Aula 00 - Prof. Paolla Ramos e Raphael Lacerda

CAGEPA (Analista de Sistemas - Sistemas de TI) Desenvolvimento de software - 2024 (Pós-Edital)

Autor:

Felipe Mathias, Paolla Ramos

20 de Outubro de 2024

Índice

1) Desenvolvimento de Software - Apresentação do Professor 3

2) Apresentação Flashcards 5

3) Análise Estática de Código-Fonte - Conceitos Básicos - Teoria 7

4) Análise Estática de Código-Fonte - Conceitos Básicos - Questões Comentadas 12

5) Análise Estática de Código-Fonte - Conceitos Básicos - Lista de Questões 14

6) Análise Estática de Código-Fonte - Clean Code - Teoria 16

7) Análise Estática de Código-Fonte - Clean Code - Questões Comentadas 21

8) Análise Estática de Código-Fonte - Clean Code - Lista de Questões 25

9) Análise Estática de Código-Fonte - SonarQube - Teoria 28

10) Análise Estática de Código-Fonte - SonarQube - Questões Comentadas 31

11) Análise Estática de Código-Fonte - SonarQube - Lista de Questões 32

APRESENTAÇÃO

PROF. PAOLLA RAMOS

FORMADA EM SISTEMAS DE INFORMAÇÃO PELA UNIVERSIDADE FEDERAL DE OURO PRETO,
PÓS-GRADUADA EM SISTEMAS DE INFORMAÇÃO
DIREITO TRIBUTÁRIO
DIREITO ADMINISTRATIVO
AUDITORA FISCAL ESPECIALISTA EM TI.

Olá, pessoal!! Meu nome é Paolla Ramos, sou Auditora Fiscal especialista em TI do ISS-Aracaju. Trabalhar nesse fisco incrível tem sido uma experiência fantástica!!

Pessoal, eu sou uma pessoa normal, assim como vocês. No início, achava que conquistar a aprovação em um concurso de alto nível era quase impossível, até que provei o contrário! Querem saber qual foi o segredo? Foi o hiper foco, galera! Não existe uma fórmula mágica, e eu nunca fui considerada "superinteligente" ou a primeira aluna na turma. No entanto, sempre fui MUITO DETERMINADA, PERSISTENTE.

A equipe de TI e eu estamos aprimorando nossas aulas de forma gradativa para oferecer o melhor conteúdo possível. Sabemos que o estudo pode ser complexo, especialmente por meio de livros eletrônicos, por isso, recomendo estudar em conjunto com as vídeo-aulas.

Além disso, informo que estamos trabalhando na atualização dos cursos neste exato momento! Estamos refazendo a formatação, adicionando questões e diagramas, entre outros aprimoramentos. Gradualmente, os cursos ficaram mais completos e aprofundados. E, para acompanhar as tecnologias mais recentes, novas videoaulas também estão a caminho.

Caso surja alguma demanda, não hesitem em contactar no fórum. Se preferirem, também podem entrar em contato pelo Instagram **@prof.paollaramos**. Eu amo ajudar os alunos e estou disponível para esclarecer quaisquer dúvidas que possam surgir.

A minha missão aqui é dar o meu melhor para ajudar cada um de vocês a conquistar a aprovação também! Podem contar comigo sempre que precisarem.

Então, minha ideia aqui é fazer o meu melhor para que você também consiga ser aprovado! Sempre que precisar, pode contar comigo. Meu instagram é:

ESTRATÉGIA FLASHCARDS

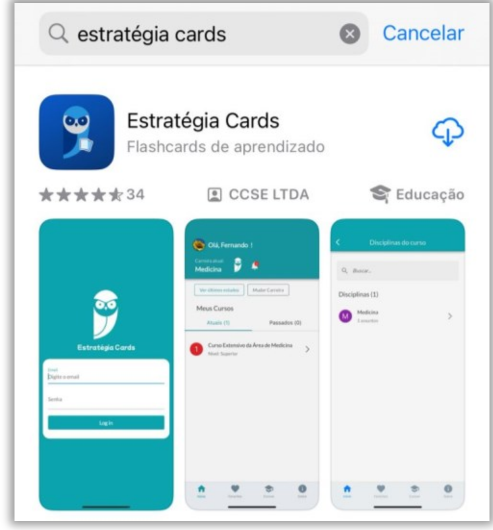
📚 Você tem dificuldade de estudar, memorizar e revisar os conteúdos que estuda em nossas aulas? Então nós temos a ferramenta perfeita para você!

Apresentamos o **Estratégia Cards**: app de flashcards que vai revolucionar sua forma de **estudar** e **revisar** conteúdos de provas de concurso público. Com nossa tecnologia inovadora e interface amigável, você dominará os tópicos mais complexos de maneira eficiente e divertida.

✨ Recursos do Estratégia Cards:

Recurso	Descrição
Curadoria de Flashcards	Flashcards criados e revisados por professores especializados em cada área, com qualidade e voltados para concursos públicos.
Flashcards Personalizados	Crie seus próprios flashcards, cobrindo os principais tópicos e matérias dos concursos públicos.
Repetição Espaçada	Técnica de aprendizagem que envolve revisar informações em intervalos crescentes para melhorar a retenção de longo prazo e combater o esquecimento.
Estatísticas Personalizadas	Visualize graficamente o percentual de acertos, erros ou dúvidas dos decks estudados.
Modo Offline	Estude em qualquer lugar, mesmo sem conexão à internet, fazendo o download dos decks.
Estudo por Áudio	*Está dirigindo ou fazendo esteira e quer continuar estudando?* Basta utilizar a opção "Escutar".
Decks Favoritos	Você pode escolher decks específicos como favoritos e visualizá-los em uma aba separada do app.
Opções de Estudo	Você poderá estudar todos os cards de um deck; ou apenas os que você errou; ou apenas os que você não estudou ainda; entre outras opções.

📲 E como eu consigo baixar?



É muito fácil! Basta pesquisar por "Estratégia Cards" na loja oficial do seu smartphone.

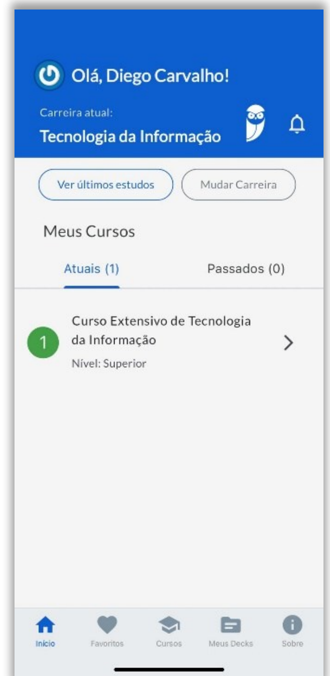
Se você tiver um Android, basta acessar a **Google Play**;

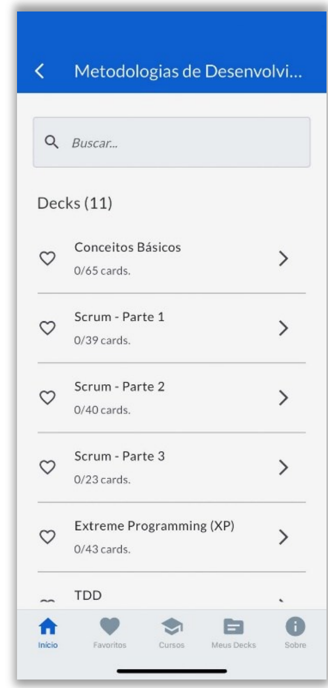
Se for tiver um iPhone, basta acessar a **App Store (iOS)**.

