

Engenharia de Software

Prof. Vinicius Cardoso Garcia

vcg@cin.ufpe.br :: [@vinicius3w](https://twitter.com/vinicius3w) :: viniciusgarcia.me

[IF977] Engenharia de Software

<http://bit.ly/vcg-es>

Licença do material

Este Trabalho foi licenciado com uma Licença

Creative Commons - Atribuição-NãoComercial-
Compartilhual 3.0 Não Adaptada



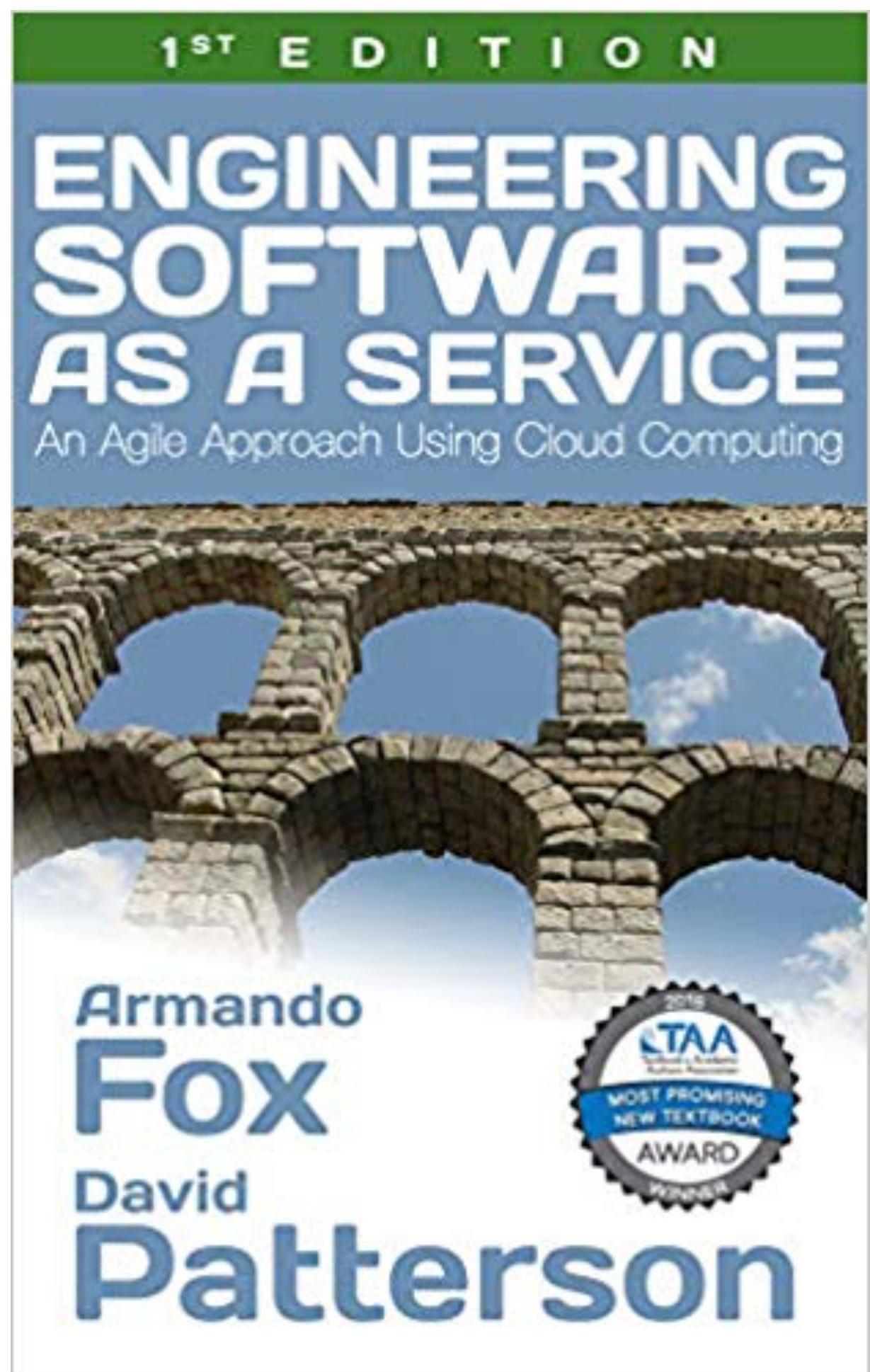
Mais informações visite

[http://creativecommons.org/licenses/by-nc-sa/
3.0/deed.pt](http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt)



Referências

- A biblioteca do Desenvolvedor de Software dos dias de hoje
 - <http://bit.ly/31WYK5f>
- SWEBOK: Guide to the Software Engineering Body of Knowledge (SWEBOK)
 - <http://www.computer.org/web/swebok>
- Engineering Software as a Service: An Agile Approach Using Cloud Computing
 - <http://www.saasbook.info/>
- Marco Tulio Valente. Engenharia de Software Moderna
 - <https://engsoftmoderna.info/>





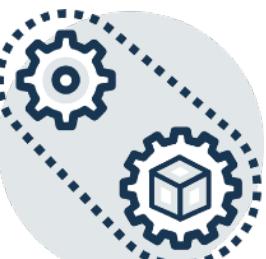
Evolução e Garantia de Qualidade

Introdução a Testes de Software



Mudança é inevitável

- Mudança de software é inevitável
 - Novos requisitos surgem quando o software é usado
 - O ambiente de negócio muda
 - Erros devem ser reparados
 - Novos computadores e equipamentos são adicionados ao sistema;
 - O desempenho ou a confiabilidade do sistema deve ser melhorada
- Um problema-chave para as organizações é a implementação e o gerenciamento de mudanças em seus sistemas



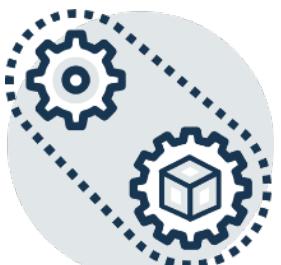
Importância da evolução

- As organizações fazem grandes investimentos em seus sistemas de software – eles são **ativos críticos** de negócios
- Para manter o valor desses ativos de negócio, eles devem ser mudados e atualizados
- A maior parte do orçamento de software nas grandes organizações é voltada para evolução, ao invés do desenvolvimento de sistemas novos



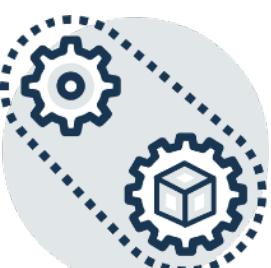
Dinâmica da evolução de programas

- **Dinâmica de evolução de programas** é o estudo de mudança de sistema
- Lehman e Belady propuseram que havia uma série de '**leis**' (hipóteses) que se aplicavam a todos os sistemas quando eles evoluiam
- Na prática, são observáveis, de fato, mas com ressalvas
 - Aplicáveis principalmente a sistemas de grande porte



