Estratégia Concursos

Aula 00

Engenharia de Software p/ SEFAZ-RJ (Auditor - Área TI) - 2021 - Pré-Edital

Autor:
Diego Carvalho, Equipe Informática e TI

05 de Janeiro de 2021

Sumário

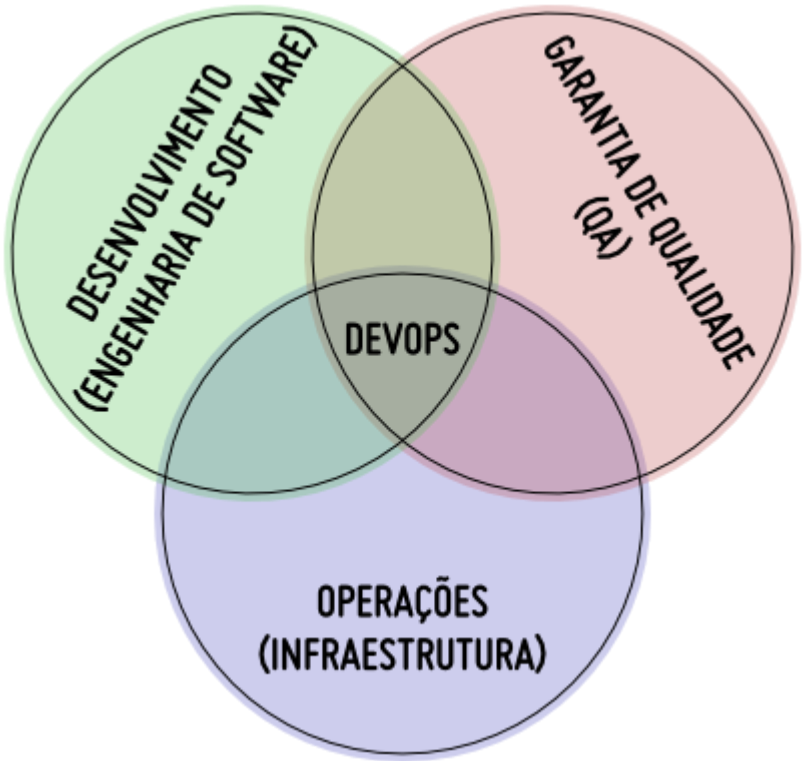
NOÇÕES DE DEVOPS

1 – Conceitos Básicos

De um tempo para cá, nós temos vivenciado novos paradigmas em tecnologias de software. **As metodologias ágeis, por exemplo, vieram com um conjunto de práticas que desencadearam diversas outras práticas, de forma a obter software de maneira mais ágil e mais adaptável a possíveis mudanças**. No entanto, o grande lance é que as mudanças ocorrem com um intervalo de tempo cada vez menor.

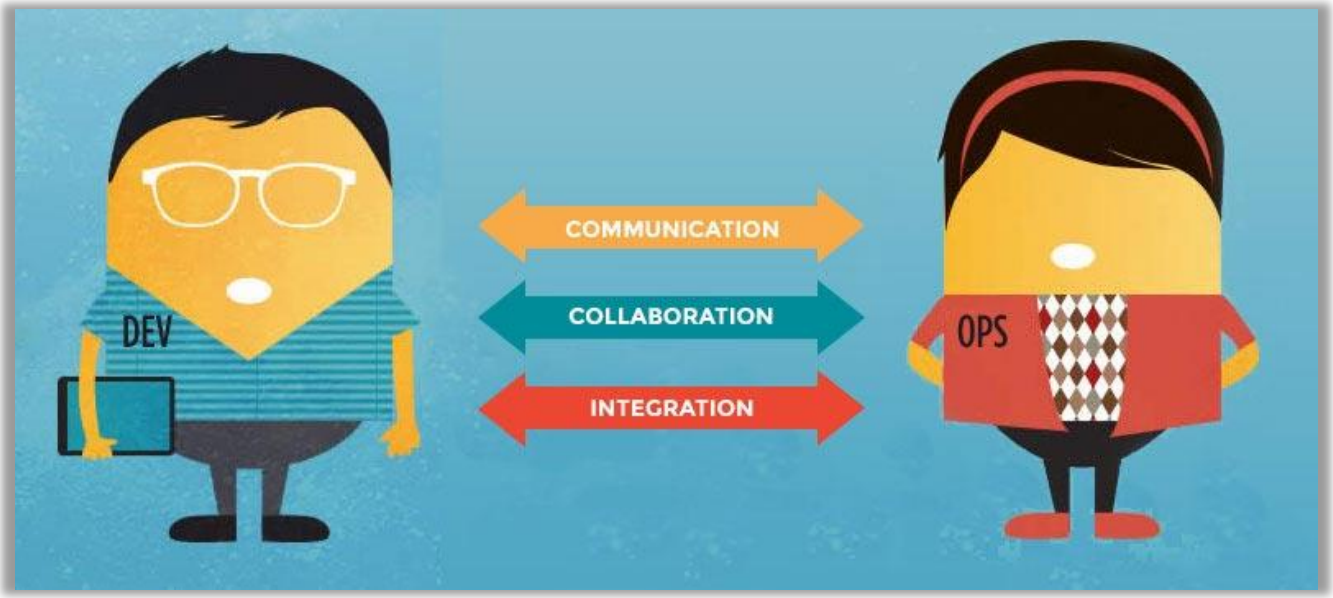
Vocês já devem ter percebido isso! Antigamente, softwares lançavam atualizações em intervalos de 18 a 24 meses (ou até mais). **Atualmente, a dinâmica de consumo de aplicações de tecnologia da informação sofreu uma reviravolta e a demanda dos clientes é insana.** As empresas de software atualmente são duramente pressionadas para lançar e atualizar aplicativos no mercado o mais rápido possível.

Pois é, o mundo mudou! O ciclo para a criação de novos aplicativos de software dura cerca de três meses para uma versão inicial e mais seis meses para o conjunto completo de recursos**. Não só o ciclo de vida foi encurtado, mas os aplicativos se tornaram muito mais complexos e exigem colaboração e integração cruzada entre os diversos componentes de tecnologia da informação, como Dev, Ops e Q&A.**

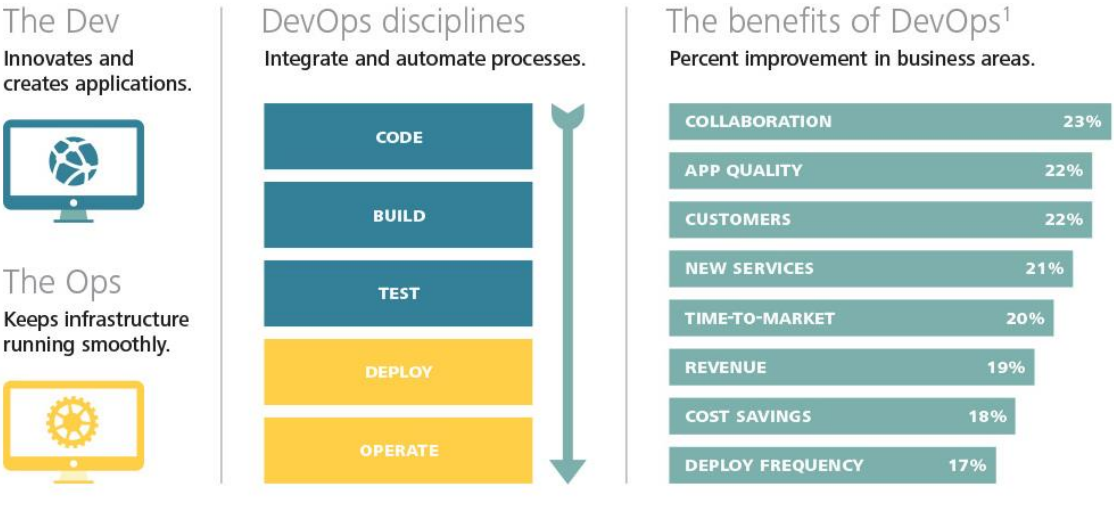


Professor, espera um pouco aí! O que acontece se eu juntar conceitos de Desenvolvimento de Software, Operação de Sistemas e Garantia de Qualidade? Surgirá, então, o conceito de DevOps!

Trata-se de um conceito muito simples, como apresenta a imagem acima. Essas ideias são tratadas em conjunto, em vez de separadas. O lance é ter uma maior comunicação, colaboração e integração entre essas áreas.

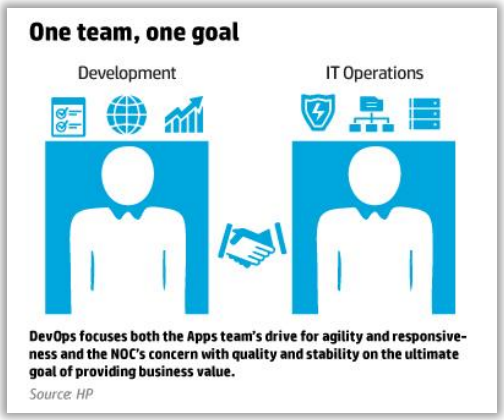


Professor, o que é exatamente DevOps? É uma cultura? É uma técnica? É uma metodologia de desenvolvimento de software? Ainda não existe uma resposta precisa para essas perguntas! *Por que, professor?* **Porque foi um movimento que começou ao mesmo tempo em diversos lugares diferentes, tratando de infraestrutura e desenvolvimento, mas não houve um manifesto formal, como o manifesto ágil.**



Parece simples fazer a infraestrutura conversar de forma harmônica com o desenvolvimento, mas não é tão fácil! *Qual é o papel da infraestrutura?* É sustentar os sistemas em produção; monitorar o funcionamento e a performance; cuidar da estabilidade, segurança, níveis de

serviço; planejar mudanças, minimizando riscos; entre outros. *O que acontece se uma aplicação em produção parar?*



Isso pode significar prejuízo financeiro ou institucional. Em suma, podemos dizer que a infraestrutura se preocupa em proteger o valor de negócio. *E o desenvolvimento?* Esses caras se preocupam com inovação e criatividade, baseado nos requisitos do usuário. Desenvolvedor fica louco quando sai uma biblioteca, componente ou tecnologia nova. A consequência disso é que cada inovação significa um novo Deploy (feito pela rapaziada da infraestrutura).

E se ocorrer algum problema? Deve ser realizado um rollback (também pelo pessoal da infraestrutura). **Podemos afirmar, então, que o desenvolvedor se preocupa em aumentar o valor do negócio.** Vocês já devem ter notado que há um conflito interessante nessa conversa. Ora, o desenvolvedor quer colocar suas aplicações no ar o mais rápido possível para que fique disponível para o cliente.

No entanto, a galera da infraestrutura quer ter certeza de que a aplicação está suficientemente estável para ir para produção sem gerar incidentes que parem o que já está funcionando. *O que ocorria antigamente?* As empresas permitiam apenas um deploy por semana ou por mês! *Tem coisa mais não-ágil que isso?* Vai totalmente de encontro aos ideais do manifesto ágil. *Lembram-se dos conceitos de entrega, integração e teste contínuos?*

Pois é, a infraestrutura teve que se adaptar a realizar deploys diários. No entanto, os desenvolvedores – muitas vezes – se esqueciam de considerar algumas diferenças importantes entre ambientes de desenvolvimento e produção. **Isso gerava alguns incidentes, o cliente reclamava e começava uma briga muito comum representada pelas duas imagens anteriores.** Sim, galera... rola essa briga!

Desenvolvedores afirmando que a Infraestrutura é engessada, lenta e que não oferece um ambiente adequado para o desenvolvimento de aplicações; já a Infraestrutura afirmava que os desenvolvedores faziam código ruim e instável, e que a culpa não era deles. **Pessoal, voltem agora para a primeira imagem e percebam o que DevOps tenta integrar: Desenvolvimento, Infraestrutura e Qualidade!**

Parece briga de marido e mulher, mas ambos os lados têm que reconhecer seus erros, ceder e se adaptar! **O Desenvolvimento precisa pensar mais na Infraestrutura e controlar as fases de deploy (Ex: Deployment Pipeline). Já a Infraestrutura tem que evoluir para o mundo ágil.** Começar a trabalhar de forma automatizada e dinâmica, ser mais veloz para subir ambientes; reconstruir ou duplicar ambientes de acordo com as necessidades do Desenvolvimento.

Galera, as principais características do DevOps são: colaboração entre equipes; fim de divisões; relação saudável entre áreas; teste, integração e entrega contínuos; automação de deploy; controle e monitoração; gerenciamento de configuração; orquestração de serviços; avaliação de métricas e desempenho; logs e integração; velocidade de entrega; feedback intenso; e comunicação constante.

Em suma, podemos dizer que ele é um movimento, um conceito, uma cultura, uma abordagem que trata do feedback, comunicação e colaboração entre áreas de tecnologia da informação com o objetivo de garantir qualidade do software, com menor custo, mais rapidez, menor risco e maior eficiência. **É como se fosse a utilização de metodologias ágeis no desenvolvimento e na infraestrutura.** *Bacana?*

Para o DevOps, o código gerado pela infraestrutura é apenas mais um artefato qualquer de desenvolvimento e, não, uma parte separada. Trata-se de uma prática importante, já que não basta somente o desenvolvedor escrever o código do sistema, é necessário que a área de infraestrutura também atue para liberar, controlar e entregar a versão do sistema de forma contínua, periódica e preferencialmente automática.

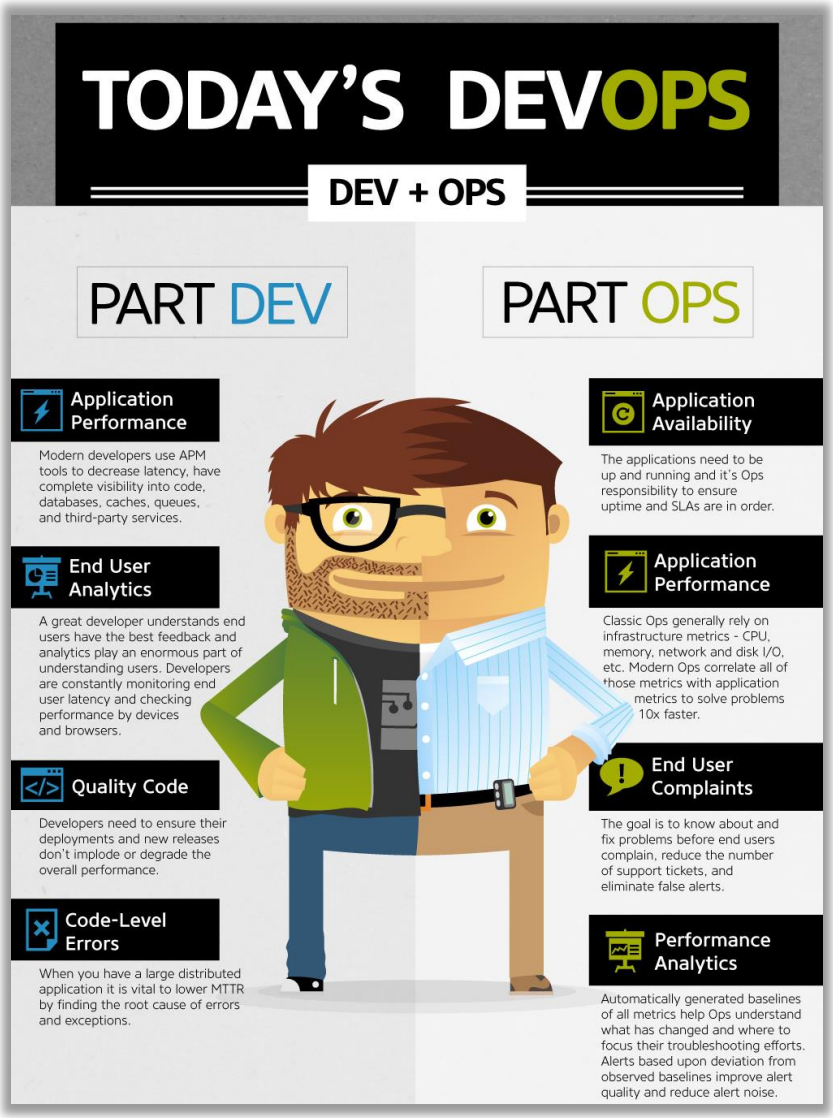
Vamos resumir: **a preocupação é com os objetivos quase diametralmente opostos da galera de Desenvolvimento (DEV) e Operações (OPS).** Os desenvolvedores querem programar novos recursos, melhorar o produto; corrigir bugs, etc. As operações querem colocar tudo em funcionamento e nunca mudar, já que as alterações causam novos erros, bugs, problemas de desempenho, entre outros.

O objetivo do DevOps é aliviar a tensão entre esses dois campos! **Ter pessoas de Operações nas trincheiras de Desenvolvimento é a principal maneira de alcançar esse objetivo.** Seu trabalho

é facilitar ao máximo que desenvolvedores e operações façam o que precisam fazer. *Como assim?* Por exemplo: fornecendo ambientes idênticos de desenvolvimento, testes, homologação, *staging* e qualidade; configuração de pipelines de teste e implementação automáticos.

além de estar envolvido no processo de desenvolvimento para que eles estejam mais preparados para lidar com erros de produção. **Tradicionalmente, as operações são quase alheias à base de código, uma vez que se preocupam apenas com sua infraestrutura.** O objetivo é tornar todo o processo transparente de forma que você possa fazer milhares de implantações de produção por dia; ao contrário dos métodos tradicionais de configuração de uma janela de implantação uma vez a cada trimestre ou por mais tempo.

