Resumos para Revisão Sistemática de Literatura

Artigo 1 - A Study on the Role of Software Architecture in the Evolution and Quality of Software

Dados do Trabalho:

- Título: A Study on the Role of Software Architecture in the Evolution and Quality of Software
- Ano de Publicação: 2015
- Autoria: Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek e Yuanfang Cai
- Fonte: Artigo disponível no ResearchGate. Link para acesso

Resumo do Trabalho: "Um Estudo sobre o Papel da Arquitetura de Software na Evolução e Qualidade do Software"

Introdução

O trabalho aborda o problema de pesquisa sobre o **impacto da arquitetura de software na evolução e qualidade de sistemas de software**. A relevância da pesquisa se dá pela necessidade de entender como a arquitetura influencia a capacidade de manutenção e a qualidade do software. O objetivo principal é **investigar se co-mudanças que abrangem múltiplos módulos arquitetônicos têm um impacto diferente na qualidade do software em comparação com co-mudanças localizadas dentro de um único módulo**. Os objetivos específicos incluem:

- Quantificar a dispersão de co-mudanças entre módulos arquitetônicos.
- Avaliar se diferentes visões arquitetônicas (surrogates) impactam a relação entre dispersão de co-mudanças e defeitos.
- Comparar a correlação de métricas de co-mudança que consideram a arquitetura com métricas que não a consideram.
  O estudo busca responder às seguintes questões de pesquisa:
- **RQ1: Co-mudanças dispersas entre múltiplos módulos arquitetônicos são mais propensas a defeitos do que co-mudanças localizadas dentro de um módulo?**
- **RQ2: Diferentes surrogates para a visão de módulo exibem resultados diferentes na relação entre dispersão de co-mudanças e defeitos?**
- **RQ3: Uma métrica que diferencia co-mudanças entre módulos tem maior correlação com defeitos do que uma métrica que não considera a arquitetura?**

Fundamentação Teórica

A base teórica inclui conceitos de **arquitetura de software como uma abstração para lidar com a complexidade do software**, o **princípio da separação de preocupações**, a **importância da arquitetura no projeto e manutenção** e o conceito de **co-mudanças como indicadores de acoplamento lógico**. O trabalho se relaciona com a área de engenharia de software e utiliza conceitos de **módulos arquitetônicos**, **co-mudanças intra e entre módulos**, **acoplamento lógico** e **análise de repositórios de software**. Os autores se baseiam em trabalhos anteriores sobre a relação entre mudanças de código, dependências e qualidade de software.

Metodologia

Foi utilizada uma abordagem **empírica**, com **análise de dados de repositórios de código-fonte de projetos open source**. As técnicas e ferramentas empregadas incluem:

- **Co-change Extractor:** para extrair grupos de arquivos que foram modificados juntos.
- **Defect Extractor:** para identificar mudanças que introduziram defeitos.
- **Architecture Explorer:** para reconstruir a visão de módulo da arquitetura, usando diferentes métodos de engenharia reversa (surrogate views).
- **Negative Binomial Regression (NBR):** para modelar a relação entre co-mudanças e defeitos.
  Os dados foram coletados de repositórios SVN de projetos open source, com um período de tempo fixo para coleta de co-mudanças e defeitos. A análise envolveu a definição de métricas de co-mudança intra e entre módulos (IMC e CMC), além do uso de regressão binomial negativa para análise estatística.

Desenvolvimento

O trabalho foi estruturado em etapas principais:

1. **Extração de co-mudanças e defeitos:** Coleta de dados de repositórios de software.
2. **Reconstrução da arquitetura:** Uso de diferentes técnicas de engenharia reversa para gerar "surrogate views" da arquitetura (Package View, Bunch View, ArchDRH View, LDA View e ACDC View).
3. **Definição de métricas:** Cálculo de métricas de co-mudança intra e entre módulos (IMC e CMC).
4. **Análise estatística:** Uso de regressão binomial negativa para avaliar a relação entre co-mudanças e defeitos, controlando o tamanho do arquivo (LOC).
   Os autores implementaram as ferramentas para coleta de dados, utilizando a biblioteca SVNKit e métodos de análise estatística.

Resultados

As principais descobertas incluem:

- **Co-mudanças que cruzam módulos arquitetônicos (CMC) são mais correlacionadas com defeitos do que co-mudanças dentro de um módulo (IMC)**, apoiando a hipótese de que a dispersão arquitetônica de mudanças é um indicador de defeitos.
- **As diferentes "surrogate views" (Bunch, ArchDRH, ACDC, Package Low-level, LDA) produzem resultados semelhantes em termos da relação entre dispersão de co-mudanças e defeitos**, exceto a Package View em alto nível.

- **Uma métrica de co-mudança que considera a arquitetura (CMC) tem maior correlação com defeitos do que uma métrica que não considera (NCF - número de arquivos co-alterados)**.
  A análise revelou que **a dispersão das mudanças entre os módulos arquitetônicos, e não apenas a magnitude das mudanças, é um fator crítico na introdução de defeitos**.

