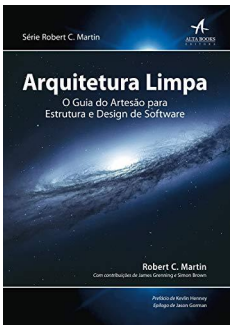

AS NOTAS DO KINDLE PARA:

Arquitetura Limpa: O guia do artesão para estrutura e design de software (Robert C. Martin)

de Robert C. Martin

Visualização instantânea gratuita do Kindle: https://a.co/gplWlvD

134 destaques

Destaque (Amarelo) | Posição 305

> O apelo óbvio da arquitetura é a estrutura, que domina os paradigmas e discussões sobre o desenvolvimento de software — componentes, classes, funções, módulos, camadas e serviços, micro ou macro.

Destaque (Amarelo) | Posição 328

> A velocidade do processador e a largura de banda da rede podem fornecer um veredito duro sobre a performance de um sistema.

Destaque (Amarelo) | Posição 329

> A memória e o armazenamento podem limitar as ambições de qualquer base de código.

Destaque (Amarelo) | Posição 340

> uma arquitetura não deve apenas atender às demandas dos usuários, desenvolvedores e proprietários em um determinado momento, mas também corresponder a essas expectativas ao longo do tempo.

Destaque (Amarelo) | Posição 349

> A arquitetura é o conjunto de decisões que você queria ter tomado logo no início de um projeto, mas, como todo mundo, não teve a imaginação necessária. — Ralph Johnson

Destaque (Amarelo) | Posição 386

> As regras da arquitetura são sempre as mesmas! Isso é surpreendente porque os sistemas que criei eram radicalmente diferentes entre si.

Destaque (Amarelo) | Posição 388

> as regras da arquitetura de software são independentes de todas as outras variáveis.

Destaque (Amarelo) | Posição 425

> As regras não mudaram. Apesar das novas linguagens, frameworks e paradigmas, as regras continuam as mesmas de quando Alan Turing escreveu seu primeiro código de máquina em 1946.

Destaque (Amarelo) | Posição 472

> fazer algo funcionar — uma vez — não é tão difícil. Fazer direito é outra questão.

Destaque (Amarelo) | Posição 474

> Criar software de maneira correta é difícil. Requer conhecimentos e habilidades que a maioria dos jovens programadores ainda não adquiriu.

Destaque (Amarelo) | Posição 475

> Requer um grau de raciocínio e insight que a maioria dos programadores não se empenha em desenvolver.

Destaque (Amarelo) | Posição 475

> Requer um nível de disciplina e dedicação que a maioria dos programadores nunca sonhou que precisaria.

Destaque (Amarelo) | Posição 487

> Já trabalhou em sistemas tão interconectados e intrincadamente acoplados que qualquer mudança, por mais trivial que seja, levava semanas e envolvia grandes riscos?

Destaque (Amarelo) | Posição 488

> Já experimentou a impedância de um código ruim e um péssimo design?

Destaque (Amarelo) | Posição 497

> Tem havido muita confusão entre os termos design e arquitetura ao longo dos anos.

Destaque (Amarelo) | Posição 498

> O que é design? O que é arquitetura?

Destaque (Amarelo) | Posição 500

> não há diferença entre os dois termos.

Destaque (Amarelo) | Posição 516

> O objetivo da arquitetura de software é minimizar os recursos humanos necessários para construir e manter um determinado sistema.

Destaque (Amarelo) | Posição 518

> A medida da qualidade do design corresponde à medida do esforço necessário para satisfazer as demandas do cliente. Se o esforço for baixo e se mantiver assim ao longo da vida do sistema, o design é bom. Se o esforço aumentar a cada novo release ou nova versão, o design é ruim. Simples assim.

Destaque (Amarelo) | Posição 544

> Quando sistemas são desenvolvidos às pressas, quando o número total de programadores se torna o único gerador de resultados, e quando houver pouca ou nenhuma preocupação com a limpeza do código e a estrutura do design, você pode ter certeza que irá seguir

---

Destaque (Amarelo) | Posição 545

> esta curva até o seu terrível final.

Destaque (Amarelo) | Posição 568

> é necessário tomar uma ação imediata para evitar um desastre.