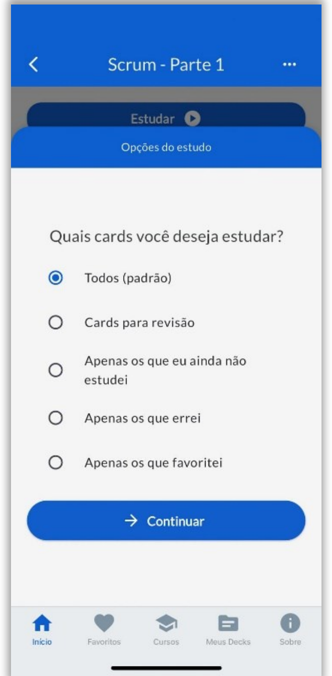
E para acessar?

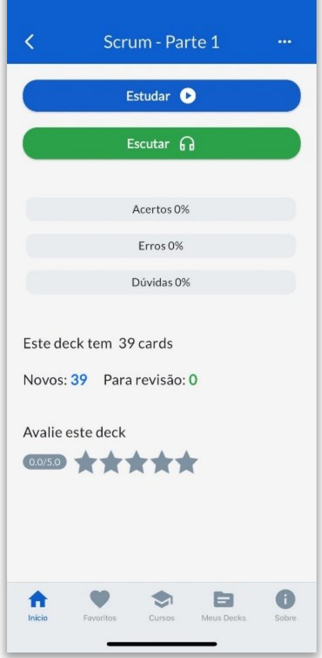
Para acessar, basta ter uma conta no Estratégia Concursos. Em seguida, utilize suas credenciais de login e senha para acessar o aplicativo. Por fim, acessa a carreira de Tecnologia da Informação.

Como utilizar o app:









ANÁLISE ESTÁTICA DE CÓDIGO-FONTE

A Análise Estática de Código-Fonte trata do processo de detecção de erros código sem, de fato, executá-lo – pode ser considerado um processo de revisão automática de código1! Em geral, é utilizada para encontrar bugs ou garantir conformidade com boas práticas de programação. Querem um exemplo comum? Um compilador! Ele é capaz de encontrar erros léxicos, sintáticos e, até mesmo, semânticos.

Sommerville define como: "Técnica automatizada de análise de programa na qual o programa é analisado detalhadamente para encontrar condições potencialmente errôneas". Inspeções são uma forma de análise estática, em que se examina um programa sem executá-lo. As inspeções são frequentemente dirigidas por checklists de erros e heurísticas que identificam erros comuns em diferentes linguagens.

Para alguns erros e heurísticas, é possível automatizar o processo de verificação de programas em relação a essa lista, o que resultou no desenvolvimento de analisadores estáticos automatizados para diferentes linguagens de programação. Analisadores estáticos são ferramentas de software que varrem o código-fonte de um programa e detectam possíveis defeitos e anomalias.

Eles analisam o código e, assim, reconhecem os tipos de declarações no programa. Podem portanto detectar se as declarações estão bem formuladas, fazer inferências sobre o fluxo de controle do programa e, em muitos casos, computam o conjunto de todos os valores possíveis para os dados de programa. Eles complementam os recursos de detecção de erros providos pelo compilador da linguagem.

Podem ser usados como parte do processo de inspeção ou como uma atividade separada do processo de verificação e validação. A intenção da análise estática automática é chamar a atenção do inspetor para anomalias do programa, como variáveis usadas sem serem iniciadas, variáveis não usadas ou dados cujos valores poderiam ficar fora de sua extensão.

As anomalias são frequentemente um resultado de erros de programação ou omissões, de modo que eles enfatizam coisas que poderiam sair erradas quando o programa fosse executado. Entretanto, você deve compreender que essas anomalias não são necessariamente defeitos de programa. Eles podem ser determinados ou não ter consequências adversas.

O BSTQB define Análise Estática de Código-Fonte como a prática de encontrar defeitos no código-fonte e na modelagem. Análise Estática é feita sem a execução do software examinado pela ferramenta; já o teste dinâmico executa o software. Análise estática pode localizar defeitos

[1] Na maioria das vezes, a análise é executada em alguma versão do código-fonte, e em outros casos é executada no código-objeto (i.e., código de máquina ou de montagem).

que são dificilmente encontrados em testes. Como as revisões, a análise estática encontra defeitos ao invés de falhas.

Ferramentas de análise estática analisam o código do programa (ex.: fluxo de controle e fluxo de dados), gerando, como saída, arquivos do tipo HTML e XML, por exemplo. Ferramentas de análises estáticas são tipicamente usadas por desenvolvedores antes e durante o teste de componente e de integração e por projetistas durante a modelagem do software.

BENEFÍCIOS DA ANÁLISE ESTÁTICA DE CÓDIGO-FONTE
Detecção de defeitos antes da execução do teste.
Conhecimento antecipado sobre aspectos suspeitos no código através de métricas, por exemplo, na obtenção de uma medida da alta complexidade.
Identificação de defeitos dificilmente encontrados por testes dinâmicos.
Detecção de dependências e inconsistências em modelos de software, como links perdidos.
Aprimoramento da manutenibilidade do código e construção.
Prevenção de defeitos, se as lições forem aprendidas pelo desenvolvimento.

DEFEITOS MAIS COMUNS DESCOBERTOS POR ANÁLISE ESTÁTICA
Referência a uma variável com valor indefinido.
Inconsistências entre as interfaces dos módulos e componentes.
Variáveis que nunca são usadas ou impropriamente declaradas.
Código morto.
Falta de lógica ou lógica errada (loops infinitos).
Construções excessivamente complicadas.
Violação de padrões de programação.

Vulnerabilidade na segurança.
Violação de sintaxe e de modelos.

Os estágios envolvidos na análise estática incluem:

- Análise de Fluxo de Controle: identifica e enfatiza os loops com vários pontos de saída ou de entrada e código inacessível. Código inacessível é aquele código cercado por declarações incondicionais 'goto' ou em uma ramificação de uma declaração condicional na qual a condição de guarda nunca pode ser verdadeira, portanto nunca é acessada.

- Análise de Uso de Dados: enfatiza como as variáveis do programa são usadas. Detecta variáveis usadas sem prévia iniciação, variáveis escritas duas vezes sem uma tarefa de impedimento e variáveis declaradas mas que nunca são usadas. A análise de uso de dados também descobre testes não eficientes nos quais a condição do teste é redundante.