Leis de Lehman

1. Manutenção é um processo inevitável.
 - Ambiente do sistema muda e requisitos mudam
2. Alteração do sistema degrada sua estrutura
 - Como evitar? Manutenção preventiva
3. Sistemas de grande porte possuem dinâmica própria (estágios iniciais de desenv.).
Tamanho e complexidade dificultam alteração
4. Estado saturado. Mudanças de recursos e pessoal não têm efeito. Sistemas grandes e overhead de comunicação
5. Novas funcionalidades introduzem novos defeitos. Não orçar grandes incrementos sem pensar nas correções de defeitos
6. Declínio da qualidade e insatisfação dos usuários
7. Aprimoramento a partir de sistemas de feedback



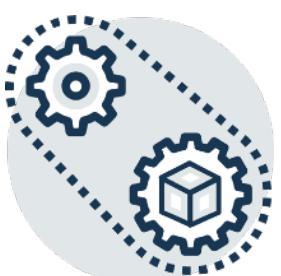
Manutenção de software

- É a modificação de um programa após ter sido colocado em uso.
- A manutenção **normalmente** não envolve mudanças **consideráveis** na arquitetura do sistema.
- As mudanças são implementadas pela modificação de componentes existentes e pela adição de novos componentes ao sistema



A Manutenção é Inevitável

- Os sistemas estão fortemente **acoplados** ao seu **ambiente**
 - Quando um sistema é **implantado** em um ambiente, ele **muda** esse ambiente e, portanto, mudam os **requisitos de sistema**
 - Portanto, os sistemas **DEVEM** ser mantidos se forem úteis em um ambiente



Tipos de manutenção

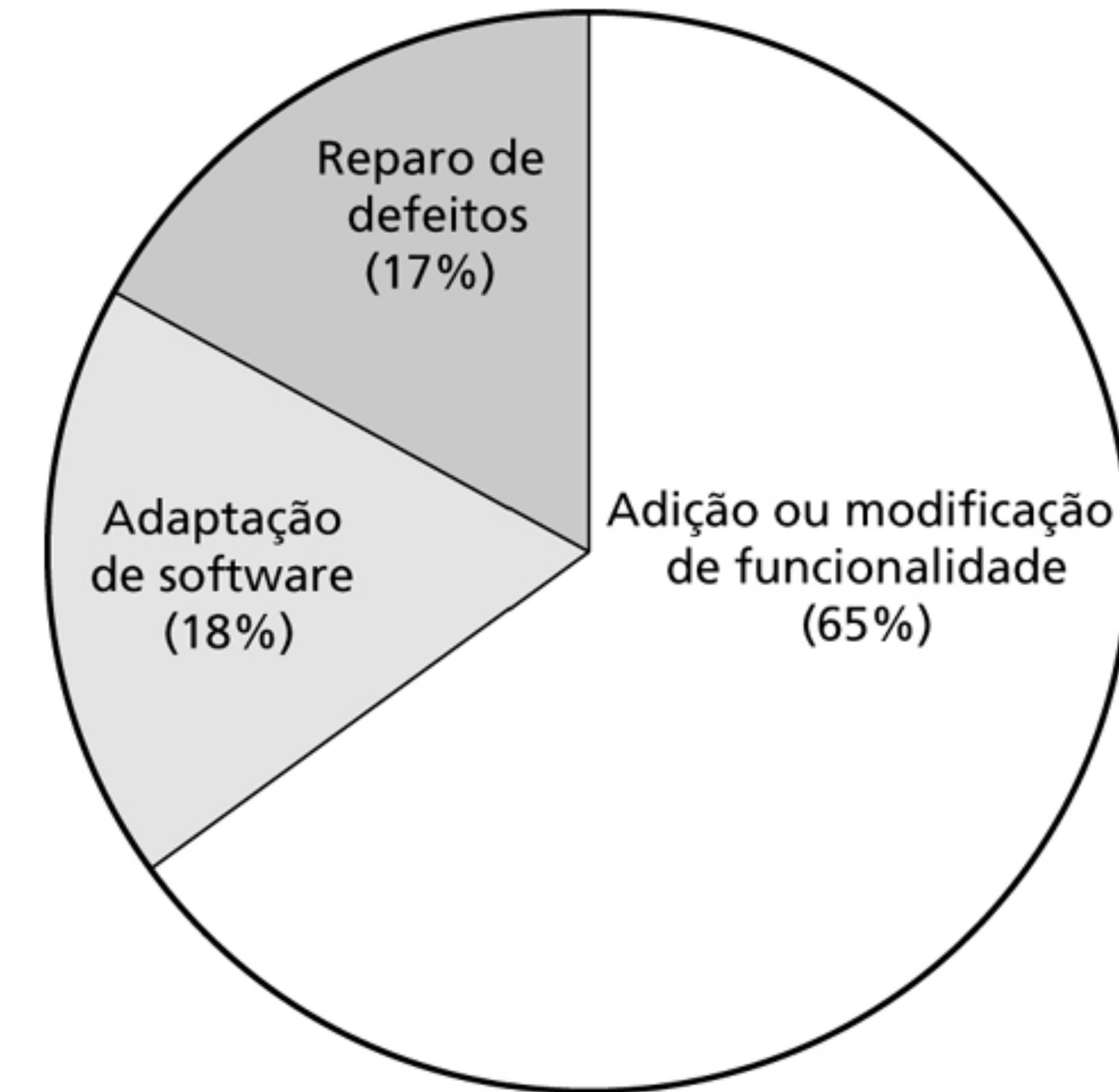
- Manutenção para **reparar** defeitos de software
 - Mudança em um sistema para corrigir deficiências de maneira a atender seus requisitos.
- Manutenção para **adaptar** o software a um ambiente operacional diferente
 - Mudança de um sistema de tal maneira que ele opere em um ambiente diferente (computador, OS, etc.) a partir de sua implementação inicial.
- Manutenção para **adicionar** funcionalidade ao sistema ou modificá-lo
 - Modificação do sistema para satisfazer a novos requisitos.



Distribuição de esforços de manutenção

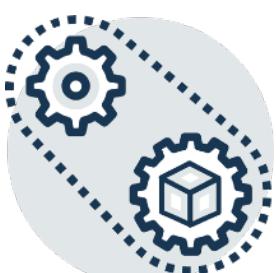
Figura 21.2

Distribuição de esforços de manutenção.



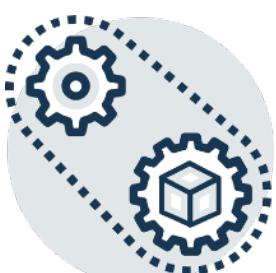
Custos de manutenção

- Geralmente, são **maiores que os custos de desenvolvimento** (de 2 a 100 vezes, dependendo da aplicação).
- São afetados por fatores **técnicos e não técnicos**
- A manutenção **corrompe** a estrutura do software, tornando a manutenção posterior **mais difícil**
 - **Design Erosion**
 - Software "em envelhecimento" pode ter altos custos de suporte (por exemplo, linguagens antigas, compiladores, etc.)