Destaque (Amarelo) | Posição 570

> Mas que ação pode ser tomada? O que deu errado?

Destaque (Amarelo) | Posição 581

> Esses desenvolvedores acreditam em uma mentira bem conhecida: "Podemos limpar tudo depois, mas, primeiro, temos que colocá-lo no mercado!"

Destaque (Amarelo) | Posição 583

> A preocupação de lançar o produto no mercado o quanto antes significa que você agora tem uma horda de concorrentes nos seus calcanhares e precisa ficar à frente deles, correndo o mais rápido que puder.

Destaque (Amarelo) | Posição 586

> Nesse ritmo, a bagunça vai se acumulando e a produtividade continua a se aproximar assintoticamente do zero.

Destaque (Amarelo) | Posição 589

> Mas a crescente bagunça no código, que mina a sua produtividade, nunca dorme e nunca cede. Se tiver espaço, ela reduzirá a produtividade a zero em uma questão de meses.

Destaque (Amarelo) | Posição 590

> A maior mentira em que os desenvolvedores acreditam é a noção de que escrever um código bagunçado permite fazer tudo mais rápido a curto prazo e só diminui o ritmo a longo prazo.

Destaque (Amarelo) | Posição 594

> fazer bagunça é sempre mais lento do que manter tudo limpo,

---

Destaque (Amarelo) | Posição 598

> desenvolvimento guiado por testes (TDD — do inglês test-driven development)

Destaque (Amarelo) | Posição 608

> A única maneira de seguir rápido é seguir bem.

Destaque (Amarelo) | Posição 616

> a melhor opção para a organização de desenvolvimento é reconhecer e evitar o seu próprio excesso de confiança e começar a levar a sério a qualidade da arquitetura do software.

Destaque (Amarelo) | Posição 619

> Para construir um sistema com um design e uma arquitetura que minimizem o esforço e maximizem a produtividade, você precisa saber quais atributos da arquitetura do sistema podem concretizar esses objetivos.

Destaque (Amarelo) | Posição 621

> arquiteturas e designs limpos e eficientes

Destaque (Amarelo) | Posição 626

> Todo sistema de software fornece dois valores diferentes para os stakeholders:

Destaque (Amarelo) | Posição 626

> comportamento e estrutura.

Destaque (Amarelo) | Posição 627

> Desenvolvedores de software são responsáveis por garantir que ambos esses valores permaneçam altos.

Destaque (Amarelo) | Posição 630

> O primeiro valor do software é seu comportamento.

Destaque (Amarelo) | Posição 631

> Nós fazemos isso ajudando os stakeholders a desenvolverem uma

---

Destaque (Amarelo) | Posição 632

> especificação funcional

Destaque (Amarelo) | Posição 632

> ou um

Destaque (Amarelo) | Posição 632

> documento de requisitos.

Destaque (Amarelo) | Posição 635

> programadores

Destaque (Amarelo) | Posição 635

> acreditam que seu trabalho é fazer a máquina implementar os requisitos e corrigir qualquer bug.

Destaque (Amarelo) | Posição 639

> O software foi inventado para ser "suave". Foi planejado para ser um meio de mudar facilmente o comportamento das máquinas.

Destaque (Amarelo) | Posição 642

> o software deve ser suave — ou seja, deve ser fácil de mudar.

Destaque (Amarelo) | Posição 643

> A dificuldade em fazer tal mudança deve ser proporcional apenas ao escopo da mudança, e não à forma da mudança.

Destaque (Amarelo) | Posição 645

> É essa diferença entre escopo e forma que muitas vezes impulsiona o crescimento nos custos do desenvolvimento de software.

Destaque (Amarelo) | Posição 647

> Do ponto de vista dos stakeholders, eles estão simplesmente fornecendo um fluxo de mudanças de escopo praticamente similar.

Destaque (Amarelo) | Posição 648

> Do ponto de vista dos desenvolvedores, os stakeholders estão dando a eles um fluxo de peças que eles devem encaixar em um quebra-cabeças de complexidade cada vez maior.

Destaque (Amarelo) | Posição 649

> Cada novo pedido é mais difícil de encaixar do que o anterior, porque a forma do sistema não combina com a forma do pedido.