- Análise de Interface: verifica a consistência das declarações de rotina e de procedimento e seus usos. Ela é desnecessária se uma linguagem com tipagem forte for usada, uma vez que o compilador realiza essas verificações. Detecta erros de tipo em linguagens com tipagem fraca; funções e procedimentos declarados e nunca chamados; ou resultados de funções que nunca são usados.

- Análise de Fluxo de Informações: identifica as dependências entre variáveis de entrada e de saída. Embora não detecte anomalias, mostra como o valor de cada variável é derivada de outros valores de variáveis. Com essas informações, a inspeção de código deve ser capaz de encontrar valores que foram erroneamente computados.

- Análise de Caminho: identifica todos os caminhos possíveis por meio do qual o fluxo de um programa pode passar durante sua execução e estabelece as declarações executadas naquele caminho. Essencialmente, ela elucida o controle do programa e permite que cada predicado possível seja analisado individualmente.

Analisadores estáticos são particularmente valiosos quando uma linguagem de programação, como C, for usada. A linguagem C não possui regras de tipagem estritas e a verificação que o compilador C pode executar é limitada. Portanto, é fácil para programadores cometerem erros e a ferramenta de análise estática pode descobrir automaticamente alguns dos defeitos de programa resultantes.

Isso é particularmente importante quando C (e, em menor extensão, C++) for usada para o desenvolvimento de sistemas críticos. Nesse caso, a análise estática pode descobrir um grande número de erros potenciais e reduzir significativamente os custos de teste. No entanto, os

projetistas de linguagens de programação modernas, como Java[2], têm removido algumas características propensas a erro.

Todas as variáveis devem ser iniciadas, não há declarações 'goto'; assim, os códigos inacessíveis são menos prováveis de serem criados acidentalmente e o gerenciamento de armazenamento é automático. Essa abordagem de **prevenção de erros** em vez de detecção de erros é mais eficiente no aprimoramento da confiabilidade de programa.

Voltando um pouco na história, a Revisão de Código é um dos métodos mais antigos e seguros para detecção de defeitos. Em geral, trata-se da leitura atenciosa do código e recomendações de como melhorá-lo. Um programador geralmente não revisa seu próprio código, visto que é mais fácil encontrar erros nos códigos alheios do que em seu próprio código – entretanto, esse é um processo caro.

Já imaginaram alocar diversos programadores regularmente para revisar códigos, e eventualmente re-revisar após receber recomendações de melhoria? Pois é, pode se tornar um processo inviável! Logo, pode-se fazer uso de ferramentas automáticas de análise de código[3], que farão recomendações ao programador. Apesar de não substituir um programador e sua experiência, é mais barato e viável.

As questões que podem ser resolvidas por ferramentas de análise de estática de código-fonte podem ser divididas em três categorias: detecção de erros em programas; recomendações de formatação de código; e cálculo de métricas. Existem dezenas de ferramentas de análise estática de código-fonte com suporte a diversas linguagens de programação (C, C++, C#, Java, Ada, Fortran, Perl, etc).

Deve ser ficar cristalino que essa técnica possui seus pontos fortes e pontos fracos. Nenhum método de teste de software é ideal – diferentes métodos produzirão diferentes resultados para diferentes classes de software. Apenas com a combinação de vários métodos, será possível alcançar o mais alto nível de qualidade de um software. Captaram a mensagem?

A grande vantagem da análise estática de código-fonte é que ela permite que você reduza enormemente o preço de eliminação de defeitos em um software, visto que quanto mais cedo se detecta um erro, menor será o preço para consertá-lo. De acordo com Steve McConnell, consertar um erro na fase de testes custa dez vezes mais do que na fase de implementação.

---

[2] Embora analisadores estáticos para Java estejam disponíveis, eles não são amplamente usados.

[3] O termo *Análise Estática*, em geral, refere-se a ferramentas automáticas. O termo Revisão de Código, em geral, refere-se à análise feita por programadores.

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

Static code analysis

Custo médio de conserto de defeitos dependendo do momento em que foram feitos e detectados.

Por fim, é importante citar – com relação às práticas ágeis na análise estática de código-fonte – que a revisão por pares apresenta similaridades ao processo proposto pela programação em pares (pair programming) das práticas ágeis. Quanto à automatização da análise estática, não há menção nem mesmo recomendação entre as práticas ágeis para essa técnica automatizada.

Dessa forma, pode-se dizer que os processos de análise estática nas práticas ágeis são contemplados parcialmente. Consequentemente, sob a ótica de que há necessidade de se realizarem técnicas de análise estática de código juntamente com análise dinâmica (testes), as práticas ágeis não podem ser consideradas completas e eficientes. Bacana? Isso já foi tema de discursiva!

Questões Comentadas – Análise Estática de Código-Fonte - Multibancas

1. **(CESGRANRIO – 2010 – PETROBRÁS – Analista de Sistemas – B) Ferramentas de análise estática do código permitem obter métricas de qualidade de um produto de software, tais como o grau de dependência entre seus componentes.**

Comentários:

Perfeito! O cálculo de métricas de qualidade pode ser obtido por meio de ferramentas de análise estática de código. Entre as métricas de qualidade de software, podemos citar o acoplamento, que mede o grau de dependência entre componentes de um software. **Gabarito: C**

2. **(CESPE – 2010 – INMETRO – Analista de Sistemas – E) Teste é uma abordagem de controle de qualidade de um software, e o teste pode ser desenvolvido por meio de técnicas de análise estática de código, embora sejam mais comuns as técnicas dinâmicas, incluindo simulação.**

Comentários:

Essa questão é esquisita! Por que? Bem, eu nunca encontrei uma bibliografia que falasse qual das duas técnicas é a mais comum. Por que eu acho que é a análise estática? Porque uma leitura do código procurando um erro, é uma análise estática; a compilação é uma análise estática; a inspeção é uma análise estática. No entanto, vocês poderiam me dizer dezenas de tipos de testes dinâmicas também! Portanto, na minha opinião, é impossível dizer – logo caberia recurso! **Gabarito: E**

3. **(CESPE – 2010 – TCU – Analista de Sistemas) No projeto a ser desenvolvido, será apropriado adotar a revisão estática de código, pois tal abordagem produz resultados precisos, objetivos e completos acerca do grau de vulnerabilidade do código analisado, especialmente quando se utilizam ferramentas de software de análise estática que simulem o comportamento da aplicação a partir de seu código-fonte.**

Comentários:

Completos? Não! Já vimos que todos os testes têm limitações. Um alto grau de qualidade é alcançado quando temos a combinação de vários métodos. **Gabarito: E**

4. (FCC – 2007 – TRF/4 – Analista de Sistemas) Os requisitos específicos dos usuários, que sugerem os casos de teste que os colocarão à prova, são isolados por Ferramentas CASE de testes para:

a) Gerenciamento de testes.
b) Aquisição de dados.
c) Análise estática.
d) Análise dinâmica.
e) Avaliações transfuncionais.