Claro que para fazer isso, é necessário utilizar um conjunto de práticas e ferramentas. *Quais, professor?* Ferramentas de Controle de versão (Git, CVS, Tortoise), Servidores de Integração Contínua (Jenkins, Bamboo, Travis), Docker/Vagrant, Gerenciamento de Configuração (SaltStack, Chef, Puppet). Isso reduz a dor de cabeça tanto para os desenvolvedores quanto para os operadores e reduz a quantidade de problemas de desempenho e erros de codificação.



(CESPE / STF – 2013) Integração contínua, entrega contínua, teste contínuo, monitoramento contínuo e feedback são algumas práticas do DevOps.

_____

Comentários: as principais características do DevOps são: colaboração entre equipes; fim de divisões; relação saudável entre áreas; **teste, integração e entrega contínuos**; automação de deploy; **controle e monitoração**; gerenciamento de configuração; orquestração de serviços; avaliação de métricas e desempenho; logs e integração; velocidade de entrega; **feedback intenso**; e comunicação constante (Correto).

(CESPE / STF – 2013) Teste contínuo é uma prática do DevOps que, além de permitir a diminuição dos custos finais do teste, ajuda as equipes de desenvolvimento a balancear qualidade e velocidade.

_____

Comentários: teste, integração e entrega contínuos são realmente práticas do DevOps que permitem reduzir custos de teste e ajuda a equipe de desenvolvimento a balancear a qualidade e velocidade (Correto).

(CESPE / TCU – 2015) De acordo com a abordagem DevOps (development – operations), os desafios da produção de software de qualidade devem ser vencidos com o envolvimento dos desenvolvedores na operação dos sistemas com os quais colaboraram no desenvolvimento.

_____

Comentários: essa questão permite duas interpretações de 'envolvimento': (1) no sentido de interação e colaboração entre equipes; (2) no sentido de mão na massa mesmo - na operação dos sistemas. Por conta da ambiguidade, a questão foi anulada (Anulada).

(CESPE / TRE-PE – 2017) O DevOps consiste em:

a) um processo similar ao IRUP (IBM Rational Unified Process), que tem como objetivo dividir o processamento em fases e disciplinas de software para paralelizar as ações de desenvolvimento e de manutenção das soluções.

b) uma plataforma aberta cuja função é substituir a virtualização de aplicações e serviços em containers e, com isso, agilizar a implantação de soluções de software.

c) um aplicativo que permite o gerenciamento de versões de códigos-fonte e versões de programas, bem como a implantação da versão mais recente de um software em caso de falha.

d) um processo de promoção de métodos que objetivam aprimorar a comunicação, tornando a colaboração eficaz especialmente entre os departamentos de desenvolvimento e teste e entre os departamentos de operações e serviço para o negócio.

e) uma metodologia ágil que, assim como a XP (extreme programming) e o Scrum, tem foco na gestão de produtos complexos relativos à equipe de desenvolvimento.

_____

Comentários: DevOps não é um processo similar ao iRUP, não é uma plataforma aberta, não é um aplicativo e não é uma metodologia ágil. DevOps é um processo de promoção de métodos que objetivam aprimorar a comunicação, tornando a colaboração eficaz especialmente entre os departamentos de desenvolvimento e teste e entre os departamentos de operações e serviço para o negócio (Letra D).

(CESPE / TRT-PA e AP – 2016) Acerca de DevOps, assinale a opção correta.

a) O DevOps concentra-se em reunir diferentes processos e executá-los mais rapidamente e com mais frequência, o que gera baixa colaboração entre equipes.

b) O DevOps tem como princípio produzir, a partir da avaliação dos times de desenvolvimento do serviço, grandes mudanças e farta documentação com valor agregado para os usuários, assemelhando-se, por isso, com objetivos dos métodos iterativos e em cascata.

c) A infraestrutura de nuvem de provedores internos e externos vem restringindo o uso de DevOps pelas organizações.

d) O DevOps parte da premissa de adoção de grandes equipes de especialistas, com a menor interação possível, visando à padronização de processos e à mínima automação de atividades.

e) Atividades típicas em DevOps compreendem teste do código automatizado, automação de fluxos de trabalho e da infraestrutura e requerem ambientes de desenvolvimento e produção idênticos.

_____

Comentários: (a) Errado, isso gera alta colaboração entre equipes; (b) Errado, não se assemelha com objetivos de métodos em cascata, visto que essa metodologia possui entregas menos frequentes e é menos dinâmica; (c) Errado, é justamente o inverso – elas usam com cada vez mais frequência; (d) Errado, recomenda-se a maior interação possível entre as equipes; (e) Correto, testes automatizados, automação de fluxos e infraestrutura são atividades frequentes que realmente exigem ambientes de desenvolvimento e produção idênticos (Letra E).

(CESPE / STJ – 2015) DevOps é um conceito pelo qual se busca entregar sistemas melhores, com menor custo, em menor tempo e com menor risco.

_____

Comentários: perfeito, perfeito, perfeito – todas são características do DevOps (Correto).

(CESPE / STJ – 2015) O profissional especialista em DevOps deve atuar e conhecer as áreas de desenvolvimento (engenharia de software), operações e controle de qualidade, além de conhecer, também, de forma ampla, os processos de desenvolvimento ágil.

Comentários: ele é a combinação entre desenvolvimento, operação e controle de qualidade – portanto questão perfeita (Correto).

(CESPE / SLU-DF – 2019) Em DevOps, o princípio monitorar e validar a qualidade operacional antecipa o monitoramento das características funcionais e não funcionais dos sistemas para o início do seu ciclo de vida, quando as métricas de qualidade devem ser capturadas e analisadas.

Comentários: o princípio monitorar e validar a qualidade operacional transfere o monitoramento para o início do ciclo de vida do software, para identificar antecipadamente problemas de qualidade que podem ocorrer no desenvolvimento. Na implantação e teste, as métricas de qualidade são capturadas e analisadas, sendo que essas métricas devem ser entendidas por todos os stakeholders (Correto).

(CESPE / MPE-PI – 2018) A infraestrutura como código é uma prática DevOps caracterizada pela infraestrutura provisionada e gerenciada por meio de técnicas de desenvolvimento de código e de software, como, por exemplo, controle de versão e integração contínua.

Comentários: trata-se realmente de uma prática em que a infraestrutura provisionada e gerenciada por meio de técnicas de desenvolvimento de código e de software – ela oferece controle de versão, entrega e integração contínua (Correto).

(CESPE / STJ – 2018) Apesar de ser um processo com a finalidade de desenvolver, entregar e operar um software, o DevOps é incompatível com a aplicação de métodos ágeis como o Scrum ou, ainda, com o uso de ferramentas que permitam visualizar os fluxos do processo.

Comentários: pelo contrário, é completamente compatível com as aplicações de métodos ágeis como o Scrum ou, ainda, com o uso de ferramentas que permitam visualizar os fluxos do processo (Errado).

(CESPE / STJ – 2018) O gerenciamento de desenvolvimento de software por meio do Scrum pode ser combinado com o ciclo de vida do DevOps, haja vista que o DevOps combina práticas e ferramentas que aumentam a capacidade de uma organização de distribuir aplicativos e serviços; logo, a integração contínua do software pode ser realizada na sprint do Scrum junto com a operação dos serviços da organização.

Comentários: ele realmente combina práticas e ferramentas que aumentam a capacidade de uma organização de distribuir aplicativos e serviços e pode ser realizado na sprint do Scrum por meio da integração contínua (Correto).

TEST DRIVEN DEVELOPMENT (TDD)

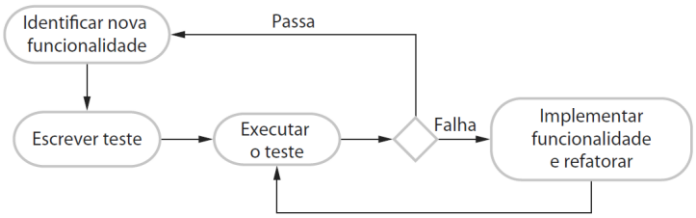
1 – Conceitos Básicos

O Test-Driven Development (TDD) é uma abordagem de desenvolvimento de software em que se intercalam testes e desenvolvimento de código. Essencialmente, você desenvolve um código de forma incremental, em conjunto com um teste para esse incremento. Você não caminha para o próximo incremento até que o código desenvolvido passe no teste. O desenvolvimento dirigido a testes foi apresentado como parte dos métodos ágeis, como o XP.

Por outro lado, ele também pode ser utilizado em processos de desenvolvimento dirigido a planos. Trata-se de uma abordagem que **se baseia na repetição de um ciclo de desenvolvimento curto focado em testes unitários**. A ideia fundamental dessa abordagem consiste em escrever o teste, encontrar uma falha nesse mesmo teste e depois refatorá-lo. Vamos agora ver as etapas do processo de desenvolvimento dirigido a testes:

ETAPA	DESCRIÇÃO
1	Você começa identificando o incremento de funcionalidade necessário. Este, normalmente, deve ser pequeno e implementável em poucas linhas de código.
2	Você escreve um teste para essa funcionalidade e o implementa como um teste automatizado. Isso significa que o teste pode ser executado e relatará se passou ou falhou.
3	Você, então, executa o teste, junto com todos os outros testes implementados. Inicialmente, você não terá implementado a funcionalidade, logo o novo teste falhará. Isso é proposital, pois mostra que o teste acrescenta algo ao conjunto de testes.
4	Você, então, implementa a funcionalidade e executa novamente o teste. Isso pode envolver a refatoração do código existente para melhorá-lo e adicionar um novo código sobre o que já está lá.
5	Depois que todos os testes forem executados com sucesso, você caminha para implementar a próxima parte da funcionalidade.

Como o código é desenvolvido em incrementos muito pequenos, você precisa ser capaz de executar todos os testes cada vez que adicionar funcionalidade ou refatorar o programa. Dessa forma, os testes são embutidos em um programa separado que os executa e invoca o sistema que está sendo testado. Usando essa abordagem, é possível rodar centenas e centenas de testes separados em poucos segundos.



Um argumento forte a favor do desenvolvimento dirigido a testes é que ele ajuda os programadores a clarear suas ideias sobre o que um segmento de código supostamente deve fazer**. Para escrever um teste, você precisa entender a que ele se destina, e como esse entendimento faz que seja mais fácil escrever o código necessário.** Certamente, se você tem conhecimento ou compreensão incompleta, o desenvolvimento dirigido a testes não ajudará.

Se você não sabe o suficiente para escrever os testes, não vai desenvolver o código necessário. Por exemplo: se seu cálculo envolve divisão, você deve verificar se não está dividindo o número por zero. Se você se esquecer de escrever um teste para isso, então o código para essa verificação nunca será incluído no programa. Além de um melhor entendimento do problema, outros benefícios do desenvolvimento dirigido a testes são:

1. **Cobertura de código:** em princípio, todo segmento de código que você escreve deve ter pelo menos um teste associado. Assim, você pode ter certeza de que todo o código no sistema foi realmente executado. Cada código é testado enquanto está sendo escrito; assim, os defeitos são descobertos no início do processo de desenvolvimento.

2. **Teste de Regressão:** um conjunto de testes é desenvolvido de forma incremental enquanto um programa é desenvolvido. Você sempre pode executar testes de regressão para verificar se as mudanças no programa não introduziram novos bugs.

3. **Depuração Simplificada:** quando um teste falha, a localização do problema deve ser óbvia. O código recém-escrito precisa ser verificado e modificado. Você não precisa usar as ferramentas de depuração para localizar o problema. Alguns relatos de uso de desenvolvimento dirigido a testes sugerem que, em desenvolvimento dirigido a testes, quase nunca é necessário usar um sistema automatizado de depuração.

4. **Documentação de Sistema:** os testes em si mesmos agem como uma forma de documentação que descreve o que o código deve estar fazendo. Ler os testes pode tornar mais fácil a compreensão do código.

Um dos benefícios mais importantes de desenvolvimento dirigido a testes é que ele reduz os custos dos testes de regressão. O teste de regressão envolve a execução de conjuntos de testes que tenham sido executados com sucesso, após as alterações serem feitas em um sistema. Ele também verifica se essas mudanças não introduziram novos bugs no sistema e se o novo código interage com o código existente conforme o esperado.

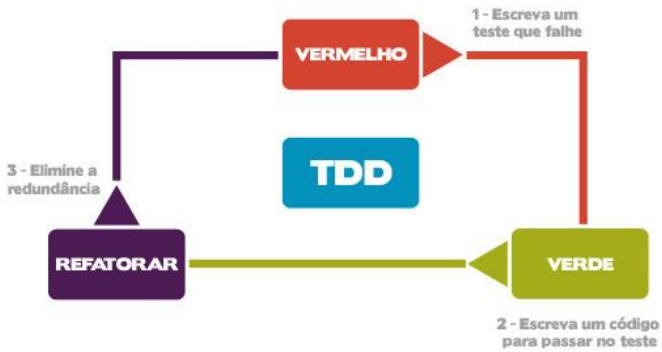
O teste de regressão é muito caro e geralmente impraticável quando um sistema é testado manualmente, pois os custos com tempo e esforço são muito altos. Em tais situações, você precisa tentar escolher os testes mais relevantes para executar novamente, e é fácil perder testes importantes. No entanto, testes automatizados – fundamentais para o desenvolvimento *test-first* – reduzem drasticamente os custos com testes de regressão.

Os testes existentes podem ser executados novamente de forma rápida e barata. **Após se fazer uma mudança para um sistema em desenvolvimento *test-first*, todos os testes existentes devem ser executados com êxito antes de qualquer funcionalidade ser adicionada.** Como um programador, você precisa ter certeza de que a nova funcionalidade não tenha causado ou revelado problemas com o código existente.

O desenvolvimento dirigido a testes é de maior utilidade no desenvolvimento de softwares novos, em que a funcionalidade seja implementada no novo código ou usando bibliotecas-padrão já testadas. **Se você estiver reusando componentes de código ou sistemas legados grandes, você precisa escrever testes para esses sistemas como um todo.** O desenvolvimento dirigido a testes também pode ser ineficaz em sistemas multi-threaded.

As *threads* diferentes podem ser intercalados em tempos diferentes, em execuções diferentes, e isso pode produzir resultados diferentes. Se você usa o desenvolvimento dirigido a testes, ainda precisa de um processo de teste de sistema para validar o sistema, isto é, verificar se atende aos requisitos dos stakeholders. O teste de sistema também testa o desempenho, a confiabilidade, e verifica se o sistema não faz coisas que não deveria, como produzir resultados indesejados etc.

O desenvolvimento dirigido a testes revelou-se uma abordagem de sucesso para projetos de pequenas e médias empresas. Geralmente, os programadores que adotaram essa abordagem estão satisfeitos com ela e acham que é uma maneira mais produtiva de desenvolver softwares. Em alguns experimentos, foi mostrado que essa abordagem gera melhorias na qualidade do código. Bem, agora vamos falar sobre o ciclo vermelho, verde e refatoração.



ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

Vamos lá! Nós sabemos que – para cada parte da aplicação – adiciona-se um teste escrito antes mesmo do desenvolvimento do código em si. *Por que?* **Porque eles podem ajudar a reduzir riscos de possíveis problemas no código.** Executamos o teste e ele... falha! Ele deve

necessariamente falhar! *Por que?* Ora, porque ele é o primeiro teste e você nem criou a funcionalidade ainda, logo ele não irá funcionar!

Então nós adicionamos uma nova funcionalidade ao sistema apenas para que ele passe no teste e execute novamente (agora ele deve passar no teste). **Então, nós adicionamos um novo teste e rodamos o teste anterior e esse novo teste.** Se algum deles falhar, modifica-se o código da funcionalidade e rodam-se todos os testes novamente, e assim por diante – nós já vimos aquela imagem anterior que mostra como tudo isso funciona...

Galera, vocês percebem que o feedback sobre a nova funcionalidade ocorre de maneira bem rápido? **Além disso, cria-se um código mais limpo, visto que o código para passar nos testes deve ser bastante simples.** Há mais segurança na correção de eventuais bugs; aumenta-se a produtividade, visto que se perde menos tempo com depuradores; e o código se torna mais flexível, menos acoplado e mais coeso.

Nós podemos afirmar que, em geral, utilizam-se testes unitários, testes de integração ou testes de aceitação – sendo os dois primeiros os mais comuns. **Algumas ferramentas que podem ser utilizadas para implementar o processo de desenvolvimento orientado a testes:** JUnit, TesteNG, PHPUnit, SimpleTest, NUnit, Jasmine, CUnit, PyUnit, etc. Pessoal, agora um detalhe que nós passamos direto sobre a abordagem de desenvolvimento...

O TEST DRIVEN DEVELOPMENT (TDD) NÃO É UMA ABORDAGEM PARA REALIZAR TESTES – TRATA-SE DE UMA ABORDAGEM PARA DESENVOLVER SOFTWARES. ELA PODE EVENTUALMENTE SER CONSIDERADA TAMBÉM UMA TÉCNICA DE PROGRAMAÇÃO!