Fatores de custo de manutenção

- Estabilidade da equipe
 - Os custos de manutenção são reduzidos se o mesmo pessoal estiver envolvido por algum tempo
- Responsabilidade contratual
 - Os desenvolvedores de um sistema podem não ter responsabilidade contratual pela manutenção, portanto, não há incentivo para projetar para mudanças futuras
- Habilidade do pessoal
 - O pessoal da manutenção geralmente é inexperiente e tem conhecimento limitado de domínio
- Idade e estrutura do programa
 - À medida que os programas envelhecem, sua estrutura é degradada e se torna mais difícil de ser compreendida e modificada



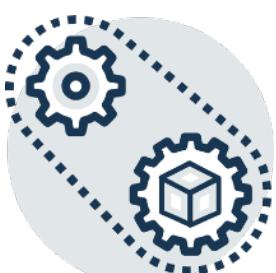
Previsão de manutenção

- Avaliação de quais partes do sistema podem causar problemas e ter altos custos de manutenção
 - A aceitação de mudança depende da facilidade de manutenção dos componentes afetados por ela
 - A implementação de mudanças degrada o sistema e reduz a sua facilidade de manutenção
 - Os custos de manutenção dependem do número de mudanças, e os custos de mudança dependem da facilidade de manutenção



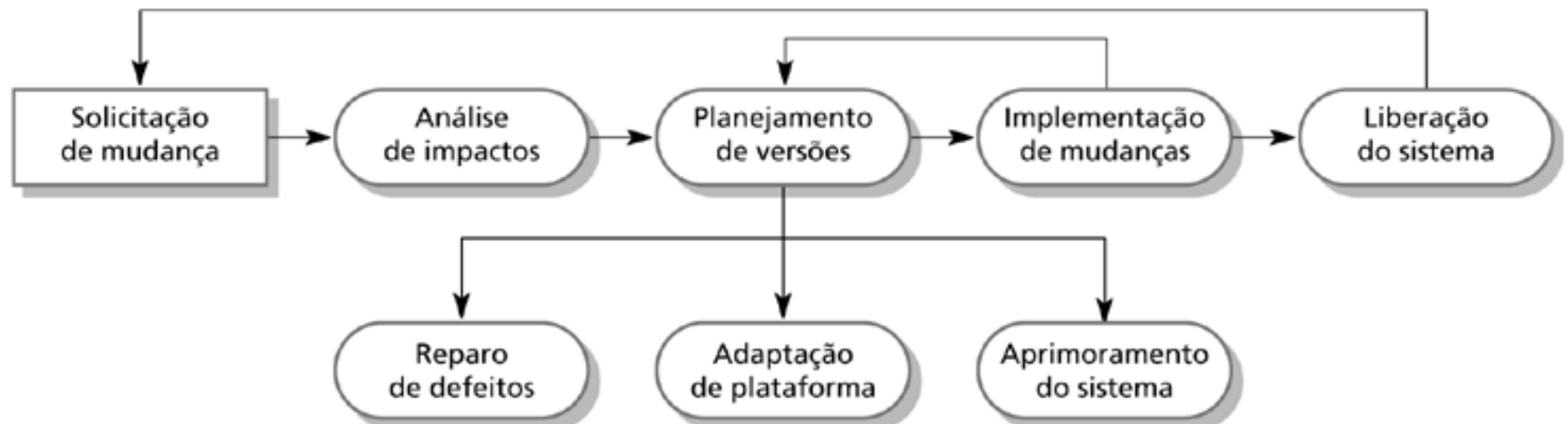
Previsão de mudanças

- A previsão do número de mudanças requer o entendimento dos relacionamentos entre um sistema e seu ambiente
- Sistemas fortemente acoplados requerem mudanças sempre que o ambiente é mudado
- Fatores que influenciam esse relacionamento são
 - O número e a complexidade das interfaces de sistema
 - O número de requisitos de sistema inherentemente voláteis
 - Os processos de negócio nos quais o sistema é usado



O processo de evolução de sistema

Figura 21.6 Processo de evolução de sistema.



Reengenharia de sistema

- É a reestruturação ou reescrita de parte ou de todo um sistema sem mudar sua funcionalidade
 - Importante ressaltar: **reestruturação de grande porte!**
 - Aplicável onde partes de um sistema de grande porte necessitam de manutenção frequente
 - Envolve a adição de esforço para tornar o sistema mais fácil de manter
 - **Simplicidade** é um objetivo **complexo**



Atividades do processo de reengenharia

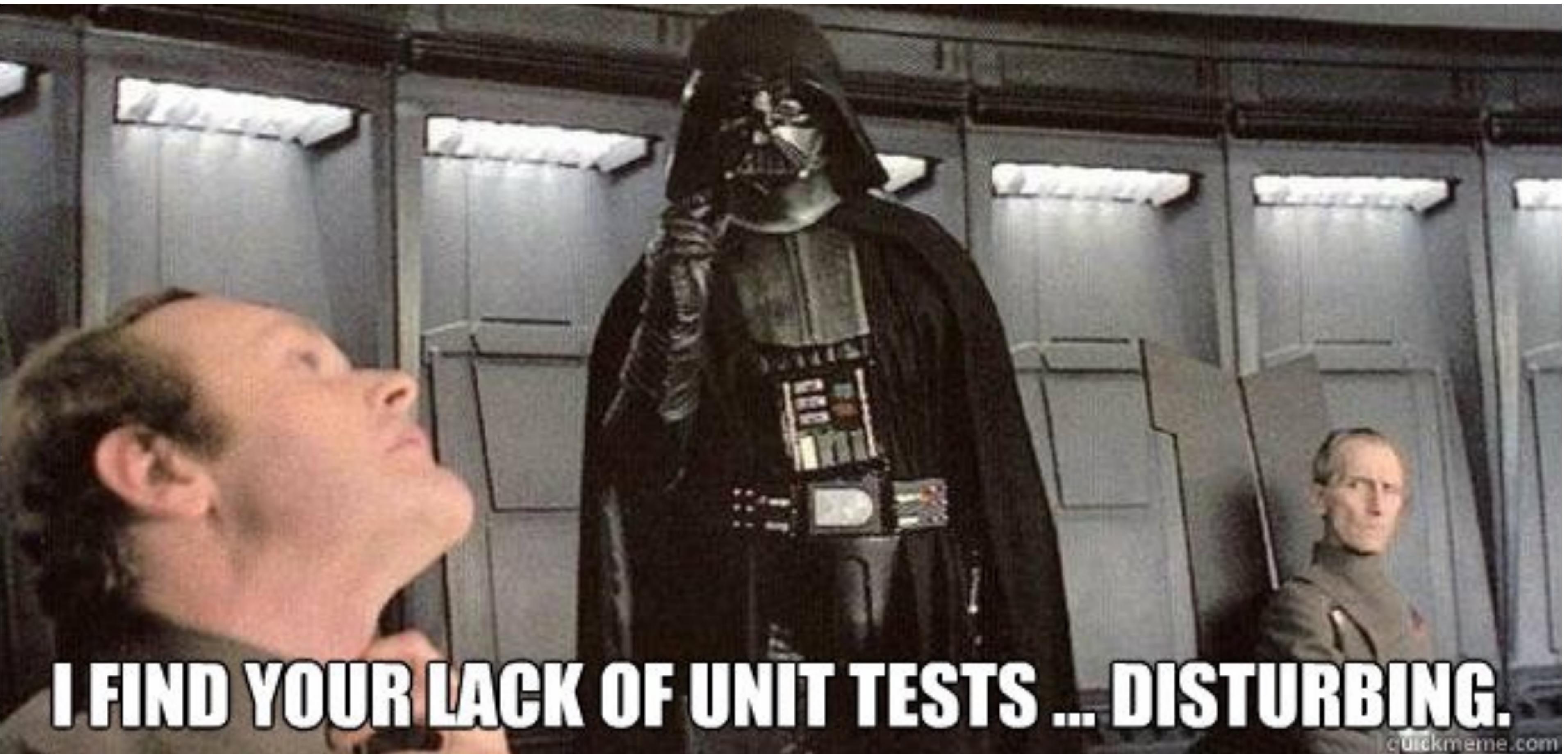
- Conversão de código-fonte
 - Converter o código para uma nova linguagem
- Engenharia reversa
 - Analisar o programa para compreendê-lo
- Aprimoramento da estrutura de programa
 - Analisar e modificar a estrutura para facilidade de entendimento
- Modularização de programa
 - Reorganizar a estrutura do programa
- Reengenharia de dados
 - Limpar e reestruturar os dados do sistema