Destaque (Amarelo) | Posição 652

> Desenvolvedores de software frequentemente sentem que são forçados a encaixar peças quadradas em buracos redondos.

Destaque (Amarelo) | Posição 653

> O problema, é claro, é a arquitetura do sistema.

Destaque (Amarelo) | Posição 654

> as arquiteturas devem ser tão agnósticas em sua forma quanto práticas.

Destaque (Amarelo) | Posição 656

> Função ou arquitetura?

Destaque (Amarelo) | Posição 657

> É mais importante que o sistema de software funcione ou é mais importante que ele seja fácil de mudar?

Destaque (Amarelo) | Posição 661

> Se você me der um programa que funcione perfeitamente, mas seja impossível de mudar, então ele não funcionará quando as exigências mudarem e eu não serei capaz de fazê-lo funcionar.

Destaque (Amarelo) | Posição 663

> Se você me der um programa que não funcione, mas seja fácil de mudar, então eu posso fazê-lo funcionar, e posso mantê-lo funcionando à medida que as exigências mudarem.

Destaque (Amarelo) | Posição 666

> existem sistemas praticamente impossíveis de mudar, porque o custo da mudança excede seu benefício. Muitos sistemas alcançam esse ponto em alguns de seus recursos ou configurações. Se você perguntar aos gerentes de negócios se eles querem ser capazes de fazer mudanças, eles dirão que é claro que querem, mas podem, então, qualificar sua resposta ao notar que a funcionalidade atual é mais importante

---

Destaque (Amarelo) | Posição 670

> do que qualquer flexibilidade posterior.

Destaque (Amarelo) | Posição 670

> se os gerentes de negócio lhe pedirem uma mudança, e seus custos estimados para ela forem inacessivelmente altos, os gerentes de negócios provavelmente ficarão furiosos que você tenha permitido que o sistema chegasse ao ponto onde a mudança é impraticável.

Destaque (Amarelo) | Posição 675

> Eu tenho dois tipos de problemas, os urgentes e os importantes. Os urgentes não são importantes e os importantes nunca são urgentes.

Destaque (Amarelo) | Posição 681

> O primeiro valor do software — comportamento — é urgente,

Destaque (Amarelo) | Posição 682

> mas nem sempre é particularmente importante.

Destaque (Amarelo) | Posição 683

> O segundo valor do software — arquitetura — é importante, mas nunca é particularmente urgente.

Destaque (Amarelo) | Posição 691

> eles falham em separar aqueles recursos que são urgentes, mas não são importantes,

---

Destaque (Amarelo) | Posição 692

> daqueles que são realmente urgentes e importantes.

Destaque (Amarelo) | Posição 693

> a ignorar a importante arquitetura do sistema em favor de recursos não importantes do sistema.

Destaque (Amarelo) | Posição 694

> os gerentes de negócios não estão equipados para avaliar a importância da arquitetura.

Destaque (Amarelo) | Posição 695

> é responsabilidade da equipe de desenvolvimento de software garantir a importância da arquitetura sobre a urgência dos recursos.

Destaque (Amarelo) | Posição 705

> Arquitetos de software são, por virtude da descrição de seu emprego, mais focados na estrutura do sistema do que em seus recursos e funções.

Destaque (Amarelo) | Posição 706

> recursos e funções

Destaque (Amarelo) | Posição 706

> facilmente desenvolvidos, facilmente modificados e facilmente ampliados.

Destaque (Amarelo) | Posição 708

> se a arquitetura vier por último, então o sistema ficará cada vez mais caro para desenvolver e, por fim, a mudança será praticamente impossível para parte ou para todo o sistema.

Destaque (Amarelo) | Posição 715

> A arquitetura de software começa com o código.

Destaque (Amarelo) | Posição 727

> Os paradigmas são maneira de programar, relativamente não relacionadas às linguagens.

Destaque (Amarelo) | Posição 728

> paradigma diz quais estruturas de programação usar e quando usá-las.

Destaque (Amarelo) | Posição 733

> Os três paradigmas abordados neste capítulo de panorama são os modelos de programação estruturada, orientada a objetos e funcional.

Destaque (Amarelo) | Posição 741

> A programação estruturada impõe disciplina sobre a transferência direta do controle.