Contribuições

O trabalho inova ao **investigar o impacto da arquitetura de software na qualidade do software usando dados de repositórios** e **ao comparar diferentes métodos para reconstruir a arquitetura do software**. As principais contribuições são:

- **Evidência empírica da importância da arquitetura de software na qualidade e evolução de sistemas**.
- **Métricas para quantificar o impacto da dispersão de co-mudanças entre módulos arquitetônicos**.
- **Comparação de diferentes métodos para reconstrução da arquitetura de sistemas de software**.
- **Sugestões para melhorar modelos de previsão de defeitos, considerando a arquitetura**.
- **Um possível método para identificar "bad smells" arquitetônicos**.
  O trabalho tem potencial impacto em:
- **Desenvolvimento de modelos de previsão de defeitos mais precisos**.
- **Melhoria da qualidade de sistemas de software através da consideração da arquitetura**.
- **Identificação de partes do sistema que necessitam de refatoração devido a problemas arquitetônicos**.

Conclusões e Trabalhos Futuros

Os autores concluem que **a arquitetura de software tem um papel crucial na qualidade do software** e que **co-mudanças que cruzam módulos arquitetônicos estão mais associadas a defeitos do que as localizadas dentro de um único módulo**. Os trabalhos futuros sugerem:

- Explorar o uso da arquitetura para entender as causas de defeitos.
- Correlacionar o histórico de revisão/defeitos do software com a arquitetura para encontrar as causas dos problemas.
- Investigar se os defeitos que afetam múltiplos componentes levam à degeneração da arquitetura.
  As limitações identificadas incluem:
- Possíveis erros na vinculação de defeitos com arquivos.
- Ameaças à validade dos métodos de engenharia reversa usados.
- O fato de que os projetos estudados são implementados em Java.

Análise Crítica

- **Pontos Fortes:**
  - Abordagem empírica rigorosa com análise estatística detalhada.
  - Uso de diferentes surrogate views para avaliar a influência da arquitetura.

    - Comparação com um projeto industrial e um projeto open source com arquitetura documentada.
    - Contribuições práticas para a previsão de defeitos e identificação de "bad smells" arquitetônicos.
- **Limitações:**
    - Amostra limitada a projetos Java com histórico de bugs em logs de commits.
    - Potenciais vieses na forma como os bugs foram associados aos commits.
    - Dependência de ferramentas de engenharia reversa para obter a arquitetura.
- **Relevância para a área:**
    - Fornece evidências empíricas sólidas da importância da arquitetura para a qualidade do software.
    - Avança o estado da arte em modelagem e análise de arquitetura de software.
    - Oferece insights práticos para arquitetos e desenvolvedores de software.

Palavras-chave

Arquitetura de Software, Co-mudanças, Qualidade de Software, Engenharia Reversa, Previsão de Defeitos.

Artigo 2 - Adesão da Arquitetura de Microsserviços nas Grandes Corporações para Desenvolvimento ou Migração de Aplicações

Dados do Trabalho:

- Título: Adesão da Arquitetura de Microsserviços nas Grandes Corporações para Desenvolvimento ou Migração de Aplicações
- Ano de Publicação: 2023
- Autoria: Gabriel Pinheiro de Santana Lima e José Roberto de Araújo Fontoura
- Fonte: Artigo disponível no Saber Aberto UNEB. Link para acesso

Resumo do Trabalho: "Adesão da Arquitetura de Microsserviços nas Grandes Corporações para Desenvolvimento ou Migração de Aplicações"

Introdução

O trabalho aborda o **problema de pesquisa** sobre o impacto da arquitetura de microsserviços em grandes corporações, questionando qual o efeito que essa arquitetura pode trazer para o desenvolvimento ou migração de aplicações. A **justificativa da pesquisa** reside na crescente tendência de grandes corporações em adotar microsserviços para melhorar a qualidade de seus softwares, e na necessidade de compreender melhor os benefícios e desafios associados a essa implementação. O **objetivo principal** da investigação é analisar como a arquitetura de microsserviços impacta positivamente grandes corporações. Os **objetivos específicos** incluem:

- Comparar as diferenças entre arquitetura de microsserviços e arquitetura monolítica.
- Verificar como os microsserviços funcionam.

- Demonstrar os benefícios que a arquitetura de microsserviços traz para as empresas.
  O estudo se propõe a responder à seguinte **questão de pesquisa**: Qual o impacto que a arquitetura de microsserviços pode trazer às grandes corporações?.

Fundamentação Teórica

A base teórica inclui os conceitos de **arquitetura monolítica** e **arquitetura de microsserviços**, e as diferenças entre elas. Os autores se apoiam em autores como Fowler, Newman e Richardson. O trabalho se relaciona com a **área de arquitetura de software** e utiliza conceitos de **modularização**, **acoplamento**, **escalabilidade**, **resiliência** e **comunicação entre serviços**.

Metodologia

Foi utilizada uma abordagem de **natureza básica**, com uma metodologia **quali-quantitativa**, e um **método hipotético-dedutivo**. Os procedimentos adotados foram **bibliográficos e documentais**, com revisão de literatura e análise de casos. A coleta de dados incluiu uma **observação simples**. Não há detalhes sobre o tamanho ou características da amostra, mas a pesquisa analisa casos como Amazon, Netflix e Uber. A análise envolveu a interpretação dos dados coletados para verificar o impacto dos microsserviços nas empresas.