Comentários:

Pessoal! Em geral, testes estáticos são realizados nas primeiras fases do ciclo de vida de desenvolvimento de software e testes dinâmicos são realizados na fase final. Na implementação, ambos são utilizados. Portanto, os requisitos do usuário são feitos por ferramentas CASE de testes para Análise Estática. Gabarito: C

5. (CESPE – 2016 – TCE/SC – Analista de Sistemas) As técnicas estáticas de verificação centram-se na análise manual ou automatizada do código-fonte do programa, enquanto a validação dinâmica tem por objetivo identificar defeitos no programa e demonstrar se ele atende a seus requisitos.

Comentários:

Técnicas estáticas de verificação, de fato, centram-se na análise do código-fonte (i.e., sem executar o programa); já técnicas de validação dinâmica executam o software para encontrar defeitos e demonstrar se ele atende aos seus requisitos. Gabarito: C

LISTA DE QUESTÕES - ANÁLISE ESTÁTICA DE CÓDIGO-FONTE - MULTIBANCAS

1. (CESGRANRIO – 2010 – PETROBRÁS – Analista de Sistemas – B) Ferramentas de análise estática do código permitem obter métricas de qualidade de um produto de software, tais como o grau de dependência entre seus componentes.

2. (CESPE – 2010 – INMETRO – Analista de Sistemas – E) Teste é uma abordagem de controle de qualidade de um software, e o teste pode ser desenvolvido por meio de técnicas de análise estática de código, embora sejam mais comuns as técnicas dinâmicas, incluindo simulação.

3. (CESPE – 2010 – TCU – Analista de Sistemas) No projeto a ser desenvolvido, será apropriado adotar a revisão estática de código, pois tal abordagem produz resultados precisos, objetivos e completos acerca do grau de vulnerabilidade do código analisado, especialmente quando se utilizam ferramentas de software de análise estática que simulem o comportamento da aplicação a partir de seu código-fonte.

4. (FCC – 2007 – TRF/4 – Analista de Sistemas) Os requisitos específicos dos usuários, que sugerem os casos de teste que os colocarão à prova, são isolados por Ferramentas CASE de testes para:

   a) Gerenciamento de testes.
   b) Aquisição de dados.
   c) Análise estática.
   d) Análise dinâmica.
   e) Avaliações transfuncionais.

5. (CESPE – 2016 – TCE/SC – Analista de Sistemas) As técnicas estáticas de verificação centram-se na análise manual ou automatizada do código-fonte do programa, enquanto a validação dinâmica tem por objetivo identificar defeitos no programa e demonstrar se ele atende a seus requisitos.

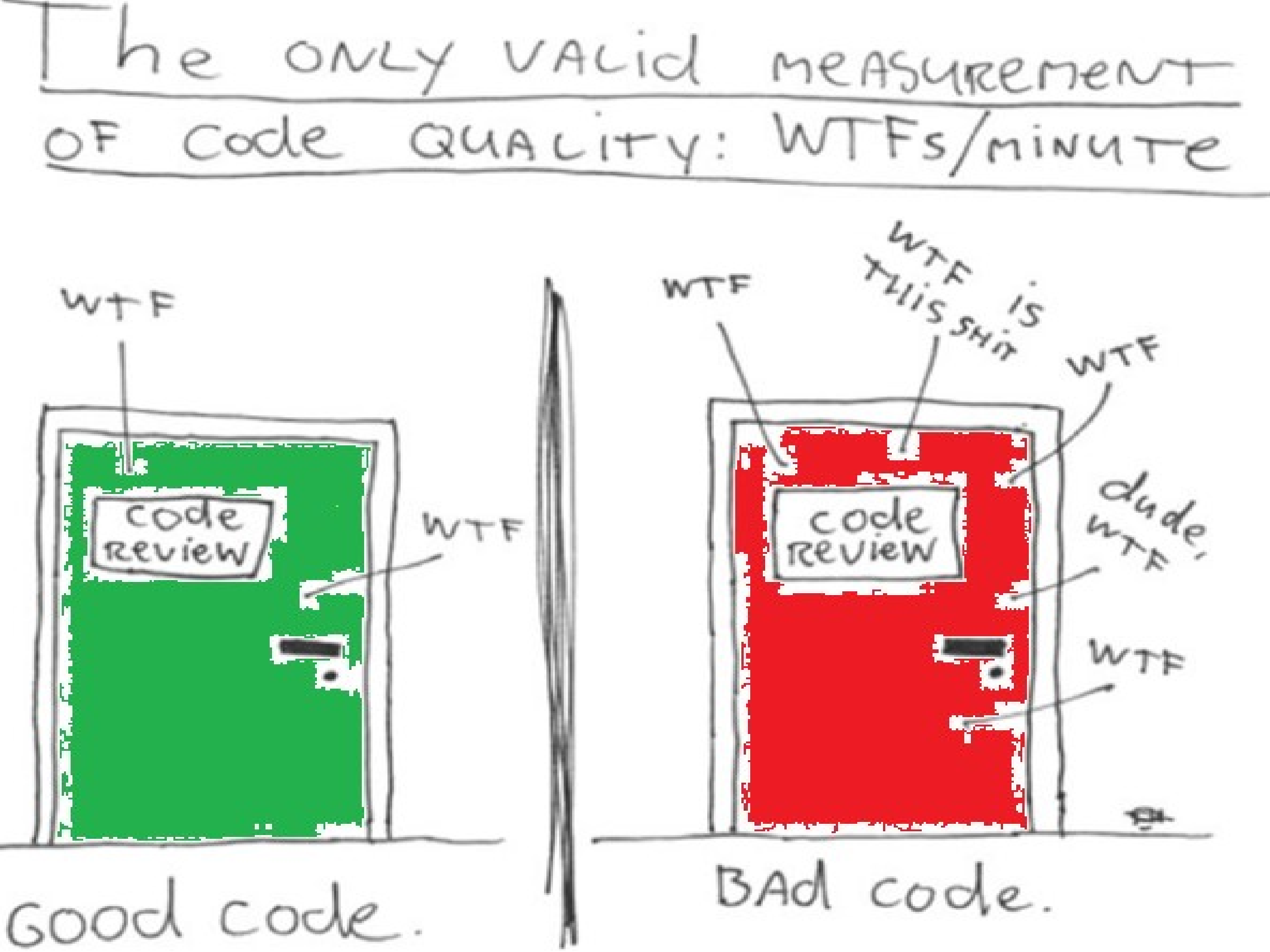
GABARITO

1. C
2. E
3. E
4. C
5. C

CLEAN CODE (CÓDIGO LIMPO)

Quando falamos em Clean Code, necessariamente passamos por Robert Martin! Esse cara é um dos autores do Manifesto Ágil e ele foi o grande idealizador do Clean Code. Em seu livro: "Clean code: a Handbook of Agile Software Craftsmanship", ele devora esse assunto em pouco mais de 400 páginas! Nós não vamos entrar em tantos detalhes, porque ainda é um assunto incipiente e que só o CESPE cobrou...

De acordo com Robert Martin, o CleanCode trata do uso disciplinado de uma miríade de pequenas técnicas aplicadas por meio de uma sensibilidade meticulosamente adquirida sobre 'limpeza' – a sensibilidade ao código é o segredo. Alguns de nós já nascemos com ela; outros precisam se esforçar para adquiri-la. Um código limpo sempre parece que foi escrito por alguém que se importa.