Destaque (Amarelo) | Posição 750

> A programação orientada a objetos impõe disciplina sobre a transferência indireta do controle.

Destaque (Amarelo) | Posição 757

> uma linguagem funcional não tem nenhuma declaração de atribuição.

Destaque (Amarelo) | Posição 760

> A programação funcional impõe disciplina sobre a

Destaque (Amarelo) | Posição 760

> atribuição.

Destaque (Amarelo) | Posição 763

> cada um dos paradigmas

Destaque (Amarelo) | Posição 763

> remove capacidades do programador. Nenhum deles adiciona novas capacidades.

Destaque (Amarelo) | Posição 766

> Os três paradigmas juntos removem declarações goto, ponteiros de função

---

Destaque (Amarelo) | Posição 767

> e atribuições.

Destaque (Amarelo) | Posição 817

> todos os

Destaque (Amarelo) | Posição 817

> programas podem ser construídos a partir de apenas três estruturas: sequência, seleção e iteração.

Destaque (Amarelo) | Posição 863

> método científico.

Destaque (Amarelo) | Posição 872

> Essa é a natureza das teorias e leis científicas: elas são refutáveis e não comprováveis.

Destaque (Amarelo) | Posição 878

> A ciência não tem a função de provar que as afirmações são verdadeiras, mas sim de provar que as afirmações são falsas.

Destaque (Amarelo) | Posição 886

> um programa pode ser provado como incorreto por um teste, mas um teste não pode provar que um programa está correto.

Destaque (Amarelo) | Posição 899

> É essa habilidade de criar unidades de programação que podem ser testadas pela negação do correto, que faz da programação estruturada um modelo importante atualmente.

Destaque (Amarelo) | Posição 903

> Em todos os níveis, da menor função ao maior componente, o software é como uma ciência e, portanto, orientado pela refutabilidade.

Destaque (Amarelo) | Posição 914

> a base de uma boa arquitetura é a compreensão e aplicação dos princípios do design orientado a objetos (OO).

Destaque (Amarelo) | Posição 915

> Mas o que é OO? "Uma combinação de dados e funções", seria uma resposta para essa pergunta.

Destaque (Amarelo) | Posição 916

> No entanto, embora muito citada, essa resposta é bastante insatisfatória, pois implica que o.f() é, de alguma forma, diferente de f(o).

Destaque (Amarelo) | Posição 923

> três palavras mágicas para explicar a natureza da OO:

Destaque (Amarelo) | Posição 924

> encapsulamento, herança e polimorfismo.

Destaque (Amarelo) | Posição 925

> uma linguagem OO deveria dar suporte a esses três conceitos.

Destaque (Amarelo) | Posição 984

> para que o encapsulamento fosse parcialmente corrigido, foi preciso incluir as palavra-chave

---

Destaque (Amarelo) | Posição 984

> public, private e protected

Destaque (Amarelo) | Posição 995

> A herança é simplesmente a redeclaração de um grupo de variáveis e funções dentro de um escopo fechado.

Destaque (Amarelo) | Posição 1101

> polimorfismo é uma aplicação de ponteiros em funções.

Destaque (Amarelo) | Posição 1104

> As linguagens OO podem não ter criado o polimorfismo, mas o tornaram muito mais seguro e muito mais conveniente.

Destaque (Amarelo) | Posição 1174

> Como consequência, as regras de negócio, a UI e o banco de dados podem ser compiladas em três componentes separados ou unidades implantáveis (por exemplo, arquivos jar, DLLs ou arquivos Gem) que tenham as mesmas dependências de código fonte.

Destaque (Amarelo) | Posição 1177

> Por sua vez, as regras de negócio podem ser implantadas independentemente da UI e do banco de dados.

Destaque (Amarelo) | Posição 1179

> Resumindo, quando o código fonte muda em um componente, apenas esse componente deve ser reimplantado.

Destaque (Amarelo) | Posição 1180

> Isso se chama implantação independente.

Destaque (Amarelo) | Posição 1184

> a OO é a habilidade de obter controle absoluto, através do uso do polimorfismo, sobre cada dependência de código fonte do sistema.