Desenvolvimento

O trabalho foi estruturado em dez seções que abordam:

- A introdução, contextualizando o tema e os objetivos.
- As características da arquitetura monolítica e da arquitetura de microsserviços, incluindo um comparativo entre ambas.
- O funcionamento dos microsserviços e seus benefícios para as empresas.
- A metodologia empregada.
- A análise dos resultados obtidos.
- As considerações finais.
  O trabalho utiliza exemplos práticos de empresas que adotaram microsserviços.

Resultados

As principais descobertas incluem:

- A arquitetura de microsserviços proporciona melhorias significativas para as empresas, superando desafios de escalabilidade e melhorando o desempenho das aplicações.
- A adoção de microsserviços permite que as empresas sejam mais ágeis, flexíveis e eficientes no desenvolvimento e manutenção de sistemas.
- Empresas como Amazon, Netflix e Uber obtiveram sucesso com a adoção de microsserviços, melhorando a funcionalidade, escalabilidade e reduzindo custos.
  A análise revelou que a migração para microsserviços não é apenas uma mudança técnica, mas também cultural, exigindo uma mentalidade ágil e colaborativa.

Contribuições

O trabalho inova ao fornecer uma **compreensão mais profunda dos conceitos, princípios e benefícios** associados à adoção de microsserviços por grandes corporações. As principais contribuições são:

- **Avanço do conhecimento** em arquitetura de software e microsserviços.

- **Evidências** do impacto positivo dos microsserviços em grandes empresas.
- **Insights** valiosos para empresas interessadas em explorar essa arquitetura.
- **Apoio à tomada de decisão** sobre a adoção de microsserviços como estratégia para impulsionar a inovação, eficiência e competitividade.
  O trabalho tem potencial impacto em empresas que desejam melhorar seus processos de desenvolvimento e se destacar no mercado.

Conclusões e Trabalhos Futuros

Os autores concluem que a **adoção da arquitetura de microsserviços tem um impacto positivo** nas grandes corporações, permitindo maior agilidade, flexibilidade e eficiência. A pesquisa confirma a hipótese de que a adoção de microsserviços traz benefícios como **rapidez no desenvolvimento, melhoria da qualidade e redução de custos operacionais**. Não há sugestões explícitas para trabalhos futuros no texto. As limitações são referentes à análise de casos específicos de empresas, com a necessidade de mais estudos empíricos.

Análise Crítica

- **Pontos Fortes:**
  - Revisão bibliográfica abrangente sobre arquitetura de microsserviços e monolítica.
  - Análise de casos de sucesso de grandes empresas como Amazon, Netflix e Uber.
  - Discussão detalhada sobre os benefícios e desafios da arquitetura de microsserviços.
  - Apresentação clara do funcionamento dos microsserviços.
  - Confirmação da hipótese de que microsserviços trazem impactos positivos.
- **Limitações:**
  - A metodologia de coleta de dados foi por observação simples de documentos, o que pode limitar a profundidade da análise.
  - Não há detalhes sobre o processo de seleção dos casos de empresas.
  - Falta de uma análise mais aprofundada dos desafios e contradições teóricas em relação aos microsserviços.
  - Não são exploradas as dificuldades de transição de arquiteturas monolíticas para microsserviços, além de alguns desafios de implementação.
- **Relevância para a área:**
  - Fornece informações valiosas para empresas que consideram a adoção de microsserviços.
  - Contribui para o avanço do conhecimento em arquitetura de software.
  - Ajuda a entender o impacto dos microsserviços em grandes corporações.

Palavras-chave

Arquitetura de Microsserviços, Grandes Corporações, Competitividade.

Artigo 3 - Análise Comparativa entre Ferramentas de Desenvolvimento de Aplicativos Móveis Multiplataforma utilizando Características da ISO/IEC 25010 por meio da Implementação de um Cenário Pré-definido

Dados do Trabalho:

- Título: Análise Comparativa entre Ferramentas de Desenvolvimento de Aplicativos Móveis Multiplataforma utilizando Características da ISO/IEC 25010 por meio da Implementação de um Cenário Pré-definido
- Ano de Publicação: 2019
- Autoria: Gustavo Ribeiro Miranda
- Fonte: Artigo disponível no IFSP Pergamum. Link para acesso

Resumo do Trabalho: "Análise Comparativa entre Ferramentas de Desenvolvimento de Aplicativos Móveis Multiplataforma"

Introdução

O trabalho aborda o **problema de pesquisa** da comparação entre ferramentas de desenvolvimento de aplicativos móveis multiplataforma, especificamente **React Native (híbrido-nativo) e Ionic (híbrido)**. A **justificativa da pesquisa** reside na crescente relevância de aplicativos móveis e na necessidade de escolher as ferramentas mais adequadas para desenvolvimento, considerando que existem muitas opções disponíveis no mercado. O **objetivo principal** do estudo é comparar as duas ferramentas, utilizando como base as normas ISO/IEC 25000, também conhecidas como SQuaRE. Os **objetivos específicos** incluem:

- Analisar a Eficiência de Desempenho, Adequação Funcional e Usabilidade de cada ferramenta.
- Implementar um cenário de desenvolvimento de um aplicativo com as mesmas funcionalidades em ambas as ferramentas.
- Coletar dados através de questionários respondidos pelos usuários, baseados em sua experiência com as ferramentas.
  O estudo se propõe a responder qual ferramenta se destaca em quais características, para resolução do mesmo problema.