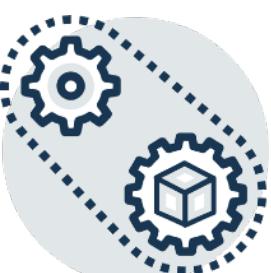
Fatores do custo de reengenharia

- A qualidade do software que deve passar pela reengenharia
- O apoio de ferramentas disponíveis para reengenharia
- Extensão da conversão de dados
- A disponibilidade do pessoal especializado
 - Isso pode ser um problema com sistemas antigos baseados em tecnologia que não são mais amplamente usadas

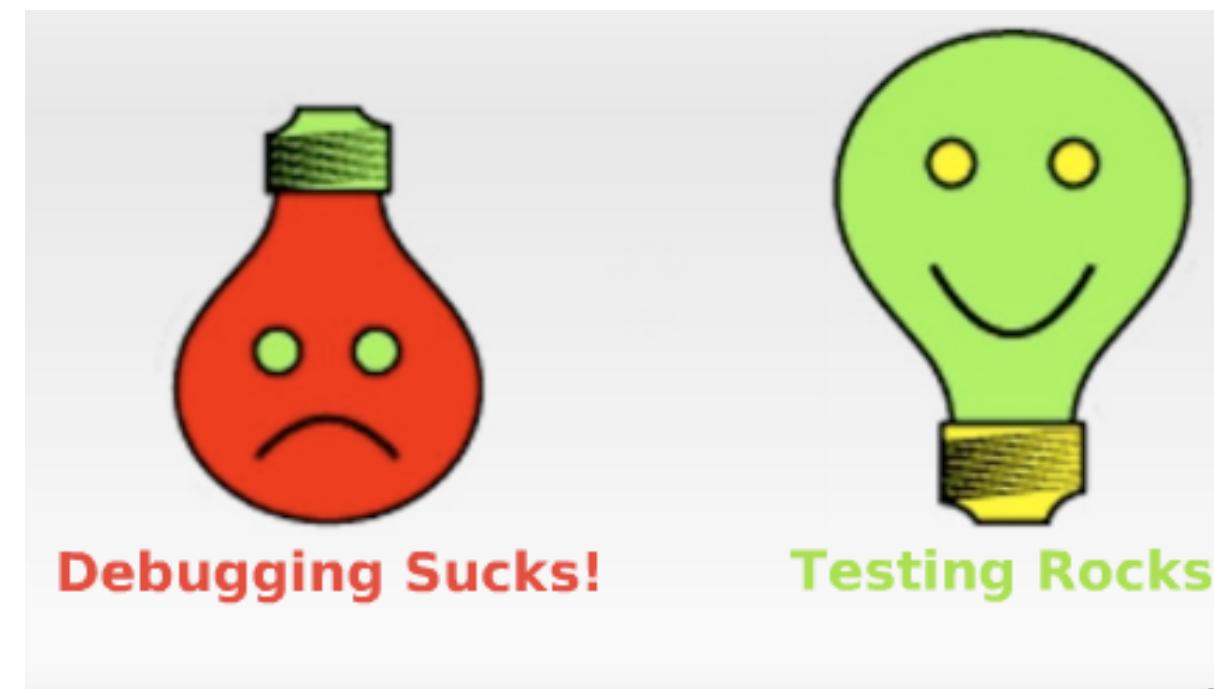




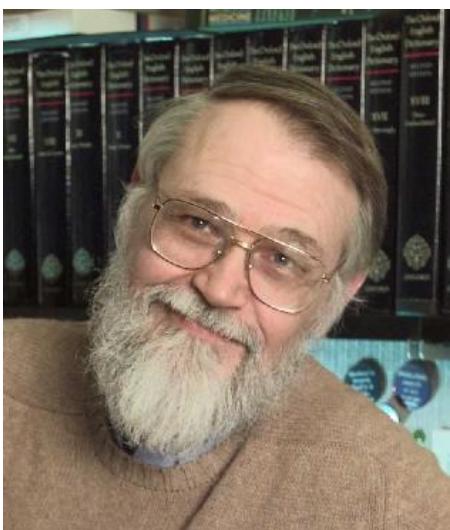
Testes de Software



Testar pra quê?

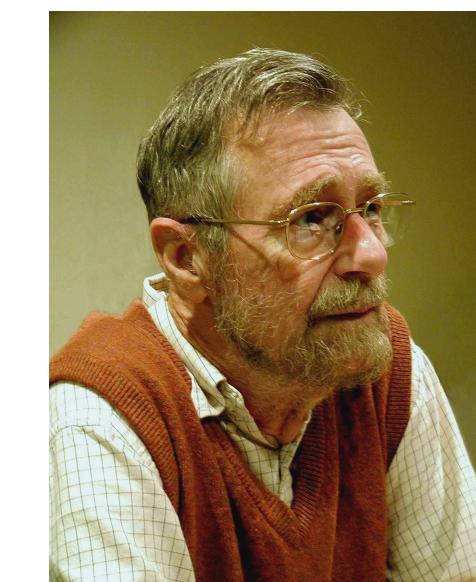


A depuração é duas vezes mais difícil que escrever o código em primeiro lugar. Portanto, se você escrever o código o mais inteligentemente quanto possível, você, por definição, não é inteligente o suficiente para depurá-lo.



Turing Award winner Brian Kernighan, co-inventor of the C language

Testes nunca vão demonstrar a **ausência** de erros no software, somente sua **presença**!



Turing Award winner Edsger Dijkstra, "father of object-oriented programming"

Motivação

- Ocorrência de **falhas humanas** no processo de desenvolvimento de software é considerável
- Processo de testes é **indispensável** na garantia de qualidade de software
- Custos associados às **falhas** de software justificam um processo de testes cuidadoso e bem planejado



Survey Finds 58% of Software Bugs Result from Test Infrastructure and Process, Not Design Defects

Developers Prefer Taxes to Dealing with Software Testing

Sunnyvale, Calif. — June 2, 2010 Electric Cloud®, the leading provider of software production management (SPM) solutions, today released the results of a survey conducted in partnership with Osterman Research showing that the majority of software bugs are attributed to poor testing procedures or infrastructure limitations rather than design problems. Additionally, the software test process is generally considered an unpleasant process, with software development professionals rating the use of their companies' test systems more painful than preparing taxes.

