Introdução e conceitos:

1. Qualidades em projeto de software:
    a. Estimativas -> métricas
    b. Reuso
    c. Padrões, padrões arquiteturais e frameworks, design patterns
    d. Testes
    e. Integração contínua
2. Projeto Estruturado X Projeto Orientado a Objetos

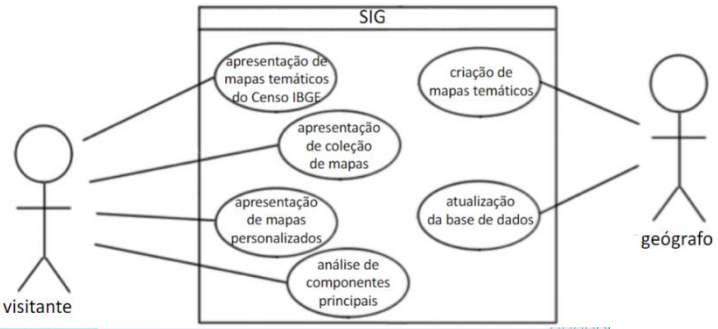
    Estruturado - 1971:

    - Top- down
    - Modular
    - Módulo caixa preta
    - Refinamento sucessivo

    OO - 1986:

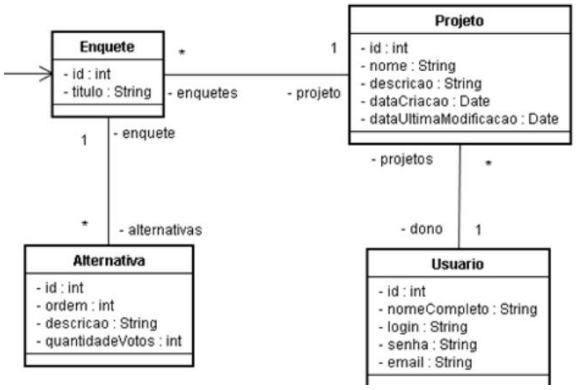
    - Abstração, a modularização e a reutilização de código
3. Projeto preliminar: Caso de Uso , Interace e Robustez
4. Projeto detalhado: Interação de objetos(Sequencia e colaboração) Pacotes e componentes e execução

Casos de uso:



Identificador	CSU01
Lista de atores	Visitante
Descrição	Apresentação de mapas personalizados
Pré-condições	Página de mapas personalizados ativa
Fluxo básico de eventos	1 – Usuário seleciona a variável (cursos, matrículas, ingressos, matrículas por mil) 2 – Sistema atualiza o mapa e estatísticas 3 – Usuário seleciona a área de conhecimento (total, grande área, área, subárea) 4 – Sistema atualiza o mapa e estatísticas 5 – Usuário seleciona o grau dos cursos (total, bacharelado, licenciatura, tecnólogo) 6 – Sistema atualiza o mapa e estatísticas 7 – Usuário seleciona a modalidade (total, presencial, a distância) 8 – Sistema atualiza o mapa e estatísticas
Fluxo alternativo	
Fluxo de exceção	
Pós-condições	Dados apresentados no mapa de acordo com as seleções nos campos

Classe:



Robustez:

Consiste na produção incremental e em paralelo de um conjunto de artefatos que retratam as visões dinâmica e estática de um sistema, privilegiando a “rastreabilidade” e a robustez.

Fazer a análise de robustez:

Criar Diagramas de Robustez usando os estereótipos de classes boundary, control, e entity – Atualizar o modelo do domínio, com os novos Objetos e atributos descobertos:

Regras:

- Os actores podem comunicar com o sistema através de Objetos fronteira.
- Os Objetos fronteira comunicam apenas com actores e Objetos de controle.
- Os Objetos entidade comunicam apenas com Objetos de controle.
- Os Objetos de controle comunicam apenas com Objetos de fronteira e de entidade

Exemplo:

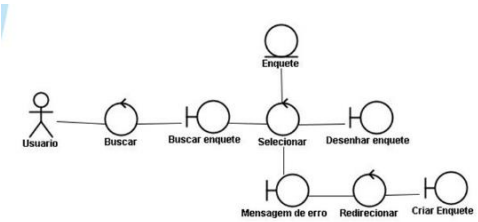
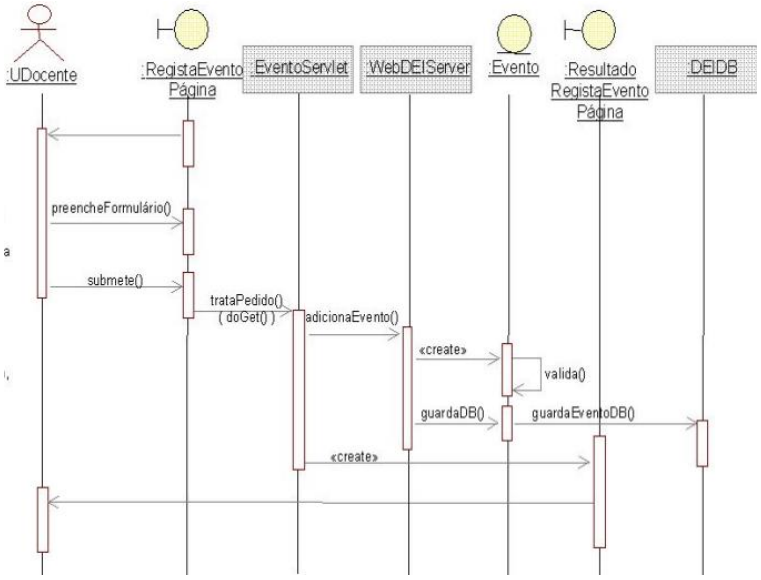
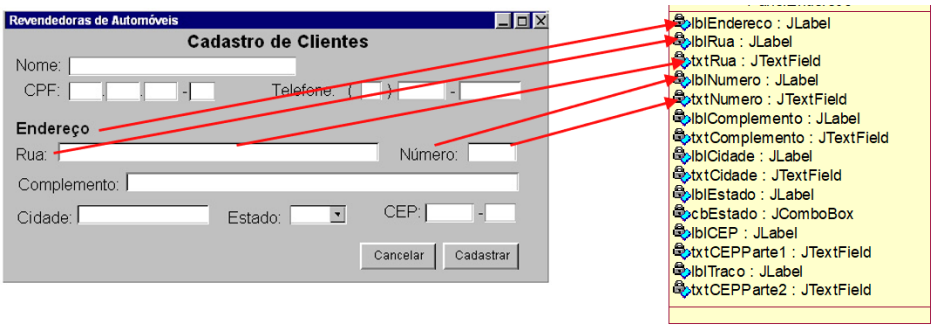


Diagram : Sequencia:



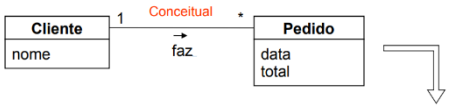
Projeto de Interface – Projeto detalhado(inclui tudo até aqui)

- Interface é uma área cinza com diversas possibilidades de comportamento
  - **Forma** que possibilita informação
  - **Estrutura** que possibilita interação
  - **Função** possibilita a experiencia
- Design:
  - Conformidade
  - Articulação
  - Referência
- Padrões de design → organização e estrutura
- Regras:
  - Cada tela é considerada uma classe
  - Criar um relacionamento de dependência entre a classe da tela e as classes de negócio que possuem os valores a serem exibidos
  - Agrupamentos podem ser considerados como uma classe
  - Analise as ações que o usuário pode fazer com a janela
  - Verifique quais deles precisam de especificar a ação
- Exemplo:

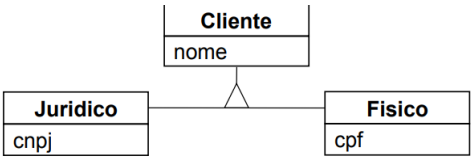


Projeto de Persistência:

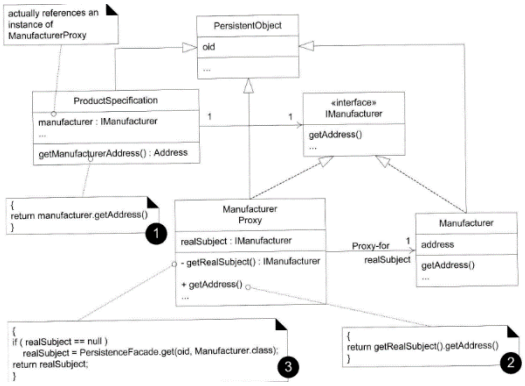
- Projeto do banco de dados:
    - Mapeamento OO/Relacional (MOR) ou Object-relational mapping(ORM)
    - Linguagem de Consulta
    - API de acesso aos dados
- Mapeamento Objeto Relacional(MOR):
- Regras:
    - Associações – losango no meio :



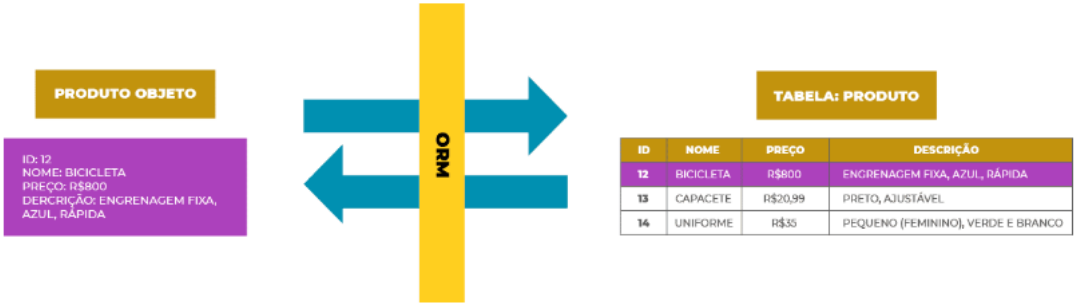
    - Herança:



- Exemplo:



- Mapeamento ORM:



    - Hibernate para java
    - Django para Python
    - Sequelize para JS
- Projeto Arquitetural (Arquitetura Lógica X Arquitetura Física)
- Arquitetura Lógica: especifica as propriedades funcionais do
- sistema (objetos de negócio)
- - Arquitetura Física: aborda os aspectos não funcionais do sistema, como:
    - Segurança
    - Compatibilidade (Portabilidade)

    - Ambiente de execução e acesso a recursos
- Física:
    - Visão de Componentes - Diagramas de Componentes
    - Visão de Concorrência - Diagramas de Implementação (Diagramas de Componentes; Diagramas de Execução)

Diagrama de Componentes:

Componentes de Código-fonte

Arquivos contendo código-fonte que implementam uma ou mais classes do sistema

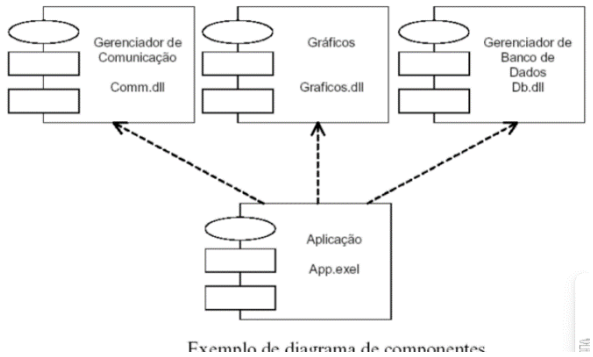
Componentes Binários

- Arquivos com código objeto, resultante da compilação de um ou mais componentes de código-fonte

Componentes Executáveis

- Arquivos de programa executável resultante da ligação dos componentes binários
- Representam unidades de software que podem ser executadas por um computador

Exemplo:



Exemplo de diagrama de componentes

Diagramas de Execução:

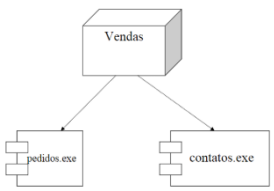
É composto por:

• Componentes de software

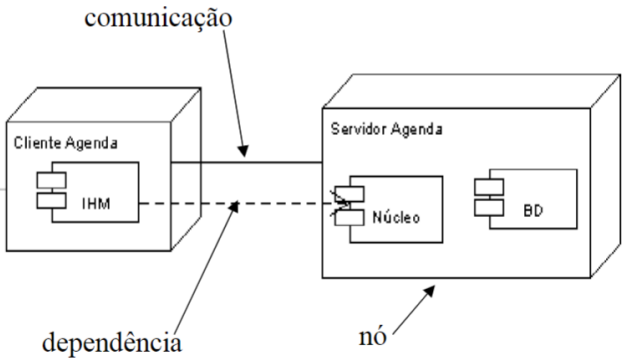
• Nós

• Conexões

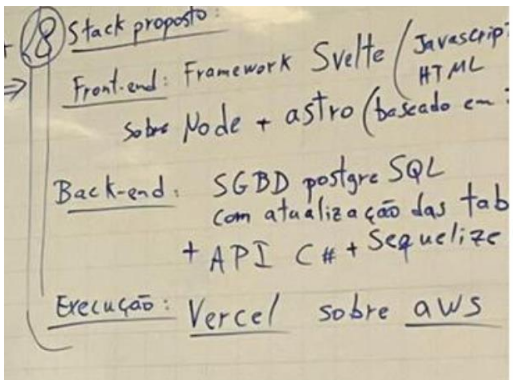
Nós e componentes:



Exemplo:



Stack proposto:



Prova II

Estimativa → Métricas:

- O uso de métricas de software torna-se essencial para medir (tamanho, custo, prazo) e, consequentemente gerenciar melhor o desenvolvimento do software.
- Por isso a importância de se estimar bem para a qualidade de um projeto de software.
- Pontos por Caso de Uso:
  - Trata de estimar o tamanho de um sistema de acordo com:
    - o modo como os usuários o utilizarão;
    - a complexidade de ações requerida por cada tipo de usuário;
    - uma análise em alto nível dos passos necessários para a realização de cada tarefa;

- Calculo UAW:

Tipo de Ator	Peso	Nº de atores	Resultado
Ator Simples	1	0	0
Ator Médio	2	0	0
Ator Complexo	3	4	12
		Total UAW	**12**

- Calculo UUCW:

  - **Simples (peso 5):** Tem até 3 transações, incluindo os passos alternativos, e envolve menos de 5 entidades;
  - **Médio (peso 10):** Tem de 4 a 7 transações, incluindo os passos alternativos, e envolve de 5 a 10 entidades;
  - **Complexo (peso 15):** Tem acima de 7 transações, incluindo os passos alternativos, e envolve pelo menos de 10 entidades;

Tipo	Peso	Nº de Casos de Uso	Resultado
Simples	5	7	35
Médio	10	13	130
Complexo	15	3	45
		Total UUCW	**210**

- Calculo UUCP = UAW + UUCW
- Calculo TFACTOR:

- **Passo 4:** Cálculo do Tfactor
  - Para cada requisito listado na tabela, deve ser atribuído um valor que determina a influência do requisito no sistema, variando entre 0 e 5;

Fator	Requisito	Peso	Influência	Resultado
T1	Sistema distribuído	2	1	2
T2	Tempo de resposta	2	3	6
T3	Eficiência	1	3	3
T4	Processamento complexo	1	3	3
T5	Código reusável	1	0	0
T6	Facilidade de instalação	0.5	0	0
T7	Facilidade de uso	0.5	5	2.5
T8	Portabilidade	2	0	0
T9	Facilidade de mudança	1	3	3
T10	Concorrência	1	0	0
T11	Recursos de segurança	1	0	0
T12	Acessível por terceiros	1	0	0
T13	Requer treinamento especial	1	0	0
			Tfactor	**19,5**

- Cálculo do TCF :

$$TCF = 0.6 + (0.01 \times Tfactor)$$

  - No caso do exemplo:

$$TCF = 0.6 + (0.01 \times 19.5) = 0.795$$

-

- Calculo EFactor:

Passo 6: Cálculo do Efactor

- Para cada requisito listado na tabela, deve ser atribuído um valor que determina a influência do requisito no sistema, variando entre 0 e 5;

Fator	Descrição	Peso	Influência	Resultado
E1	Familiaridade com RUP ou outro processo formal	1.5	5	7.5
E2	Experiência com a aplicação em desenvolvimento	0.5	0	0
E3	Experiência em Orientação a Objetos	1	5	5
E4	Presença de analista experiente	0.5	5	2.5
E5	Motivação	1	5	5
E6	Requisitos estáveis	2	3	6
E7	Desenvolvedores em meio-expediente	-1	0	0
E8	Linguagem de programação difícil	-1	0	0
			Efactor	**26**

- Passo 7: Cálculo do ECF (Environmental Complexity Factor)
- ECF = 1.4 + (-0.03 Efactor)
- No caso do exemplo:
- ECF = 1.4 + (-0.03 26) = 0.62

Passo 8: Cálculo dos UCP (Use Case Points)

UCP = UUCP TCF ECF

Passo 9: Cálculo do tempo de trabalho estimado

Para simplificar, utilizaremos a média de 20 horas

por Ponto de Casos de Uso

No caso do exemplo:

Tempo estimado = 109 * 20 = 2180 horas de

Trabalho

Estimativas Ageis:

Conceitos Gerais

- **Estimativas são projeções imprecisas:** Variabilidade alta nas fases iniciais dos projetos.
- **Cone da incerteza:** Representa a variabilidade das estimativas ao longo do tempo.

Técnicas de Estimativas Ágeis

1. **Técnicas de Contagem:**
    - Contar diretamente em ambientes similares e fazer analogias.
    - Quando não possível, usar cálculos baseados em dimensões.

- Em última instância, usar julgamentos de especialistas.

2. **Unidades Relativas:**
    - Utilizar pontos em vez de dias para facilitar comparações e evitar previsões imprecisas.

Processos e Ferramentas

1. **SCRUM:** Visão geral do processo SCRUM é apresentada.
2. **T-Shirt Sizes:**
    - Técnica de estimativa em alto nível usando tamanhos (PP, P, M, G, GG).
    - Exemplo de previsibilidade de release: 5 estórias com diferentes tamanhos e tempos de ciclo.
    - Importância de manter a cadência e respeitar o WIP no Kanban.
3. **Planning Poker:**
    - Recomendado para estimar user stories com poucos itens (até 10).
    - Utiliza a sequência de Fibonacci para votações e busca de consenso.
4. **Affinity Mapping:**
    - Agrupamento de itens em categorias similares para manter consistência nos story points.
    - Triangulação para visão comparativa e verificação das estimativas, acelerando o processo.

Padrões:

Design Patterns: Padrões de Projeto

Introdução

Em 1995, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides publicaram o livro "Design Patterns: Elements of Reusable Object-Oriented Software". Este livro apresenta 23 Design Patterns que detalham soluções para problemas recorrentes no desenvolvimento de software orientado a objetos. Devido a esta contribuição, os autores são conhecidos como a "Gangue dos Quatro" (Gang of Four ou GoF).