Professor, uma curiosidade: isso já caiu em alguma prova discursiva? **Sim, galera... o enunciado dessa prova requisitava ao aluno informar as vantagens do emprego do TDD em relação a outras metodologias ágeis.** Poderíamos responder essa pergunta afirmando que o software desenvolvido, em geral, apresenta maior qualidade, na medida em que é implementado direcionado às expectativas do cliente.

Poderíamos dizer também que há a possibilidade de se testar todo o código desenvolvido, o que oferece maior confiabilidade ao sistema. Por fim, em geral, o código é mais modularizado, flexível e extensível, visto que a metodologia requer que os desenvolvedores imaginem o

software como pequenas unidades[1] que podem ser reescritas, desenvolvidas e testadas de forma independente e integradas em momento posterior.

Essa mesma prova perguntava também quais são os princípios da Metodologia Extreme Programming (XP) apoiados pelo TDD. Uma resposta adequada poderia afirmar que o XP apresenta diversas práticas que podem ser relacionadas com o TDD. *Qual é a mais óbvia?* **A mais óbvia é o *Test-First* (Teste Primeiro), ratificando a característica básica recomendada veementemente pelo desenvolvimento orientado a testes.**

O TDD pode apoiar esse princípio por fornecer detalhes para a realização dos testes de unidade e de funcionalidade, que são importantes e necessários. Ademais, o desenvolvimento orientado a testes apresenta relação intrínseca com a refatoração, tendo em vista que confere ao programador maior segurança para identificar e remover o código duplicado, e permite, assim, a melhoria contínua do programa.

Galera, nós temos também uma variação chamada Acceptance Test-Driven Development (ATDD). **Ele é método ágil de desenvolvimento de software que se baseia na comunicação entre clientes do negócio, desenvolvedores e testadores.** Ele se difere do TDD, na medida em que possui um foco maior na comunicação entre os colaboradores. São utilizados testes de aceitação a partir do ponto de vista dos usuários.

O ATDD é focado na captura de requisitos em testes de aceitação e os utiliza para guiar o desenvolvimento do sistema. Ele ajuda a assegurar que todos os membros do projeto entendam precisamente o que é necessário fazer e implementar, estabelecendo critérios a partir da perspectiva do usuário e criando exemplos concretos.

As equipes que experimentam ATDD normalmente concluem que apenas o ato de se definir testes de aceitação ao discutir requisitos resulta numa melhor compreensão destes requisitos. Os testes em ATDD nos forçam a chegar a um ponto de acordo concreto sobre o exato comportamento que se espera que o software deva ter. *Entenderam?*

A proposta do ATDD é favorecer uma colaboração e comunicação maior entre todos os envolvidos no desenvolvimento de um produto, o que resulta em um entendimento mais claro e refinado dos requisitos, possibilitando um acordo entre ambas as partes do que será desenvolvido durante uma iteração/sprint. **No final, o resultado estará alinhado às expectativas do cliente.**

[1] Atenção: apesar de serem tipicamente realizados testes de unidade, não é obrigatória a utilização desse tipo de teste.

EXERCÍCIOS COMENTADOS

1. **(CESPE / INMETRO / 2009)** A rotina diária dos desenvolvedores, ao empregar processos baseados no TDD (Test-Driven Development), é concentrada na elaboração de testes de homologação.

Comentários:

A rotina dos desenvolvedores é concentrada, na verdade, na elaboração de testes unitários e, não, de homologação. Lembrem-se: constrói o teste, constrói a funcionalidade e refatora a funcionalidade.

Gabarito: Errado

---

2. **(CESPE / INPI / 2013)** Usando-se o TDD, as funcionalidades devem estar completas e da forma como serão apresentadas aos seus usuários para que possam ser testadas e consideradas corretas.

Comentários:

ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

Pelo contrário, primeiro são feitos os testes e depois desenvolvem-se as funcionalidades.

Gabarito: Errado

---

3. **(CESPE / ANCINE / 2013)** No desenvolvimento de software conforme as diretivas do TDD (Test-Driven Development), deve-se elaborar primeiramente os testes e, em seguida, escrever o código necessário para passar pelos testes.

Comentários:

ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;

VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

Perfeito! Primeiro, criam-se os testes, depois cria-se o código.

Gabarito: Correto

---

4. **(CESPE / INMETRO / 2009)** Considerando uma organização na qual a abordagem de Test Driven Development (TDD) esteja implementada, assinale a opção correta.

a) Nessa organização, ocorre a execução de iterações com ciclo longo, isto é, com duração de alguns meses.

b) No início de cada iteração, a primeira atividade realizada pela equipe de desenvolvimento é produzir o código que será validado através de testes.

c) O refactoring é uma das primeiras atividades realizada no início de cada iteração.

d) Entre as atividades finais de cada iteração, o desenvolvedor escreve casos de teste automatizados, cuja execução verifica se houve a melhoria desejada ou se uma nova funcionalidade foi implementada.

e) Há coerência e inter-relação com os princípios promovidos pela prática da extreme programming (XP).

Comentários:

(a) Errado, são ciclos curtos e, não, longos; (b) Errado, são produzidos primeiramente os testes e, depois, os códigos; (c) Errado, a refatoração é a última atividade realizada em uma iteração; (d) Errado, escrever casos de testes é uma das atividades iniciais; (e) Correto, trata-se inclusive de uma das práticas recomendadas pelo XP.

Gabarito: Letra E

---

5. **(CESPE / MPOG / 2013)** Ao realizar o TDD (test-driven development), o programador é conduzido a pensar em decisões de design antes de pensar em código de implementação, o que cria um maior acoplamento, uma vez que seu objetivo é pensar na lógica e nas responsabilidades de cada classe.

Comentários:

Em Engenharia de Software, há dois conceitos importantíssimos: coesão e acoplamento. Quando eu estudava, eu decorava uma frase pequena para entender: "*Coesão é a divisão de responsabilidades e Acoplamento é a dependência entre componentes*". Há outra que dizia assim: "*Uma boa arquitetura de software deve ter componentes de projeto com baixo acoplamento e alta coesão*".

O Acoplamento trata do nível de dependência entre módulos ou componentes de um software. *Por que é bom ter baixo acoplamento?* Porque se os módulos pouco dependem um do outro, modificações de um não afetam os outros, além de não prejudicar o reúso. Se esse princípio não for observado durante a construção da arquitetura de um sistema de software, pode haver problemas sérios de manutenção futura!

Voltando à questão: se o programador pensa em decisões de design antes de pensar em código de implementação, isso diminui o acoplamento - os componentes ficam menos dependentes, tornando a arquitetura bem mais flexível.

Gabarito: Errado

---

6. **(CESPE / MPU – 2013)** Na metodologia TDD, ou desenvolvimento orientado a testes, cada nova funcionalidade inicia com a criação de um teste, cujo planejamento permite a identificação dos itens e funcionalidades que deverão ser testados, quem são os responsáveis e quais os riscos envolvidos.

Comentários:

Perfeito! Essa metodologia permite um aprendizado maior sobre o problema a ser resolvido, permitindo (não obrigatoriamente) a identificação de itens, funcionalidades, responsáveis e riscos.

Gabarito: Correto

---

7. **(CESPE / STF – 2013)** No TDD, o primeiro passo do desenvolvedor é criar o teste, denominado teste falho, que retornará um erro, para, posteriormente, desenvolver o código e aprimorar a codificação do sistema.

Comentários:

ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

Perfeito! Cria-se o teste falho, desenvolve-se um código e só então ocorre a refatoração.

Gabarito: Correto

---

8. **(CESPE / TRT-17 – 2013)** TDD consiste em uma técnica de desenvolvimento de software com abordagem embasada em perspectiva evolutiva de seu desenvolvimento. Essa abordagem envolve a produção de versões iniciais de um sistema a partir das quais é possível realizar verificações de suas qualidades antes que ele seja construído.

Comentários:

A questão diz que versões iniciais são produzidas e, a partir dessas versões, é possível realizar verificações de suas qualidades antes que ele seja construído. Na verdade, testes são criados inicialmente para verificar sua qualidade e, a partir daí, versões são produzidas. Logo, a questão inverteu os conceitos!

Gabarito: Correto

---

9. **(CESPE / ALRN – 2013)** Um típico ciclo de vida de um projeto em TDD consiste em:

I. Executar os testes novamente e garantir que estes continuem tendo sucesso.
II. Executar os testes para ver se todos estes testes obtiveram êxito.
III. Escrever a aplicação a ser testada.
IV. Refatorar (refactoring).
V. Executar todos os possíveis testes e ver a aplicação falhar.
VI. Criar o teste.

A ordem correta e cronológica que deve ser seguida para o ciclo de vida do TDD está expressa em:

a) IV – III – II – V – I – VI.
b) V – VI – II – I – III – IV.
c) VI – V – III – II – IV – I.
d) III – IV – V – VI – I – II.
e) III – IV – VI – V – I – II.

Comentários:

A ordem correta é: (VI) Criar o teste; (V) Executar todos os possíveis testes e ver a aplicação falhar; (III) Escrever a aplicação a ser testada; (II) Executar os testes para ver se todos estes testes obtiveram êxito; (IV) Refatorar (refactoring); (I) Executar os testes novamente e garantir que estes continuem tendo sucesso.

Gabarito: Letra C

---

10. (FGV / ALMT – 2013) Com relação ao desenvolvimento orientado (dirigido) a testes (do Inglês Test Driven Development – TDD), analise as afirmativas a seguir.

I. TDD é uma técnica de desenvolvimento de software iterativa e incremental.

II. TDD implica escrever o código de teste antes do código de produção, um teste de cada vez, tendo certeza de que o teste falha antes de escrever o código que irá fazê-lo passar.

III. TDD é uma técnica específica do processo XP (Extreme Programming), portanto, só pode ser utilizada em modelos de processos ágeis de desenvolvimento de software.

Assinale:

a) Se somente as afirmativas I e II estiverem corretas.
b) Se somente as afirmativas I e III estiverem corretas.
c) Se somente as afirmativas II e III estiverem corretas.
d) Se somente a afirmativa III estiver correta.
e) Se somente a afirmativa I estiver correta.

Comentários:

(I) Correto, é iterativo e incremental (lembrando que, em sua imensa maioria, metodologias ágeis são iterativas/incrementais); (II) Correto, escrevem-se os testes antes, fá-lo falhar e só depois escreve-se o código da aplicação; (III) Errado, ele é uma abordagem independente que pode ser utilizado com metodologias ágeis ou tradicionais.

Gabarito: Letra A

---

11. (FCC / TRT-MG – 2015) Um analista de TI está participando do desenvolvimento de um software orientado a objetos utilizando a plataforma Java. Na abordagem de desenvolvimento adotada, o código é desenvolvido de forma incremental, em conjunto com o teste para esse incremento, de forma que só se passa para o próximo incremento quando o atual passar no teste. Como o código é desenvolvido em incrementos muito pequenos e são executados testes a cada vez que uma funcionalidade é adicionada ou que o programa é refatorado, foi necessário definir um ambiente de testes automatizados utilizando um framework popular que suporta o teste de programas Java.

A abordagem de desenvolvimento adotada e o framework de suporte à criação de testes automatizados são, respectivamente,

a) Behavior-Driven Development e JTest.
b) Extreme Programming e Selenium.
c) Test-Driven Development e Jenkins.
d) Data-Driven Development and Test e JUnit.
e) Test-Driven Development e JUnit.

Comentários:

A abordagem claramente é o TDD e o framework evidentemente é o JUnit (ferramenta de suporte à criação de testes unitários automatizados). Lembrando que: DDD é um modelo de programação em que os próprios dados controlam o fluxo do programa e não a lógica do programa; XP é uma metodologia ágil de gerenciamento de projetos que suporta lançamentos frequentes em curtos ciclos de desenvolvimento para melhorar a qualidade do software e permitir que os desenvolvedores respondam às mudanças nos requisitos dos clientes; BDD é um método de desenvolvimento ágil que encoraja a colaboração entre desenvolvedores; Selenium é uma ferramenta usada para testes funcionais em aplicações web; e JTest é um ramework de análise estática que usa a linguagem Java.

Gabarito: Letra E

---

12. (CESPE / TRE-PE – 2017) O desenvolvimento orientado a testes (TDD):

a) é um conjunto de técnicas que se associam ao XP (extreme programming) para o desenvolvimento incremental do código que se inicia com os testes.

b) agrega um conjunto de testes de integração para avaliar a interconexão dos componentes do software com as aplicações a ele relacionadas.

c) avalia o desempenho do desenvolvimento de sistemas verificando se o volume de acessos/transações está acima da média esperada.

d) averigua se o sistema atende aos requisitos de desempenho verificando se o volume de acessos/transações mantém-se dentro do esperado.

e) testa o sistema para verificar se ele foi desenvolvido conforme os padrões e a metodologia estabelecidos nos requisitos do projeto.

Comentários:

O único item que faz algum sentido em relação ao TDD é o primeiro, isto é, conjunto de técnicas intimamente ligadas ao XP para o desenvolvimento de software que se inicia com os testes.

Gabarito: Correto

---

13. (CESPE / STM – 2018) O TDD (test driven development) parte de um caso de teste que caracteriza uma melhoria desejada ou nova funcionalidade a ser desenvolvida, de modo a confirmar o comportamento correto e possibilitar a evolução ou refatoração do código.

Comentários:

TDD é um método para construir software que enfatiza a criação de testes antes da criação do código-fonte. Logo, faz-se o teste - se não passou, refatora!

Gabarito: Correto

---

14. (CESPE / TRE/PI – 2016) O TDD (test driven development):

a) apresenta como vantagem a leitura das regras de negócio a partir dos testes, e, como desvantagem, a necessidade de mais linhas de códigos que a abordagem tradicional, o que gera um código adicional.

b) impede que seja aplicada a prática de programação em pares, que é substituída pela interação entre analista de teste, testador e programador.

c) é um conjunto de técnicas associadas ao eXtremme Programing e a métodos ágeis, sendo, contudo, incompatível com o Refactoring, haja vista o teste ser escrito antes da codificação.

d) refere-se a uma técnica de programação cujo principal objetivo é escrever um código funcional limpo, a partir de um teste que tenha falhado.

e) refere-se a uma metodologia de testes em que se devem testar condições, loops e operações; no entanto, por questão de simplicidade, não devem ser testados polimorfismos.

Comentários:

(a) Errado, esse item não faz nenhum sentido em relação ao TDD; (b) Errado, não impede a programação em pares; (c) Errado, ela é totalmente compatível e dependente da refatoração; (d) Correto, ela pode ser vista como uma técnica de programação cujo objetivo é escrever um código funcional limpo a partir de um teste falho; (e) Errado, não se trata de uma metodologia de testes.

Gabarito: Letra D

---

15. (UFRRJ / UFRRJ – 2015) Os testes de unidade têm papel central na metodologia de implementação dirigida por testes, popularizada pelo processo XP e adotada em outros métodos. Esses testes são criados primeiro, exercitando o contrato de cada operação

implementada pelos métodos. Em seguida, o código dos métodos é escrito para cumprir os contratos e, portanto, passar nos testes de unidade. Esse cenário corresponde à abordagem

a) TDD.
b) MDD.
c) DDC.
d) MDE.
e) FDD.

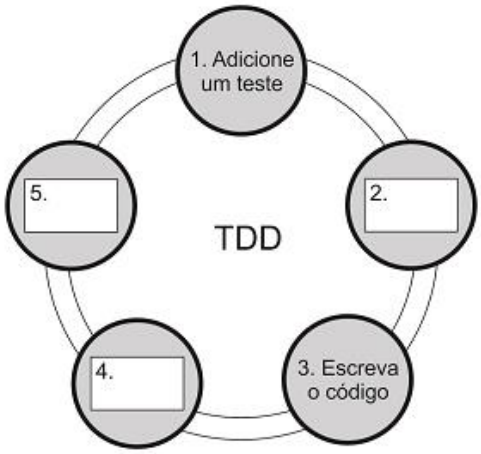
Comentários:

Testes de unidade têm papel central? Metodologia dirigida a testes? Popularizada pelo XP? Testes são criados primeiro? Tudo isso nos remente a... TDD!

Gabarito: Letra A

---

16. (FCC / TRE-PR – 2017) Considere o ciclo do Test-Driven Development – TDD.



A Caixa:

a) 2. corresponde a "Execute os testes automatizados".
b) 4. corresponde a "Refatore o código".
c) 5. corresponde a "Execute os testes novamente e observe os resultados".
d) 4. corresponde a "Execute os testes automatizados".
e) 5. corresponde a "Faça todos os testes passarem".

Comentários:

(a) Errado, corresponde a "Execute o teste e observe o resultado"; (b) Errado, corresponde a "Execute os testes automatizados"; (c) Errado, corresponde a "Refatore os códigos"; (d) Correto, corresponde realmente a "Execute os testes automatizados"; (e) Errado, corresponde a "Refatore os códigos".