Fundamentação Teórica

A base teórica inclui os conceitos de **desenvolvimento híbrido e híbrido-nativo**, além de explicar o funcionamento de ferramentas como **Apache Cordova**, **Ionic**, **Angular**, **TypeScript**, **React Native** e **React.js**. O trabalho se relaciona com a **área de desenvolvimento de software para dispositivos móveis** e utiliza conceitos de **qualidade de software** definidos na norma **ISO/IEC 25000 SQuaRE**.

Metodologia

Foi utilizada uma **abordagem comparativa** para analisar as ferramentas, baseada nas características da norma ISO/IEC 25000. A coleta de dados foi realizada através de **questionários**, elaborados com base na abordagem **GQM (Goal-Question-Metrics)**, que visam medir o software através de objetivos, perguntas e métricas. O **tamanho da amostra** foi de cinco (5) participantes, todos estudantes do curso de Análise e Desenvolvimento de Sistemas (ADS). A análise envolveu a interpretação das respostas dos questionários, utilizando a escala Likert para avaliar os critérios da ISO.

Desenvolvimento

O trabalho foi estruturado em etapas principais:

- Estudo da literatura sobre desenvolvimento multiplataforma e a norma técnica SQuaRE.
- Seleção das ferramentas (React Native e Ionic) e dos critérios de avaliação (Eficiência de Desempenho, Adequação Funcional e Usabilidade).
- Definição de um cenário de desenvolvimento de um aplicativo com três abas, incluindo componentes com acesso a API nativa, geolocalização e OAuth.
- Elaboração de um questionário com base na abordagem GQM, para coletar dados dos participantes.
- Análise dos resultados obtidos nos questionários.
  A implementação envolveu o desenvolvimento do aplicativo nas duas ferramentas e a aplicação do questionário aos participantes.

Resultados

As principais descobertas incluem:

- Em relação à **Eficiência de Desempenho**, o **Ionic** obteve melhor avaliação devido ao uso do navegador para preview, que proporciona um feedback mais rápido ao desenvolvedor. O React Native, por utilizar um emulador, apresentou um tempo de resposta menor.
- Quanto à **Adequação Funcional**, o **React Native** foi considerado ligeiramente melhor na **Integridade Funcional**, enquanto o **Ionic** se destacou na **Adequação Funcional**.
- Na **Usabilidade**, o **Ionic** foi considerado superior, tanto na **Operacionalidade** quanto na **Apreensibilidade**, demonstrando ser mais fácil de operar e aprender a utilizar.

Contribuições

O trabalho inova ao aplicar **critérios de qualidade da norma ISO/IEC 25000** para comparar ferramentas de desenvolvimento de aplicativos móveis, fornecendo uma análise prática e detalhada. As principais contribuições são:

- **Comparação detalhada** entre Ionic e React Native com base em critérios de qualidade.
- **Insights** sobre as vantagens e desvantagens de cada ferramenta em relação ao tempo de resposta, facilidade de uso e adequação funcional.
- **Aplicação da norma ISO/IEC 25000** para avaliar ferramentas de desenvolvimento de aplicativos móveis, utilizando uma abordagem de métricas orientada por objetivos.

- **Utilização da abordagem GQM** para criação de questionários, a fim de coletar métricas úteis para a análise dos resultados obtidos.
  O trabalho tem potencial impacto em desenvolvedores de aplicativos móveis que precisam escolher a melhor ferramenta para seus projetos.

Conclusões e Trabalhos Futuros

Os autores concluem que, com base nos resultados obtidos, o **Ionic** demonstrou ser mais adequado para os desafios propostos neste trabalho, principalmente devido à sua facilidade de uso e feedback rápido, utilizando tecnologias como HTML, CSS e JS com Angular. No entanto, o **React Native** se destaca na **reutilização de código** em grandes aplicações. As **limitações** do estudo incluem o pequeno número de participantes e a possível influência de suas experiências prévias com as ferramentas. Como **trabalhos futuros**, sugere-se expandir a pesquisa para um maior número de pessoas e incluir outras ferramentas.

Análise Crítica

- **Pontos Fortes:**
  - Uso da norma ISO/IEC 25000 para comparar as ferramentas, o que fornece uma base sólida para a avaliação.
  - Aplicação da abordagem GQM para a elaboração do questionário.
  - Criação de um cenário de desenvolvimento prático para testar as ferramentas.
  - Análise dos resultados com base nas opiniões dos participantes.
- **Limitações:**
  - O número de participantes da pesquisa é pequeno, o que pode limitar a generalização dos resultados.
  - Os participantes são todos estudantes de ADS, o que pode não refletir a opinião de desenvolvedores experientes no mercado.
  - A pesquisa não aborda outros aspectos relevantes das ferramentas, como custo, facilidade de manutenção e disponibilidade de documentação.
- **Relevância para a área:**
  - Fornece informações valiosas para desenvolvedores que precisam escolher entre Ionic e React Native.
  - Apresenta uma metodologia baseada em normas técnicas para avaliar ferramentas de desenvolvimento.
  - Contribui para o avanço do conhecimento sobre as vantagens e desvantagens de diferentes abordagens de desenvolvimento de aplicativos móveis.