O objetivo dos padrões de projeto é criar componentes reutilizáveis que facilitem a padronização e agilizem a solução de problemas recorrentes no desenvolvimento de sistemas.

Conceito de Padrão

Existem dois conjuntos de padrões de projeto conhecidos na engenharia de software: os padrões GoF e os padrões GRASP (General Responsibility Assignment Software Patterns).

Padrões GRASP

Os padrões GRASP ajudam a definir a atribuição de responsabilidades de decisão em objetos. Os principais padrões GRASP incluem:

- Especialista na Informação
- Criador
- Controlador
- Acoplamento Fraco
- Coesão Alta
- Polimorfismo
- Invenção Pura
- Indireção
- Variações Protegidas

Padrões GoF

Os padrões GoF são classificados em três categorias: criação, estruturais e comportamentais.

Padrões de Criação

Esses padrões tratam da criação de objetos de maneira a atender diversas necessidades, reduzindo o acoplamento e permitindo maior controle sobre as instâncias.

1. **Factory Method**: Define uma interface para criar um objeto, deixando as subclasses decidirem qual classe instanciar.
2. **Abstract Factory**: Cria famílias de objetos relacionados ou interdependentes sem especificar suas classes concretas.
3. **Singleton**: Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela.
4. **Builder**: Permite a construção de objetos complexos passo a passo.
5. **Prototype**: Cria novos objetos clonando uma instância existente.

Padrões Estruturais

Esses padrões tratam da composição de classes ou objetos para formar estruturas maiores.

1. **Adapter**: Converte a interface de uma classe em outra interface esperada pelos clientes.
2. **Bridge**: Separa uma abstração de sua implementação para que ambas possam variar independentemente.
3. **Composite**: Trata objetos individuais e composições de objetos de maneira uniforme.
4. **Decorator**: Adiciona responsabilidades a objetos dinamicamente.

5. **Facade**: Fornece uma interface simplificada para um conjunto de interfaces em um subsistema.
6. **Flyweight**: Usa compartilhamento para suportar grandes quantidades de objetos de forma eficiente.
7. **Proxy**: Controla o acesso a um objeto através de um substituto.

Padrões Comportamentais

Esses padrões tratam da comunicação entre objetos.

1. **Chain of Responsibility**: Passa uma solicitação por uma cadeia de handlers.
2. **Command**: Encapsula uma solicitação como um objeto.
3. **Interpreter**: Representa gramáticas e interpreta sentenças nessa gramática.
4. **Iterator**: Fornece uma maneira de acessar elementos de uma coleção sequencialmente.
5. **Mediator**: Define um objeto que encapsula como um conjunto de objetos interage.
6. **Memento**: Captura e restaura o estado interno de um objeto sem violar seu encapsulamento.
7. **Observer**: Define uma dependência um-para-muitos entre objetos.
8. **State**: Permite que um objeto altere seu comportamento quando seu estado interno muda.
9. **Strategy**: Define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis.
10. **Template Method**: Define o esqueleto de um algoritmo em uma operação, diferindo alguns passos para subclasses.
11. **Visitor**: Representa uma operação a ser realizada em elementos de uma estrutura de objeto.

Princípios SOLID

Os princípios SOLID são um conjunto de cinco princípios de design de software que visam tornar o código mais compreensível, flexível e de fácil manutenção. Eles foram definidos por Robert C. Martin e são amplamente utilizados em programação orientada a objetos.

1. **Single Responsibility Principle (SRP)**: Uma classe deve ter apenas uma responsabilidade, ou seja, uma única razão para mudar. Isso significa que cada classe deve lidar com uma única parte da funcionalidade do software e encapsular apenas essa parte.
2. **Open/Closed Principle (OCP)**: As entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação. Isso permite que o comportamento de uma classe possa ser

estendido sem alterar seu código fonte, geralmente através da herança ou implementação de interfaces.