Gabarito: Letra D

---

17. (IESES / TRE/MA – 2015) A respeito da técnica de testes TDD é correto afirmar que:

a) Testa o software com base no comportamento esperado.
b) É uma prática para desenvolvimento de testes unitário que pode utilizar o processo Red-Green-Refactor.
c) Utiliza-se da estrutura Dado, Quando e Então para montar os testes.
d) Prega que os testes devem ser realizados sempre após a implementação ser concluída.

Comentários:

(a) Errado, não vejo nada de errado nesse item – ele pode testar o software com base no comportamento esperado! No entanto, a banca considerou o item errado; (b) Correto, ele realmente utiliza o processo RED/GREEN/REFACTOR; (c) Errado, esse item não faz qualquer sentido; (d) Errado, testes são realizados antes de a implementação ser concluída.

Gabarito: Letra B

---

18. (CESPE / TRE/RS – 2015) Projeto para o desenvolvimento de software que utilize TDD deve:

a) realizar sprints a cada quinzena.
b) desenvolver pequenos releases.
c) apresentar grande quantidade de testes unitários de código-fonte previamente desenvolvidos.

d) apresentar linguagem de programação estruturada.
e) recomendar a preparação dos testes para que, posteriormente, seja desenvolvido o código.

Comentários:

(a) Errado, não há nenhuma relação entre TDD e Sprints; (b) Errado, são desenvolvidas pequenas unidades e, não, releases; (c) Errado, os testes unitários são escritos iterativamente a medida que o desenvolvimento ocorre; (d) Errado, não há nenhuma relação ou dependência com linguagens de programação; (e) Correto, recomenda-se preparar testes antes do desenvolvimento do código.

Gabarito: Letra E

---

19. **(FCC / TRE/AP – 2015)** O TDD – Test Driven Development (Desenvolvimento orientado a teste):

a) é parte das metodologias ágeis UP – Unified Process e XP – Extreme Programming, tendo sido criado para ser usado em metodologias que respeitam os 4 princípios do Manifesto Ágil.

b) transforma o desenvolvimento, pois deve-se primeiro implementar o sistema antes de escrever os testes. Os testes são utilizados para facilitar no entendimento do projeto e para clarear o que se deseja em relação ao código.

c) baseia-se em um ciclo simples: escreve-se um código -> cria-se um teste para passar no código -> refatora-se.

d) propõe a criação de testes que validem o código como um todo para reduzir o tempo de desenvolvimento.

e) beneficia-se de testes que seguem o modelo FIRST: F (Fast) I (Isolated) R (Repeatable) S (Self-verifying) T (Timely).

Comentários:

ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

(a) Errado, Unified Process (UP) não é uma metodologia ágil; (b) Errado, devem ser implementados os testes antes do sistema; (c) Errado, cria-se um teste falho > escreve-se o

código que passa > refatora-se; (d) Errado, criam-se testes que validam unidades e, não, o código como um todo; (e) Correto, vamos falar um pouquinho sobre isso agora:

Existe uma representação chamada Modelo FIRST: **F (Fast):** devem ser rápidos, pois testam apenas uma unidade; **I (Isolated):** testes unitários são isolados, testando individualmente as unidades e não sua integração; **R (Repeateble):** repetição nos testes, com resultados de comportamento constante; **S (Self-verifying):** a auto verificação deve verificar se passou ou se deu como falha o teste; **T (Timely):** o teste deve ser oportuno, sendo um teste por unidade.

Gabarito: Letra E

---

20. (CESPE / TRE/TO – 2017) O TDD (Test-Driven Development), que vem sendo adotado para testar os projetos de software,

a) utiliza os testes de caixa preta antes da entrega do software.
b) agiliza os testes por amostragem sem compatibilidade retroativa.
c) cobre amplamente os testes unitários.
d) escreve o teste antes da codificação do software.
e) realiza refactoring antes de escrever a aplicação a ser testada.

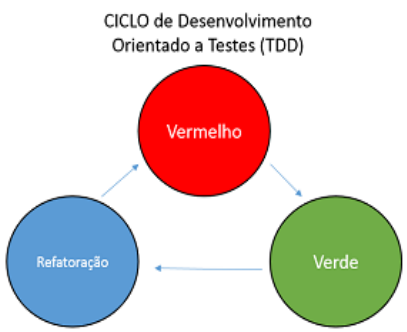
Comentários:

Eu já começo discordando do enunciado – ele vem sendo utilizado para desenvolver software utilizando testes, mas não para testar projetos de software. Ignorando a redação da questão, vamos aos comentários: (a) Errado, ele tipicamente utiliza testes de unidade antes da entrega do software; (b) Errado, esse item não faz qualquer sentido; (c) Errado, ele não tem o intuito principal de cobrir amplamente testes unitários; (d) Correto, o teste é escrito antes da codificação do software; (e) Errado, não faz sentido fazer *refactoring* antes de escrever a aplicação.

Gabarito: Letra D

---

21. (FGV / IBGE – 2016) O Desenvolvimento Orientado a Testes (TDD) é um método de desenvolvimento criado e disseminado por Kent Beck em seu livro "Test-driven development". O método define regras, boas práticas e um ciclo de tarefas com 3 etapas: a etapa vermelha, a etapa verde e a etapa de refatoração, ilustrado na imagem abaixo:



Com relação às regras e boas práticas de TDD e ao seu ciclo, é correto afirmar que:

a) pode-se escrever testes que não compilam na etapa vermelha;
b) na etapa verde deve-se escrever código que testa uma funcionalidade a fundo de forma criteriosa e detalhada;
c) código novo só é escrito se um teste automatizado passar;
d) a duplicação é tolerada na etapa de refatoração;
e) é uma boa prática de TDD iniciar o desenvolvimento do código de uma funcionalidade e, logo em seguida, testá-la.

Comentários:

ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

(a) Correto. Nessa etapa, o objetivo do desenvolvedor é escrever um pequeno teste que não funcione e que talvez nem mesmo compile inicialmente; (b) Errado. Nessa etapa, escreve-se o código que apenas passa no teste, sem necessidade de aprofundamento em detalhes de implementação; (c) Errado. O código novo é escrito para o teste automatizado passar; (d) Errado. Não é tolerada a duplicação, uma vez que o objetivo é eliminar redundâncias e melhorar o código; (e) Errado. Vimos centenas de vezes que o primeiro passo é criar o teste e depois escrever.

Gabarito: Letra A

---

22. **(IADES / EBSERH – 2013)** Assinale a alternativa que não corresponde a uma das fases do processo de desenvolvimento, dirigido a testes (TDD).

a) Executar o teste, com os outros testes implementados, que rodarão e fornecerão o resultado de que o software está sem problemas.
b) Escrever o teste para a funcionalidade e implementação.
c) Realizar a identificação do incremento de funcionalidade.
d) Implementar a funcionalidade e executar novamente o teste.
e) Implementar a próxima parte da funcionalidade, após todos os testes terem sido executados, com sucesso

Comentários:

(a) Errado. Na terceira fase, executa-se realmente o teste junto com todos os outros testes implementados. No entanto, inicialmente você não terá implementado a funcionalidade, logo o novo teste falhará; (b) Correto, essa é a segunda fase; (c) Correto, essa é a primeira fase; (d) Correto, essa é a quarta fase; (e) Correto, essa é a quinta fase.

Gabarito: Letra A

---

23. (PR-4 UFRJ / UFRJ – 2018) O ciclo do TDD - Test Driven Development, ou, em português, Desenvolvimento Guiado por Testes consiste em:

a) implementar teste unitário falho, tornar o teste bem-sucedido e refatorar.
b) implementar a funcionalidade, executar teste unitário e refatorar.
c) implementar teste unitário falho, refatorar e tornar o teste bem-sucedido.
d) implementar a funcionalidade, refatorar e tornar o teste bem-sucedido.
e) refatorar, executar teste unitário e implementar a funcionalidade.

Comentários:

(a) Correto, as atividades são exatamente essas e nessa ordem; (b) Errado, o teste vem antes da implementação da funcionalidade; (c) Errado, primeiro você torna o teste bem sucedido e depois refatora; (d) Errado, não faz sentido refatorar antes do teste; (e) Errado, não faz sentido refatorar antes do teste.

Gabarito: Letra A

---

24. (CESPE / STJ – 2015) Um dos passos executados no ciclo de atividades do processo TDD é a criação de novos testes para as falhas encontradas no código original, sem alteração deste.

Comentários:

Nooooope... testes devem realmente encontrar falhas – o que se altera é o código para passar no teste e, não, o teste em si.

Gabarito: Errado

---

25. (CS-UFG / AL-GO – 2015) O desenvolvimento dirigido a testes (TDD, do Inglês Test-Driven Development) é uma abordagem de desenvolvimento de software na qual se intercalam testes e desenvolvimento de código. Uma das características da abordagem TDD é:

a) a sua utilidade no desenvolvimento de softwares novos.
b) o maior custo associado aos testes de regressão.
c) a redução da importância da automatização dos testes.
d) a sua adequação a processos de software sequenciais.

Comentários:

(a) Correto. Ela é mais ideal para o desenvolvimento de softwares novos; (b) Errado, ela reduz custos de testes de regressão; (c) Errado, ela aumenta a importância da automatização de testes; (d) Errado, ela é adequada a processos iterativos – lembrem-se do ciclo de testes, falhas e refatorações.

Gabarito: Letra A

---

26. (UECE-CEV / FUNCEME – 2018) Test-driven Development (TDD) é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código (Sommerville, I. Engenharia de Software, 9a edição, 2011).

A respeito do TDD, é correto afirmar que:

a) consiste em um processo iterativo que se inicia escrevendo um código de uma funcionalidade do sistema e, logo em seguida, testa-o para saber se a implementação foi correta.
b) apesar de útil, não diminui o custo de testes de regressão do sistema.
c) sua utilização elimina a necessidade de testes de validação do sistema, uma vez que ele já foi testado incrementalmente.
d) apesar de ter sido apresentado como parte dos métodos ágeis, também pode ser usado em outros processos de desenvolvimento de software.

Comentários:

(a) Errado, inicia-se escrevendo o teste; (b) Errado, diminui – sim – os custos de testes de regressão; (c) Errado, não elimina em nenhuma hipótese testes de validação/aceitação; (d) Correto, ele pode ser utilizado com métodos ágeis ou tradicionais.

Gabarito: Letra D

---

27. (FAUGRS / UFRGS – 2018) _____ é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código. Essencialmente, desenvolve-se um código de forma incremental em conjunto com um teste para este incremento. Não se avança para o próximo incremento até que o código desenvolvido passe no teste. Essa abordagem foi introduzida como parte de métodos ágeis, mas pode ser também usada em processos de desenvolvimento dirigido a planos. Assinale a alternativa que preenche corretamente a lacuna do texto acima.

a) Desenvolvimento Guiado por Testes (TDD)
b) Desenvolvimento em Espiral

c) Engenharia Dirigida a Modelos (MDD)
d) Rational Unified Process (RUP)
e) Teste de Sistema

Comentários:

TDD é uma **abordagem para o desenvolvimento** de programas em que se **intercalam testes e desenvolvimento de código**. Essencialmente, desenvolve-se um código de forma incremental em conjunto com um teste para este incremento. Não se avança para o próximo incremento até que o código desenvolvido passe no teste. Essa abordagem foi introduzida como parte de métodos ágeis, mas pode ser também usada em processos de desenvolvimento dirigido a planos.

Gabarito: Letra A

---

28. (VUNESP / TCE-SP – 2015) No Desenvolvimento Orientado a Testes (TDD), os casos de teste que definem o recurso a ser implementado devem ser elaborados:

a) assim que o código do teste estiver pronto.
b) antes de o código do recurso ser desenvolvido.
c) após o código do recurso ter sido completamente documentado.
d) simultaneamente com o desenvolvimento do código do recurso.
e) somente se o código do recurso apresentar erros.

Comentários:

(a) Errado, os casos de testes já devem estar prontos antes do código do teste ser feito; (b) Correto, devem ser elaborados antes de o código do recurso ser desenvolvido; (c) Errado, não existe o conceito de documentação formal, os próprios testes agem como uma forma de documentação que descreve o que o código deve fazer; (d) Errado, as etapas de desenvolvimento do código do recurso e a implementação dos casos de testes ocorrem em etapas distintas e não concomitantes; (e) Errado, casos de testes sempre são implementados porque eles representam a funcionalidade a ser desenvolvida.

Gabarito: Letra B

---

29. (IBFC / EMBASA – 2017) No Ciclo de Desenvolvimento do TDD (Test-Driven Development), utiliza-se a estratégia que aplica três palavras-chaves (em inglês), que é denominada:

a) Red, Green, Refactor
b) White, Gray, Black
c) White, Black, Refactor
d) Green, Yellow, Red

Comentários:

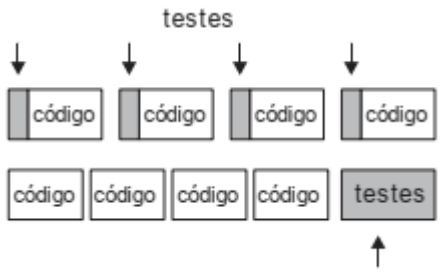
ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

Trata-se do RED > GREEN > REFACTOR.

Gabarito: Letra A

---

30. (FCC / CREMESP – 2016) Considere a figura abaixo que apresenta duas abordagens de teste.



A figura:

a) ilustra as duas fases do TDD, que correspondem a escrever pequenos testes e testá-los no final.
b) mostra o ciclo conhecido como Vermelho-Verde-Refatora.
c) apresenta a diferença entre testes automatizados e testes manuais no XP.
d) mostra que um desenvolvedor que pratica TDD tem mais feedbacks do que um que escreve testes ao final.
e) evidencia que TDD é impraticável, pois o desenvolvedor gasta muito tempo escrevendo código de testes.

Comentários:

Qual é a diferença entre a abordagem superior e inferior? Na parte superior, testes são aplicados individualmente em cada unidade de código. Na parte inferior, todos os códigos foram escritos e, somente depois, foram testados. Logo, existem mais feedbacks na abordagem superior (TDD) do que na abordagem inferior.

Gabarito: Letra D

---

31. **(CESPE / ANATEL – 2014)** Em se tratando de desenvolvimento de softwares dirigidos a testes (TDD), a execução dos testes é realizada antes da implementação da funcionalidade.

Comentários:

Perfeito! A execução dos testes de realmente ocorrem antes da implementação da funcionalidade.

Gabarito: Correto

---

32. **(CESPE / TC/DF – 2014)** No TDD, o refatoramento do código deve ser realizado antes de se escrever a aplicação que deve ser testada.

Comentários:

Opaaaaa... como você vai refatorar um código que ainda não foi escrito? Não faz sentido!

Gabarito: Correto

---

33. **(CESPE / STJ – 2015)** No método de desenvolvimento TDD (Test Driven Development), o desenvolvedor escreve primeiro um caso de teste e, posteriormente, o código.

Comentários:

Perfeito! Esse é o conceito do *test-first*, isto é, o desenvolvedor escreve primeiro um caso de teste e, posteriormente, o código.

Gabarito: Correto

---

34. **(CESPE / MPE-PI – 2018)** O TDD possibilita o desenvolvimento de softwares fundamentado em testes. O ciclo de desenvolvimento do TDD segue os seguintes passos:

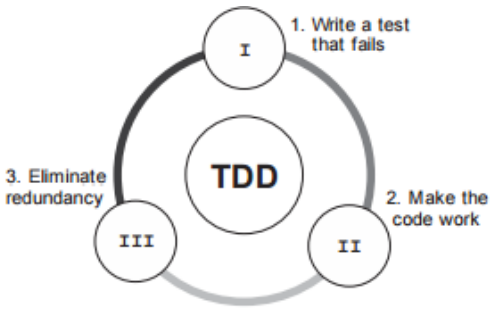
- escrever um teste que inicialmente não passa;
- adicionar uma nova funcionalidade do sistema;
- fazer o teste passar;
- realizar a integração contínua do código;
- escrever o próximo teste.

Comentários:

A ordem realmente está correta, no entanto há uma pegadinha: na quarta etapa, realiza-se a refatoração do código e, não, a integração contínua.

Gabarito: Errado

35. **(FCC / Pref. de Teresina/PI – 2016)** O Test Driven Development – TDD é uma das práticas sugeridas na eXtreme Programming – XP, onde o programador escreve o teste antes de escrever o código. O ciclo de desenvolvimento utilizando TDD é mostrado abaixo.



Considere:

I. Etapa inicial, onde se escreve um teste que falha, para alguma funcionalidade que ainda será Escrita.
II. Já com o teste criado, é o momento de executar o teste.
III. Eliminar códigos redundantes, remover acoplamentos, enfim, identificar pontos de melhoria no código.

As etapas I, II e III são, respectivamente,

a) Iniciação, Execução e Controle.
b) Red, Green e Refactor.
c) Iniciação, Atuação e Otimização.
d) Plan, Do e Check.
e) Planejamento, Execução e Melhoria.

Comentários:

ETAPA	DESCRIÇÃO
VERMELHO (RED)	Escreva um teste que falha: casos de teste são escritos sem que exista o recurso a ser testado, levando o teste a falhar;
VERDE (GREEN)	Escreva código para passar no teste: o recurso é implementado com o mínimo de código necessário para que passe no teste.
REFATORAR (REFACTOR)	Elimine redundâncias: o código que implementa o recurso é refinado e melhorado, sem que seja adicionada novas funcionalidades.

