Engenharia de Software

Prova, 13.03.2023

Tipos de Sistemas (ABC)

Sistemas A (Acute ou Críticos)

- Nada pode dar errado, pois o custo é imenso
- Perdas significativas financeiramente e/ou em vidas humanas

Sistemas B (Business ou Negócios)

- Sistemas importantes para uma organização
- Risco: não usarem técnicas de ES e se tornarem um passivo, em vez de um ativo para as organizações

Sistemas C (Casual)

- Sistemas pequenos, comuns e sem muita importância
- Não precisam ser perfeitos, podem ter bugs; às vezes, são descartáveis
- Desenvolvidos por 1-2 engenheiros, são pouco complexos
- Risco: "over-engineering"

Processos

- Definem quais atividades devem ser seguidas para construir um sistema
- Qual a importância? *A definição de um processo de software em equipes que estão desenvolvendo projetos complexos é de extrema importância pois é através dele que os membros serão capazes de se organizar, seguindo passos e realizando tarefas de maneira ordenada para atingir seu objetivo. Sem um processo estabelecido para coordenar as atividades da equipe, o trabalho pode muito facilmente se tornar caótico e com isso prejudicar os interesses de todas as partes interessadas na conclusão do sistema.*

Escopo fechado

- Cliente define requisitos ("fecha escopo")
- Empresa desenvolvedora: preço + prazo

Escopo aberto

- Escopo definido a cada iteração
- Pagamento por homem/hora
- Contrato renovado a cada iteração
- Exige maturidade e acompanhamento do cliente
- **Vantagens:**
    - Privilegia qualidade
    - Não vai ser "enganado" ("entregar por entregar")
    - Pode mudar de fornecedor

Modelo Waterfall ("Cascata")

- “Big Design Up Front” ou BDUF
- Inspirado em processos usados em engenharias tradicionais, como Civil, Mecânica, Elétrica, etc
- 5 fases sequenciais, uma tem de estar completa antes de passar para a próxima
    1. Análise e definição de requisitos
    2. Projeto de sistema e software
    3. Implementação e teste de unidade
    4. Integração e teste de sistema
    5. Operação e manutenção
- Primeiro modelo a organizar as atividades de desenvolvimento
- Todas as fases envolvem atividades de validação
- **Com o que cascata funciona bem?** Quando as especificações não mudam, é um sistema sólido e bem definido
- **Problemas do modelo cascata:**
    - Particionamento inflexível do projeto em estágios
    - Dificulta a resposta aos requisitos de mudança do cliente, levantamento inicial demanda tempo
        - Clientes às vezes não sabem o que querem
    - Documentos “completamente elaborados” são necessários para fazer as transições entre estágios
        - Documentação é demorada e custosa
        - E rapidamente se torna obsoletas
    - Apropriado somente quando os requisitos são bem compreendidos e quando as mudanças são raras
    - Poucos sistemas de negócio têm requisitos estáveis
        - Quando ficar pronto, o "mundo já mudou"

Ciclo de Vida Espiral

- Transição do tradicional para o ágil
    - *Os métodos de Processo Unificado e Espiral não são considerados ágeis pois surgiram em um período de transição entre eles e o antigo modelo Waterfall, apresentando características de ambos porém com diferenças o suficiente para que não se encaixem em nenhuma das duas definições. Apesar de fazer uso das iterações incrementais, os ciclos desenvolvidos nesses métodos são de duração bem maior do que aqueles feitos nos modelos ágeis. Além disso, eles também se diferenciam do Waterfall por não serem estritamente sequenciais.*
- Use protótipos para obter feedback do cliente até a versão "final" criada
    - Cada "iteração" oferece um novo protótipo
- Sobrepõe os estágios de planejamento e execução de várias iterações
- **Pontos positivos:**
    - Iterações envolvem o cliente antes do produto estar completo
        - Reduz as chances de mal entendidos
    - Gerenciamento de Riscos no ciclo de vida
    - Monitoramento do projeto facilitado

