se deve procurar por [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos).

Voltando ao problema da variável i, no código de geração de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), a renomeamos para valorMaximo. Note que o número 2 se comporta como um "[**número mágico**](http://www.refactoring.com/catalog/replaceMagicNumberWithSymbolicConstant.html)" neste código: apenas lendo-o não temos como saber qual é o seu significado exato no programa. [**Literais**](http://en.wikipedia.org/wiki/String_literal) espalhados ao longo do código frequentemente têm essa característica: quem escreve o código é capaz de compreendê-los (pelo menos logo depois de escrever), mas outros programadores não conseguem entender seu significado rapidamente, o que prejudica a [**manutenção**](http://en.wikipedia.org/wiki/Software_maintenance). Resolvemos isso introduzindo uma [**constante**](http://en.wikipedia.org/wiki/Variable#Constant), cujo nome expressa o significado do número. Veja a [**Listagem 6**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#6).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. import junit.framework.TestCase;
3. public class GeradorPrimosTeste extends TestCase {
4. public void testePrimosGeradosAteNumeroDois() throws Exception {
5. verificaPrimosGerados("2", 2);
6. }
8. public void testePrimosGeradosAteNumeroTres() throws Exception {
9. verificaPrimosGerados("2, 3", 3);
10. }
12. private void verificaPrimosGerados(String listaEsperada, int numeroMaximo)
13. throws Exception {
14. GeradorPrimos geradorPrimos = new GeradorPrimos();
15. assertEquals(listaEsperada, geradorPrimos.gerarPrimosAte(numeroMaximo));
16. }
17. }

**Listagem 5**: [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode) de código eliminada com verificaPrimosGerados().

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public class GeradorPrimos {
2. public static final int MENOR\_PRIMO = 2;
4. public String gerarPrimosAte(int valorMaximo) {
5. if (valorMaximo == MENOR\_PRIMO)
6. return "2";
7. else
8. return "2, 3";
9. }
10. }

**Listagem 6**: introdução de uma [**constante**](http://en.wikipedia.org/wiki/Variable#Constant) no lugar do [**literal**](http://en.wikipedia.org/wiki/String_literal) 2.

## Testando erros

Agora que o código de teste e o código do gerador de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) estão mais organizados, podemos avançar com segurança na solução do problema. Antes de verificar o que acontece ao tentar gerar [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até 4, percebemos que não existem testes para o caso de algum usuário tentar utilizar como valor máximo um número menor que 2. Em princípio, faria pouco sentido, mas é importante tratar essa possibilidade.

O comportamento desejado para esses casos é que o método lance a exceção ValorMaximoInvalidoException. Utilizamos um teste para expressar esse comportamento, como demonstrado na [**Listagem 7**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#7). A estrutura de teste apresentada no método testeSeRejeitaValorMaximoUm() é tipicamente utilizada quando se deseja assegurar que um código lance uma exceção sob certas circunstâncias.

Analisando o processamento desse método, note que, se o gerador de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) estiver funcionando corretamente, a chamada geradorPrimos.gerarPrimosAte(1) irá lançar a exceção esperada e o processamento será desviado para o bloco catch. Lá dentro, a instrução assertTrue(true) serve apenas para informar ao leitor deste código que alcançar o bloco catch é o comportamento esperado do teste.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeSeRejeitaValorMaximoUm() throws Exception {
2. GeradorPrimos geradorPrimos = new GeradorPrimos();
3. try {
4. geradorPrimos.gerarPrimosAte(1);
5. fail("Deveria ter lancado ValorMaximoInvalidoException");
6. } catch (ValorMaximoInvalidoException e) {
7. assertTrue(true);
8. }
9. }

**Listagem 7**: testa se gera exceção quando um valor máximo inválido é informado.

Se a exceção não for lançada, a linha contendo a instrução fail() será executada forçando o **[JUnit](http://www.junit.org/)** a apresentar uma falha com a descrição passada ao método fail(): "Deveria ter lançado ValorMaximoInvalidoException". Inicialmente esse código não é [**compilável**](http://pt.wikipedia.org/wiki/Compilador), pois a classe ValorMaximoInvalidoException ainda não existe. Vamos criá-la:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public class ValorMaximoInvalidoException extends Exception {
2. public ValorMaximoInvalidoException() {
3. super("O valor maximo deve ser maior ou igual a 2");
4. }
5. }

Finalmente, para que o código compile, é necessário que o gerador de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) declare lançar esta exceção:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public String gerarPrimosAte(int valorMaximo)
2. throws ValorMaximoInvalidoException {
3. (...)