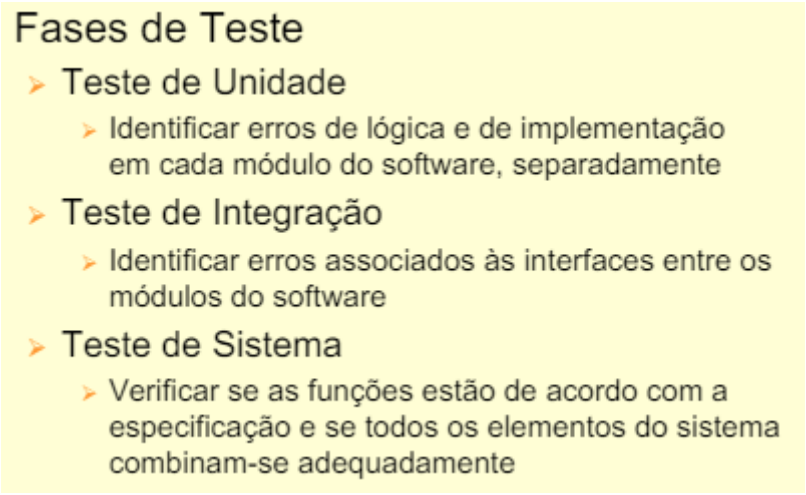
3. **Liskov Substitution Principle (LSP)**: Objetos de uma classe base devem poder ser substituídos por objetos de uma classe derivada sem alterar a correção do programa. Em outras palavras, uma subclasse deve ser substituível pela sua superclasse.
4. **Interface Segregation Principle (ISP)**: Muitas interfaces específicas são melhores do que uma interface única e geral. Este princípio sugere que os clientes não devem ser forçados a depender de interfaces que não utilizam.
5. **Dependency Inversion Principle (DIP)**: Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações. Isso inverte a maneira tradicional de dependências de módulos, promovendo o uso de interfaces ou classes abstratas.

Conclusão

Neste artigo, foi descrito como surgiu o conceito de "Design Patterns" e os principais padrões utilizados, incluindo os padrões GoF e GRASP. Além disso, foram apresentados os princípios SOLID, que são fundamentais para a criação de um código flexível e de fácil manutenção. Esses padrões e princípios fornecem uma base sólida para a construção de software robusto e escalável.

Testes de Software:

Fases de teste:



Testes de Software

Os testes de software são fundamentais para garantir a qualidade, funcionalidade, desempenho e segurança de um aplicativo antes de sua disponibilização aos usuários finais. Eles ajudam a identificar bugs e erros precocemente, evitando problemas mais caros e demorados de resolver em produção.

Como Funcionam os Testes de Software?

Os testes de software envolvem várias etapas:

1. **Planejamento**: Definição de objetivos, escopo, abordagem e cronograma dos testes.
2. **Preparação do Ambiente**: Configuração do ambiente de teste com a instalação da solução em um cenário controlado.
3. **Execução dos Testes**: Realização dos testes conforme o planejado, manualmente ou por meio de automação.
4. **Registro de Inconsistências**: Identificação e documentação de quaisquer bugs ou problemas encontrados.

Importância dos Testes de Software

Os testes são cruciais para:

- Garantir a funcionalidade correta do software.
- Aumentar a satisfação do cliente.
- Melhorar a segurança contra ataques cibernéticos.
- Verificar a escalabilidade e desempenho do software.

Tipos de Testes de Software

1. **Testes Unitários**:
    - **Objetivo**: Verificar funções e classes individualmente.
    - **Vantagens**: Baixo custo, automatizável, detecta bugs cedo.
2. **Testes de Integração**:
    - **Objetivo**: Verificar a interação entre diferentes módulos.
    - **Vantagens**: Valida a integração e funcionamento conjunto dos módulos.
3. **Testes Funcionais**:
    - **Objetivo**: Validar as saídas do software com base nas entradas especificadas no documento SRS.
    - **Vantagens**: Verifica funcionalidades e usabilidade, sem foco no código-fonte.
4. **Testes de Aceitação**:
    - **Objetivo**: Validar o software do ponto de vista do usuário final.
    - **Tipos**:
        - **Teste de Aceitação Formal**: Funções e recursos são previamente conhecidos pelos testadores.

        - **Teste de Aceitação Informal**: Procedimentos não definidos claramente, mais subjetivos.
        - **Teste Beta**: Usuários finais testam o software em um ambiente real.

5. **Testes de Desempenho**:
    - **Objetivo**: Avaliar tempo de resposta, velocidade, escalabilidade e estabilidade sob carga.
    - **Vantagens**: Detecta gargalos de desempenho, assegura a confiabilidade sob alta carga.
6. **Testes de Segurança**:
    - **Objetivo**: Detectar vulnerabilidades e ameaças ao sistema.
    - **Tipos**:
        - **SAST (Static Application Security Test)**: Realizado durante o desenvolvimento para detectar vulnerabilidades.

Escolha e Gerenciamento dos Testes

A escolha dos testes deve basear-se nos requisitos específicos do projeto e nas necessidades dos usuários. Exemplos de aplicação incluem:

- **Software de Gestão Comercial**:
    - **Testes Funcionais**: Verificar cadastro de produtos, gestão de estoque, emissão de notas fiscais.
    - **Testes de Integração**: Verificar a integração com outras ferramentas.
    - **Testes de Desempenho**: Assegurar responsividade com muitos usuários simultâneos.
    - **Testes de Segurança**: Garantir a proteção dos dados confidenciais.