Fifty-eight percent of respondents pointed to problems in the testing process or infrastructure as the cause of their last major bug found in delivered or deployed software, not design defects.

Specifically, the survey found:

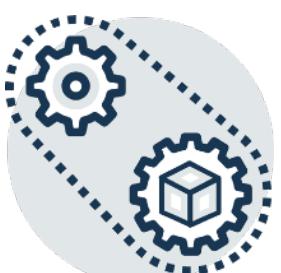


- ✓ Completely automated software testing environments are still rare, with just 12 percent of software development organizations using fully automated test systems. Almost 10 percent reported that all testing was done manually.



Testes hoje em dia

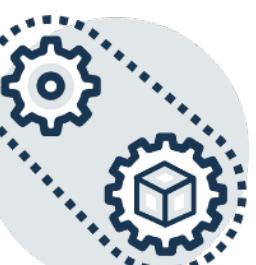
- Antes
 - Desenvolvedores finalizam o código, alguns testes ad-hoc
 - “jogar pro alto a Garantia de Qualidade [QA]”
 - Equipe de QA manualmente exercita o software
- Hoje em dia / Ágil
 - Testes é parte de **TODA** iteração Ágil
 - Desenvolvedores testam seu **próprio** código
 - Ferramentas e processos de testes **automatizados**
 - Equipe de testes/QA melhora as **ferramentas/testabilidade**



Testes hoje em dia

- Antes
 - Desenvolvedores finalizam o código, alguns testes ad-hoc
 - “jogar pra ver” – Quality Assurance [QA]
 - Equipe de QA
- Hoje em dia
 - Testes é parte do processo
 - Desenvolvedores fazem os próprios testes
 - Ferramentas e processos de testes **automatizados**
 - Equipe de testes/QA melhora as **ferramentas/testabilidade**

Qualidade de Software é o resultado de um bom **processo**, ao invés de responsabilidade de um grupo específico



Falha, Falta e Erro

- Falha
 - Incapacidade do software de realizar a função requisitada (aspecto externo)
 - Exemplo: Terminação anormal, restrição temporal violada
- Falta
 - Causa de uma falha
 - Exemplo: Código incorreto ou faltando
- Erro
 - Estado intermediário (instabilidade), provém de uma falta
 - Pode resultar em falha, se propagado até a saída



Falha, Falta e Erro



System Specification

The system should calculate $(3 * a) ^ 4 / 2$ for a given input a.

<u>Code</u>	<u>Runtime</u>
int a = 9;	a = 9
a = double(a);	a = 18 Error!!
a = a ^ 4;	a = 104976 Error!!
result = a / 2;	result = 52488 Error!!
Sys0(result);	User Output Failure!!

Expected Result

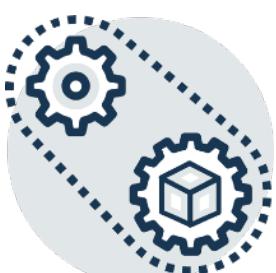
Input: 9
Output: 265720.5 Failure!!

Fault: happens in line 2, when the developer use double function instead of multiply by 3.



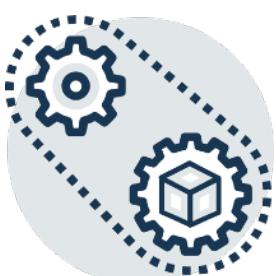
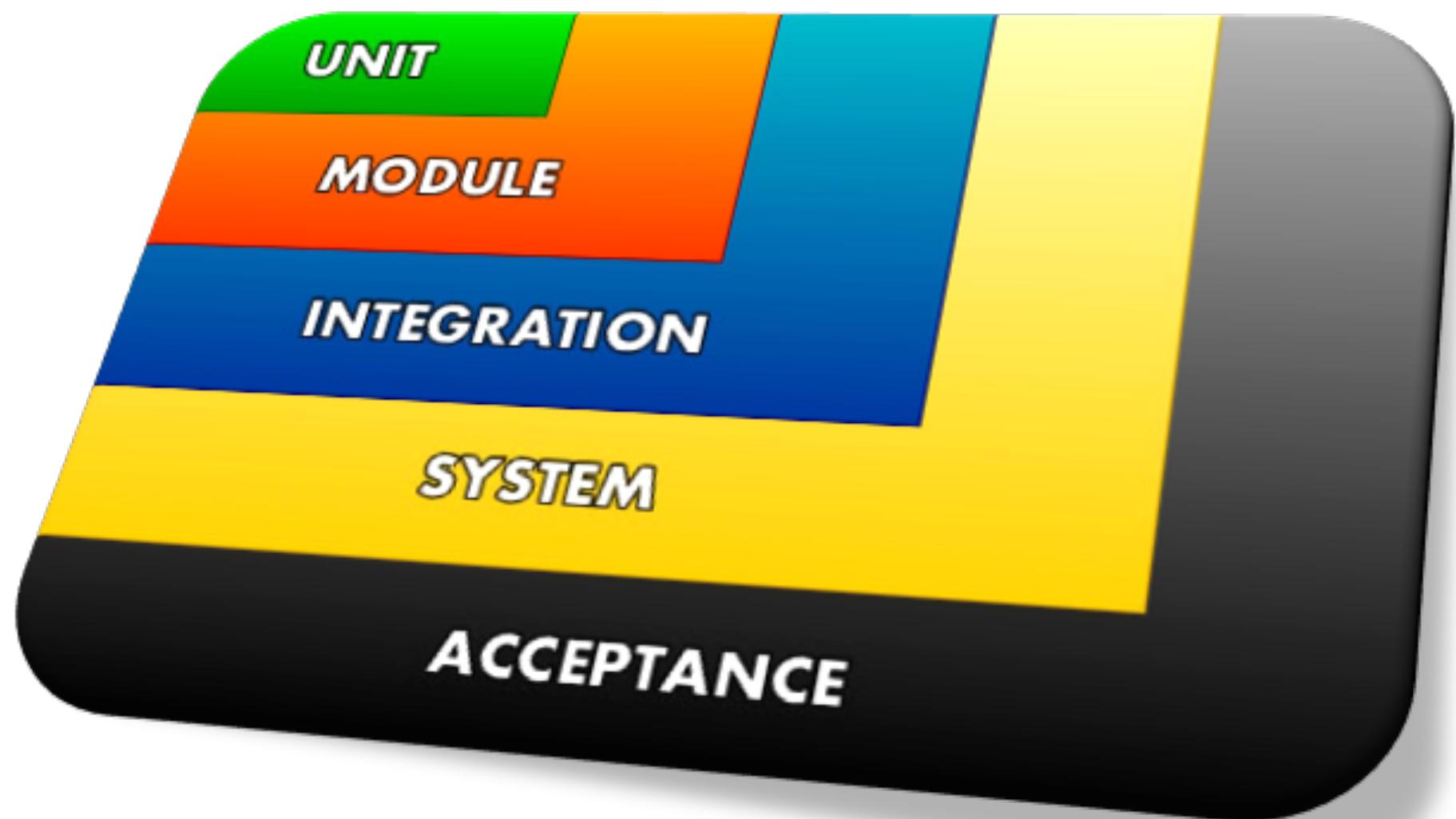
Revisando