Agora o teste compila, e gera a falha abaixo quando executado:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. testeSeRejeitaValorMaximoUm()
2. junit.framework.AssertionFailedError: Deveria ter lancado
3. ValorMaximoInvalidoException

Como sempre, primeiro esperamos que o teste falhe, depois o fazemos passar. Fazendo a correção apresentada na [**Listagem 8**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#8), o teste funciona.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public String gerarPrimosAte(int valorMaximo)
2. throws ValorMaximoInvalidoException {
3. if (valorMaximo < MENOR\_PRIMO)
4. throw new ValorMaximoInvalidoException();
5. if (valorMaximo == MENOR\_PRIMO)
6. return "2";
7. else
8. return "2, 3";
9. }
10. }

**Listagem 8**: lança exceção quando o valor máximo é inválido.

## Refatorando os testes

O novo teste, introduzido na [**Listagem 7**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#7), [**duplicou**](http://c2.com/cgi/wiki?DuplicatedCode) a instanciação da variável geradorPrimos. Para evitar esse problema, **[refatoramos](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** o teste, tornando a variável um atributo da classe. Veja a [**Listagem 9**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#9) e compare-a com os métodos apresentados nas Listagens [**5**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#5) e [**7**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#7).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. import junit.framework.TestCase;
3. public class GeradorPrimosTeste extends TestCase {
5. GeradorPrimos geradorPrimos = new GeradorPrimos();
7. public void testePrimosGeradosAteNumeroDois() throws Exception {
8. verificaPrimosGerados("2", 2);
9. }
11. public void testePrimosGeradosAteNumeroTres() throws Exception {
12. verificaPrimosGerados("2, 3", 3);
13. }
15. private void verificaPrimosGerados(String listaEsperada, int numeroMaximo) throws
16. Exception {
17. GeradorPrimos geradorPrimos = new GeradorPrimos();
18. assertEquals(listaEsperada,
19. geradorPrimos.gerarPrimosAte(numeroMaximo));
20. }
22. public void testeSeRejeitaValorMaximoUm() throws Exception {
23. try {
24. geradorPrimos.gerarPrimosAte(1);
25. fail("Deveria ter lancado ValorMaximoInvalidoException");
26. } catch (ValorMaximoInvalidoException e) {
27. assertTrue(true);
28. }
29. }
30. }

**Listagem 9**: elimina [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode) da instanciação da variável geradorPrimos introduzindo um atributo na classe.

Analisando a classe GeradorPrimos, notamos que a legibilidade do método gerarPrimosAte() foi prejudicada, porque ele cuida primeiro da exceção e depois se preocupa com o processamento do roteiro que seria natural caso o parâmetro de entrada tivesse sido válido. É recomendável que os métodos primeiro cuidem do roteiro natural de processamento e depois tratem os casos excepcionais. Resolvemos isso com a **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** apresentada na [**Listagem 10**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#10).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public String gerarPrimosAte(int valorMaximo)
2. throws ValorMaximoInvalidoException {
3. if (valorMaximo > MENOR\_PRIMO) {
4. if (valorMaximo == MENOR\_PRIMO)
5. return "2";
6. else
7. return "2, 3";
8. }
9. else {
10. throw new ValorMaximoInvalidoException();
11. }
12. }

**Listagem 10**: **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** para tratar a exceção após o roteiro natural de execução do método.

Ao fazer **[refatorações](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)**, devemos sair de um estado no qual todos os testes estão passando para outro no qual os testes continuem funcionando. Para verificar isso, sempre executamos todos os testes após **[refatorar](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** o código. Fazendo isso obtemos a resposta abaixo:

testePrimosGeradosAteNumeroDois()

Error:

ValorMaximoInvalidoException: O valor maximo deve ser maior ou igual a 2

Opa! Ao **[refatorar](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)**, o código deixou de funcionar. Sorte nossa termos um teste para apontar o problema imediatamente! Aparentemente, o método não consegue mais gerar primos até 2. Analisando melhor o que foi feito, podemos notar o uso de if (valorMaximo > MENOR\_PRIMO), enquanto o correto teria sido if (valorMaximo >= MENOR\_PRIMO), um erro comum fruto de falta de atenção. Fazendo-se essa pequena correção, todos os testes voltaram a funcionar.