(RED) Etapa inicial, onde se escreve um teste que falha, para alguma funcionalidade que ainda será Escrita; (GREEN) Já com o teste criado, é o momento de executar o teste; (REFACTOR) Eliminar códigos redundantes, remover acoplamentos, enfim, identificar pontos de melhoria no código.

Gabarito: Letra B

---

36. (FAUGRS / BANRISUL – 2018) Considere as ações abaixo, executadas em desenvolvimento orientado a testes, Test-Driven Design (TDD).

I - Escrever código de teste.
II - Verificar se o teste falha.
III - Escrever código de produção.
IV - Executar teste até passar (reescrevendo o código de produção, se for necessário, até que o teste passe).
V - Refatorar código de produção e/ou de teste para melhorá-lo.

Considerando que se deseja incluir um novo caso de teste, assinale a alternativa que apresenta a sequência de ações que devem obrigatoriamente ocorrer para essa inclusão, segundo o TDD.

a) I, III e IV.
b) III, I e IV.
c) I, II, III e IV.
d) I, III, IV e V.
e) I, II, III, IV e V.

Comentários:

A ordem correta é: (I) Escrever código de teste; (II) Verificar se o teste falha; (III) Escrever código de produção; (IV) Executar teste até passar (reescrevendo o código de produção, se for necessário, até que o teste passe); (V) Refatorar código de produção e/ou de teste para melhorá-lo.

Gabarito: Letra C

---

37. (FGV / IBGE – 2017) Test Driven Development (TDD) é uma prática muito utilizada no processo de desenvolvimento de sistemas computacionais. Analise as afirmativas a seguir sobre o uso da prática de TDD:

I. Tornam os testes de regressão mais demorados porque o desenvolvedor precisará fazer testes manuais várias vezes por dia.

II. Garante que os requisitos do sistema sejam atendidos porque o desenvolvedor escreverá o código de testes sempre que acabar a implementação do código do sistema.

III. Ajuda o desenvolvedor a escrever código de qualidade porque ele gastará parte do seu tempo escrevendo código de testes.

Está correto o que se afirma em:

a) somente I;
b) somente II;
c) somente III;
d) somente II e III;
e) I, II e III.

Comentários:

(I) Errado. De acordo com Ian Sommerville, um dos benefícios mais importantes de desenvolvimento dirigido a testes é que ele reduz os custos dos testes de regressão, uma vez que testes automatizados – fundamentais para o desenvolvimento *test-first* – reduzem drasticamente os custos com testes de regressão; (II) Errado. O desenvolver escreverá o código de testes antes de acabar a implementação do código do sistema; (III) Correto. O TDD realmente ajuda o desenvolvedor a escrever código de qualidade porque ele gastará parte do seu tempo escrevendo código de testes.

Gabarito: Letra C

---

38. **(CESPE / ANATEL – 2014)** Na atividade de TDD (test-driven development), a escrita de teste primeiro define implicitamente tanto uma interface quanto uma especificação do comportamento para a funcionalidade que está sendo desenvolvida, estando, entretanto, a viabilidade do uso dessa abordagem limitada aos processos de desenvolvimento de software que seguem as práticas ágeis.

Comentários:

Opa... a interface deve ser definida explicitamente! Além disso, conforme afirma Ian Sommerville, ele também pode ser utilizado em processos de desenvolvimento dirigido a planos (tradicionais) e, não só, aos ágeis.

Gabarito: Errado

---

39. **(CESPE / TRE/MT – 2015)** Considere as seguintes etapas de um processo do tipo desenvolvimento orientado a testes (TDD).

I Implementar funcionalidade e refatorar.
II Identificar nova funcionalidade.
III Executar o teste.

IV Escrever o teste.
V Implementar a próxima parte da funcionalidade.

Assinale a opção que apresenta a sequência correta em que essas etapas devem ser realizadas.

a) I; IV; III; II; V
b) IV; III; II; I; V
c) I; IV; II; III; V
d) II; IV; III; I; V
e) IV; II; III; I; V

Comentários:

A ordem correta é: (II) Identificar nova funcionalidade; (IV) Escrever o teste; (III) Executar o teste; (I) Implementar funcionalidade e refatorar; (V) Implementar a próxima parte da funcionalidade.

Gabarito: Letra D

---

40. **(FCC / SEFAZ-SC – 2018)** O Test-Driven Development (TDD) é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código. As etapas do processo fundamental de TDD são mostradas abaixo em ordem alfabética:

I. Escrever um teste para a funcionalidade identificada e implementá-lo como um teste automatizado.

II. Executar o teste, junto com os demais testes já implementados, sem implementar a nova funcionalidade no código.

III. Identificar e implementar uma outra funcionalidade, após todos os testes serem executados com sucesso.

IV. Identificar uma nova funcionalidade pequena para ser incrementada com poucas linhas em um código.

V. Implementar a nova funcionalidade no código e reexecutar o teste.

VI. Refatorar o código com melhorias incrementais até que o teste execute sem erros.

VII. Revisar a funcionalidade e o teste, caso o código execute sem falhar.

Considerando o item IV a primeira etapa e o item III a última etapa, a sequência intermediária correta das etapas do processo é:

a) I – II – VII – V e VI.
b) I – V – II – VII e VI.
c) I – VI – V – VII e II.
d) V – I – II – VII e VI.
e) V – I – VI – VII e II.

Comentários:

São somente cinco etapas: I. Escrever um teste para a funcionalidade identificada e implementá-lo como um teste automatizado; II. Executar o teste, junto com os demais testes já implementados, sem implementar a nova funcionalidade no código; VII. Revisar a funcionalidade e o teste, caso o código execute sem falhar; V. Implementar a nova funcionalidade no código e reexecutar o teste; VI. Refatorar o código com melhorias incrementais até que o teste execute sem erros.

Gabarito: Letra A

LISTA DE EXERCÍCIOS

1. **(CESPE / INMETRO / 2009)** A rotina diária dos desenvolvedores, ao empregar processos baseados no TDD (Test-Driven Development), é concentrada na elaboração de testes de homologação.

2. **(CESPE / INPI / 2013)** Usando-se o TDD, as funcionalidades devem estar completas e da forma como serão apresentadas aos seus usuários para que possam ser testadas e consideradas corretas.

3. **(CESPE / ANCINE / 2013)** No desenvolvimento de software conforme as diretivas do TDD (Test-Driven Development), deve-se elaborar primeiramente os testes e, em seguida, escrever o código necessário para passar pelos testes.

4. **(CESPE / INMETRO / 2009)** Considerando uma organização na qual a abordagem de Test Driven Development (TDD) esteja implementada, assinale a opção correta.

   a) Nessa organização, ocorre a execução de iterações com ciclo longo, isto é, com duração de alguns meses.

   b) No início de cada iteração, a primeira atividade realizada pela equipe de desenvolvimento é produzir o código que será validado através de testes.

   c) O refactoring é uma das primeiras atividades realizada no início de cada iteração.

   d) Entre as atividades finais de cada iteração, o desenvolvedor escreve casos de teste automatizados, cuja execução verifica se houve a melhoria desejada ou se uma nova funcionalidade foi implementada.

   e) Há coerência e inter-relação com os princípios promovidos pela prática da extreme programming (XP).

5. **(CESPE / MPOG / 2013)** Ao realizar o TDD (test-driven development), o programador é conduzido a pensar em decisões de design antes de pensar em código de implementação, o que cria um maior acoplamento, uma vez que seu objetivo é pensar na lógica e nas responsabilidades de cada classe.

6. **(CESPE / MPU – 2013)** Na metodologia TDD, ou desenvolvimento orientado a testes, cada nova funcionalidade inicia com a criação de um teste, cujo planejamento permite a identificação dos itens e funcionalidades que deverão ser testados, quem são os responsáveis e quais os riscos envolvidos.

7. **(CESPE / STF – 2013)** No TDD, o primeiro passo do desenvolvedor é criar o teste, denominado teste falho, que retornará um erro, para, posteriormente, desenvolver o código e aprimorar a codificação do sistema.

8. **(CESPE / TRT-17 – 2013)** TDD consiste em uma técnica de desenvolvimento de software com abordagem embasada em perspectiva evolutiva de seu desenvolvimento. Essa abordagem envolve a produção de versões iniciais de um sistema a partir das quais é possível realizar verificações de suas qualidades antes que ele seja construído.

9. **(CESPE / ALRN – 2013)** Um típico ciclo de vida de um projeto em TDD consiste em:

   I. Executar os testes novamente e garantir que estes continuem tendo sucesso.
   II. Executar os testes para ver se todos estes testes obtiveram êxito.
   III. Escrever a aplicação a ser testada.
   IV. Refatorar (refactoring).
   V. Executar todos os possíveis testes e ver a aplicação falhar.
   VI. Criar o teste.

   A ordem correta e cronológica que deve ser seguida para o ciclo de vida do TDD está expressa em:

   a) IV – III – II – V – I – VI.
   b) V – VI – II – I – III – IV.
   c) VI – V – III – II – IV – I.
   d) III – IV – V – VI – I – II.
   e) III – IV – VI – V – I – II.

10. **(FGV / ALMT – 2013)** Com relação ao desenvolvimento orientado (dirigido) a testes (do Inglês Test Driven Development – TDD), analise as afirmativas a seguir.

    I. TDD é uma técnica de desenvolvimento de software iterativa e incremental.

    II. TDD implica escrever o código de teste antes do código de produção, um teste de cada vez, tendo certeza de que o teste falha antes de escrever o código que irá fazê-lo passar.

    III. TDD é uma técnica específica do processo XP (Extreme Programming), portanto, só pode ser utilizada em modelos de processos ágeis de desenvolvimento de software.

    Assinale:

    a) Se somente as afirmativas I e II estiverem corretas.
    b) Se somente as afirmativas I e III estiverem corretas.
    c) Se somente as afirmativas II e III estiverem corretas.
    d) Se somente a afirmativa III estiver correta.

e) Se somente a afirmativa I estiver correta.

11. **(FCC / TRT-MG – 2015)** Um analista de TI está participando do desenvolvimento de um software orientado a objetos utilizando a plataforma Java. Na abordagem de desenvolvimento adotada, o código é desenvolvido de forma incremental, em conjunto com o teste para esse incremento, de forma que só se passa para o próximo incremento quando o atual passar no teste. Como o código é desenvolvido em incrementos muito pequenos e são executados testes a cada vez que uma funcionalidade é adicionada ou que o programa é refatorado, foi necessário definir um ambiente de testes automatizados utilizando um framework popular que suporta o teste de programas Java.

    A abordagem de desenvolvimento adotada e o framework de suporte à criação de testes automatizados são, respectivamente,

    a) Behavior-Driven Development e JTest.
    b) Extreme Programming e Selenium.
    c) Test-Driven Development e Jenkins.
    d) Data-Driven Development and Test e JUnit.
    e) Test-Driven Development e JUnit.

12. **(CESPE / TRE-PE – 2017)** O desenvolvimento orientado a testes (TDD):

    a) é um conjunto de técnicas que se associam ao XP (extreme programming) para o desenvolvimento incremental do código que se inicia com os testes.

    b) agrega um conjunto de testes de integração para avaliar a interconexão dos componentes do software com as aplicações a ele relacionadas.

    c) avalia o desempenho do desenvolvimento de sistemas verificando se o volume de acessos/transações está acima da média esperada.

    d) averigua se o sistema atende aos requisitos de desempenho verificando se o volume de acessos/transações mantém-se dentro do esperado.

    e) testa o sistema para verificar se ele foi desenvolvido conforme os padrões e a metodologia estabelecidos nos requisitos do projeto.

13. **(CESPE / STM – 2018)** O TDD (test driven development) parte de um caso de teste que caracteriza uma melhoria desejada ou nova funcionalidade a ser desenvolvida, de modo a confirmar o comportamento correto e possibilitar a evolução ou refatoração do código.

14. **(CESPE / TRE/PI – 2016)** O TDD (test driven development):

a) apresenta como vantagem a leitura das regras de negócio a partir dos testes, e, como desvantagem, a necessidade de mais linhas de códigos que a abordagem tradicional, o que gera um código adicional.

b) impede que seja aplicada a prática de programação em pares, que é substituída pela interação entre analista de teste, testador e programador.

c) é um conjunto de técnicas associadas ao eXtremme Programing e a métodos ágeis, sendo, contudo, incompatível com o Refactoring, haja vista o teste ser escrito antes da codificação.

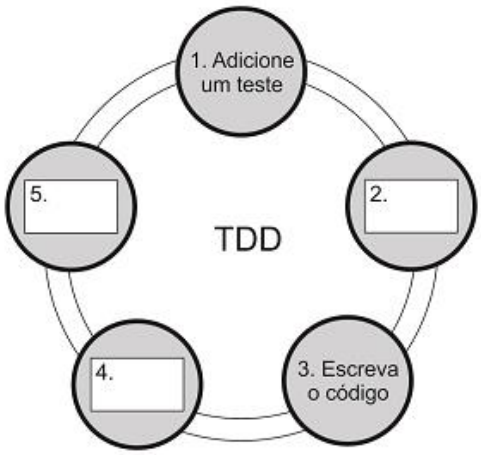
d) refere-se a uma técnica de programação cujo principal objetivo é escrever um código funcional limpo, a partir de um teste que tenha falhado.

e) refere-se a uma metodologia de testes em que se devem testar condições, loops e operações; no entanto, por questão de simplicidade, não devem ser testados polimorfismos.

15. **(UFRRJ / UFRRJ – 2015)** Os testes de unidade têm papel central na metodologia de implementação dirigida por testes, popularizada pelo processo XP e adotada em outros métodos. Esses testes são criados primeiro, exercitando o contrato de cada operação implementada pelos métodos. Em seguida, o código dos métodos é escrito para cumprir os contratos e, portanto, passar nos testes de unidade. Esse cenário corresponde à abordagem

a) TDD.
b) MDD.
c) DDC.
d) MDE.
e) FDD.

16. **(FCC / TRE-PR – 2017)** Considere o ciclo do Test-Driven Development – TDD.



A Caixa:

a) 2. corresponde a "Execute os testes automatizados".
b) 4. corresponde a "Refatore o código".
c) 5. corresponde a "Execute os testes novamente e observe os resultados".
d) 4. corresponde a "Execute os testes automatizados".
e) 5. corresponde a "Faça todos os testes passarem".

17. **(IESES / TRE/MA – 2015)** A respeito da técnica de testes TDD é correto afirmar que:

a) Testa o software com base no comportamento esperado.
b) É uma prática para desenvolvimento de testes unitário que pode utilizar o processo Red-Green-Refactor.
c) Utiliza-se da estrutura Dado, Quando e Então para montar os testes.
d) Prega que os testes devem ser realizados sempre após a implementação ser concluída.

18. **(CESPE / TRE/RS – 2015)** Projeto para o desenvolvimento de software que utilize TDD deve:

a) realizar sprints a cada quinzena.
b) desenvolver pequenos releases.
c) apresentar grande quantidade de testes unitários de código-fonte previamente desenvolvidos.
d) apresentar linguagem de programação estruturada.
e) recomendar a preparação dos testes para que, posteriormente, seja desenvolvido o código.

19. **(FCC / TRE/AP – 2015)** O TDD – Test Driven Development (Desenvolvimento orientado a teste):

a) é parte das metodologias ágeis UP – Unified Process e XP – Extreme Programming, tendo sido criado para ser usado em metodologias que respeitam os 4 princípios do Manifesto Ágil.

b) transforma o desenvolvimento, pois deve-se primeiro implementar o sistema antes de escrever os testes. Os testes são utilizados para facilitar no entendimento do projeto e para clarear o que se deseja em relação ao código.

c) baseia-se em um ciclo simples: escreve-se um código -> cria-se um teste para passar no código -> refatora-se.

d) propõe a criação de testes que validem o código como um todo para reduzir o tempo de desenvolvimento.

e) beneficia-se de testes que seguem o modelo FIRST: F (Fast) I (Isolated) R (Repeatable) S (Self-verifying) T (Timely).

20. **(CESPE / TRE/TO – 2017)** O TDD (Test-Driven Development), que vem sendo adotado para testar os projetos de software,

a) utiliza os testes de caixa preta antes da entrega do software.
b) agiliza os testes por amostragem sem compatibilidade retroativa.
c) cobre amplamente os testes unitários.
d) escreve o teste antes da codificação do software.
e) realiza refactoring antes de escrever a aplicação a ser testada.

21. **(FGV / IBGE – 2016)** O Desenvolvimento Orientado a Testes (TDD) é um método de desenvolvimento criado e disseminado por Kent Beck em seu livro "Test-driven development". O método define regras, boas práticas e um ciclo de tarefas com 3 etapas: a etapa vermelha, a etapa verde e a etapa de refatoração, ilustrado na imagem abaixo:

Com relação às regras e boas práticas de TDD e ao seu ciclo, é correto afirmar que:

a) pode-se escrever testes que não compilam na etapa vermelha;
b) na etapa verde deve-se escrever código que testa uma funcionalidade a fundo de forma criteriosa e detalhada;
c) código novo só é escrito se um teste automatizado passar;
d) a duplicação é tolerada na etapa de refatoração;
e) é uma boa prática de TDD iniciar o desenvolvimento do código de uma funcionalidade e, logo em seguida, testá-la.