    - Cronograma e custo mais realista através do tempo
- **Pontos negativos:**
    - Iterações longas de 6 a 24 meses
    - Tempo para os clientes mudarem de ideia
    - Muita documentação por iteração
    - Muitas regras para seguir, pesado para todo projeto
    - Custo alto do processo
    - Complicado de atingir metas de investimento e marcos no cronograma

Ágil (Incremental ou iterativo)

- Crítica a modelos sequenciais e pesados

    "Individuals and interactions over processes & tools
    Working software over comprehensive documentation
    Customer collaboration over contract negotiation
    Responding to change over following a plan"
- Aceita as mudanças como um fato da vida: melhoria contínua
- Os desenvolvedores refinam continuamente o protótipo em funcionamento, mas incompleto, até que os clientes fiquem satisfeitos, com feedback do cliente a cada iterações (1-2 semanas)
- Todos os elementos do ciclo de vida em todas as iterações
- Enfatiza o **Test-Driven Development (TDD)** para reduzir erros, especificando **User Stories** para validar os requisitos do cliente, **Velocity** para medir o progresso
    - Velocity: Número de story points que o time consegue implementar em um sprint

XP ("Extreme Programming")

- Iterações mais curtas
- Produção de coisas mais simples
- Mais testes, a todo momento
- Revisão de código continua, com programação em pares

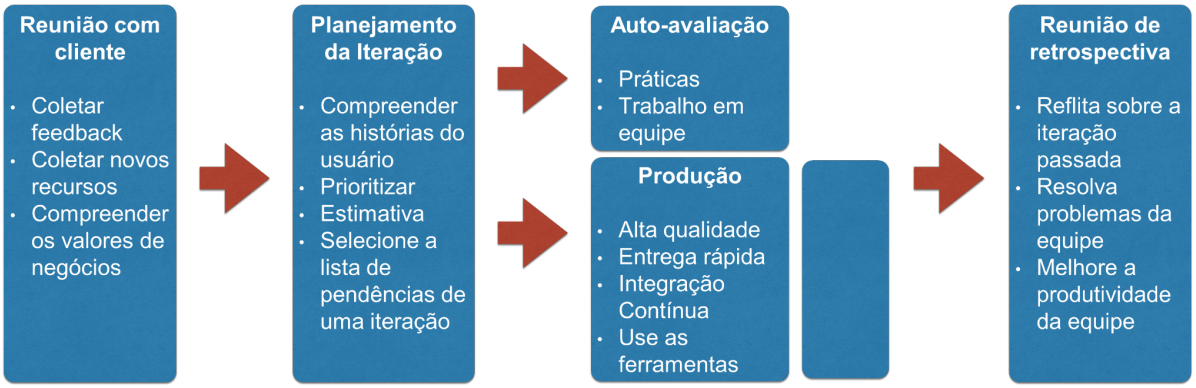
SCRUM

- Não é apenas para projetos de software, então não define práticas de programação
- Processo mais rígido que XP
- Eventos, papéis e artefatos bem claros
- Desenvolvimento é dividido em sprints (iterações)
- Duração de um sprint: até 1 mês, normalmente 15 dias
- **O que se faz em uma sprint?**
    - Implementa-se algumas histórias dos usuários
    - Histórias = funcionalidades (ou features) do sistema
- **Histórias de usuário**
    - Escritas pelo Product Owner (PO)
        - Papel obrigatório em times Scrum
        - Especialista no domínio do sistema
        - Escreve histórias dos usuários
        - Explica histórias para os devs, durante o sprint

        - Define "testes de aceitação" de histórias
        - Prioriza histórias
        - Mantém o backlog do produto
    - Backlog do Produto = Lista de histórias do usuário
        - Priorizada: histórias do topo têm maior prioridade
        - Dinâmica: histórias podem sair e entrar com a evolução do sistema
- **Quais histórias vão entrar no próximo sprint?**
    - Essa decisão é tomada no início do sprint, na reunião de planejamento
    - PO propõe histórias que gostaria de ver implementadas
    - Devs decidem se têm velocidade para implementá-las