Chegamos até aqui para assegurar que o gerador de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) rejeita números menores que 2 como entrada. Acabamos de verificar que isso já está sendo feito para o caso do número 1. Mas, será que o mesmo acontece para zero e números negativos? A melhor forma de saber é escrevendo mais um teste, como o apresentado na [**Listagem 11**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#11). Esse teste funciona de primeira, mas para isso, [**duplicamos**](http://c2.com/cgi/wiki?DuplicatedCode) muito código. Podemos solucionar isso rapidamente [**extraindo um método**](http://www.refactoring.com/catalog/extractMethod.html), como o mostrado na [**Listagem 12**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#12).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeSeRejeitaValorMaximoZero() throws Exception {
2. GeradorPrimos geradorPrimos = new GeradorPrimos();
3. try {
4. geradorPrimos.gerarPrimosAte(0);
5. fail("Deveria ter lancado ValorMaximoInvalidoException");
6. } catch (ValorMaximoInvalidoException e) {
7. assertTrue(true);
8. }
9. }

**Listagem 11**: testa se lança exceção quando se tenta gerar [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até o valor máximo zero.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeSeRejeitaValorMaximoZero() throws Exception {
2. verificaSeRejeitaNumerosMenoresQueDois(0);
3. }
5. public void testeSeRejeitaValorMaximoUm() throws Exception {
6. verificaSeRejeitaNumerosMenoresQueDois(1);
7. }
9. private void verificaSeRejeitaNumerosMenoresQueDois(int valorMaximo) {
10. try {
11. geradorPrimos.gerarPrimosAte(valorMaximo);
12. fail("Deveria ter lancado ValorMaximoInvalidoException");
13. } catch (ValorMaximoInvalidoException e) {
14. assertTrue(true);
15. }
16. }

**Listagem 12**: **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** para eliminar a [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode) no método que testa a rejeição de valores máximos inválidos.

Continuando com os testes, devemos verificar também se o programa rejeita números negativos. Para isso, adicionamos um teste para verificar o caso do -1. Agora, que já fizemos uma **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** que deu origem ao método verificaSeRejeitaNumerosMenoresQueDois() começamos a colher frutos, já que esse novo teste se revela trivial:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeSeRejeitaValorMaximoNegativo() throws Exception {
2. verificaSeRejeitaNumerosMenoresQueDois(-1);
3. }

O tempo investido **[refatorando](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** permitiu adicionar outro teste de maneira mais rápida, mantendo o código organizado. Como se vê, a **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** normalmente demanda um pequeno investimento inicial, porém gera economia de tempo futura, mantendo o código organizado. É comum ocorrer situações em que [**extraímos métodos**](http://www.refactoring.com/catalog/extractMethod.html) como o verificaSeRejeitaNumerosMenoresQueDois(), que são então utilizados inúmeras vezes em uma mesma classe de teste. Nestes casos, especialmente, deixar de **[refatorar](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** é mais custoso e o código fica mais difícil de compreender e [**manter**](http://en.wikipedia.org/wiki/Software_maintenance).

Poderíamos adicionar outros testes para números negativos, mas não parece que sejam necessários. Iremos inferir que o que temos até o momento é suficiente para cuidar dos casos em que o valor máximo informado tenha que ser rejeitado.

## Até onde testar?

Podemos prosseguir com o desenvolvimento do gerador de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), tentando fazer com que ele gere números além do valor máximo 3. Para isso, criamos um novo teste verificando se o programa funciona para o valor máximo 4:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testePrimosGeradosAteNumeroQuatro() throws Exception {
2. verificaPrimosGerados("2, 3", 4);
3. }

Mais uma vez, colhemos os frutos da refatoração, pois o método verificaPrimosGerados(), extraído mais cedo, nos ajudou a criar esse novo método de teste mais facilmente. Poderíamos esperar que o teste falhasse, porque a classe ainda não estava preparada para gerar [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até 4, mas o teste funcionou. Olhando melhor o código, observa-se que isso faz sentido, embora o caso não tivesse sido previsto. De qualquer forma, será difícil passar no teste a seguir:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testePrimosGeradosAteNumeroCinco() throws Exception {
2. verificaPrimosGerados("2, 3, 5", 5);
3. }