Quem é programador aí? Qual porta representa seu código ou da empresa/órgão que você trabalha? Martin Robert diz que codificar é um artesanato, e todo artesanato é composto de conhecimento e mão na massa! Você deve adquirir o conhecimento sobre princípios, padrões, práticas e heurísticas de um artesão de software, mas você precisa também pôr a mão na massa e praticar muito.

Uma comparação bacana é mais ou menos assim: é possível ensinar a física por trás de uma bicicleta. É possível ensinar a matemática, a gravidade, atrito, momento angular, centro de massa, entre outros; tudo isso em menos de uma página de equações e, ainda assim, você cairia na primeira vez que andasse de bike! Codificação não é diferente! Nós vamos ver os princípios, mas é necessário praticar.

Pessoal, os programadores não são maus profissionais – há que se entender o contexto que envolve o desenvolvimento de software na maioria dos lugares:

- Cronogramas apertados: o programador tem um monte de coisas para fazer em pouco tempo. Se ele conseguir de alguma maneira fazer o código funcionar, evita-se de mexer nele, porque os programadores têm mais uma porrada de coisas para fazer no pouco tempo que lhes resta. Qual a consequência disso? Código ineficiente, mal implementado e mal documentado.

- Necessidades urgentes: o usuário chega com uma necessidade absurdamente urgente que precisa ser feita em meia hora. O que o programador faz? Tenta resolver rapidamente, enfiando linhas e mais linhas de código no meio do programa com o único intuito de fazer o código funcionar da maneira que o usuário deseja; novamente, sem se preocupar com outras coisas.

- Pressão do gerente: vem lá seu gerente dizendo que está tudo atrasado, que o deadline está chegando, que o usuário está irritado, entre outros. O programador tenta fazer tudo o mais rápido, feio e ineficiente possível. Resolve o problema? Muitas vezes, não; outras vezes, sim, mas com uma programação orientada a gambiarra completa.

- Apenas produtividade: muitos programadores querem mostrar que são excelentes codificadores, talvez para impressionar seus chefes ou colegas de trabalho: “Vejam como eu implemento isso rápido”. No entanto, muitas vezes eles se esquecem da qualidade do código; ora, implementar rapidamente um código ruim não é vantagem; implementar rapidamente código bom, é vantagem.

Tudo isso serve para qualquer profissão, mas por alguma razão, algumas pessoas acham que a área de tecnologia da informação é diferente das demais. Ora, vocês já imaginaram um médico fazendo uma cirurgia, quando recebe uma mensagem do chefe do hospital dizendo: “Doutor, você está enrolando demais nas cirurgias, acelera um pouco isso aí, porque você já está atrasado os próximos pacientes”.

"Outra coisa, essa mulher tinha pedido para colocar silicone nos seios, mas acabou de desistir e, na verdade, ela vai querer uma plástica no nariz; não demore mais do que vinte minutos, ok?!". Existem muitos programadores ruins por aí, mas existem muitos que são bons programadores, porém fazem códigos ruins por conta de todos esses motivos supracitados.

É preciso que todos tenham noção de que código ruim tem um preço alto. Por que? Porque eventualmente ele irá falhar e, quando isso acontecer, alguém vai ter que ler o código e entendê-lo para dar manutenção. Quanto pior escrito, mais difícil de entender, mais demora para consertar. Fora a frustração e a irritação de mexer em um código mal escrito. Quem já passou por isso sabe...

Acho que já ficou claro que um código de baixa qualidade, pode matar um projeto e até destruir uma empresa. Professor, o que seria um código limpo? É um código eficiente, simples, direto, elegante, não-redundante, pouco dependente, fácil de manutenir, ler e entender, coberto por testes, e que seguem padrões definidos. Galera, um código limpo que pode ser lido quase como uma conversa. Vejam só:

Martin Fowler afirma que qualquer um é capaz de escrever um código que um computador entenda. No entanto, bons programadores escrevem código que humanos entendem. Professor, por que o nome 'código limpo'? Cara, esse nome veio de uma das regras do escotismo que diz: "Sempre deixe a área de acampamento mais limpa do que estava" – isso serve para os programadores e seus códigos.

Vamos ver agora alguns princípios do CleanCode:

Nomes significativos: escolher bons nomes (para variáveis, métodos, funções, etc); nomes devem ter significado (i.e., revelar a intenção); nomes devem estar de acordo com o contexto empregado; nomes devem ser pronunciáveis (evitar siglas); nomes não devem ser abreviados; nomes devem

ser descritíveis; por fim, devem ser não-ambíguos, fáceis de ler e de compreender no contexto em que se insere.

```
int d; //tempo gasto em dias
```

Em geral, se você precisa de um comentário para explicar o que significa uma variável, algo está errado – o nome não revela a intenção. Agora vejam a diferença na forma de escrever abaixo. Não é necessário comentário, porque ela já revela sua intenção e seu significado. Programadores, parem de preguiça de criar variáveis com nomes grandes, afinal toda IDE atualmente tem ctrl + space para completar.

```
int tempoGastoEmDias;
```

Nomes de classes não devem ser verbos, devem ser substantivos; já nomes de métodos não devem ser substantivos, devem ser verbos. Quanto às funções: devem ser pequenas e devem fazer apenas uma coisa (responsabilidade única); não deve ter nível de endentação maior que dois; preferencialmente, devem ter poucos parâmetros; recomenda-se não repetir código (evitar redundância).

Quanto aos comentários: somente são úteis se forem colocados nos lugares certos; como comentários, em geral, não são atualizados, eles podem distorcer a realidade, logo não são confiáveis; além disso, comentários por si só não transformam um código sujo em um código limpo; deve-se evitar fazer controle de versão por meio de comentários (//revisado por Diego).

Quanto à formatação: classes menores são mais fáceis de entender; conceitos relacionados devem ficar próximos; endentar bem o código é fundamental; limite o tamanho máximo da linha (Ex: 120 caracteres); recomenda-se combinar um estilo de formatação entre toda equipe de desenvolvimento; use espaços entre operadores, parâmetros e vírgulas; organize as declarações de variáveis e métodos;

Quanto às classes: também devem ser pequenas e ter responsabilidade única; organize as declarações de variáveis e métodos. Galera, é isso... existe muito mais no CleanCode – como eu disse, há um livro inteiro sobre isso. No entanto, é um tema que caiu apenas uma vez até hoje, logo vamos nos limitar ao que foi dado. Bacana? Vamos ver uns exercícios.

VIDADEPROGRAMADOR
.COM.BR
real historia;
string sender;
sender = "Elias dy Paula";
BY NC SA
#785

POR QUE TEM ESSES COMENTÁRIOS NO CÓDIGO FONTE?

DEPOIS DE CRIAR UMA FUNÇÃO OU ALGO MAIS COMPLEXO, VOCÊ CRIA UM COMENTÁRIO PARA QUE QUEM FOR DAR MANUTENÇÃO ENTENDA MAIS FÁCIL ESSE TRECHO DE CÓDIGO...
NÃO É OBRIGATÓRIO, MAS É UMA BOA PRÁTICA ENTRE OS DESENVOLVEDORES

DIAS DEPOIS, VENDO UM CÓDIGO DO ALONSO...
IF ($num_pagina<10)
{
// Não faz nada
}
ELSE
{
echo "Exibir paginacao";
}
PLOFT!