Resumindo:

- Elementos principais
    - Sprint (evento)
    - PO e Devs (papeis)
    - Backlog do produto (artefato)
- Em um time scrum, todos têm o mesmo nível hierárquico
    - PO não é o "chefe" dos Devs
    - Devs têm autonomia para dizer que não vão conseguir implementar tudo que o PO quer em um único sprint

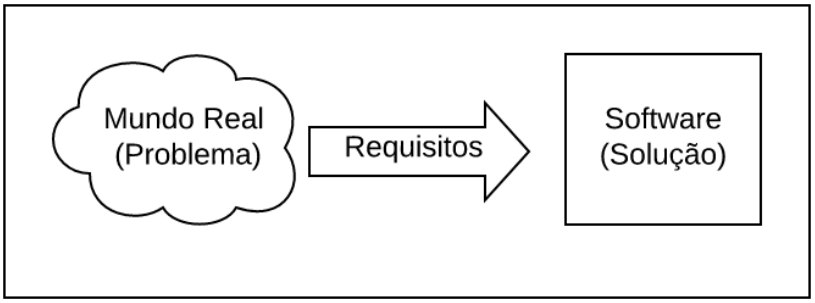


Controle de riscos

- Waterfall: Longas fases de planejamento e modelagem do sistema que tentam tratar todos os possíveis casos conflituosos do sistema antes que este seja definitivamente implementado. A existência de muita documentação garante que tudo que é feito é devidamente registrado.
- Scrum: Por fazer uso dos sprints em ciclos flexíveis, ele é capaz de gerenciar os riscos e erros no projeto através da correção destes em iterações futuras do processo que se beneficiaram do feedback que é um ingrediente essencial desse modelo, além das interações constantes com o cliente, o que garante que o projeto não está se distanciando do que é requisitado por ele. Identificam-se erros cedo pois eles são enfrentados logo na implementação durante o sprint, e isso garante um fluxo saudável das operações como um todo.
- Kanban: Garante um fluxo sustentável de trabalho através do uso de um quadro que permite uma visualização rápida e clara das tarefas que devem ser desempenhadas pela equipe.

Requisitos

- "A parte mais difícil da construção de um software é a definição do que se deve construir".
- Tradução de problemas reais em soluções em Software



Requisitos funcionais

- O que sistema deve fazer
- *Para o iFood: Cadastrar cartão de crédito, visualizar restaurantes, pesquisar comida, pedir comida, agendar entrega*

Requisitos não funcionais

- Restrições que o sistema precisa obedecer
- *Para o iFood: Precisa ficar disponível de 5 da manhã até 3 da manhã, tem que ser rápido que ligar para o restaurante (3 min), visibilidade clara se o restaurante está aberto, aceitar várias formas de pagamento, confiança na compra e em seus dados*

História de usuário

- São descrições de pedaços de funcionalidades do sistema.
- Inclui: usuário específico, problema a ser resolvido, solução, critério de aceite, critério de sucesso
- O formato mais comum é Como [usuário] Eu quero [problema] Então [solução]
- Os critérios de aceite são parâmetros para conferir se aquela história foi entregue, servem para os membros do time confirmarem sua realização
    - Para isso, geralmente são descritos casos de uso, que identificam possíveis cenários de teste
- **Características de boas histórias (INVEST)**
    - Independentes, Abertas para Negociação, Agregar Valor, Estimáveis, Sucintas, Testáveis
    - Acrônimo que descreve características importantes de boas US

Casos de Uso

- São uma espécie de verificação para saber se o sistema está tendo um comportamento esperado dada situação. Muitas vezes é utilizado dentro dos critérios de aceite.
- Documento mais detalhado de especificação de requisitos
- Um ator realizando alguma operação com o sistema