O teste falha com a seguinte mensagem:

testePrimosGeradosAteNumeroCinco()

junit.framework.ComparisonFailure: expected:<..., 5> but was:<...>;

Agora é hora de implementar o [**Crivo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php). Começamos isolando o código que irá gerar os [**números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) em um novo método chamado numerosPrimos(), conforme a [**Listagem 13**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#13).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public String gerarPrimosAte(int valorMaximo) throws ValorMaximoInvalidoException {
2. if (valorMaximo >= MENOR\_PRIMO) {
3. return numerosPrimos(valorMaximo);
4. } else {
5. throw new ValorMaximoInvalidoException();
6. }
7. }
9. private String numerosPrimos(int valorMaximo) {
10. boolean [] candidatos = inicializaListaCandidatos(valorMaximo);
11. if (valorMaximo == MENOR\_PRIMO)
12. return "2";
13. else
14. return "2, 3";
15. }

**Listagem 13**: **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** para isolar o método que acomodará a parte principal do [**Algoritmo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php).

Vamos precisar de uma lista representando possíveis candidatos de [**números primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos). Usamos um vetor de booleanos, no qual true indica que o número é [**primo**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos). Por exemplo, candidatos[3] = true indica que o número três é um [**primo**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos). Para inicializar esse vetor, criamos o método inicializaListaCanditados(), que retorna o vetor preenchido com true em todas as suas posições, exceto às referentes ao número zero e um, números que já sabemos antecipadamente que não são [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos). Seguindo o TDD, começamos a implementação do método a partir de um teste.

Há um pequeno inconveniente. O método deveria ser privado, pois não há necessidade de torná-lo público; também não seria bom fazê-lo apenas para torná-lo testável. Entretanto, se for mantido privado, será difícil testá-lo a partir da classe de teste. É possível resolver esse problema flexibilizando as restrições. Ao invés de mantê-lo privado, faremos com que tenha visibilidade de pacote. Assim, se a classe de teste for colocada no mesmo pacote da classe que desejamos testar, será possível acessar o método através do teste, sem a necessidade de torná-lo público. A [**Listagem 14**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#14) apresenta o teste escrito para esse método.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeInicializacaoListaCandidatos() throws Exception {
2. int valorMaximo = 5;
3. boolean [] candidatos =
4. geradorPrimos.inicializaListaCandidatos(valorMaximo);
5. assertFalse(candidatos[0]);
6. assertFalse(candidatos[1]);
7. for (int i = GeradorPrimos.MENOR\_PRIMO; i <= valorMaximo; i++) {
8. assertTrue(candidatos[i]);
9. }
10. }

**Listagem 14**: testa a inicialização do array de números candidatos a [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos).

Nossa implementação inicial do método inicializaListaCandidatos() é apresentada abaixo:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. boolean[] inicializaListaCandidatos(int valorMaximo) {
2. return null;
3. }

O código compila, mas o teste não passa, como era de se esperar inicialmente. O **[JUnit](http://www.junit.org/)** gera uma barra vermelha com a seguinte mensagem:

testeInicializacaoListaCandidatos()

Error:

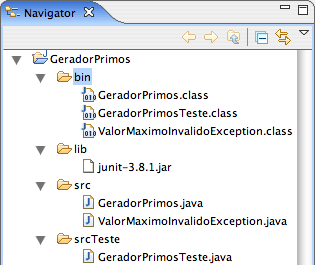
java.lang.NullPointerException

Lembre-se que a primeira coisa que buscamos quando criamos um novo método de teste é assegurar que ele gere uma barra vermelha, introduzindo um erro proposital na classe que estamos testando. Isso foi feito neste caso quando fizemos o método inicializaListaCandidatos() retornar null.

## Localização de Classes

É possível colocar classes de testes no mesmo pacote no qual se encontra a classe a ser testada, permitindo acessar métodos e atributos com visibilidade de pacote. Entretanto, há o inconveniente de misturar classes da aplicação e de teste no mesmo pacote.