- **Validação** -> avaliação durante, ou ao final do ciclo de desenvolvimento de software para determinar se satisfaz aos requisitos especificados
- **Verificação** -> avaliação para determinar se os produtos de uma dada fase do ciclo de desenvolvimento satisfazem as condições impostas no inicio daquela



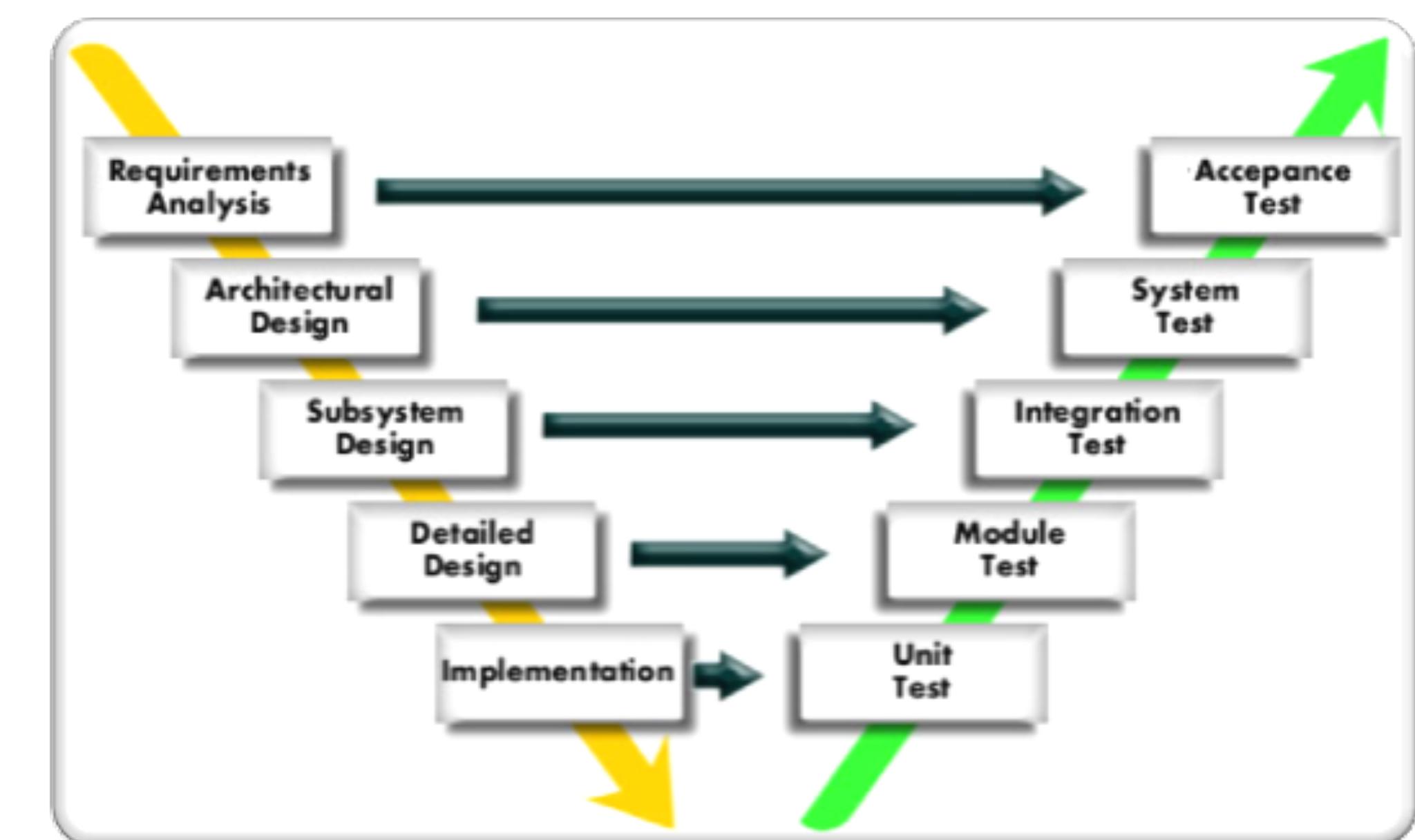
Níveis de Teste (Amman, 2008)

- Teste Unitário
 - Avalia o software com relação a implementação
- Teste de Módulo
 - Avalia o software com respeito a detalhes do design
- Teste de Integração
 - Avalia o software com respeito ao design de subsistemas
- Teste de Sistemas
 - Avalia o software com respeito ao design da arquitetura
- Teste de Aceitação
 - Avalia o software com respeito a seus requisitos
- Alpha/Betha testing
 - Commercial Off-The-Shelf