- Incluem fluxo normal e extensões
- Extensões: Exceções (ou erros) e detalhamento

MVP

- Produto = já pode ser usado
- Mínimo = menor conjunto de funcionalidades (menor custo)
- Viável = objetivo é obter dados e validar uma ideia
- Ciclo Construir-Medir-Aprender
    - Realizar alguns ajustes e rodar ciclo de novo
- Pivotar: mudar foco do produto (ex.: aluguel de músicas)
    - Desistir (dinheiro acabou!)
    - Deu certo! Vamos construir um produto mais robusto
- Se a ideia der certo, pode-se depois reimplementar o sistema
- Desempenho, usabilidade, estabilidade, etc

Testes A/B

- Existe o cenário onde duas implementações de requisitos "competem" entre si
- Versão original x Nova versão, proposta por alguns devs
    - Vale a pena migrar para nova versão? No teste A/B, os dados decidem
- No final do experimento analisam-se as métricas
- Requerem amostras grandes
- **O que é um experimento?** É uma maneira de testar se uma hipótese atende os resultados esperados. Podemos utilizar um teste A/B como experimento.
- Alguns outros tipos de experimentos:
    - Landing Pages
    - Teste de Canal
    - Fake Door
    - Concierge
    - Mágico de Oz
- Como o Teste A/B funciona, quando devemos aplicar e quando não devemos aplicar Testes A/B? Cite exemplos. *Nos testes A/B os desenvolvedores implementam duas versões idênticas do sistema, diferindo apenas em um requisito específico para cada versão, que são necessariamente distintos. Normalmente, a versão A é a 'normal', do sistema como era antes; e a versão B inclui alguma funcionalidade nova que está sendo testada com o público. Essas versões são distribuídas para os usuários e analisa-se qual delas despertou mais interesse e foi mais bem sucedida. Esse tipo de teste é útil quando queremos verificar a reação do público a uma atualização que pode ser considerada disruptiva. Dito isso, testes A/B não são recomendados quando não se tem um público grande o suficiente para que seja extraída uma base de dados significativa, ou quando a mudança no sistema não é marcante o suficiente para não gerar mudança notável no comportamento dos usuários.*

Arquitetura de Software

Qualidades para avaliação de uma arquitetura

- Performance - latency time
- Reliability – system operating over the time
- Availability – system is up and running
- Security – resistance to attacks
- Modifiability - changes
- Portability – run under different environments
- Functionality – do the work that is expected
- Variability - reuse
- Conceptual integrity
- Quais são as principais vantagens, na sua opinião, da Arquitetura MVC? *A arquitetura MVC define que as classes devem ser organizadas em 3 grupos (Visão, Controladoras e Modelo), também conhecida como Model-View-Controller. É uma arquitetura que permite a flexibilidade e a especialização do trabalho, pois podemos ter por exemplo uma pessoa trabalhando na interface (front) e uma pessoa trabalhando com modelos no fundo, o qual não precisaria se preocupar com a interface. Outra vantagem é a testabilidade, pois é mais fácil testar ao separar objetos de apresentação dos objetos de Modelo, já que não estão relacionadas com a interface gráfica.*
- Como você vê a relação entre a Lei de Conway e microsserviços? *As empresas normalmente adotam arquiteturas de software parecidas com suas estruturas organizacionais, logo, a arquitetura dos sistemas de uma empresa vão se espalhar por todo o organograma e influenciar as áreas. Dessa forma, os microsserviços normalmente acabam por ser usados por grandes empresas de internet a qual possuem centenas de times de desenvolvimento distribuídos em diversos países.*
- O que significa desacoplamento no espaço e desacoplamento no tempo? Cite exemplos. *No desacoplamento no espaço, não se faz necessário que os clientes tenham conhecimento dos servidores, e vice-versa. Basicamente, o cliente apenas produz informações, e não precisa saber quem irá fazer usa destas; por outro lado, o servidor apenas consome informações, sem saber quem às produz. Ele trás bastante flexibilidade às soluções baseados em filas de mensagens, já que os times de desenvolvimento podem trabalhar em paralelo, sem afetar o desenvolvimento uns dos outros. Já no desacoplamento no tempo, os clientes e servidores não precisam estar simultaneamente disponíves para que ocorra comunicação entre eles. Em caso de falha do servidor, os clientes continuarão produzindo informações, deixando-as em espera até o servidor voltar a funcionar. Sua principal vantagem é que torna a solução robusta contra falhas, pois quedas no serviço não afetam diretamente o cliente.*
- Quando uma empresa deve considerar o uso de uma arquitetura baseada em filas de mensagens ou uma arquitetura publish/subscribe? *É vantajoso para uma empresa optar por uma arquitetura baseada em filas de mensagens quando sua comunicação é feita de maneira assíncrona e ponto-a-ponto. Ela também protege a integridade dos dados, permitindo a escalabilidade horizontal e melhorando a resiliência geral do sistema. Já no caso da arquitetura publish/subscribe, ela é mais utilizada para situações que é necessário a implementação de uma comunicação distribuída em vários pontos. Nessa arquitetura, os produtores publicam mensagens ("eventos") em*