22. **(IADES / EBSERH – 2013)** Assinale a alternativa que não corresponde a uma das fases do processo de desenvolvimento, dirigido a testes (TDD).

a) Executar o teste, com os outros testes implementados, que rodarão e fornecerão o resultado de que o software está sem problemas.
b) Escrever o teste para a funcionalidade e implementação.
c) Realizar a identificação do incremento de funcionalidade.
d) Implementar a funcionalidade e executar novamente o teste.
e) Implementar a próxima parte da funcionalidade, após todos os testes terem sido executados, com sucesso

23. **(PR-4 UFRJ / UFRJ – 2018)** O ciclo do TDD - Test Driven Development, ou, em português, Desenvolvimento Guiado por Testes consiste em:

a) implementar teste unitário falho, tornar o teste bem-sucedido e refatorar.
b) implementar a funcionalidade, executar teste unitário e refatorar.
c) implementar teste unitário falho, refatorar e tornar o teste bem-sucedido.
d) implementar a funcionalidade, refatorar e tornar o teste bem-sucedido.
e) refatorar, executar teste unitário e implementar a funcionalidade.

24. **(CESPE / STJ – 2015)** Um dos passos executados no ciclo de atividades do processo TDD é a criação de novos testes para as falhas encontradas no código original, sem alteração deste.

25. **(CS-UFG / AL-GO – 2015)** O desenvolvimento dirigido a testes (TDD, do Inglês Test-Driven Development) é uma abordagem de desenvolvimento de software na qual se intercalam testes e desenvolvimento de código. Uma das características da abordagem TDD é:

a) a sua utilidade no desenvolvimento de softwares novos.
b) o maior custo associado aos testes de regressão.
c) a redução da importância da automatização dos testes.
d) a sua adequação a processos de software sequenciais.

26. **(UECE-CEV / FUNCEME – 2018)** Test-driven Development (TDD) é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código (Sommerville, I. Engenharia de Software, 9a edição, 2011).

A respeito do TDD, é correto afirmar que:

a) consiste em um processo iterativo que se inicia escrevendo um código de uma funcionalidade do sistema e, logo em seguida, testa-o para saber se a implementação foi correta.
b) apesar de útil, não diminui o custo de testes de regressão do sistema.
c) sua utilização elimina a necessidade de testes de validação do sistema, uma vez que ele já foi testado incrementalmente.
d) apesar de ter sido apresentado como parte dos métodos ágeis, também pode ser usado em outros processos de desenvolvimento de software.

27. **(FAUGRS / UFRGS – 2018)** _____ é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código. Essencialmente, desenvolve-se um código de forma incremental em conjunto com um teste para este incremento. Não se avança para o próximo incremento até que o código desenvolvido passe no teste. Essa abordagem foi introduzida como parte de métodos ágeis, mas pode ser também usada em processos de desenvolvimento dirigido a planos.

Assinale a alternativa que preenche corretamente a lacuna do texto acima.

a) Desenvolvimento Guiado por Testes (TDD)
b) Desenvolvimento em Espiral
c) Engenharia Dirigida a Modelos (MDD)
d) Rational Unified Process (RUP)
e) Teste de Sistema

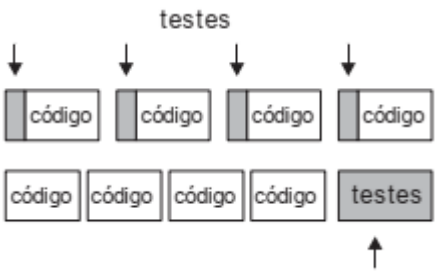
28. **(VUNESP / TCE-SP – 2015)** No Desenvolvimento Orientado a Testes (TDD), os casos de teste que definem o recurso a ser implementado devem ser elaborados:

a) assim que o código do teste estiver pronto.
b) antes de o código do recurso ser desenvolvido.
c) após o código do recurso ter sido completamente documentado.
d) simultaneamente com o desenvolvimento do código do recurso.
e) somente se o código do recurso apresentar erros.

29. **(IBFC / EMBASA – 2017)** No Ciclo de Desenvolvimento do TDD (Test-Driven Development), utiliza-se a estratégia que aplica três palavras-chaves (em inglês), que é denominada:

a) Red, Green, Refactor
b) White, Gray, Black
c) White, Black, Refactor
d) Green, Yellow, Red

30. **(FCC / CREMESP – 2016)** Considere a figura abaixo que apresenta duas abordagens de teste.



A figura:

a) ilustra as duas fases do TDD, que correspondem a escrever pequenos testes e testá-los no final.
b) mostra o ciclo conhecido como Vermelho-Verde-Refatora.
c) apresenta a diferença entre testes automatizados e testes manuais no XP.
d) mostra que um desenvolvedor que pratica TDD tem mais feedbacks do que um que escreve testes ao final.
e) evidencia que TDD é impraticável, pois o desenvolvedor gasta muito tempo escrevendo código de testes.

31. **(CESPE / ANATEL – 2014)** Em se tratando de desenvolvimento de softwares dirigidos a testes (TDD), a execução dos testes é realizada antes da implementação da funcionalidade.

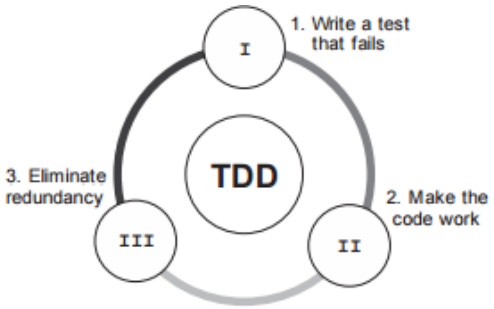
32. **(CESPE / TC/DF – 2014)** No TDD, o refatoramento do código deve ser realizado antes de se escrever a aplicação que deve ser testada.

33. **(CESPE / STJ – 2015)** No método de desenvolvimento TDD (Test Driven Development), o desenvolvedor escreve primeiro um caso de teste e, posteriormente, o código.

34. **(CESPE / MPE-PI – 2018)** O TDD possibilita o desenvolvimento de softwares fundamentado em testes. O ciclo de desenvolvimento do TDD segue os seguintes passos:

- escrever um teste que inicialmente não passa;
- adicionar uma nova funcionalidade do sistema;
- fazer o teste passar;
- realizar a integração contínua do código;
- escrever o próximo teste.

35. **(FCC / Pref. de Teresina/PI – 2016)** O Test Driven Development – TDD é uma das práticas sugeridas na eXtreme Programming – XP, onde o programador escreve o teste antes de escrever o código. O ciclo de desenvolvimento utilizando TDD é mostrado abaixo.



Considere:

I. Etapa inicial, onde se escreve um teste que falha, para alguma funcionalidade que ainda será Escrita.
II. Já com o teste criado, é o momento de executar o teste.
III. Eliminar códigos redundantes, remover acoplamentos, enfim, identificar pontos de melhoria no código.

As etapas I, II e III são, respectivamente,

a) Iniciação, Execução e Controle.
b) Red, Green e Refactor.
c) Iniciação, Atuação e Otimização.

d) Plan, Do e Check.
e) Planejamento, Execução e Melhoria.

36. **(FAUGRS / BANRISUL – 2018)** Considere as ações abaixo, executadas em desenvolvimento orientado a testes, Test-Driven Design (TDD).

I - Escrever código de teste.
II - Verificar se o teste falha.
III - Escrever código de produção.
IV - Executar teste até passar (reescrevendo o código de produção, se for necessário, até que o teste passe).
V - Refatorar código de produção e/ou de teste para melhorá-lo.

Considerando que se deseja incluir um novo caso de teste, assinale a alternativa que apresenta a sequência de ações que devem obrigatoriamente ocorrer para essa inclusão, segundo o TDD.

a) I, III e IV.
b) III, I e IV.
c) I, II, III e IV.
d) I, III, IV e V.
e) I, II, III, IV e V.

37. **(FGV / IBGE – 2017)** Test Driven Development (TDD) é uma prática muito utilizada no processo de desenvolvimento de sistemas computacionais. Analise as afirmativas a seguir sobre o uso da prática de TDD:

I. Tornam os testes de regressão mais demorados porque o desenvolvedor precisará fazer testes manuais várias vezes por dia.

II. Garante que os requisitos do sistema sejam atendidos porque o desenvolvedor escreverá o código de testes sempre que acabar a implementação do código do sistema.

III. Ajuda o desenvolvedor a escrever código de qualidade porque ele gastará parte do seu tempo escrevendo código de testes.

Está correto o que se afirma em:

a) somente I;
b) somente II;
c) somente III;
d) somente II e III;
e) I, II e III.

38. **(CESPE / ANATEL – 2014)** Na atividade de TDD (test-driven development), a escrita de teste primeiro define implicitamente tanto uma interface quanto uma especificação do comportamento para a funcionalidade que está sendo desenvolvida, estando, entretanto, a viabilidade do uso dessa abordagem limitada aos processos de desenvolvimento de software que seguem as práticas ágeis.

39. **(CESPE / TRE/MT – 2015)** Considere as seguintes etapas de um processo do tipo desenvolvimento orientado a testes (TDD).

I Implementar funcionalidade e refatorar.
II Identificar nova funcionalidade.
III Executar o teste.
IV Escrever o teste.
V Implementar a próxima parte da funcionalidade.

Assinale a opção que apresenta a sequência correta em que essas etapas devem ser realizadas.

a) I; IV; III; II; V
b) IV; III; II; I; V
c) I; IV; II; III; V
d) II; IV; III; I; V
e) IV; II; III; I; V

40. **(FCC / SEFAZ-SC – 2018)** O Test-Driven Development (TDD) é uma abordagem para o desenvolvimento de programas em que se intercalam testes e desenvolvimento de código. As etapas do processo fundamental de TDD são mostradas abaixo em ordem alfabética:

I. Escrever um teste para a funcionalidade identificada e implementá-lo como um teste automatizado.

II. Executar o teste, junto com os demais testes já implementados, sem implementar a nova funcionalidade no código.

III. Identificar e implementar uma outra funcionalidade, após todos os testes serem executados com sucesso.

IV. Identificar uma nova funcionalidade pequena para ser incrementada com poucas linhas em um código.

V. Implementar a nova funcionalidade no código e reexecutar o teste.

VI. Refatorar o código com melhorias incrementais até que o teste execute sem erros.

VII. Revisar a funcionalidade e o teste, caso o código execute sem falhar.

Considerando o item IV a primeira etapa e o item III a última etapa, a sequência intermediária correta das etapas do processo é:

a) I – II – VII – V e VI.
b) I – V – II – VII e VI.
c) I – VI – V – VII e II.
d) V – I – II – VII e VI.
e) V – I – VI – VII e II.

GABARITO

1. ERRADO
2. ERRADO
3. CORRETO
4. LETRA E
5. ERRADO
6. CORRETO
7. CORRETO
8. CORRETO
9. LETRA C
10. LETRA A
11. LETRA E
12. CORRETO
13. CORRETO
14. LETRA D
15. LETRA A
16. LETRA D
17. LETRA B
18. LETRA E
19. LETRA E
20. LETRA D
21. LETRA A
22. LETRA A
23. LETRA A
24. ERRADO
25. LETRA A
26. LETRA D
27. LETRA A
28. LETRA B
29. LETRA A
30. LETRA D
31. CORRETO
32. CORRETO
33. CORRETO
34. ERRADO
35. LETRA B
36. LETRA C
37. LETRA C
38. ERRADO
39. LETRA D
40. LETRA A

REFATORAÇÃO

Uma importante atividade sugerida por diversas metodologias ágeis de desenvolvimento de software, a Refatoração é uma técnica (inclusive preconizada pelo XP) de **reorganização que simplifica o projeto (ou código) de um componente de software sem modificar sua função ou seu comportamento**. Martin Fowler define refatoração da seguinte maneira:

> *"É o processo de mudar um sistema de software de tal forma que não altere o comportamento externo do código, embora melhore sua estrutura interna".*

Quando um software é refatorado ou refabricado, o projeto existente é examinado em termos de redundância, elementos de projeto não utilizados, algoritmos ineficientes ou desnecessários, estruturas de dados mal construídas ou inapropriadas, **ou qualquer outra falha de projeto que possa ser corrigida para produzir um projeto melhor**. *Vamos ver um exemplo?*

Uma primeira iteração de projeto poderia gerar um componente que apresentasse baixa coesão (realizar três funções que possuem apenas relacionamento limitado entre si). Após cuidadosa consideração, talvez decidamos que o componente devesse ser refabricado em três componentes distintos, cada um apresentando alta coesão. **O resultado será um software mais fácil de se integrar, testar e manter**.

Portanto a Refatoração consiste da melhoria da estrutura interna do código-fonte de um programa, enquanto preserva seu comportamento externo. Prestem bastante atenção: Refatoração não é reescrever o código! Refatoração não é consertar bugs! Refatoração não é melhorar aspectos observáveis do software, tais como sua interface.

Entre os benefícios esperados, podemos afirmar que a refatoração melhora atributos objetos de código (tamanho, duplicação, acoplamento, coesão, complexidade ciclomática, entre outros) que estão relacionados com facilidade de manutenção. **Além disso, ele ajuda a compreensão do código-fonte e encoraja o desenvolvedor a pensar sobre suas decisões de projeto**. *Bacana?*

EXERCÍCIOS COMENTADOS

1. **(FCC – 2010 – DPE/SP – Analista de Sistemas)** A refatoração é o processo de modificar um sistema de software para melhorar a estrutura interna do código sem alterar seu comportamento externo.

Comentários:

Portanto a Refatoração consiste da melhoria da estrutura interna do código-fonte de um programa, enquanto preserva seu comportamento externo*. Prestem bastante atenção: Refatoração não é reescrever o código! Refatoração não é consertar bugs! Refatoração não é melhorar aspectos observáveis do software, tais como sua interface.*

Conforme vimos em aula, a questão está perfeita!

Gabarito: Correto

---

2. **(CESPE – 2006 – CENSIPAM – Analista de Sistemas)** A refatoração modifica a estrutura interna de um software visando facilitar o entendimento e as futuras modificações sem alterar o comportamento apresentado pelo software. Não é uma prática que possa ser aplicada em processos de desenvolvimento ágeis, pois requer a construção de modelos tanto para o projeto de alto nível quanto para o projeto detalhado.

Comentários:

Portanto a Refatoração consiste da melhoria da estrutura interna do código-fonte de um programa, enquanto preserva seu comportamento externo*. Prestem bastante atenção: Refatoração não é reescrever o código! Refatoração não é consertar bugs! Refatoração não é melhorar aspectos observáveis do software, tais como sua interface.*

Uma importante atividade sugerida por diversas metodologias ágeis de desenvolvimento de software, a Refatoração é uma técnica ***(inclusive preconizada pelo XP****) de* ***reorganização que simplifica o projeto (ou código) de um componente de software sem modificar sua função ou seu comportamento****. Martin Fowler define refatoração da seguinte maneira:*

Conforme vimos em aula, a segunda sentença está incorreta! Ele tanto é aplicado que é explicitamente referenciado pelo XP.

Gabarito: Errado

---

3. **(CESPE – 2006 – CENSIPAM – Analista de Sistemas)** A refatoração é aplicável quando são identificados fragmentos de código que podem ser agrupados, expressões complicadas, atributos acessados mais por outras classes que pelas classes das quais são membros, enunciados condicionais complexos, códigos duplicados, longos métodos, longas classes, muitos parâmetros, métodos ou classes pouco usadas.

Comentários:

Entre os benefícios esperados, podemos afirmar que a refatoração melhora atributos objetos de código (tamanho, duplicação, acoplamento, coesão, complexidade ciclomática, entre outros) que estão relacionados com facilidade de manutenção. ***Além disso, ele ajuda a compreensão do código-fonte e encoraja o desenvolvedor a pensar sobre suas decisões de projeto****. Bacana?*

Conforme vimos em aula, a questão está perfeita!

Gabarito: Correto

---

4. **(CESPE – 2009 – INMETRO – Analista de Sistemas)** As técnicas de refatoração de código compreendem, entre outras, a remoção de números mágicos e a introdução de padrões de desenho.

Comentários:

Entre os benefícios esperados, podemos afirmar que a refatoração melhora atributos objetos de código (tamanho, duplicação, acoplamento, coesão, complexidade ciclomática, entre outros) que estão relacionados com facilidade de manutenção. ***Além disso, ele ajuda a compreensão do código-fonte e encoraja o desenvolvedor a pensar sobre suas decisões de projeto****. Bacana?*

Conforme vimos em aula, está perfeito! A refatoração é a reorganização interna do código, logo bastante ligada aos padrões de projeto. Já os número mágicos são aqueles não tem um significado documentado e esclarecido no código e que, em geral, não estão atribuídos diretamente a uma variável. Eles foram ficando por conta de modificações no sistema e atualmente ninguém sabe o que eles significam.