QUESTÕES COMENTADAS - CLEANCODE - CEBRASPE

1. (CESPE – 2013 – STF – Analista de Sistemas) O desenvolvimento de sistemas mediante a utilização de CLEAN CODE baseia-se em um ciclo curto de repetições, em que o responsável pela codificação descreve testes automatizados que definem uma funcionalidade elicitada. Após se definir o teste, desenvolve-se o código que será validado pela equipe de teste e, posteriormente, refatorado.

Comentários:

De acordo com Robert Martin, o CleanCode trata do uso disciplinado de uma miríade de pequenas técnicas aplicadas por meio de uma sensibilidade meticulosamente adquirida sobre 'limpeza' – a sensibilidade ao código é o segredo. Alguns de nós já nascemos com ela; outros precisam se esforçar para adquiri-la. Um código limpo sempre parece que foi escrito por alguém que se importa. Conforme vimos em aula, isso não tem nada a ver com CleanCode! Na verdade, isso está mais para Test-Driven Development (TDD).
Gabarito: E

2. (CESPE – 2013 – STF – Analista de Sistemas) Os nomes de classes devem conter verbos, ao passo que os métodos devem ser indicados por substantivos.

Comentários:

Nomes de classes não devem ser verbos, devem ser substantivos; já nomes de métodos não devem ser substantivos, devem ser verbos. Quanto às funções: devem ser pequenas e devem fazer apenas uma coisa (responsabilidade única); não deve ter nível de endentação maior que dois; preferencialmente, devem ter poucos parâmetros; recomenda-se não repetir código (evitar redundância). Conforme vimos em aula, a questão inverteu os conceitos. Gabarito: E

3. (CESPE – 2013 – STF – Analista de Sistemas) No contexto de Clean Code, as funções devem ter tamanho reduzido.

Comentários:

Nomes de classes não devem ser verbos, devem ser substantivos; já nomes de métodos não devem ser substantivos, devem ser verbos. Quanto às funções: devem ser pequenas e devem fazer apenas uma coisa (responsabilidade única); não deve ter nível de endentação maior que dois; preferencialmente, devem ter poucos parâmetros; recomenda-se não repetir código (evitar redundância). Conforme vimos em aula, realmente as funções devem ter tamanho reduzido.
Gabarito: C

4. (CESPE – 2016 – TCE/SC – Analista de Sistemas) De acordo com as diretivas do Clean Code, o número de argumentos de uma função não deve ser igual ou superior a três, devido a sua influência no entendimento da função.

Comentários:

Robert C. Martin diz: "The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification — and then shouldn't be used anyway". O grande Robert C. Martin diz que três argumentos devem ser evitados e, se houver mais de três (i.e., quatro, cinco, seis, etc), deve haver uma justificativa especial. Logo, recomenda-se que uma função não deve ser igual ou superior a três. Gabarito: C

5. (CESPE – 2015 – TRE/MT – Analista de Sistemas) Assinale a opção que apresenta instruções de elaboração corretas de acordo com a técnica Clean Code.

a) Os nomes utilizados devem ser pronunciáveis e devem ter sentido conhecido.

b) Os nomes de classes devem ser verbos no infinitivo e os de métodos devem ser substantivos.

c) Os nomes de funções e de métodos devem ser longos e descritivos.

d) Os parâmetros devem ser aglutinados em funções, e cada função deve ter, no máximo, três parâmetros.

e) O comando return deve ser evitado, ao passo que continue e break devem ser priorizados, assim como o goto.

Comentários:

Nomes significativos: escolher bons nomes (para variáveis, métodos, funções, etc); nomes devem ter significado (i.e., revelar a intenção); nomes devem estar de acordo com o contexto empregado; nomes devem ser pronunciáveis (evitar siglas); nomes não devem ser abreviados; nomes devem ser descritivos; por fim, devem ser não-ambíguos, fáceis de ler e de compreender no contexto em que se insere.

(a) Conforme vimos em aula, nomes devem ser pronunciáveis e que tenham um propósito.

Nomes de classes não devem ser verbos, devem ser substantivos; já nomes de métodos não devem ser substantivos, devem ser verbos. Quanto às funções: devem ser pequenas e devem fazer apenas uma coisa (responsabilidade única); não deve ter nível de endentação maior que dois;

preferencialmente, devem ter poucos parâmetros; recomenda-se não repetir código (evitar redundância).

(b) Conforme vimos em aula, nomes de classes não devem ser verbos – mas substantivos.

Nomes significativos: escolher bons nomes (para variáveis, métodos, funções, etc); nomes devem ter significado (i.e., revelar a intenção); nomes devem estar de acordo com o contexto empregado; nomes devem ser pronunciáveis (evitar siglas); nomes não devem ser abreviados; nomes devem ser descritivos; por fim, devem ser não-ambíguos, fáceis de ler e de compreender no contexto em que se insere.

(c) Conforme vimos em aula, nomes devem ser descritivos, mas isso não significa necessariamente longo.

Robert C. Martin diz: "The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification — and then shouldn't be used anyway".

(d) Conforme vimos em aula, o ideal é que uma função tenha zero argumentos e cada função deve ter no máximo dois (e, não, três) funções.

(e) Essa foi de graça - não tem nada a ver com CleanCode. Galera, goto deve ser priorizado? Nunca! Jamais! Comandos de salto de instruções devem ser evitados. Para quem não sabe o que é goto, algumas linguagens possuem instruções que permitem que se saia de uma linha de código para qualquer outra linha de código – isso é péssimo para o entendimento, para manutenção, entre outros.

Gabarito: A

6. (CESPE – 2016 – TRE/PI – Analista de Sistemas) Acerca do Clean Code, assinale a opção correta.

a) A segurança do código é vital, por isso os programadores devem deixar o código o mais obscuro possível.

b) Se um valor deve ser utilizado em múltiplos locais do código, é imperativo atribuir esse valor a uma variável ou a uma constante com nome amigável.

c) As classes devem possuir nome amigável oriundo de verbos, escolhidos no infinitivo, e não no gerúndio.

d) Para customizar o código, deve-se utilizar o mesmo termo para duas diferentes ideias.

e) Os nomes das variáveis devem ser simplificados, de forma a não criar códigos gordos (fat codes) — por exemplo, o uso de x para o nome de uma variável é mais apropriado que MediadosAlunosAprovados.

Comentários:

(a) A técnica se chama CleanCode, logo a ideia é deixar o código o mais claro possível; (b) Essa questão é polêmica, porque – de fato – quando existe um valor utilizado em vários locais, recomenda-se atribuir esse valor a uma variável ou constante. No entanto, dizer que é imperativo é pesado – eu acho que essa questão deveria ser anulada, mas ela foi considerada correta; (c) Nome de classe deve ser substantivo e, não, verbo; (d) Usar o mesmo termo para duas diferentes ideias pode causar ambiguidade, dificultando a manutenção; (e) Pelo contrário, nomes de variáveis devem ser descritivos – você pode usar ‘x’, desde que tenha um escopo menor e local.