um tópico específico, e os consumidores se inscrevem nos tópicos relevantes para receber as mensagens.

SOLID

- Cinco princípios de design de software que visam a criação de código flexível, fácil de manter e extensível. Isso pode ajudar a reduzir os custos de desenvolvimento e tornar o software mais robusto e confiável.

Single Responsibility

- Cada classe deve ter apenas uma única responsabilidade, isto é, ela deve ter apenas um motivo para mudar. Isso significa que a classe deve ter apenas uma única função ou tarefa a ser executada.

Open/Closed

- Uma classe deve estar aberta para extensão, mas fechada para modificação. Isso significa que as mudanças devem ser feitas adicionando novas funcionalidades sem alterar o código existente.

Liskov Substitution

- As classes derivadas devem ser substituíveis por suas classes base. Isso significa que uma classe derivada deve ser capaz de ser usada no lugar da classe base sem quebrar o código.

Interface Segregation

- As interfaces devem ser segregadas, isto é, elas devem ter apenas os métodos necessários para cumprir suas responsabilidades. Isso significa que as interfaces não devem ser sobrecarregadas com muitos métodos, mas sim ter apenas os métodos relevantes para sua tarefa.

Dependency Inversion

- Os módulos de alto nível não devem depender dos módulos de baixo nível, mas ambos devem depender de abstrações. Isso significa que as dependências devem ser abstratas e não específicas, o que permite que o código seja mais flexível e fácil de manter.

Testes

- Verificam se um programa apresenta um resultado esperado, ao ser executado com alguns casos de teste
- Podem ser manuais ou automatizados

Motivação

- Ocorrência de falhas humanas no processo de desenvolvimento de software
- 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

Depuração: "método científico"

- O que você esperava que acontecesse vs O que realmente aconteceu
- Extrair informações das mensagens de erro
    - De onde veio a mensagem?

    - Se não houver mensagem, identificar o último ponto em que as coisas estavam da maneira que você espera
    - Qual foi a última coisa relacionada que você tentou que funcionou conforme o esperado? O que mudou?
- Forme uma hipótese que possa explicar a discrepância
- Crie uma maneira de testar a hipótese