Palavras-chave

Desenvolvimento Multiplataforma, React Native, Ionic, ISO/IEC 25000, GQM.

Artigo 4 - Aplicando refatorações em uma arquitetura de software monolítica para uma solução utilizando microsserviços

Dados do Trabalho:

- Título: Aplicando refatorações em uma arquitetura de software monolítica para uma solução utilizando microsserviços
- Ano de Publicação: 2021
- Autoria: Alan Ricardo S. Silva e Ana Claudia Rossi
- Fonte: Artigo disponível no Repositório Institucional Mackenzie. Link para acesso

Resumo do Trabalho: "Aplicando Refatorações em uma Arquitetura de Software Monolítica para uma Solução utilizando Microsserviços"

Introdução

O trabalho aborda o **problema de pesquisa** da refatoração de uma arquitetura de software monolítica para uma arquitetura de microsserviços, motivado pelo **crescimento e complexidade** dos sistemas atuais, que tornam a arquitetura monolítica inviável em muitos casos. A **justificativa da pesquisa** reside na necessidade de sistemas mais escaláveis, de fácil manutenção e seguros, qualidades oferecidas pela arquitetura de microsserviços. O **objetivo principal** da pesquisa é aplicar um padrão de refatoração em uma arquitetura monolítica genérica para um modelo que utiliza microsserviços. Os **objetivos específicos** incluem extrair, formular e conduzir o processo de refatoração, e avaliar os impactos dessa refatoração.

Fundamentação Teórica

A base teórica inclui os conceitos de **arquitetura monolítica**, onde todos os módulos são agrupados em uma única peça, e **arquitetura de microsserviços**, onde os serviços são independentes e implantados individualmente. O trabalho também aborda o conceito de **refatoração**, que consiste em modificar a estrutura interna do código sem alterar seu comportamento externo. A pesquisa se apoia em autores como **Newman** e **Fowler**, e se relaciona com a **área de arquitetura de software**.

Metodologia

Foi utilizada a metodologia **Design Science Research (DSRM)**, dividida em três etapas: identificação do problema e motivação, definição dos objetivos e solução, e projeto e desenvolvimento da solução. Essa metodologia foi escolhida devido à necessidade de identificar um problema real em um ambiente de software que utilizava uma arquitetura monolítica. A **técnica** de coleta de dados foi a aplicação prática da refatoração em um sistema real.

Desenvolvimento

O trabalho foi estruturado em etapas principais:

- **Identificação** de um sistema monolítico (o sistema PAR).
- **Análise** da arquitetura monolítica do sistema.

- **Aplicação** do padrão Strangler Fig para migrar gradualmente o sistema para microsserviços.
- **Implementação** de um microsserviço para o plano de atividades do recurso (PAR).
- **Decomposição** do banco de dados, adotando o padrão de banco de dados por contexto delimitado.
- **Desenvolvimento de um modelo de arquitetura de microsserviços.**
- **Apresentação** do modelo de arquitetura final.
  As principais **tecnologias** e abordagens utilizadas foram: padrões de refatoração, arquitetura de microsserviços, e o padrão Strangler Fig. Os **parâmetros** considerados foram a escalabilidade, manutenção, disponibilidade e a liberdade de escolha de tecnologia.

Resultados

As principais descobertas incluem:

- A viabilidade da migração de um sistema monolítico para microsserviços, usando o padrão Strangler Fig.
- A possibilidade de decompor o sistema em serviços independentes, cada um responsável por uma tarefa específica.
- A importância de escolher uma estratégia adequada para a decomposição do banco de dados, adotando o padrão de banco de dados por contexto delimitado.
- A criação de um modelo de arquitetura que garante a liberdade da equipe de desenvolvimento na escolha de tecnologia, facilidade em resolver problemas de manutenção, escalabilidade e disponibilidade.

Contribuições

O trabalho inova ao aplicar o processo de refatoração de um sistema monolítico para microsserviços em um cenário real, utilizando um sistema de planejamento de recursos genérico como base. As principais contribuições são:

- Apresentação de um modelo de arquitetura de microsserviços, baseado em um sistema real.
- Utilização do padrão Strangler Fig para migrar o sistema monolítico para uma arquitetura de microsserviços.
- Implementação de um padrão de decomposição do banco de dados por contexto delimitado.
- Discussão sobre as vantagens e desvantagens das abordagens de decomposição do banco de dados.
  O trabalho tem potencial impacto em desenvolvedores que precisam migrar sistemas monolíticos para microsserviços.

Conclusões e Trabalhos Futuros

Os autores concluem que é possível migrar um sistema monolítico para microsserviços, obtendo benefícios como escalabilidade, manutenção e disponibilidade. Como **trabalhos futuros**, eles sugerem explorar o processo de deploy de aplicações que adotam o padrão de microsserviços e a aplicação de diferentes tecnologias em um mesmo projeto utilizando essa arquitetura. As **limitações** identificadas não são explicitamente mencionadas no resumo.