Normalmente é preferível manter as classes separadas e utilizar uma técnica para continuar se beneficiando da visibilidade de pacote. Basta adotar duas raízes de código fonte, com a mesma estrutura, fazendo com que o código gerado por ambas seja direcionado para uma mesma raiz de **[bytecodes](http://pt.wikipedia.org/wiki/Bytecode)** (.class). Veja, por exemplo, a organização dos arquivos deste projeto no [**Eclipse**](http://www.eclipse.org/), na [**Figura 5**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#figura5).

**  
Figura 5**. Organização de pacotes no [**Eclipse**](http://www.eclipse.org/).

Depois de assegurar que o teste é capaz de detectar um erro proposital, queremos que ele passe quando retornamos a resposta esperada da forma mais óbvia possível:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. boolean[] inicializaLista
2. Candidatos(int valorMaximo) {
3. return new boolean[] {false, false, true, true, true, true};
4. }

O teste passou como esperado. Agora que temos confiança de que o teste está correto, é hora de implementar o método inicializaListaCandidatos(). A [**Listagem 15**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#15) mostra a implementação que criamos, que, a princípio, parece estar correta. Vejamos se o teste confirma isso. Infelizmente ele falhou com uma mensagem enigmática:

testeInicializacaoListaCandidatos()

junit.framework.AssertionFailedError

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. boolean[] inicializaListaCandidatos(int valorMaximo) {
2. boolean [] resultado = new boolean[valorMaximo];
3. resultado[0] = resultado [1] = false;
4. for (int i = 0; i < resultado.length; i++) {
5. resultado[i] = true;
6. }
7. return resultado;
8. }

**Listagem 15**: inicializa vetor com candidatos a [**número primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos).

Erramos no código da classe e felizmente o teste detectou isso. Porém, a mensagem que o **[JUnit](http://www.junit.org/)** forneceu nos dá poucas pistas sobre o que deu errado. Podemos melhorar o teste para tentar identificar a causa do erro mais facilmente. Veja as mudanças que fizemos no teste mostradas na [**Listagem 16**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#16).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeInicializacaoListaCandidatos() throws Exception {
2. int valorMaximo = 5;
3. boolean [] candidatos =
4. geradorPrimos.inicializaListaCandidatos(valorMaximo);
5. assertEquals("candidatos[0]", false, candidatos[0]);
6. assertEquals("candidatos[1]", false, candidatos[1]);
7. for (int i = GeradorPrimos.MENOR\_PRIMO; i <= valorMaximo; i++) {
8. assertEquals("candidatos[" + i + "]:", true, candidatos[i]);
9. }
10. }

**Listagem 16**: adiciona mensagem ao assert para facilitar a [**depuração de erros**](http://en.wikipedia.org/wiki/Debug).

## Melhorando o feedback das falhas nos testes

O **[JUnit](http://www.junit.org/)** permite especificar uma mensagem a ser exibida quando ocorre uma falha, para facilitar a [**depuração**](http://en.wikipedia.org/wiki/Debug). No caso do assertFalse(), por exemplo, existem duas sobrecargas:

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. assertFalse(boolean condição)
2. assertFalse(String mensagem, boolean condição)

Isso também ocorre com o método assertEquals().

[**view plainprint?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. assertEquals(Object valorEsperado, Object valorObtido)
2. assertEquals(String mensagem, Object valorEsperado, Object valorObtido)

Na primeira versão do teste de inicialização, na [**Listagem 14**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#14), foi usada a forma mais simples do assertFalse(), mas ela não ajudou muito na [**depuração**](http://en.wikipedia.org/wiki/Debug). Para melhorar o feedback do código, substituiremos o assertFalse() por assertEquals(String mensage, Object valorEsperado, Object valorObtido). Executando o teste com a nova implementação, apresentada na [**Listagem 16**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#16), o **[JUnit](http://www.junit.org/)** informa:

testeInicializacaoListaCandidatos()

junit.framework.AssertionFailedError: candidatos[0] expected:<false>; but was:<true>;

Agora está mais fácil identificar o problema, sabemos exatamente a posição do vetor preenchida com o valor incorreto. De alguma forma, estamos inicializando a posição [0] com true, quando deveria ser false. Analisando o código da classe GeradorPrimos, observamos que foi utilizada uma instrução for cuja variável i começa assumindo o valor 0. O correto seria ela começar com o valor 2, que representa o menor [**número primo**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos). A correção é apresentada na [**Listagem 17**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#17).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. boolean[] inicializaListaCandidatos(int valorMaximo) {
2. boolean [] resultado = new boolean[valorMaximo];
3. resultado[0] = resultado [1] = false;
4. for (int i = MENOR\_PRIMO; i < resultado.length; i++) {
5. resultado[i] = true;
6. }
7. return resultado;
8. }