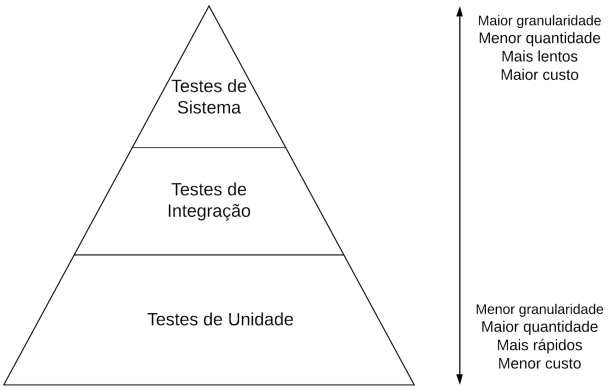
Testes antigamente
- Desenvolvedores finalizam o código, alguns testes ad-hoc
- "Jogar pro alto a Garantia de Qualidade [QA]"
- Equipe de QA manualmente exercita o software

Hoje em dia
- 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

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

Verificação vs Validação
- Verificação: estamos implementando o sistema *corretamente*?
    - Avaliação para determinar se os produtos de uma dada fase do ciclo de desenvolvimento satisfazem as condições impostas nela
    - De acordo com os requisitos e especificações
- Validação: estamos implementando o sistema *correto*?
    - Avaliação durante, ou ao final do ciclo de desenvolvimento de software para determinar se satisfaz aos requisitos especificados
    - Aquele que os clientes querem
    - Testes de aceitação com os usuários

Níveis de Teste
- 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



Testes com Métodos Ágeis

- Automatizados
    - Algumas vezes, implementados antes do código (TDD)
    - Escritos pelo próprio desenvolvedor do código sob testes
- Outras funções:
    - Detectar regressões
    - Documentação

Noção de confiabilidade

- Algumas faltas escaparão tanto dos testes quanto da depuração
- Perturbação depende do que se trate e em qual frequência irá surgir para o usuário

Confiabilidade de software

- Estimativa probabilística que mede a frequência com que um software irá executar sem falha em dado ambiente e por determinado período de tempo

Políticas de teste

- Somente testes exaustivos podem mostrar que um programa está livre de defeitos, mas eles são impossíveis na prática
- 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

Cobertura de Testes

- Cobertura de testes = (número de comandos executados pelos testes) / (total de comandos do programa)
- A cobertura ideal varia de projeto para projeto, em geral ~60%

BDD+TDD: Visão Geral

Behavior-driven design (BDD)

- Desenvolver histórias do usuário para descrever como a aplicação deve funcionar
- Ideia: HUs se tornam testes de aceitação e integração

Test-driven development (TDD)

- Prática ágil para garantir que a qualidade é construída desde o início
- Definir uma nova HU pode requerer a escrita de novo código
- TDD diz: escreva testes unitários e funcionais primeiro, antes do código propriamente dito
- Etapas: escrever um teste automatizado que descreva o comportamento esperado do sistema; executar o teste e verificar se ele falha (já que ainda não há implementação); e, por fim, escrever o código de produção que faça o teste passar.
- Se baseia em dois conceitos principais: Testes Unitários & Funcionais e Refactoring
    - *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*
- Evita que os devs esqueçam de escrever os testes
- Incentiva a escrita de código com testabilidade; cobertura pode chegar a 90%
- Processo de desenvolvimento mais simples e produtivo
- Melhora o design e/ou usabilidade do código
- Melhoria na comunicação
- Melhoria no entendimento dos comportamentos dos Requisitos do Software
- Ajuda como documentação
- Facilita refactorings

TDD vs debugging

- *Resumidamente, o Test-Driven Development envolve a criação de testes antes de escrever o código, enquanto o debugging é a abordagem tradicional de encontrar e corrigir erros durante a execução do código. O TDD tem a vantagem de detectar erros mais rapidamente e aumentar a confiabilidade do código a longo prazo, enquanto o debugging é mais demorado e depende da habilidade do desenvolvedor para encontrar as falhas, mas é capaz de detectar erros que não foram previstos pela equipe. Embora a automatização de testes possa ajudar a identificar e corrigir erros, ela não elimina completamente a necessidade de debugar o código. Mesmo com testes automatizados, ainda é possível que ocorram erros que não foram antecipados, que só serão encontrados através de um debugging minucioso.*