Análise Crítica

- **Pontos Fortes:**
  - Aplicação prática da refatoração em um sistema real.
  - Utilização do padrão Strangler Fig para migração gradual.
  - Apresentação de um modelo de arquitetura de microsserviços.
  - Discussão sobre a decomposição do banco de dados.
- **Limitações:**
  - O resumo não detalha as limitações encontradas no processo de refatoração.
  - A falta de dados quantitativos sobre os impactos da refatoração.
- **Relevância para a área:**
  - Fornece um guia prático para migrar sistemas monolíticos para microsserviços.
  - Apresenta uma abordagem baseada em padrões de arquitetura de software.
  - Contribui para o avanço do conhecimento sobre refatoração de sistemas.

Palavras-chave

Refatoração, Microsserviços, Arquitetura Monolítica, Strangler Fig, Design Science Research.

Artigo 5 - Arquitetura Hiper-Hexagonal: Um Novo Paradigma de Computação Paralela

Dados do Trabalho:

- Título: Arquitetura Hiper-Hexagonal: Um Novo Paradigma de Computação Paralela
- Ano de Publicação: 1998
- Autoria: Guiou Kobayashi
- Fonte: Artigo disponível no Repositório de Teses e Dissertações da USP. Link para acesso

Resumo do Trabalho "Arquitetura Hiper-Hexagonal: Um Novo Paradigma de Computação Paralela"

Introdução

Este trabalho, intitulado **"Arquitetura Hiper-Hexagonal: Um Novo Paradigma de Computação Paralela"**, publicado em **1998**, foi desenvolvido por Guiou Kobayashi em São Paulo. O estudo aborda a **proposta de uma nova arquitetura de computação paralela denominada hiper-hexagonal**. Este tema é relevante devido à **necessidade de novas arquiteturas para computação paralela que superem as limitações das arquiteturas tradicionais**, oferecendo melhor desempenho e escalabilidade. A pesquisa visa introduzir um **novo modelo de interconexão entre processadores para computação paralela**, explorando as características da estrutura hexagonal. O objetivo principal é **apresentar a arquitetura hiper-hexagonal como um paradigma alternativo para a construção de computadores paralelos**. As principais questões investigadas incluem: **Quais são as propriedades da arquitetura hiper-hexagonal? Como essa arquitetura se compara às arquiteturas paralelas existentes? Quais são as vantagens e desafios da arquitetura hiper-hexagonal?**.

Metodologia

O trabalho apresenta uma abordagem teórica e de modelagem para descrever e analisar a arquitetura hiper-hexagonal. A metodologia envolveu a **construção de modelos matemáticos e visuais para descrever a topologia de interconexão dos processadores**. A arquitetura é definida por **uma estrutura onde os processadores estão dispostos em células hexagonais**, formando uma rede que se expande em múltiplas dimensões, formando um hipercubo. O estudo também incluiu a **análise de propriedades como diâmetro, grau e número de caminhos na rede**. A metodologia baseou-se na **exploração das propriedades geométricas da estrutura hexagonal** para derivar a arquitetura hiper-hexagonal.

Inovação e Contribuições

Este trabalho se destaca por sua abordagem inovadora na **proposição de uma arquitetura paralela baseada em uma topologia não convencional: a hiper-hexagonal**, que se distingue das arquiteturas lineares, em árvore ou em malha mais comuns. Antes deste estudo, **as arquiteturas paralelas mais comuns baseavam-se em topologias como hipercubos, árvores e malhas**. Os autores especularam que **a arquitetura hiper-hexagonal poderia oferecer um bom compromisso entre diâmetro da rede, grau dos nós, complexidade e escalabilidade**, além de um certo potencial de tolerância a falhas. Suas contribuições incluem:

- A introdução da arquitetura hiper-hexagonal como uma nova topologia para computação paralela.
- A descrição matemática e geométrica da topologia hiper-hexagonal.
- A análise de propriedades da rede como o grau dos nós, o diâmetro e a capacidade de roteamento.
- A proposição de que essa arquitetura apresenta um bom balanço entre complexidade, escalabilidade e potencial de tolerância a falhas, quando comparada com arquiteturas mais comuns.
- A demonstração de um conceito de interconexão de processadores paralelos com uma estrutura hexagonal, oferecendo caminhos diversos para comunicação.

Resultados e Conclusão

Os principais resultados encontrados foram a **definição formal da arquitetura hiper-hexagonal, com a análise de suas propriedades de rede**, como o número de nós conectados, o diâmetro da rede e o potencial de roteamento. O trabalho oferece implicações importantes para o **design de arquiteturas de computação paralela**, apresentando uma alternativa à arquitetura em hipercubo, em árvore ou em malha. Como desdobramentos, os autores sugerem **investigações futuras sobre algoritmos de roteamento específicos para a arquitetura hiper-hexagonal e sua implementação em hardware**. O trabalho também **abre possibilidades para o desenvolvimento de aplicações paralelas com topologias não convencionais**, que se adequem às propriedades da arquitetura hiper-hexagonal.