Exemplo: Suponha que um programa de cálculo trigonométrico faça uso do número π em diversos lugares. A princípio o programador usou a aproximação 3.14 e a colocou numericamente em todos os lugares que ela era necessária. O número 3.14 a princípio é facilmente reconhecível como π por qualquer pessoa com algum conhecimento de matemática. Porém nos testes o programador descobriu que precisaria de uma aproximação melhor, como 3.1415926. Agora ele tem que procurar todas as ocorrências de 3.14 no programa e substituí-la pela nova aproximação. Este procedimento é trabalhoso e sujeito a erros. Se o programador tivesse usado uma constante

com o nome PI, em vez do número mágico 3.14, bastaria mudar a aproximação na definição da constante.

Gabarito: Correto

---

5. **(CESPE – 2010 – INMETRO – Analista de Sistemas)** A técnica de refatoração, utilizada no paradigma de orientação a objetos, é mais bem enquadrada como uma técnica de reengenharia de software, isto é, que altera um software para reconstituí-lo em uma nova forma, função e implementação, que como uma técnica de engenharia reversa de software, isto é, uma técnica que analisa um software e cria novas representações abstratas do mesmo.

Comentários:

***Portanto a Refatoração consiste da melhoria da estrutura interna do código-fonte de um programa, enquanto preserva seu comportamento externo**. Prestem bastante atenção: Refatoração não é reescrever o código! Refatoração não é consertar bugs! Refatoração não é melhorar aspectos observáveis do software, tais como sua interface.*

Conforme vimos em aula, é exatamente o contrário! Lembrem-se que a refatoração muda dentro sem mudar fora, logo não se encaixa como uma técnica de reengenharia, porque essa altera o software para reconstruí-lo de uma nova forma.

Gabarito: Errado

---

6. **(CESPE – 2012 – BASA – Analista de Sistemas)** Denomina-se refatoração a atividade de reestruturação de programas, classes e métodos existentes para adaptá-los a alterações de funcionalidades e requisitos.

Comentários:

***Portanto a Refatoração consiste da melhoria da estrutura interna do código-fonte de um programa, enquanto preserva seu comportamento externo**. Prestem bastante atenção: Refatoração não é reescrever o código! Refatoração não é consertar bugs! Refatoração não é melhorar aspectos observáveis do software, tais como sua interface.*

Conforme vimos em aula, a questão está errada (mas a banca não entendeu dessa forma). Se houve alterações de funcionalidade, não é uma refatoração! Isso talvez seria uma reengenharia de software. Enfim, discordo!

Gabarito: Correto

---

7. **(CESPE – 2013 – BASA – Analista de Sistemas)** A refatoração objetiva tornar o código mais claro e limpo.

Comentários:

Entre os benefícios esperados, podemos afirmar que a refatoração melhora atributos objetos de código (tamanho, duplicação, acoplamento, coesão, complexidade ciclomática, entre outros) que estão relacionados com facilidade de manutenção. ***Além disso, ele ajuda a compreensão do código-fonte e encoraja o desenvolvedor a pensar sobre suas decisões de projeto****. Bacana?*

Conforme vimos em aula, a questão está perfeita!

Gabarito: Correto

---

8. **(CESPE – 2013 – BASA – Analista de Sistemas)** Ao refatorar um código, altera-se a funcionalidade do sistema.

Comentários:

Portanto a Refatoração consiste da melhoria da estrutura interna do código-fonte de um programa, enquanto preserva seu comportamento externo*. Prestem bastante atenção: Refatoração não é reescrever o código! Refatoração não é consertar bugs! Refatoração não é melhorar aspectos observáveis do software, tais como sua interface.*

Conforme vimos em aula, a questão está incorreta! A funcionalidade permanece a mesma!

Gabarito: Errado

---

9. **(CESPE – 2013 – UNIPAMPA – Analista de Sistemas)** No que concerne às mudanças futuras, a refatoração de um programa orientado a objetos e a manutenção preventiva têm propostas opostas.

Comentários:

Não, têm propostas similares! Ambas buscam melhorar a estrutura do código sem alterar seu comportamento externo.

Gabarito: Errado

---

10. **(CESPE – 2009 – ANAC – Analista de Sistemas)** A técnica conhecida como refactoring é constantemente aplicada no desenvolvimento baseado no método ágil extreme programming.

Comentários:

Uma importante atividade sugerida por diversas metodologias ágeis de desenvolvimento de software, a Refatoração é uma técnica ***(inclusive preconizada pelo XP****) de* ***reorganização que simplifica o projeto (ou código) de um componente de software sem modificar sua função ou seu comportamento****. Martin Fowler define refatoração da seguinte maneira:*

Conforme vimos em aula, está perfeito!

Gabarito: Correto

---

11. (CESPE – 2009 – INMETRO – Analista de Sistemas) A refabricação (ou refactoring) significa que primeiro deve ser desenvolvido um conjunto mínimo de funcionalidades que agreguem valor ao negócio e, depois, novas funcionalidades devem ser incrementadas ao produto já entregue.

Comentários:

Não faz o menor sentido! Nada de novas funcionalidades, é apenas otimização do comportamento interno.

Gabarito: Errado

---

12. (CESPE – 2010 – SAD/PE – Analista de Sistemas) Em ferramentas CASE, como refactoring, é melhor adotar-se uma abordagem formal que uma abordagem heurística.

Comentários:

Bem... a despeito de a questão tratar *refactoring* como uma ferramenta e, não, como uma técnica, é melhor adotar uma abordagem heurística, devido a imensa variedade de algoritmos, entradas, saídas, etc. Ademais, é uma abordagem puramente empírica, baseada em fatos reais, na procura de possíveis melhorias.

Gabarito: Errado

---

13. (CESPE – 2013 – STF – Analista de Sistemas) O refactoring aprimora o design de um software, reduz a complexidade da aplicação, remove redundâncias desnecessárias, reutiliza código, otimiza o desempenho e evita a deterioração durante o ciclo de vida de um código.

Comentários:

Entre os benefícios esperados, podemos afirmar que a refatoração melhora atributos objetos de código (tamanho, duplicação, acoplamento, coesão, complexidade ciclomática, entre outros) que estão relacionados com facilidade de manutenção. ***Além disso, ele ajuda a compreensão do código-fonte e encoraja o desenvolvedor a pensar sobre suas decisões de projeto****. Bacana?*

Conforme vimos em aula, a questão está perfeita!

Gabarito: Correto

---

14. (CESPE – 2015 – TCU – Analista de Sistemas) A cada nova funcionalidade de software adicionada na prática de refactoring (refatoração) em XP, a chance, o desafio e a coragem de alterar o código-fonte de um software são aproveitados como oportunidade para que o design do software adote uma forma mais simples ou em harmonia com o ciclo de vida desse software, ainda que isso implique a alteração de um código com funcionamento correto.

Comentários:

Uma importante atividade sugerida por diversas metodologias ágeis de desenvolvimento de software, a Refatoração é uma técnica ***(inclusive preconizada pelo XP)*** *de* ***reorganização que simplifica o projeto (ou código) de um componente de software sem modificar sua função ou seu comportamento****. Martin Fowler define refatoração da seguinte maneira:*

Conforme vimos em aula, a questão está completamente errada! Vimos insistentemente que a refatoração não adiciona novas funcionalidades. Quando vi que o gabarito preliminar veio como correto, achei um absurdo. Tinha certeza que a banca mudaria o gabarito, mas ela não deu o braço a torcer e apenas anulou a questão. Menos mal...

Gabarito: Anulada

---

15. (CESPE – 2017 – TRE/PE – Analista de Sistemas) Refactoring é o processo que:

a) implementa todas as funcionalidades da camada de model para depois implementar as camadas de controller e de viewer, nos casos em que a arquitetura MVC é utilizada.

b) efetua mudanças em um código existente e funcional sem alterar seu comportamento externo, com o objetivo de aprimorar a estrutura interna do código.

c) inclui funcionalidades extras no código, com o intuito de aprimorá-lo (rich source-code).

d) aprimora a extração e o refinamento iterativo dos requisitos do produto ainda na fase de planejamento do software, sendo considerado um valor na XP (extreme programming).

e) estabelece os métodos, um após o outro, para depois definir as classes e suas abstrações e implementar as interfaces.

Comentários:

Uma importante atividade sugerida por diversas metodologias ágeis de desenvolvimento de software, a Refatoração é uma técnica (inclusive preconizada pelo XP) de ***reorganização que simplifica o projeto (ou código) de um componente de software sem modificar sua função ou seu comportamento****. Martin Fowler define refatoração da seguinte maneira:*

"É o processo de mudar um sistema de software de tal forma que não altere o comportamento externo do código, embora melhore sua estrutura interna".

Conforme vimos em aula, nenhum desses itens faz sentido, exceto o segundo.

Gabarito: Letra B

LISTA DE EXERCÍCIOS

1. **(FCC – 2010 – DPE/SP – Analista de Sistemas)** A refatoração é o processo de modificar um sistema de software para melhorar a estrutura interna do código sem alterar seu comportamento externo.

2. **(CESPE – 2006 – CENSIPAM – Analista de Sistemas)** A refatoração modifica a estrutura interna de um software visando facilitar o entendimento e as futuras modificações sem alterar o comportamento apresentado pelo software. Não é uma prática que possa ser aplicada em processos de desenvolvimento ágeis, pois requer a construção de modelos tanto para o projeto de alto nível quanto para o projeto detalhado.

3. **(CESPE – 2006 – CENSIPAM – Analista de Sistemas)** A refatoração é aplicável quando são identificados fragmentos de código que podem ser agrupados, expressões complicadas, atributos acessados mais por outras classes que pelas classes das quais são membros, enunciados condicionais complexos, códigos duplicados, longos métodos, longas classes, muitos parâmetros, métodos ou classes pouco usadas.

4. **(CESPE – 2009 – INMETRO – Analista de Sistemas)** As técnicas de refatoração de código compreendem, entre outras, a remoção de números mágicos e a introdução de padrões de desenho.

5. **(CESPE – 2010 – INMETRO – Analista de Sistemas)** A técnica de refatoração, utilizada no paradigma de orientação a objetos, é mais bem enquadrada como uma técnica de reengenharia de software, isto é, que altera um software para reconstituí-lo em uma nova forma, função e implementação, que como uma técnica de engenharia reversa de software, isto é, uma técnica que analisa um software e cria novas representações abstratas do mesmo.

6. **(CESPE – 2012 – BASA – Analista de Sistemas)** Denomina-se refatoração a atividade de reestruturação de programas, classes e métodos existentes para adaptá-los a alterações de funcionalidades e requisitos.

7. **(CESPE – 2013 – BASA – Analista de Sistemas)** A refatoração objetiva tornar o código mais claro e limpo.

8. **(CESPE – 2013 – BASA – Analista de Sistemas)** Ao refatorar um código, altera-se a funcionalidade do sistema.

9. **(CESPE – 2013 – UNIPAMPA – Analista de Sistemas)** No que concerne às mudanças futuras, a refatoração de um programa orientado a objetos e a manutenção preventiva têm propostas opostas.

10. **(CESPE – 2009 – ANAC – Analista de Sistemas)** A técnica conhecida como refactoring é constantemente aplicada no desenvolvimento baseado no método ágil extreme programming.

11. **(CESPE – 2009 – INMETRO – Analista de Sistemas)** A refabricação (ou refactoring) significa que primeiro deve ser desenvolvido um conjunto mínimo de funcionalidades que agreguem valor ao negócio e, depois, novas funcionalidades devem ser incrementadas ao produto já entregue.

12. **(CESPE – 2010 – SAD/PE – Analista de Sistemas)** Em ferramentas CASE, como refactoring, é melhor adotar-se uma abordagem formal que uma abordagem heurística.

13. **(CESPE – 2013 – STF – Analista de Sistemas)** O refactoring aprimora o design de um software, reduz a complexidade da aplicação, remove redundâncias desnecessárias, reutiliza código, otimiza o desempenho e evita a deterioração durante o ciclo de vida de um código.

14. **(CESPE – 2015 – TCU – Analista de Sistemas)** A cada nova funcionalidade de software adicionada na prática de refactoring (refatoração) em XP, a chance, o desafio e a coragem de alterar o código-fonte de um software são aproveitados como oportunidade para que o design do software adote uma forma mais simples ou em harmonia com o ciclo de vida desse software, ainda que isso implique a alteração de um código com funcionamento correto.

15. **(CESPE – 2017 – TRE/PE – Analista de Sistemas)** Refactoring é o processo que:

    a) implementa todas as funcionalidades da camada de model para depois implementar as camadas de controller e de viewer, nos casos em que a arquitetura MVC é utilizada.

    b) efetua mudanças em um código existente e funcional sem alterar seu comportamento externo, com o objetivo de aprimorar a estrutura interna do código.

    c) inclui funcionalidades extras no código, com o intuito de aprimorá-lo (rich source-code).

    d) aprimora a extração e o refinamento iterativo dos requisitos do produto ainda na fase de planejamento do software, sendo considerado um valor na XP (extreme programming).

    e) estabelece os métodos, um após o outro, para depois definir as classes e suas abstrações e implementar as interfaces.

GABARITO

1. CORRETO
2. ERRADO
3. CORRETO
4. CORRETO
5. ERRADO
6. CORRETO
7. CORRETO
8. ERRADO
9. ERRAD
10. CORRETO
11. ERRADO
12. ERRADO
13. CORRETO
14. ANULADA
15. LETRA B

INTEGRAÇÃO, ENTREGA E IMPLANTAÇÃO CONTÍNUA

1 – Conceitos Básicos

A integração contínua é um termo originado na metodologia ágil XP e utilizado em diversas metodologias, consistindo em algo simples: o desenvolvedor integra o código alterado e/ou desenvolvido ao projeto principal na mesma frequência com que as funcionalidades são desenvolvidas, sendo feito idealmente muitas vezes ao dia ao invés de apenas uma vez.

O objetivo principal de utilizar a integração contínua é verificar se as alterações ou novas funcionalidades não criaram novos defeitos no projeto já existente. A prática da integração contínua pode ser feita através de processos manuais ou automatizados, utilizando ferramentas como o Jenkins, Hudson, entre outros. Um dos nossos maiores guias, Martin Fowler, já dizia:

> *"A Integração Contínua se trata de uma prática de desenvolvimento de software em que os membros de um time integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente – podendo haver múltiplas integrações por dia. Cada integração é verificada por um build automatizado (incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente. "*

Basicamente, a grande vantagem da integração contínua está no feedback instantâneo. A prática da integração contínua se mostra muito eficaz nas equipes de desenvolvimento e está sendo usada amplamente. Inúmeros projetos open-source usam várias máquinas dedicadas a serem servidores de integração, rodando muitas vezes em softwares/plataformas diferentes.

Em equipes onde há distância geográfica, recomenda-se o uso de integração contínua assíncrona e, em equipes que trabalham no mesmo espaço físico, recomenda-se o uso de integração contínua síncrona. **Em ambos casos, é importante ter pelo menos uma máquina dedicada a integração que seja um clone do ambiente de produção, pois quanto mais rápido for o feedback melhor**.

Dessa forma, podemos afirmar que a prática de Integração Contínua busca dois objetivos: minimizar a duração e esforço requeridos por cada episódio de integração; e ser capaz de entregar uma versão do produto que possa ser lançada a qualquer momento**. É requerido um procedimento de integração que seja reproduzível no mínimo e, em grande parte, automatizável**.

Essa automatização não visa apenas evitar trabalho repetitivo. Ela na verdade permite aos desenvolvedores alcançarem um nível muito maior de produtividade e qualidade. Um problema recorrente no desenvolvimento de software é que os erros ocultos nos programas que desenvolvemos são cada vez mais caros e difíceis de corrigir à medida que o tempo avança.

Qualquer pessoa que trabalhe com programação já deve ter passado pela situação de achar que um programa estava quase pronto e "só faltava testar". **Depois descobre-se que, na verdade, muita coisa**

ainda estava errada ou até mesmo faltando no código. Outra situação recorrente é a "preguiça" de testar o software, ou pelo menos a prática de postergar os testes.

Sem testes e *deploys* automatizados, o desenvolvedor tende a escrever uma grande quantidade de código antes de realmente colocar o programa para executar, afinal se o processo é trabalhoso e toma muito tempo, ele vai evitar de fazer isso ao máximo. **Só que no final, gasta-se muito tempo para corrigir todos os "detalhes" que não estavam corretos**.

Para mitigar todos esses problemas, surgiu a Integração Contínua – justamente para automatizar o processo para que, com bastante frequência, uma ou mais vezes ao dia, seja possível integrar todas as alterações de todos os desenvolvedores envolvidos no projeto e realizar um teste geral. **Tem uma metáfora que eu gosto de usar bastante para explicar esse tema: um filme de ação**.

Ora, todo filme é composto de várias cenas. Em nosso caso, uma aplicação é composta de vários builds. Toda vez que eu adiciono uma nova cena em um filme, o editor faz um teste rodando todo o filme novamente para ver se o filme faz sentido com a nova cena. Caso o teste falhe, i.e., a nova cena não se encaixe na história, ela deve ser refeita! **Em nosso caso, dizemos que ele a build foi quebrada**.