**Listagem 17**: corrige inicialização do método inicializaListaCandidatos().

Executando o teste novamente, ainda encontramos um erro:

testeInicializacaoListaCandidatos()

java.lang.ArrayIndexOutOfBoundsException: 5

Inicializamos o vetor com menos posições do que o necessário. Precisamos lembrar que a contagem começa em zero e termina no valor máximo. Portanto, o número de posições no vetor tem que ser o valor máximo + 1. A correção apresentada na <%= link\_to 'Listagem 18', '/xp/praticas/tdd#18' %> resolve essa questão fazendo o código passar no teste.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. boolean[] inicializaListaCandidatos(int valorMaximo) {
2. boolean [] resultado = new boolean[valorMaximo + 1];
3. resultado[0] = resultado [1] = false;
4. for (int i = 0; i < resultado.length; i++) {
5. resultado[i] = true;
6. }
7. return resultado;
8. }

**Listagem 18**: corrige o tamanho do vetor de candidatos a [**número primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos).

Embora a modificação no teste tenha ajudado, o código de teste ainda tem [**duplicações**](http://c2.com/cgi/wiki?DuplicatedCode). **[Refatorando](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** rapidamente, chegamos ao código da [**Listagem 19**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#19). Assim está melhor. Agora, se tivermos que fazer qualquer alteração na mensagem utilizada para [**depuração**](http://en.wikipedia.org/wiki/Debug), por exemplo, teremos de mudar apenas no método verificaSeCandidatoTemValorEsperado() (isso de fato acabará acontecendo mais adiante e a **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** se mostrará benéfica mais uma vez). Da forma como estava antes, seria necessário alterar em três lugares diferentes.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testeInicializacaoListaCandidatos() throws Exception {
2. int valorMaximo = 5;
3. boolean [] candidatos =
4. geradorPrimos.inicializaListaCandidatos(valorMaximo);
5. verificaSeCandidatoTemValorEsperado(0, false, candidatos[0]);
6. verificaSeCandidatoTemValorEsperado(1, false, candidatos[1]);
7. for (int i = GeradorPrimos.MENOR\_PRIMO; i <= valorMaximo; i++) {
8. verificaSeCandidatoTemValorEsperado(i, true, candidatos[i]);
9. }
10. }
12. private void verificaSeCandidatoTemValorEsperado(int i,
13. boolean valorEsperado, boolean candidato) {
14. assertEquals("candidatos[" + i + "]:", valorEsperado, candidato);
15. }

**Listagem 19**: **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** para retirar [**duplicação**](http://c2.com/cgi/wiki?DuplicatedCode).

Para que o programa fique completo, só falta terminar de implementar o método de geração de [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), o que é feito na [**Listagem 20**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#20).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. private String numerosPrimos(int valorMaximo) {
2. boolean [] candidatos = inicializaListaCandidatos(valorMaximo);
3. for (int valor = MENOR\_PRIMO; valor < valorMaximo; valor++) {
4. if (candidatos[valor]) {
5. for (int naoPrimos = MENOR\_PRIMO \* valor;
6. naoPrimos < valorMaximo; naoPrimos += valor) {
7. candidatos[naoPrimos] = false;
8. }
9. }
10. }
11. String resultado = String.valueOf(MENOR\_PRIMO);
12. for (int i = MENOR\_PRIMO + 1; i < valorMaximo; i++) {
13. if (candidatos[i]) {
14. resultado += ", " + i;
15. }
16. }
17. return resultado;
18. }

**Listagem 20**: implementação do [**Algoritmo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php).

## Finalizando o [Crivo de Eratóstenes](http://www.numaboa.com.br/criptologia/matematica/testRapid.php)

O resultado não agrada. O métodop parece correto, mas está grande demais e poderia se beneficiar de uma boa **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)**. Em todo o caso, vejamos se o **[JUnit](http://www.junit.org/)** realmente considera o método correto:

testePrimosGeradosAteNumeroTres()

junit.framework.ComparisonFailure: expected:<..., 3>; but was:<...>;

testPrimosGeradosAteNumeroCinco()

junit.framework.ComparisonFailure: expected:<..., 5>; but was:<...>;