Palavras-chave

Arquitetura e organização de computadores, Computação paralela

Artigo 6 - Arquitetura orientada a serviços e sua integração com software as a service

Dados do Trabalho:

- Título: Arquitetura orientada a serviços e sua integração com software as a service
- Ano de Publicação: 2023
- Autoria: FENERICH, Bruno Munaretti
- Fonte: Artigo disponível no Repositório Institucional do Centro Paula Souza. Link para acesso

Resumo do Trabalho: "Arquitetura orientada a serviços e sua integração com software as a service"

Introdução

O trabalho aborda a **Arquitetura Orientada a Serviços (SOA)** e sua integração com o modelo **Software as a Service (SaaS)**. O **problema de pesquisa** centraliza-se em como a SOA facilita a integração entre diferentes sistemas e aplicações, especialmente no contexto do SaaS. A **justificativa da pesquisa** reside na crescente adoção de tecnologias em nuvem e na necessidade de soluções de software mais eficientes, onde SOA e SaaS se destacam. O **objetivo principal** é apresentar conceitos gerais sobre SOA e SaaS. Os **objetivos específicos** incluem introduzir conceitos, terminologias, vantagens, desafios, padrões e implementações de ambas as tecnologias. O estudo visa elucidar esses conceitos cruciais e fomentar uma compreensão profunda de 'serviços' entre estudantes de tecnologia. As **questões de pesquisa** não são explicitamente formuladas, mas o trabalho busca responder como SOA e SaaS se complementam e se beneficiam mutuamente.

Fundamentação Teórica

A base teórica inclui os princípios da **Arquitetura Orientada a Serviços (SOA)**, que visa criar sistemas modulares, escaláveis e reutilizáveis através de serviços. Também explora o **Software as a Service (SaaS)**, um modelo de distribuição de software em que aplicações são hospedadas e disponibilizadas pela internet. Os autores principais mencionados são **Thomas Erl**, cujas definições de serviço e princípios de design são utilizadas. O trabalho se relaciona com a **área do conhecimento** de sistemas distribuídos e arquitetura de software. Os **conceitos fundamentais** incluem serviços, serviços compostos, inventário de serviços, computação em nuvem, arquitetura multi-inquilino, e os níveis de maturidade em SaaS.

Metodologia

Foi utilizada uma abordagem que combina a **análise de materiais já publicados** e a **experiência pessoal** do autor. O trabalho revisa conceitos por meio de **livros e artigos** relacionados aos temas de SOA e SaaS, para entender as tendências e desafios do campo. Em seguida, é incorporada a experiência pessoal de desenvolvimento de soluções hospedadas em SaaS, incluindo projetos reais e interações com outros profissionais. Os **métodos de coleta de dados** incluem revisão bibliográfica e análise da própria experiência. O **tamanho e características da amostra** não se aplicam diretamente, pois não é um estudo empírico no sentido tradicional, mas sim uma análise teórica e reflexiva. Os **procedimentos de análise** envolvem a síntese e interpretação das informações coletadas.

Desenvolvimento

O trabalho é estruturado em três etapas principais:

- **Arquitetura Orientada a Serviços (SOA)**: Aborda conceitos, princípios, vantagens, desafios, padrões e implementações da SOA. Inclui discussões sobre serviços compostos, inventário de serviços, paralelos com arquiteturas baseadas em silos e princípios de design de serviços.
- **Software as a Service (SaaS)**: Explora os fundamentos da computação em nuvem, modelos de implantação, camadas de serviço (IaaS, PaaS, SaaS), arquitetura multi-inquilino, características do SaaS, camadas da arquitetura SaaS e níveis de maturidade em SaaS.
- **Estudo de Caso Salesforce.com**: Analisa a plataforma Salesforce.com como um exemplo de SaaS, incluindo sua relação com SOA, e como os princípios da SOA são implementados nesta plataforma.

O trabalho utiliza **tecnologias/ferramentas** como os protocolos WSDL e XSD, que são usados para definir contratos de serviço. Os autores implementam conceitos de design de serviços e arquitetura de software, considerando **parâmetros/critérios importantes** como interoperabilidade, reutilização, escalabilidade e segurança.

Resultados

As principais descobertas incluem a **interconexão entre SOA e SaaS**, demonstrando que o SaaS pode ser visto como uma aplicação prática da SOA, onde os serviços são os elementos fundamentais. O trabalho também demonstra que a adoção de SOA promove maior reutilização de componentes, escalabilidade e manutenção simplificada, otimizando o desenvolvimento e a entrega de aplicações em nuvem. Os autores identificaram **padrões/tendências** na adoção de tecnologias em nuvem e como a SOA e o SaaS se complementam para fornecer soluções de software mais flexíveis e adaptáveis. A análise revelou **insights importantes** sobre como a SOA pode resolver os problemas de arquiteturas baseadas em silos, oferecendo uma abordagem mais eficiente e escalável.

Contribuições

O trabalho **inova** ao conectar teorias de SOA com a prática de SaaS, demonstrando como os princípios da SOA são aplicados em um exemplo real como a Salesforce. O **diferencial** do trabalho está na combinação de análise teórica com a experiência prática do autor no desenvolvimento de soluções hospedadas em SaaS. As **principais contribuições** são:

- Apresentação clara dos conceitos de SOA e SaaS.
- Discussão sobre a relação entre SOA e SaaS.
- Análise da aplicação prática dos conceitos através do estudo de caso Salesforce.com.