Desta feita, em vez de construir vários componentes separadamente e depois juntá-los em um software final, nós vamos integrar continuamente. Novos builds só são integrados quando o build anterior é corrigido. Percebam que esses procedimentos auxiliam a reduzir riscos e a entregar soluções livres de defeitos. A Integração Contínua segue as seguintes práticas:

AS PRÁTICAS...
1. Mantenha um único repositório de código-fonte.
2. Automatize um build.
3. Faça um build auto-testável.
4. Todo commit devem ser um build na máquina de integração.
5. Mantenha os builds rápidos.
6. Teste em uma cópia do ambiente de produção.
7. Mantenha fácil que todos consigam o último executável.
8. Todos consegue visualizar o processo.
9. Automatização do deployment.

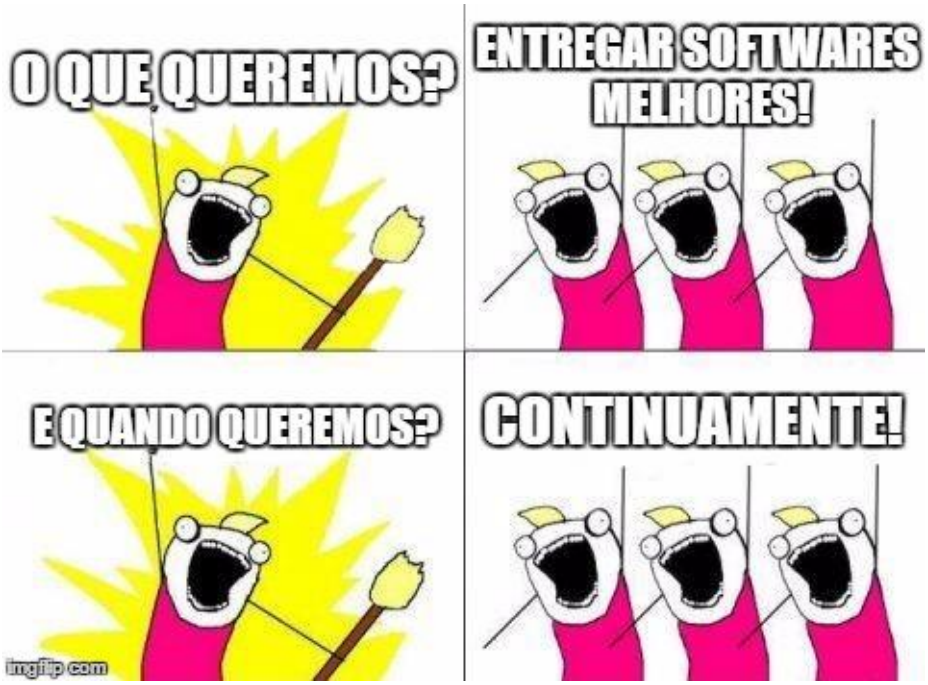
COMO FAZER...
10. Desenvolvedores devem fazer checkout do código em seus workspaces privados.
11. Quando finalizado, o commit modifica o repositório.
12. O Servidor de CI monitora o repositório e faz checkout das mudanças quando elas ocorrem.
13. O Servidor de CI constrói o sistema e roda testes de integração e testes de unidade.
14. O Servidor de CI lança artefatos implantáveis para testes.
15. O Servidor de CI atribui um rótulo de build para a versão do código que ele construiu.
16. O Servidor de CI informa ao time sobre o sucesso do build.
17. Se a build ou teste falhar, o Servidor de CI alerta a equipe.
18. A equipe corrige o problema na melhor oportunidade.
19. Continue a integrar continuamente e a testar durante o projeto.

Vamos falar um pouco sobre a entrega contínua. Uma equipe ágil geralmente entrega seu produto nas mãos dos usuários finais, ouvindo seus feedbacks (críticas e elogios). A frequência de entregas varia de acordo com aspectos técnicos e negociais do contexto que se encontra, mas se nós tivéssemos que chutar um número, diríamos que há uma entrega a cada quatro a seis iterações, no máximo.

Em contextos técnicos favoráveis, como desenvolvimento web, um ritmo mais frequente de entregas pode ser alcançado (Ex: uma entrega a cada iteração). Já algumas equipes vão ao limite dessa prática por meio de *continous deployments*. **Apresentar a última versão do produto para o gerente de projetos/produtos para teste não é suficiente, nem entregar a uma equipe de garantia de qualidade**.

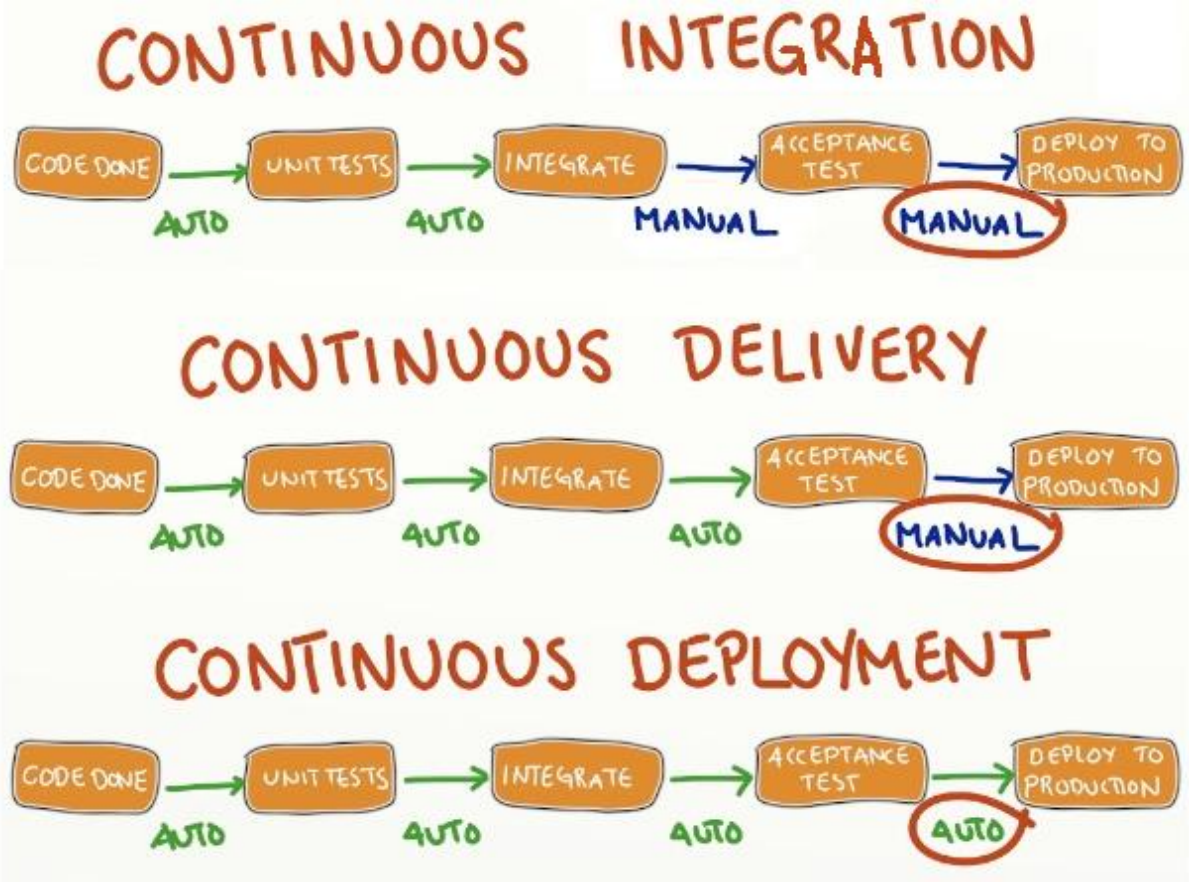
Uma entrega, em seu sentido primário, deve ser pelo menos uma versão beta avaliada por usuários representativos. Em alguns casos (como em softwares embarcados), não é possível organizar entregas contínuas para todos os usuários, mas isso não deve se tornar um pretexto para desistir das entregas contínuas mesmo que somente para alguns usuários.

Entre os benefícios esperados, busca-se mitigar a falha de planejamento bastante conhecida de descobrir atrasos muito tarde. Ela ajuda a validar o ajuste do produto ao mercado mais cedo, além de fornecer feedbacks adiantados sobre a qualidade e estabilidade do produto. **Por fim, ele permite um retorno mais rápido do investimento (*ROI – Return On Investment*) sobre o produto**.



Professor, qual a diferença entre Integração, Entrega e Implantação Contínua? **A primeira é o processo de testes e integração de um novo código com a base de código já existente – geralmente são feitos testes automatizados**. A segunda é o processo de garantir que o novo código pode ser implantado no ambiente de produção a qualquer momento – geralmente são feitos testes de comportamento.

Por fim, o terceiro trata do processo de publicação (deploy) no ambiente de produção, tão logo você esteja certo de que o código passou por todos os testes e está pronto para ser publicado. **Se o código já passou pelos testes automatizados e pelos testes manuais, visuais e de comportamento, então ele pode ser publicado de forma automatizada**. Observem a imagem abaixo:



Quem ainda não entendeu, vai entender agora! **Imaginem que temos uma equipe de desenvolvimento formada por quatro pessoas trabalhando em um sistema**. Cada desenvolvedor está focado em implementar uma funcionalidade diferente. Caso o primeiro desenvolvedor termine sua parte na codificação, ele pode fazer um *commit* do código em uma ferramenta de controle de versão (Ex: Git).

A Integração Contínua é uma prática que pode ser aplicada por meio de uma ferramenta (Ex: Jenkins). Essa ferramenta pode ser configurada para, de tempos em tempos, buscar tudo que foi *comitado* pelos desenvolvedores e realizar testes de unidade e de integração automaticamente**. Caso algum problema seja encontrado, temos um feedback imediato**.

Os desenvolvedores são notificados e ninguém faz check-in do repositório até que o problema seja resolvido. **Se os testes passarem, o primeiro desenvolvedor saberá que o código que ele implementou não quebrou o sistema**. Então, percebam que a integração contínua consiste em, periodicamente, realizar testes automatizados e integrar novos códigos ao sistema (via de regra, no ambiente de desenvolvimento).

A diferença entre Entrega Contínua e Implantação Contínua é mais sutil! A primeira é uma prática ágil na qual o software é construído de tal forma que ele pode ser colocado em produção a qualquer momento. **A segunda é uma prática ágil na qual o software é construído de tal forma que ele é colocado em produção em determinado momento (isso é configurável)**.

A grande diferença é simplesmente por conta do último ponto no nosso desenho. Em outras palavras, a Entrega Contínua sempre **apresenta uma versão candidata para publicação após cada funcionalidade passar por todos os estágios** e, então, o negócio poderá decidir quando colocá-la efetivamente em produção. Já na Implantação Contínua, a entrada em produção ocorre automaticamente.

Do início, vamos andar pelo desenho? Idealmente, o terceiro desenvolvedor pode terminar uma nova funcionalidade e realizar o *commit* do seu código para sua ferramenta de controle de versão. **Serão realizados testes de unidade e testes de integração automatizados**. Passando por esses estágios, o código é integrado ao ambiente de desenvolvimento.

Agora, todos os desenvolvedores poderão usufruir dessa nova funcionalidade implementada pelo terceiro desenvolvedor. Caso tenhamos também Entrega Contínua, o fluxo não para por aí! **De tempos em tempos, o sistema integrado passará por testes de aceitação automatizados**. Caso não haja erros, ele estará pronto para ser implantado em um ambiente de produção.

É comum que esse código fique em um ambiente de homologação, aguardando o negócio decidir quando ele deverá ser implantado, finalmente, no ambiente de produção. Caso tenhamos também a Implantação Contínua, o fluxo continua e, não mais será uma decisão do negócio a implantação ou não do software em produção – a implantação ocorrerá automaticamente. Fim :)

EXERCÍCIOS COMENTADOS

1. **(CESGRANRIO – 2010 – PETROBRÁS – Analista de Sistemas – C)** É comum, na Engenharia de Software, o uso de ferramentas de software que auxiliam na realização de diversas atividades do desenvolvimento. Nesse contexto, ferramentas de integração contínua são destinadas a automatizar a implantação do produto de software no ambiente de produção.

Comentários:

> ***A integração contínua é um termo originado na metodologia ágil XP e utilizado em diversas metodologias, consistindo em algo simples:*** *o desenvolvedor integra o código alterado e/ou desenvolvido ao projeto principal na mesma frequência com que as funcionalidades são desenvolvidas, sendo feito muitas vezes ao dia ao invés de apenas uma vez.*

Conforme vimos em aula, não se trata de automatizar a implantação, mas de oferecer feedbacks tempestivos por meio da integração a todo momento. Automatizar a implantação é apenas um detalhe!

Gabarito: Errado

---

2. **(CESPE – 2004 – TCE/PE – Analista de Sistemas)** A prática de integração contínua depende fortemente do uso de ferramentas de build e controle de versão.

Comentários:

> *Cada integração é verificada por um* ***build automatizado*** *(incluindo testes) para detectar erros de integração o mais rápido possível. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração (...)*

Perfeito! Necessita de ferramentas de build e controle de versão...

Gabarito: Correto

---

3. **(FCC – 2006 – BACEN – Analista de Sistemas – II)** A técnica de *Continuous Integration* diz que o código desenvolvido por cada par de desenvolvedores deve ser integrado ao código base constantemente. Quanto menor o intervalo entre cada integração, menor a diferença entre os códigos desenvolvidos e maior a probabilidade de identificação de erros, pois cada vez que o código é integrado, todos os *unit tests* devem ser executados, e, se algum deles falhar, é porque o código recém integrado foi o responsável por inserir erro no sistema.

Comentários:

> ***A integração contínua é um termo originado na metodologia ágil XP e utilizado em diversas metodologias, consistindo em algo simples:*** *o desenvolvedor integra o código alterado e/ou desenvolvido*

ao projeto principal na mesma frequência com que as funcionalidades são desenvolvidas, sendo feito idealmente muitas vezes ao dia ao invés de apenas uma vez.

Conforme vimos em aula, está perfeito!

Gabarito: Correto

---

4. **(CESPE – 2015 – TCU – Analista de Sistemas)** Para que a prática de integração contínua seja eficiente, é necessário parametrizar e automatizar várias atividades relativas à gerência da configuração, não somente do código-fonte produzido, mas também de bibliotecas e componentes externos.

Comentários:

Questão perfeita! Parametrizam-se e automatizam-se componentes como código-fonte, bibliotecas, scripts de build, entre outros, ajustando a dependência entre eles.

Gabarito: Correto

LISTA DE EXERCÍCIOS

1. **(CESGRANRIO – 2010 – PETROBRÁS – Analista de Sistemas – C)** É comum, na Engenharia de Software, o uso de ferramentas de software que auxiliam na realização de diversas atividades do desenvolvimento. Nesse contexto, ferramentas de integração contínua são destinadas a automatizar a implantação do produto de software no ambiente de produção.

2. **(CESPE – 2004 – TCE/PE – Analista de Sistemas)** A prática de integração contínua depende fortemente do uso de ferramentas de build e controle de versão.

3. **(FCC – 2006 – BACEN – Analista de Sistemas – II)** A técnica de *Continuous Integration* diz que o código desenvolvido por cada par de desenvolvedores deve ser integrado ao código base constantemente. Quanto menor o intervalo entre cada integração, menor a diferença entre os códigos desenvolvidos e maior a probabilidade de identificação de erros, pois cada vez que o código é integrado, todos os *unit tests* devem ser executados, e, se algum deles falhar, é porque o código recém integrado foi o responsável por inserir erro no sistema.

4. **(CESPE – 2015 – TCU – Analista de Sistemas)** Para que a prática de integração contínua seja eficiente, é necessário parametrizar e automatizar várias atividades relativas à gerência da configuração, não somente do código-fonte produzido, mas também de bibliotecas e componentes externos.

GABARITO

1. ERRADO
2. CORRETO
3. CORRETO
4. CORRETO

ESSA LEI TODO MUNDO CONHECE:
PIRATARIA É CRIME.

Mas é sempre bom revisar o porquê e como você pode ser prejudicado com essa prática.

1. 

Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.

2. 

Pirata divulga ilicitamente (grupos de rateio), utilizando-se do anonimato, nomes falsos ou laranjas (geralmente o pirata se anuncia como formador de "grupos solidários" de rateio que não visam lucro).

3. 

Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).

4. 

Pirata compra, muitas vezes, clonando cartões de crédito (por vezes o sistema anti-fraude não consegue identificar o golpe a tempo).

5. 

Pirata fere os Termos de Uso, adultera as aulas e retira a identificação dos arquivos PDF (justamente porque a atividade é ilegal e ele não quer que seus fakes sejam identificados).

6. 

Pirata revende as aulas protegidas por direitos autorais, praticando concorrência desleal e em flagrante desrespeito à Lei de Direitos Autorais (Lei 9.610/98).

7. 

Concurseiro(a) desinformado participa de rateio, achando que nada disso está acontecendo e esperando se tornar servidor público para exigir o cumprimento das leis.

8. 

O professor que elaborou o curso não ganha nada, o site não recebe nada, e a pessoa que praticou todos os ilícitos anteriores (pirata) fica com o lucro.

Deixando de lado esse mar de sujeira, aproveitamos para agradecer a todos que adquirem os cursos honestamente e permitem que o site continue existindo.