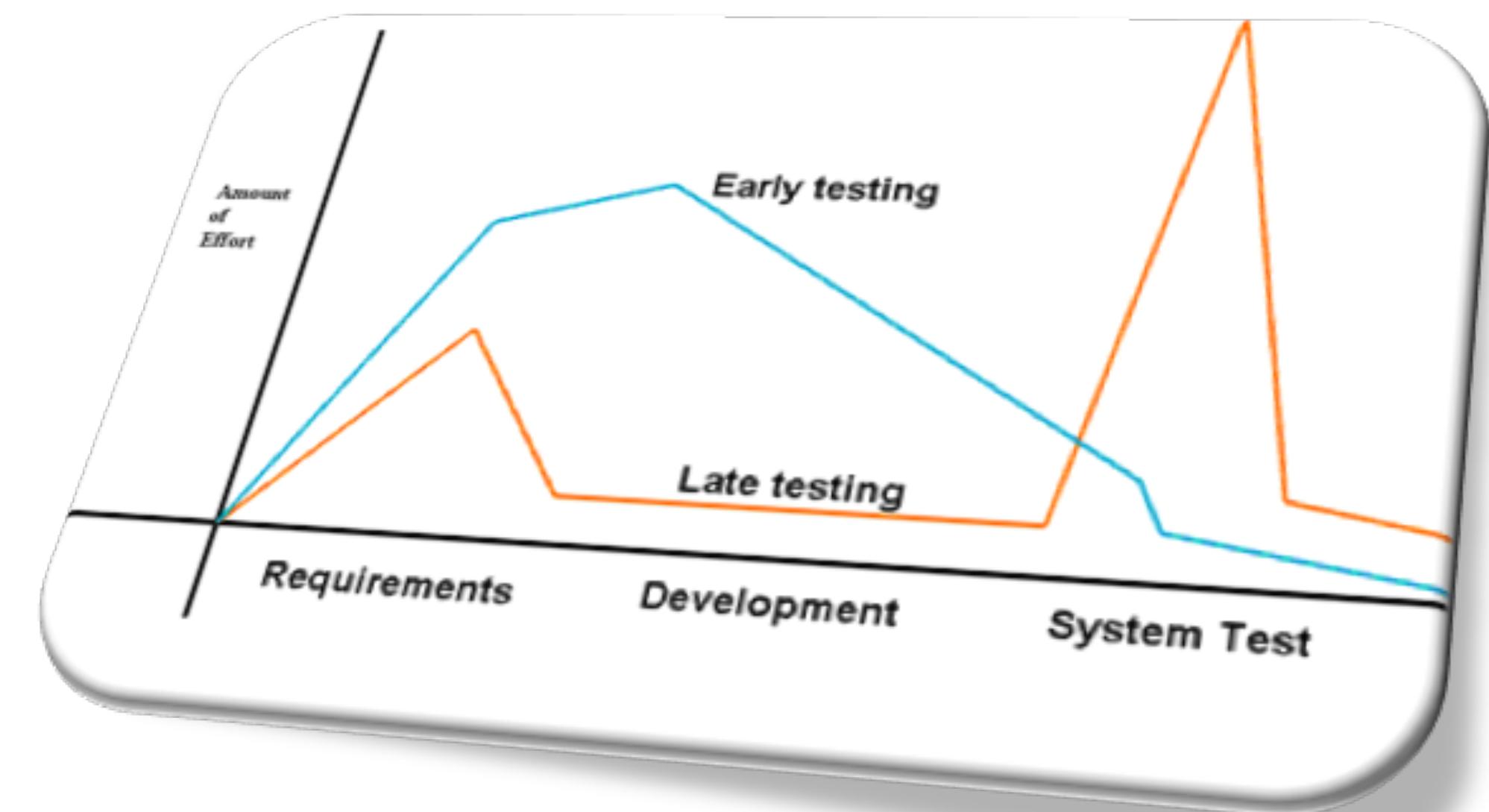
V-Model

- As atividades devem ser realizadas em paralelo com as atividades de desenvolvimento
- Mostra como as atividades de teste (verificação e validação) podem ser integradas dentro de cada fase do ciclo de vida



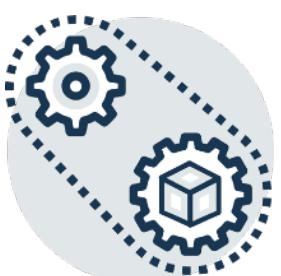
Teste e ciclo de vida

- Nas "dark ages" teste era considerado uma fase do desenvolvimento que era realizada após a implementação
- O Rational Unified Process (RUP) lista teste como uma disciplina que é ativa em todas as fases de desenvolvimento (Kruchten 03)
- A maioria dos processos de desenvolvimento incluem atividades de teste em todas as fases do ciclo de desenvolvimento



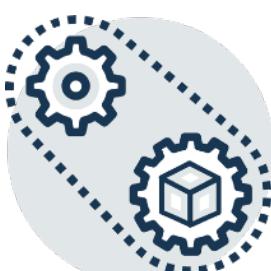
Noção de confiabilidade

- Algumas faltas **escaparão** inevitavelmente
 - Tanto dos testes quanto da depuração
- Falta pode ser mais ou menos **perturbadora**
 - Dependendo do que se trate e em qual frequência irá surgir para o usuário final
- Quando liberar ou não sistema para uso?
- Confiabilidade de software
 - É uma estimativa probabilística que mede a frequência com que um software irá executar sem falha em dado ambiente e por determinado período de tempo
 - Assim, entradas para testes devem se aproximar do ambiente do usuário final



Políticas de teste

- Somente testes exaustivos podem mostrar que um programa está livre de defeitos
 - Contudo, testes exaustivos são impossíveis
- As políticas de teste definem a abordagem a ser usada na seleção de testes de sistema:
 - Todas as funções acessadas por meio de menus devem ser testadas
 - As combinações de funções acessadas por meio dos mesmos menus devem ser testadas
 - Onde as entradas de usuário são fornecidas, todas as funções devem ser testadas com entradas corretas e incorretas



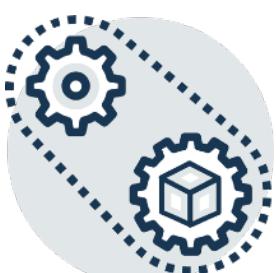
Diretrizes de teste

- Diretrizes são **recomendações** para a equipe de teste para auxiliá-los a escolher os testes que revelarão defeitos no sistema
 - Escolher entradas que forcem o sistema a gerar todas as mensagens de erro
 - Projetar entradas que causem overflow dos buffers
 - Repetir a mesma entrada ou série de entradas várias vezes
 - Forçar a geração de saídas inválidas
 - Forçar resultados de cálculo a serem muito grandes ou muito pequenos



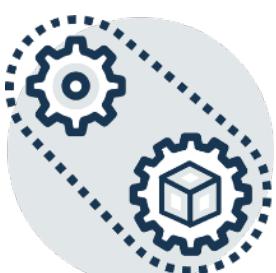
BDD+TDD: Visão Geral

- Behavior-driven design (BDD)
 - Desenvolver histórias do usuário [HU] (**as características desejadas**) para descrever como a aplicação deve funcionar
 - Ideia: HUs se tornam testes de **aceitação** e **integração**
- Test-driven development (TDD)
 - Definir uma nova HU pode requerer a escrita de **novo código**
 - TDD diz: escreva testes unitários e funcionais **primeiro**, antes do código propriamente dito
 - Isto é: escreva o teste do código que você **deseja ter**





FIRST & TDD



A propriedade FIRST

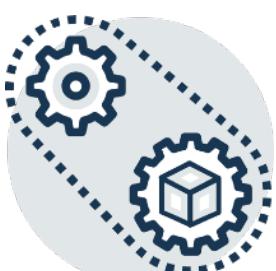
- **Fast**: executar (um conjunto de) testes rapidamente
- **Independent**: Nenhum teste depende de outro, então podemos executá-los em qualquer ordem
- **Repeatable**: Execute N vezes e obtenha o mesmo resultado (isolamento de erros e automação)
- **Self-checking**: Podem checar automaticamente se passaram (sem intervenção humana)
- **Timely**: Escritos ao mesmo tempo que o código testado (com TDD, escreva-o primeiro)



Pergunta

Que tipos de código podem ser testados **Repetidamente** e **Independentemente**?

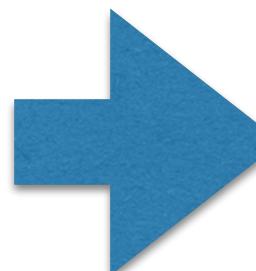
1. Código que depende da aleatoriedade (por exemplo embaralhar um baralho de cartas)
2. Código que depende da hora do dia (por exemplo, os backups executados todos os domingos à meia-noite)
 - A. Somente (1)
 - B. Somente (2)
 - C. Os dois
 - D. Nenhum dos dois



Pergunta

Que tipos de código podem ser testados **Repetidamente** e **Independentemente**?

1. Código que depende da aleatoriedade (por exemplo embaralhar um baralho de cartas)
2. Código que depende da hora do dia (por exemplo, os backups executados todos os domingos à meia-noite)
 - A. Somente (1)
 - B. Somente (2)
 - C. Os dois
 - D. Nenhum dos dois



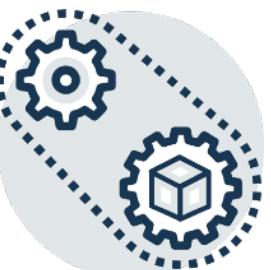
O Que é Test-Driven Development?