Gabarito: C

LISTA DE QUESTÕES - CLEANCODE - CEBRASPE

1. (CESPE – 2013 – STF – Analista de Sistemas) O desenvolvimento de sistemas mediante a utilização de CLEAN CODE baseia-se em um ciclo curto de repetições, em que o responsável pela codificação descreve testes automatizados que definem uma funcionalidade elicitada. Após se definir o teste, desenvolve-se o código que será validado pela equipe de teste e, posteriormente, refatorado.

2. (CESPE – 2013 – STF – Analista de Sistemas) Os nomes de classes devem conter verbos, ao passo que os métodos devem ser indicados por substantivos.

3. (CESPE – 2013 – STF – Analista de Sistemas) No contexto de Clean Code, as funções devem ter tamanho reduzido.

4. (CESPE – 2016 – TCE/SC – Analista de Sistemas) De acordo com as diretivas do Clean Code, o número de argumentos de uma função não deve ser igual ou superior a três, devido a sua influência no entendimento da função.

5. (CESPE – 2015 – TRE/MT – Analista de Sistemas) Assinale a opção que apresenta instruções de elaboração corretas de acordo com a técnica Clean Code.

   a) Os nomes utilizados devem ser pronunciáveis e devem ter sentido conhecido.

   b) Os nomes de classes devem ser verbos no infinitivo e os de métodos devem ser substantivos.

   c) Os nomes de funções e de métodos devem ser longos e descritivos.

   d) Os parâmetros devem ser aglutinados em funções, e cada função deve ter, no máximo, três parâmetros.

   e) O comando return deve ser evitado, ao passo que continue e break devem ser priorizados, assim como o goto.

6. (CESPE – 2016 – TRE/PI – Analista de Sistemas) Acerca do Clean Code, assinale a opção correta.

   a) A segurança do código é vital, por isso os programadores devem deixar o código o mais obscuro possível.

   b) Se um valor deve ser utilizado em múltiplos locais do código, é imperativo atribuir esse valor a uma variável ou a uma constante com nome amigável.

c) As classes devem possuir nome amigável oriundo de verbos, escolhidos no infinitivo, e não no gerúndio.

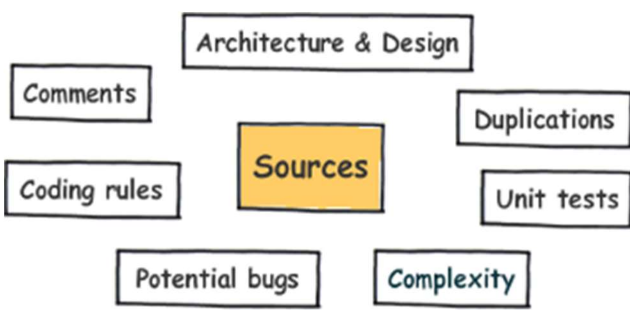
d) Para customizar o código, deve-se utilizar o mesmo termo para duas diferentes ideias.

e) Os nomes das variáveis devem ser simplificados, de forma a não criar códigos gordos (fat codes) — por exemplo, o uso de x para o nome de uma variável é mais apropriado que MediadosAlunosAprovados.

Gabarito

1. E
2. E
3. C
4. C
5. A
6. C

SONARQUBE



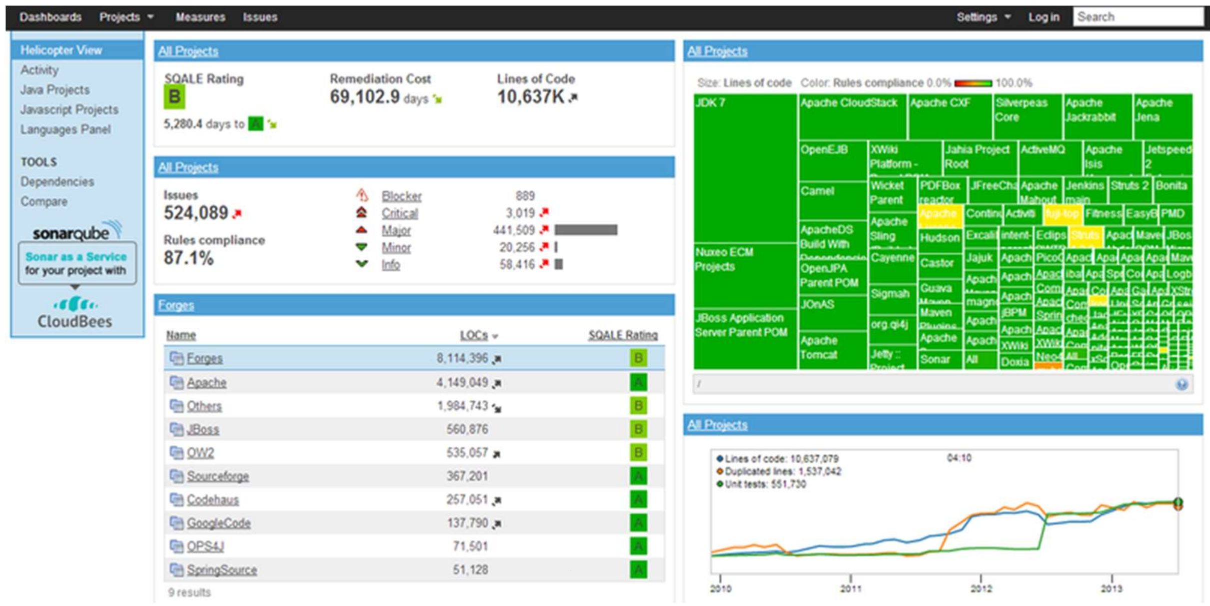
O **SonarQube** é uma ferramenta **web open-source** utilizada para gerenciar a qualidade do código – ele cobre sete grandes categorias: arquitetura e design; comentários; duplicações de código; padrões de codificação; testes (cobertura de código); complexidade ciclomática; e bugs em potencial.

Ele é escrito em Java e Ruby, é bastante organizado e customizado – além de ter uma extensa documentação. Galera, existe um conceito chamado **Entropia**! Quem aí sabe o que é isso? É o conceito de termodinâmica que mede a desordem das partículas de um sistema físico. No desenvolvimento de software, ele significa que, quanto mais se adicionam linhas de código, mais complexo o software fica.