Ele apresentou duas falhas. O problema é que em todas as instruções for, foi usado como limite superior do loop expressões do tipo valor < valorMaximo. Deveria ter sido valor <= valorMaximo. Sendo assim, chegamos ao código apresentado na [**Listagem 21**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#21).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. private String numerosPrimos(int valorMaximo) {
2. boolean [] candidatos = inicializaListaCandidatos(valorMaximo);
3. for (int valor = MENOR\_PRIMO; valor <= valorMaximo; valor++) {
4. if (candidatos[valor]) {
5. for (int naoPrimos = MENOR\_PRIMO \* valor;
6. naoPrimos <= valorMaximo; naoPrimos += valor) {
7. candidatos[naoPrimos] = false;
8. }
9. }
10. }
11. String resultado = String.valueOf(MENOR\_PRIMO);
12. for (int i = MENOR\_PRIMO + 1; i <= valorMaximo; i++) {
13. if (candidatos[i]) {
14. resultado += ", " + i;
15. }
16. }
17. return resultado;
18. }

**Listagem 21**: correção do [**Algoritmo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php).

Finalmente todos os testes passam. Já sabemos que o programa funciona bem para gerar [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos) até o número cinco. Não temos, é claro, como testar todos os possíveis [**primos**](http://pt.wikipedia.org/wiki/N%C3%BAmeros_primos), mas podemos testar para mais alguns, o suficiente para nos deixar mais tranqüilos. Sendo assim, acrescentamos os novos testes indicados na [**Listagem 22**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#22).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testePrimosGeradosAteNumeroDez() throws Exception {
2. verificaPrimosGerados("2, 3, 5, 7", 10);
3. }
4. public void testePrimosGeradosAteNumeroVinteDois() throws Exception {
5. verificaPrimosGerados("2, 3, 5, 7, 11, 13, 17, 19", 22);
6. }

**Listagem 22**: novos cenários de testes.

Esses testes também passaram. Podemos inferir que continuará funcionando para números maiores. Mas há uma última questão: o método numerosPrimos() está muito grande. Ele tem mais responsabilidades do que deveria. Além disso, algumas das variáveis têm nomes que dificultam o entendimento.

Algumas **[refatorações](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** devem resolver o problema. Podemos fazê-las tranqüilamente, pois se errarmos, os testes nos informarão. Como primeiro passo, movemos a responsabilidade de formatar o resultado para outro método, como mostrado na [**Listagem 23**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#23).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. private String numerosPrimos(int valorMaximo) {
2. boolean [] candidatos = inicializaListaCandidatos(valorMaximo);
3. for (int valor = MENOR\_PRIMO; valor <= valorMaximo; valor++) {
4. if (candidatos[valor]) {
5. for (int naoPrimos = MENOR\_PRIMO \* valor;
6. naoPrimos <= valorMaximo; naoPrimos += valor) {
7. candidatos[naoPrimos] = false;
8. }
9. }
10. }
11. return apresentaResultado(valorMaximo, candidatos);
12. }
14. private String apresentaResultado(int valorMaximo, boolean[] candidatos) {
15. String resultado = String.valueOf(MENOR\_PRIMO);
16. for (int i = MENOR\_PRIMO + 1; i <= valorMaximo; i++) {
17. if (candidatos[i]) {
18. resultado += ", " + i;
19. }
20. }
21. return resultado;
22. }

**Listagem 23**: **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** para melhorar a legibilidade do [**Algoritmo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php).

Todos os testes continuam funcionando. Agora é preciso dar um jeito na variável candidatos[]. Esse nome agora parece inadequado. Como se trata de um vetor de booleanos, seria melhor um nome do tipo ehPrimo[], como usado na [**Listagem 24**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#24). Note como fica mais fácil compreender o código depois dessa mudança.