Abordagens de Desenvolvimento: DDD, TDD e BDD

Domain-Driven Design (DDD) – no fim do processo de desenvolvimento

DDD é uma abordagem de design de software que foca em criar um modelo de domínio rico e claro para resolver problemas complexos de negócios. Os principais conceitos do DDD incluem:

- **Entidades**: Objetos com identidade distinta que persistem por longos períodos.
- **Value Objects**: Objetos que descrevem alguns aspectos do domínio sem identidade.
- **Agregados**: Conjuntos de entidades e value objects que são tratados como uma unidade.
- **Repositórios**: Interfaces que fornecem acesso aos agregados.

- **Serviços de Domínio**: Lógica de negócios que não se encaixa naturalmente em entidades ou value objects.

Test-Driven Development (TDD) – Ao longo

TDD é uma prática de desenvolvimento de software onde os testes são escritos antes do código funcional. O ciclo básico de TDD é:

1. **Escrever um Teste Falho**: Criar um teste unitário para uma funcionalidade que ainda não foi implementada.
2. **Implementar o Código**: Escrever o código mínimo necessário para fazer o teste passar.
3. **Refatorar**: Melhorar o código mantendo todos os testes passando.

Vantagens do TDD:

- Garante que o código seja testável.
- Resulta em um design mais simples e limpo.
- Facilita a detecção de bugs logo no início do desenvolvimento.

Behavior-Driven Development (BDD) – No inicio

BDD é uma extensão do TDD que se concentra no comportamento do software em vez da implementação. BDD utiliza uma linguagem comum para descrever a funcionalidade do software em termos de comportamento esperado. As principais práticas de BDD incluem:

- **User Stories**: Descrições de funcionalidades do ponto de vista do usuário.
- **Cenários**: Exemplos concretos de uso que descrevem a interação com a aplicação.
- **Gherkin Syntax**: Uma linguagem estruturada para definir cenários de BDD, utilizando palavras-chave como "Given", "When" e "Then".

Vantagens do BDD:

- Melhora a comunicação entre desenvolvedores, testers e stakeholders.
- Assegura que todos entendam claramente o comportamento esperado do sistema.
- Facilita a criação de documentação que é fácil de entender e manter

CI / CD e DEVOPS

Continuous Integration (CI) e Continuous Deployment (CD) no Contexto de DevOps na AWS

Continuous Integration (CI):

- **Definição:** Processo de integração contínua, onde desenvolvedores frequentemente fazem commits de código em um repositório compartilhado. Cada commit é verificado automaticamente, permitindo detectar erros rapidamente.

- **Práticas:** Incluem testes unitários, de integração, de aceitação e de interface de usuário, garantindo que cada alteração seja validada antes de ser integrada ao código principal.
- **Ferramentas:** Jenkins, uma ferramenta de integração contínua open-source amplamente usada, que suporta integração com serviços da AWS, como EC2 e S3, e permite escalabilidade através da adição de instâncias EC2 como trabalhadores de CI.

Continuous Deployment (CD):

- **Definição:** Processo de deployment contínuo, onde o código é automaticamente implantado em ambientes de produção após passar por todas as fases de teste no CI.
- **Práticas:** Envolvem a automação do processo de deploy, incluindo a configuração e provisionamento de infraestrutura, testes de aceitação e monitoramento pós-deploy.
- **Ferramentas:** AWS Elastic Beanstalk e AWS OpsWorks são serviços que facilitam o deployment contínuo, permitindo a criação de infraestruturas auto-recuperáveis e escaláveis, além de integração com ferramentas de configuração como Chef e Puppet.

DevOps na AWS:

- **Integração de Dev e Ops:** DevOps enfatiza a colaboração entre desenvolvedores e operadores de infraestrutura para reduzir riscos e melhorar a eficiência no ciclo de vida do software.
- **Infraestrutura como Código:** Na AWS, toda a infraestrutura é gerenciada como código, utilizando serviços como AWS CloudFormation para definir e provisionar recursos de forma programática.
- **Automação e Monitoramento:** Utilização de serviços como AWS CloudWatch para monitorar e observar o desempenho de instâncias, aplicativos e logs. A automação cobre desde testes até o deploy, garantindo que os ambientes Dev, QA e Prod sejam o mais semelhantes possível.
- **Modelos de Custo:** Aproveitamento de diferentes modelos de custo da AWS, como instâncias reservadas para repositórios de código e CI master, e instâncias spot para trabalhadores de CI, otimizando os custos de desenvolvimento e testes.

Benefícios:

- **Rapidez e Eficiência:** Processos de CI/CD automatizados resultam em deploys rápidos, confiáveis e repetíveis, com baixo risco e mínima intervenção manual.
- **Escalabilidade:** A elasticidade da AWS permite ajustar a capacidade de infraestrutura conforme necessário, reduzindo custos e melhorando a escalabilidade.

- **Qualidade:** A implementação de CI/CD melhora a qualidade do software, com testes contínuos e verificação de qualidade em todas as etapas do ciclo de desenvolvimento.