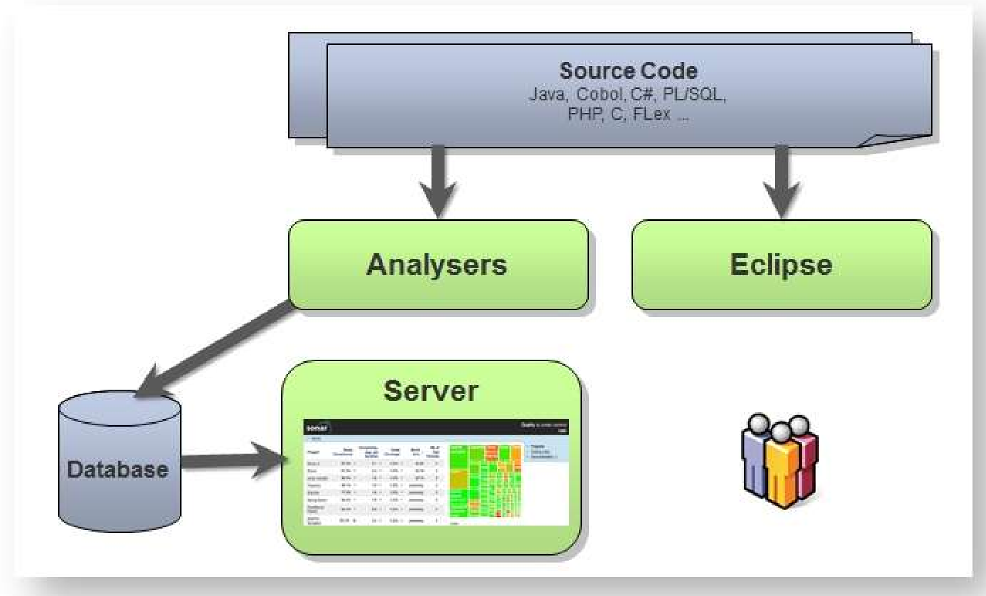
Aliás, não só complexo como também bagunçado. Imaginem um sistema estruturante de uma organização/empresa que está se deteriorando por conta dos altos custos de manutenção ou por conta da imensa quantidade de bugs ou por conta da demora para solucionar esses bugs. A entropia só tende a aumentar e, muitas vezes, exponencialmente.

Pessoal, em meu tempo de serviço público, uma coisa que eu percebi claramente é que – para se ter uma visão gerencial de um processo ou trabalho – é necessário ter métricas/números. Como você melhora a qualidade do atendimento de primeiro nível de um Service Desk se você não sabe quanto tempo demora o atendimento, a solução do problema, o transbordo para outros níveis, entre outros?

O SonarQube é bacana por conta disso! Ele consegue oferecer uma porrada de métricas sobre o código-fonte de um sistema. Dessa forma, o programador pode ter uma visão estruturada sobre a aplicação que está sendo desenvolvida e pode controlar sua qualidade. Gente, uma pancada de empresas competentíssimas utiliza o SonarQube, como IBM, EBay, HP, Nike, Cisco, PayPal, etc.



Em suma, podemos dizer que ele se propõe a ser a central de qualidade do seu código-fonte, possibilitando o controle sobre um grande número de métricas de software, e ainda apontando uma série de possíveis bugs. Tudo gerado através de uma análise completa do código-fonte, e apresentado por meio de uma interface web, em forma de dashboards e gráficos, como é mostrado acima.



A Arquitetura do SonarQube é composta de três componentes:

- **Banco de Dados:**

Deve haver apenas um! Ele mantém os resultados da análise, os projetos e configuração global, mas também mantém uma análise histórica.

- **Servidor Web:**

Deve haver apenas um! Ele é responsável por fornecer aos usuários diversos painéis de qualidade de código-fonte e por configurar a instância do SonarQube.

- **Analisadores:**

Um conjunto de analisadores de código-fonte que são agrupados e acionados por demanda – eles utilizam a configuração armazenada no banco de dados.

Por fim, ele suporta diversas linguagens de programação; pode ser utilizado para análise de desenvolvimento mobile; fornece um histórico de métricas e gráficos de evolução (chamado Time Machine) e diferentes views; integra bem com Maven, Ant, Grandle, Jenkings, Hudson, etc; integra bem com Eclipse, Mantis, LDAP; é bastante extensível por meio de plug-ins; entre outras funcionalidades.

QUESTÕES COMENTADAS - SONARQUBE - CEBRASPE

1. (CESPE – 2015 – TCU – Analista de Sistemas) Uma característica positiva da ferramenta SonarQube, quando utilizada para realizar a análise estática de código-fonte, é a conveniência de instalação e utilização em dispositivos móveis.

Comentários:

Por fim, ele suporta diversas linguagens de programação; pode ser utilizado para análise de desenvolvimento mobile; fornece um histórico de métricas e gráficos de evolução (chamado Time Machine) e diferentes views; integra bem com Maven, Ant, Grandle, Jenkings, Hudson, etc; integra bem com Eclipse, Mantis, LDAP; é bastante extensível por meio de plug-ins; entre outras funcionalidades.

Galera, ele pode ser utilizado para analisar código-fonte de aplicações móveis. No entanto, não é possível executá-lo em dispositivos móveis, porque é necessário ter uma JRE/OpenJDK instalada – e, pelo menos atualmente, não existe! **Gabarito: E**

2. (CESPE – 2016 – TCE/SC – Analista de Sistemas) Um dos modos de análise de código-fonte constante no SonarQube é o publish, que analisa completamente o código e o envia para o servidor que irá processá-lo e salvar os resultados no banco de dados.

Comentários:

Existem dois modos – Preview e Publish (padrão). Esse último analisa tudo que for possível e manda o resultado para um servidor processar e salvar o resultado no banco de dados (Publish mode performs a full analysis on the entire code base and sends it to the server, which will process it and save the results to the database). **Gabarito: C**

LISTA DE QUESTÕES - SONARQUBE - CEBRASPE

1. (CESPE – 2015 – TCU – Analista de Sistemas) Uma característica positiva da ferramenta SonarQube, quando utilizada para realizar a análise estática de código-fonte, é a conveniência de instalação e utilização em dispositivos móveis.

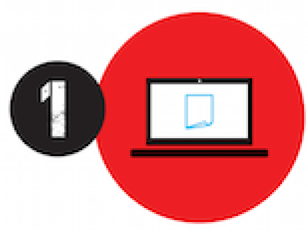
2. (CESPE – 2016 – TCE/SC – Analista de Sistemas) Um dos modos de análise de código-fonte constante no SonarQube é o publish, que analisa completamente o código e o envia para o servidor que irá processá-lo e salvar os resultados no banco de dados.

GABARITO

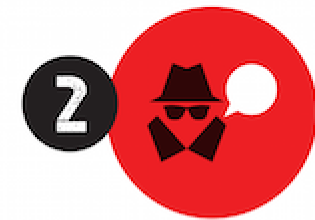
1. E
2. C

ESSA LEI TODO MUNDO CONHECE:
PIRATARIA É CRIME.

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

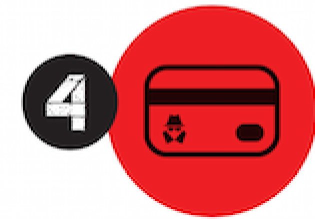


Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.



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).

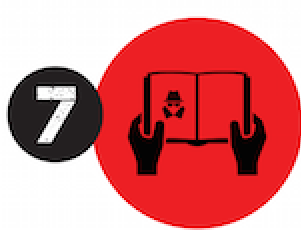
Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).



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

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).

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



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

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.