- Todo desenvolvedor sabe que testes são importantes, mas quase ninguém faz
 - Design first, Test maybe
- TDD é uma prática ágil para garantir que a qualidade é construída desde o início
 - Test first, Design always
- TDD se baseia em dois conceitos principais
 - Testes Unitários & Funcionais
 - Refactoring



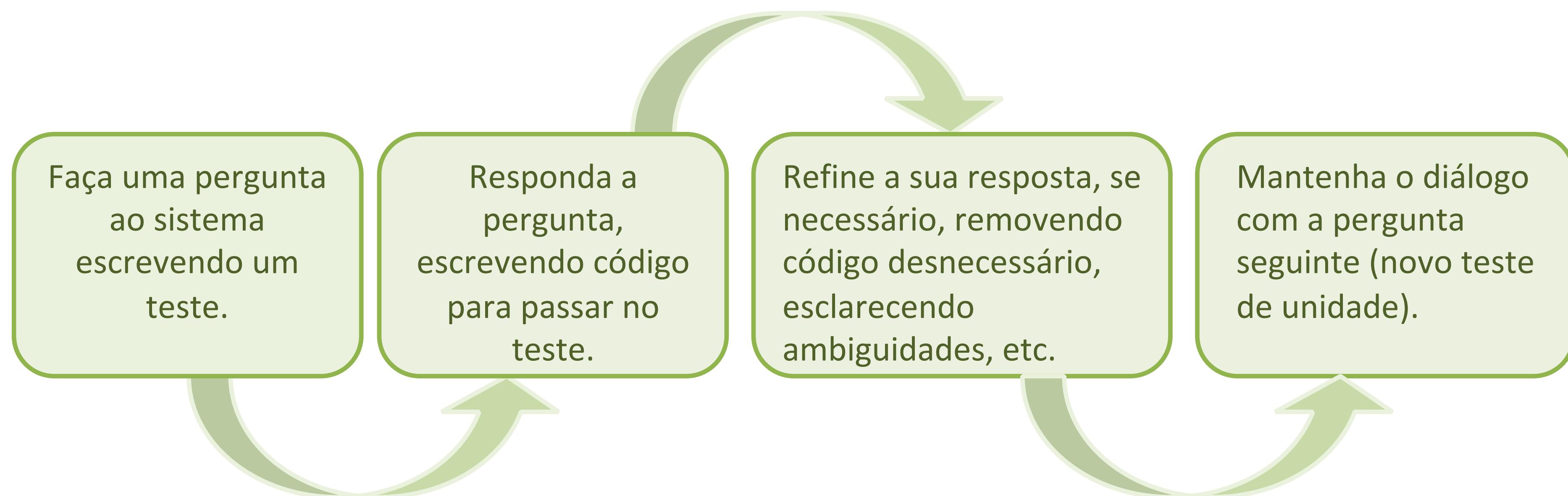
O Que é Test-Driven Development?

- Com TDD **especificamos nosso software** em detalhes no momento que vamos escrevê-lo **criando testes executáveis** e rodando-os de maneira que eles mesmos testem nosso software
- Serve para **diagnosticar precocemente erros e falhas** que podem vir a ser problemas na finalização do projeto

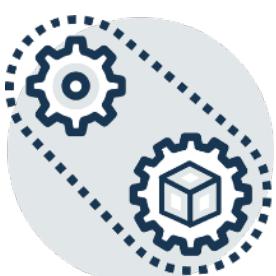


Ciclo do TDD

- O ciclo do TDD transforma a programação em um diálogo

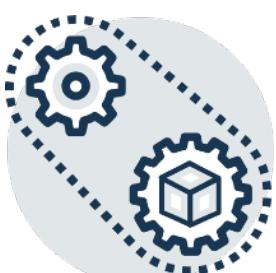


Red → Green [→ Refactor]→ Red → Green [→ Refactor]→ Red → Green [→ Refactor]→ ...



Mantra do TDD

- As cores se referem ao que você vê quando escreve e roda um teste em uma ferramenta de teste (i.e. Mocha, Jasmine, Vow...)
- **Vermelho:** Você criou um teste que expressa o que você espera que o código faça. O teste falha (fica vermelho) porque você ainda não criou o código para fazer com que o teste passe.
- **Verde:** Você escreve o código para fazer o teste passar (ficar verde). Você não se penaliza para alcançar um projeto limpo, simples e sem replicações. Você vai alcançar um projeto mais complexo adiante, quando os testes passam e você se sente mais seguro e confortável para evoluções. Se o projeto está ok, você pode ignorar a fase de refactoring e começar um novo ciclo com um novo teste.
- **Refactor:** Se necessário você pode melhorar o projeto do código que passou no teste e ir para o próximo novo teste



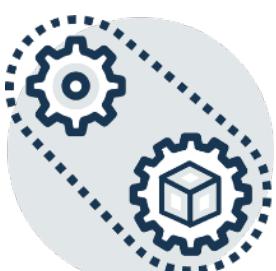
Benefícios e Limitações de TDD

- Simples, Desenvolvimento Incremental
- Processo de Desenvolvimento mais Simples e Produtivo
- Testes de Recessão Constantes
- Melhoria na Comunicação
- Melhoria no Entendimento dos Comportamentos dos Requisitos do Software
- Melhoria no Projeto do Software



Benefícios

- Simples, Desenvolvimento Incremental
- Processo de Desenvolvimento mais Simples e Produtivo
- Testes de Recessão Constantes
- Melhoria na Comunicação
- Melhoria no Entendimento dos Comportamentos dos Requisitos do Software
- Melhoria no Projeto do Software
- Incentiva a simplicidade
- Aumenta a confiança no código
- Ajuda como documentação
- Facilita refactorings



Limitações

- Custo inicial para:
 - Curva de aprendizagem
 - Configuração do ambiente
- Cultura do time



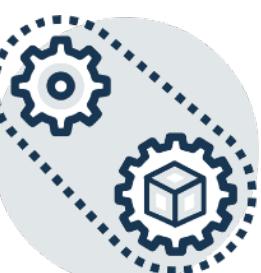
47



TDD vs. Debugging

Conventional	TDD
Write 10s of lines, run, hit bug: break out debugger	
Insert printf's to print variables while running repeatedly	
Stop in debugger, tweak/set variables to control code path	
Dammit, I thought for sure I fixed it, now I have to do this all again	

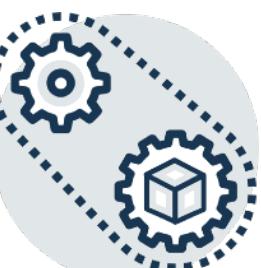
- Lição 1: TDD usa as mesmas habilidades e técnicas da depuração convencional - mas mais produtivo (FIRST)
- Lição 2: escrever testes antes do código leva mais tempo no início, mas geralmente menos tempo no geral



Pergunta

Qual afirmação **não óvia** sobre o teste é FALSA?

- A. Mesmo com 100% de cobertura de teste não é garantia de estar livre de erros
- B. Se você puder estimular uma condição causadora de erro, falha ou falta em um depurador, poderá capturá-la em um teste
- C. O teste elimina a necessidade de usar um depurador
- D. Quando você altera seu código, também precisa alterar seus testes



Pergunta

Qual afirmação **não óvia** sobre o teste é FALSA?

- A. Mesmo com 100% de cobertura de teste não é garantia de estar livre de erros
- B. Se você puder estimular uma condição causadora de erro, falha ou falta em um depurador, poderá capturá-la em um teste
- C. O teste elimina a necessidade de usar um depurador
- D. Quando você altera seu código, também precisa alterar seus testes