Destaque (Amarelo) | Posição 1245

> Meu objetivo aqui é apontar algo muito dramático sobre a diferença entre programas em Clojure e Java. O programa em Java usa uma variável mutável — uma variável que muda de estado durante a execução do programa. Essa variável é i — a variável de controle do laço. Não existe uma variável mutável no programa em Clojure. No programa em Clojure, variáveis como

---

Destaque (Amarelo) | Posição 1248

> x são inicializadas, mas nunca são modificadas.

Destaque (Amarelo) | Posição 1249

> Isso nos leva a uma declaração surpreendente: variáveis em linguagens funcionais não variam.

Destaque (Amarelo) | Posição 1251

> Por que esse ponto é importante do ponto de vista arquitetural? Por que um arquiteto ficaria preocupado com a mutabilidade das variáveis? A resposta é absurdamente simples: todas as condições de corrida (race conditions), condições de impasse (deadlock conditions) e problemas de atualizações simultâneos decorrem das variáveis mutáveis.

Destaque (Amarelo) | Posição 1253

> Você não pode ter um problema de condição de corrida ou de atualização concorrente se nenhuma variável for atualizada. Não pode ter deadlocks sem locks mutáveis.

Destaque (Amarelo) | Posição 1256

> todos os problemas que exigem várias threads e múltiplos processadores — não podem acontecer se não houver variáveis mutáveis.

Destaque (Amarelo) | Posição 1270

> Já que mudar o estado expõe esses componentes a todos os problemas de concorrência, é uma prática comum usar algum tipo de memória transacional para proteger as variáveis mutáveis de atualizações concorrentes e condições de corrida.

Destaque (Amarelo) | Posição 1293

> O ponto é que aplicações bem estruturadas devem ser segregadas entre componentes que mudam e que não mudam variáveis.

Destaque (Amarelo) | Posição 1300

> Atualmente, é comum que os processadores executem bilhões de instruções por segundo e tenham bilhões de bytes de RAM. Quanto maiores forem a memória e a velocidade das nossas máquinas, menos precisaremos de um estado mutável.

Destaque (Amarelo) | Posição 1312

> Event sourcing é uma estratégia em

Destaque (Amarelo) | Posição 1312

> que armazenamos as transações, mas não o estado. Quando o estado for solicitado, simplesmente aplicamos todas as transações desde o início.

Destaque (Amarelo) | Posição 1319

> Mais importante, nada nunca é deletado ou atualizado nesse modelo de armazenamento de dados.

Destaque (Amarelo) | Posição 1319

> Como consequência, nossas aplicações não são CRUD; são apenas CR. Além disso, como nenhuma atualização ou eliminação ocorre no armazenamento de dados, nenhum problema de atualização

---

Destaque (Amarelo) | Posição 1320

> concorrente pode ocorrer.

Destaque (Amarelo) | Posição 1321

> Se temos armazenamento e poder de processamento suficientes, podemos tornar nossas aplicações inteiramente imutáveis — e, portanto, inteiramente funcionais.

Destaque (Amarelo) | Posição 1326

> A programação estruturada é a disciplina imposta sobre a transferência direta de controle. A programação orientada a objetos é a disciplina imposta sobre a transferência indireta de controle. A programação funcional é a disciplina imposta sobre atribuição de variáveis. Cada um desses três paradigmas tirou algo de nós.

Destaque (Amarelo) | Posição 1330

> Cada um restringe algum aspecto da maneira como escrevemos código.

Destaque (Amarelo) | Posição 1333

> o software não é uma tecnologia de rápido desenvolvimento. As regras do software são as mesmas de 1946, quando Alan Turing escreveu o primeiro código a ser executado em um computador eletrônico.

Destaque (Amarelo) | Posição 1334

> As ferramentas e o hardware mudaram, mas a essência do software permanece a mesma.

Destaque (Amarelo) | Posição 1335

> O software — a matéria vital dos programas de computadores — é composto de sequência, seleção, iteração e indireção. Nada mais. Nada menos.

Destaque (Amarelo) | Posição 1346

> Bons sistemas de software começam com um código limpo.

Destaque (Amarelo) | Posição 1348

> É aí que entram os princípios SOLID.

Destaque (Amarelo) | Posição 1351

> Uma classe é apenas um agrupamento acoplado de funções e dados.