A propriedade FIRST

- Fast: executar (um conjunto de) testes rapidamente
- Independent: Um teste não depende de outro, 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 o quanto antes (com TDD, escreva-o primeiro)

Testes de Integração

- Testam uma funcionalidade ou serviço (do seu sistema)
- Incluindo classes e serviços externos (BD, por exemplo)

Testes de Sistema

- Testes ponta-a-ponta (end-to-end)
- Testam o sistema inteiro; via sua interface externa

Testes automatizados

- Testes unitários: um único trecho de código (geralmente um objeto ou uma função) é testado, isolado de outras partes
- Testes de integração: vários trechos são testados juntos, por exemplo, testando o código de acesso ao banco de dados em um banco de dados de teste
- Testes de aceitação: teste automático de todo o aplicativo, por exemplo, usando uma ferramenta como o Selenium para executar automaticamente um navegador

Outros Tipos de Testes

- Caixa-preta (funcionais)
- Caixa-branca (estruturais)
- Aceitação (manual)
- Alfa e beta (manual)
- Requisitos não-funcionais

Teste fim a fim

- As aplicações são criadas em camadas e subsistemas, com camadas de UI e API, bancos de dados externos, redes e até integrações de terceiros
- Quando um falha, o produto inteiro também falha, tornando a estabilidade de cada componente vital para o sucesso de um aplicativo
- O teste fim a fim é uma metodologia usada para testar a funcionalidade e o desempenho de uma aplicação em circunstâncias e dados semelhantes a produção
- Simula a aparência de um cenário real do usuário do início ao fim
- A conclusão deste teste não é apenas para validar o sistema em teste, mas também para garantir que seus subsistemas funcionem e se comportem conforme o esperado.

Benefícios

- Ajuda a garantir que o software esteja pronto para produção e evita riscos após a liberação
- Integridade: Validam se o software é funcional em todos os níveis - do front ao back-end -, além de fornecer uma perspectiva do desempenho em diferentes ambientes
- Expande a cobertura de teste: ao incorporar os diversos subsistemas diferentes em seu processo de teste, você expandirá efetivamente sua cobertura de teste e criará casos de teste adicionais que podem não ter sido considerados anteriormente

- Detecta bugs e aumenta a produtividade: nos testes fim a fim, o software geralmente é testado após cada iteração, o que significa que você poderá encontrar e corrigir quaisquer problemas mais rapidamente
- Reduz os esforços e os custos dos testes: com menos bugs, falhas e com testes abrangentes a cada passo, os testes fim a fim também diminuirão sua necessidade de repetir testes e, finalmente, os custos e o tempo associados a isso

Teste fim a fim	Teste de Sistema
Valida o sistema de software e os subsistemas interconectados	Valida apenas o sistema de software de acordo com as especificações de requisitos
A ênfase principal está na verificação do fluxo completo do processo de ponta a ponta.	Ele verifica as funcionalidades e recursos do sistema.
Durante a execução do teste, todas as interfaces, incluindo os processos de back-end do sistema de software, são levadas em consideração	Somente testes funcionais e não funcionais serão considerados para teste.
É executado assim que o teste do sistema é concluído	É executado após o teste de integração
O teste de ponta a ponta envolve a verificação de interfaces externas que podem ser complexas para automatizar. Portanto, o teste manual é preferido.	Manual e automação podem ser executados para teste do sistema.