Exemplos de Casos:

- **Crowdtest:** Utilização de serviços AWS como EC2, RDS e S3 para hospedar uma plataforma de testes, com automação de integração contínua e testes automáticos em múltiplas plataformas.
- **VTEX:** Implementação de Continuous Deployment com ambientes criados automaticamente através de tags no Git, permitindo publicação descentralizada e rollback fácil em caso de falhas.

A adoção de práticas de CI/CD e DevOps na AWS traz uma série de benefícios, desde a automação e eficiência no desenvolvimento e deploy, até a escalabilidade e otimização de custos, melhorando significativamente a qualidade e velocidade de entrega do software.

Deploy Contínuo: Com e Sem Revisões

Deploy Contínuo com Revisões:

- **Definição:** Processo em que o código passa por revisões antes de ser implantado em produção. As revisões podem incluir revisões de código por pares, testes adicionais e validação manual em ambientes de staging.
- **Vantagens:**
    - **Qualidade Aumentada:** Revisões adicionais ajudam a detectar e corrigir erros antes que o código chegue à produção.
    - **Controle de Qualidade:** Permite que equipes garantam que todas as alterações estejam alinhadas com os padrões de qualidade e requisitos de negócio.
    - **Confiança na Mudança:** Reduz o risco de introdução de bugs críticos, pois mais olhos avaliam o código antes do deployment.
- **Desvantagens:**
    - **Atrasos no Deploy:** Revisões podem introduzir atrasos, especialmente se os revisores não estiverem disponíveis imediatamente.
    - **Complexidade no Fluxo:** Adicionar etapas de revisão pode tornar o fluxo de trabalho mais complexo e menos ágil.

Deploy Contínuo sem Revisões:

- **Definição:** Processo em que o código é automaticamente implantado em produção após passar por testes automatizados, sem intervenção humana adicional.
- **Vantagens:**

- **Velocidade:** Reduz o tempo entre o desenvolvimento e a implantação, permitindo que novas funcionalidades e correções de bugs cheguem aos usuários rapidamente.
- **Automação:** Maximiza a automação, minimizando a necessidade de intervenção manual e potencial para erro humano.
- **Feedback Rápido:** Permite feedback rápido dos usuários e correção de problemas em tempo real.

- **Desvantagens:**
  - **Risco de Bugs:** Sem revisões humanas, há um maior risco de bugs não detectados chegarem à produção.
  - **Dependência de Testes Automatizados:** A qualidade do deploy depende fortemente da abrangência e eficácia dos testes automatizados.

Melhor Fluxo de Planejamento e Deploy

1. **Planejamento de Releases:**
   - **Backlog Priorizado:** Utilize uma abordagem ágil para manter um backlog priorizado de funcionalidades e correções.
   - **Sprints:** Organize o trabalho em sprints (ciclos de desenvolvimento curtos e iterativos) com metas claras.
   - **Planejamento de Iteração:** Defina claramente as tarefas e objetivos para cada sprint, incluindo critérios de aceitação e testes.
2. **Desenvolvimento:**
   - **Branches de Feature:** Desenvolvedores trabalham em branches de feature para isolar o desenvolvimento de novas funcionalidades.
   - **Commits Frequentes:** Práticas de commit frequentes e pequenas alterações ajudam a identificar e corrigir problemas rapidamente.
   - **Integração Contínua:** Utilize integração contínua (CI) para garantir que cada commit seja automaticamente testado e integrado ao código principal.
3. **Testes:**
   - **Testes Automatizados:** Inclua testes unitários, de integração e de aceitação automatizados no pipeline de CI.
   - **Ambientes de Staging:** Utilize ambientes de staging que espelhem o ambiente de produção para testes adicionais e validação.
4. **Revisão (se aplicável):**
   - **Revisão de Código:** Peers revisam o código, procurando por bugs, vulnerabilidades e aderência aos padrões de codificação.

- **Testes Manuais:** Testadores podem realizar testes manuais adicionais em ambientes de staging para validação final.

5. **Deploy Contínuo:**
    - **Pipeline de CI/CD:** Configure um pipeline de CI/CD que automatize a construção, teste e deployment do código.
    - **Deploy Automático:** Após passar por todos os testes (e revisões, se aplicável), o código é automaticamente implantado em produção.
    - **Monitoramento:** Monitore o desempenho e a saúde dos serviços pós-deploy usando ferramentas como AWS CloudWatch, Grafana, e sistemas de logging.
6. **Feedback e Iteração:**
    - **Monitoramento Contínuo:** Utilize métricas e logs para monitorar a aplicação e detectar problemas em tempo real.
    - **Feedback dos Usuários:** Colete feedback dos usuários para identificar melhorias e correções necessárias.
    - **Ciclo Iterativo:** Use o feedback para alimentar o backlog e iniciar novos ciclos de desenvolvimento.