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. private String numerosPrimos(int valorMaximo) {
2. boolean [] ehPrimo = inicializaListaCandidatos(valorMaximo);
3. for (int valor = MENOR\_PRIMO; valor <= valorMaximo; valor++) {
4. if (ehPrimo[valor]) {
5. for (int naoPrimos = MENOR\_PRIMO \* valor;
6. naoPrimos <= valorMaximo; naoPrimos += valor) {
7. ehPrimo[naoPrimos] = false;
8. }
9. }
10. }
11. return apresentaResultado(valorMaximo, ehPrimo);
12. }
14. private String apresentaResultado(int valorMaximo, boolean[] ehPrimo) {
15. String resultado = String.valueOf(MENOR\_PRIMO);
16. for (int i = MENOR\_PRIMO + 1; i <= valorMaximo; i++) {
17. if (ehPrimo[i]) {
18. resultado += ", " + i;
19. }
20. }
21. return resultado;
22. }
24. boolean[] inicializaListaDePrimosPotenciais(int valorMaximo) {
25. boolean [] resultado = new boolean[valorMaximo + 1];
26. resultado[0] = resultado [1] = false;
27. for (int i = MENOR\_PRIMO; i < resultado.length; i++) {
28. resultado[i] = true;
29. }
30. return resultado;
31. }

**Listagem 24**: o vetor candidatos[] foi renomeado para ehPrimo[].

Ao mudar o nome do método inicializaListaCandidatos() para inicializaListaDePrimosPotenciais(), naturalmente um ou mais métodos de teste também tiveram que ser atualizados para utilizar o novo nome.

O código está mais legível e os testes continuam funcionando. Podemos fazer a última **[refatoração](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao)** em um método de teste para levar em conta a mudança no nome do vetor candidatos[] para ehPrimo[]. Veja a [**Listagem 25**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/#25).

**[view plain](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)**[**print?**](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/)

1. public void testInicializacaoListaDePrimosPotenciais() throws Exception {
2. int valorMaximo = 5;
3. boolean [] ehPrimo =
4. geradorPrimos.inicializaListaCandidatos(valorMaximo);
5. verificaSeEhPrimo(0, false, ehPrimo[0]);
6. verificaSeEhPrimo(1, false, ehPrimo[1]);
7. for (int i = GeradorPrimos.MENOR\_PRIMO; i <= valorMaximo; i++) {
8. verificaSeEhPrimo(i, true, ehPrimo[i]);
9. }
10. }
12. private void verificaSeEhPrimo(int i, boolean esperado, boolean numero) {
13. assertEquals("ehPrimo[" + i + "]:", esperado, numero);
14. }

**Listagem 25**: o vetor candidatos[] foi renomeado para ehPrimo[].

## Conclusões

Assim encerramos a implementação do [**Crivo de Eratóstenes**](http://www.numaboa.com.br/criptologia/matematica/testRapid.php). O processo de criação e uso dos testes é mais rápido do que aparenta. Cada passo foi muito simples e levou poucos segundos para ser executado ou no máximo poucos minutos.

Usando TDD, quando acabamos, realmente acabamos. Ou seja, dificilmente temos que retornar ao código futuramente para corrigir falhas, pois possíveis falhas já foram detectadas e corrigidas durante a confecção dos testes. Além disso, se alguém alterar esse código no futuro, os testes irão dizer se a mudança foi bem sucedida ou não. O processo não é infalível, mas códigos gerados assim raramente apresentam problemas.

Lembre-se sempre dos três passos básicos do desenvolvimento orientado a testes:

1. Escrever um teste e assegurar que ele não funcione introduzindo um erro óbvio no código sendo testado.
2. Fazer o teste funcionar com a implementação mais óbvia possível.
3. [**Refatorar**](https://www.desenvolvimentoagil.com.br/xp/praticas/refatoracao) o método sendo testado e o próprio método de teste.O primeiro, para colocar a implementação desejada para a aplicação e o segundo para eliminar [**duplicações**](http://c2.com/cgi/wiki?DuplicatedCode) e melhorar a legibilidade.

No início, trabalhar com TDD pode parecer um pouco doloroso, pois temos que fazer o inverso do que estamos acostumados. Mas, como em todo aprendizado, a dificuldade vem apenas no começo e nos tornamos melhores à medida que praticamos. Pelos problemas que foram descritos no início e o impacto negativo que eles trazem para nós, para nossa indústria e nossos clientes, o esforço certamente é válido!

Desenvolvimento orientado a testes pode se tornar excessivamente trabalhoso devido a dependências entre classes da aplicação. Para lidar com elas, é importante dominar o conceito de **[mock objects](https://www.desenvolvimentoagil.com.br/xp/praticas/tdd/mock_objects)**.