Test Smells

- Como os Test Smells podem impactar a qualidade do código e quais técnicas ou políticas de Teste de Software podemos aditar para mitigar os problemas causados? *Test Smells são problemas de qualidade e na manutenção do software que ocorrem nos testes de software e representam estruturas e características que devem ser evitadas. Eles podem causar impactos negativos na eficácia, eficiência e confiabilidade do teste, como ineficiência ou fragilidade. Para evitar esses problemas, podemos adotar técnicas e políticas como refatoração, automação de testes, cobertura e revisão de código, Test-Driven Development (TDD), entre outros.*

Design Thinking

Design de Experiência

- O Design de Experiência, ou User Experience Design (UXD), é um conjunto de métodos para entender o comportamento do usuário durante toda a jornada de consumo
- É também uma estratégia do marketing de experiência que visa o encantamento de clientes, a fidelização do consumidor e, consequentemente, o aumento das vendas e do lucro do negócio
- Não basta oferecer soluções que o cliente necessita, mas sim entender como ele pensa, o que faz, o que o agrada e satisfaz, com quem convive, qual é seu estilo de vida, etc.
- Ao compreender o universo particular do cliente a empresa é capaz de fornecer experiências fantásticas que tocam níveis profundos do usuário, gerando com isso maior confiança e fidelização para com a marca
- 10 Princípios para uma Grande Experiência do Consumidor
    1. Reflete fortemente a identidade do consumidor

2. Satisfaz seus maiores objetivos
3. É planejada nos mínimos detalhes
4. Cria e atende às expectativas
5. Acontece sem esforço
6. É livre de estresse
7. Provoca os nossos sentidos
8. Mobiliza sociabilidades
9. O coloca no controle
10. Envolve nossas emoções

- UI vs UX: *O design de experiência do usuário tenta entender o comportamento do usuário durante sua jornada com o produto ou serviço, focando em proporcionar sentimentos bons como poder, exclusividade, luxo, bem-estar, etc. Já o de interface de usuário muda a aparência do site ou aplicação para facilitar a usabilidade, de maneira a se tornar mais intuitivo.*

A Jornada do Usuário

- Mapear é importante para definir quais esforços devem ser empregados em sua retenção, pelo maior tempo possível
- No processo de Design Thinking, a Jornada do Usuário é especialmente importante para compreender como este usuário se relaciona com o produto ou serviço

Design Thinking

- Processo estruturado e sistemático para geração de ideias
- Centrada no usuário, a abordagem tem como objetivo estabelecer soluções para problemas reais do usuário
- <u>As fases do Design Thinking são: Imersão, Análise, Ideação e Prototipagem</u>

Fase de Imersão

- a.k.a. Identificação dos requisitos
- São levantadas as necessidades e desafios dos usuários, através de observações, pesquisas e entrevistas - qualitativas e quantitativas
- É na imersão que a equipe se aproxima do contexto do projeto e começa a identificar e classificar os problemas a serem resolvidos na visão do usuário, na visão do sistema em que está inserido, e na visão do negócio

Fase de Análise

- Sintetizamos as informações coletadas para a geração de insights
- Posteriormente, esses insights são organizados para identificarmos padrões, possibilitando a compreensão do problema em sua essência.
- Entender com clareza em qual cenário os stakeholders estão envolvidos

Fase de Ideação

- A equipe se reúne para dar vazão ao processo criativo e apresentar sugestões para o problema, já identificado e esmiuçado
- Criação de soluções alternativas - momento de definir hipóteses de soluções para as necessidades levantadas
- Variedade de perfis de pessoas deve estar envolvida, incluindo também quem será beneficiado com as soluções propostas

Fase de Prototipagem

- Prototipar é tangibilizar uma ideia, a passagem do abstrato para o físico de forma a representar a realidade – mesmo que simplificada – e proporcionar validações
- É a fase de validação das ideias geradas. É a hora de aparar as arestas, ver o que se encaixa no projeto, juntar propostas e colocar a mão na massa
- Com o protótipo em mãos, é possível testar o produto junto ao usuário, refinando e melhorando até que se transforme em uma solução que realmente esteja alinhada às suas necessidades e possa gerar lucro
- Apesar de ser apresentada como fase final, a prototipação pode acontecer em paralelo às outras fases.
- Conforme as ideias forem surgindo elas podem ser prototipadas, testadas e, em vários casos, até implementadas.
- Mais informações: Prototipagem: o guia definitivo para colocar sua ideia na rua