O trabalho tem **potencial impacto** em áreas como desenvolvimento de software em nuvem, arquitetura de sistemas distribuídos e formação de profissionais de tecnologia.

Conclusões e Trabalhos Futuros

Os autores concluem que a **orientação a serviços e os softwares como serviço** assumirão um papel cada vez mais proeminente nos próximos anos. Os autores **sugerem como trabalhos futuros** a exploração de temas mais complexos derivados dos tópicos abordados. As **limitações** identificadas incluem a necessidade de aprofundar temas como arquitetura multi-inquilino e a complexidade da implementação de SOA em grande escala.

Análise Crítica

- **Pontos Fortes**: Apresentação didática dos conceitos, boa conexão entre teoria e prática, estudo de caso relevante.
- **Limitações**: Necessidade de maior aprofundamento em alguns temas, como a arquitetura multi-inquilino, e falta de discussão de desafios mais específicos da implementação em larga escala.
- **Relevância para a área**: O trabalho é relevante por fornecer uma introdução clara e concisa aos conceitos de SOA e SaaS, que são fundamentais para o desenvolvimento de sistemas de software modernos.

Palavras-chave

Arquitetura Orientada a Serviços, Software como Serviço, Microsserviços.

Artigo 7 - Comparison between Ports in Hexagonal Architecture and Interfaces in Clean Architecture: A Conceptual and Practical Analysis

Dados do Trabalho:

- Título: Comparison between Ports in Hexagonal Architecture and Interfaces in Clean Architecture: A Conceptual and Practical Analysis
- Ano de Publicação: 2024
- Autoria: Nagib Sabbag Filho
- Fonte 1: Artigo disponível no Leaders.tec.br. Link para acesso
- Fonte 2: Artigo disponível no ResearchGate. Link para acesso

Resumo do Trabalho: "Comparison between Ports in Hexagonal Architecture and Interfaces in Clean Architecture: A Conceptual and Practical Analysis"

Introdução

O artigo, publicado em **2024**, desenvolvido por **Nagib Sabbag Filho**, aborda a **comparação entre os conceitos de Ports na Arquitetura Hexagonal e Interfaces na Arquitetura Limpa**, um problema relevante devido à necessidade de compreender como esses conceitos moldam a estrutura e a interação de componentes de software. A investigação busca **explorar as semelhanças e diferenças** entre esses conceitos, analisando suas bases teóricas e como eles influenciam a organização e o tratamento de dependências externas.

Fundamentação Teórica

A base teórica inclui **Arquitetura Hexagonal, Arquitetura Limpa, Ports, Interfaces, padrões de projeto, separação de preocupações, e desacoplamento de componentes**. O trabalho se relaciona com a **engenharia de software** e utiliza conceitos como **dependências, comunicação, abstração, flexibilidade, modularidade e testabilidade**. O artigo menciona **Martin (2017)** como referência para as arquiteturas.

Metodologia

Foi utilizada uma abordagem de **análise conceitual e prática**, empregando **exemplos de código** para ilustrar a aplicação dos conceitos. A análise envolveu a **comparação das características e do propósito** das Ports na Arquitetura Hexagonal e das Interfaces na Arquitetura Limpa.

Desenvolvimento

O trabalho foi estruturado na apresentação e comparação das duas arquiteturas:

- **Arquitetura Hexagonal**: Apresenta a estrutura e função da Arquitetura Hexagonal, focando no uso de ports e adapters para isolar a lógica de negócios de tecnologias externas. Um exemplo de e-commerce é utilizado, com um port para processamento de pagamentos e adaptadores para diferentes métodos de pagamento. O artigo mostra um exemplo de código em C# que ilustra a interface `IPaymentProcessor`, os adaptadores `CreditCardPaymentProcessor` e `PayPalPaymentProcessor`, e o uso de injeção de dependência para configurar os processadores de pagamento.
- **Arquitetura Limpa**: Apresenta a estrutura em camadas concêntricas da Arquitetura Limpa, com o domínio no centro e as interfaces nas camadas externas. Um exemplo de sistema de gerenciamento de tarefas é utilizado para ilustrar a separação de preocupações em camadas de apresentação, aplicação, infraestrutura e domínio. O artigo apresenta um exemplo de código em C# que demonstra um controller, um serviço de aplicação, um repositório de tarefas e a entidade de domínio `TaskEntity`, além da configuração de injeção de dependência.

Resultados

As principais descobertas incluem:

- A **diferença fundamental entre ports e interfaces**, com as ports da Arquitetura Hexagonal servindo como pontos de entrada e saída para comunicação bidirecional, enquanto as interfaces da Arquitetura Limpa definem contratos entre camadas, enfatizando o isolamento da lógica de negócios.
- A **Arquitetura Hexagonal** foca na comunicação flexível com o mundo externo, enquanto a **Arquitetura Limpa** prioriza a independência do núcleo da aplicação em relação a detalhes externos.
- Ambas as arquiteturas utilizam **interfaces para promover flexibilidade e modularidade**, mas as utilizam de formas diferentes.
- A importância de **separar preocupações** e de promover a **testabilidade** em ambas as arquiteturas.

Contribuições

O trabalho inova ao **analisar e comparar dois conceitos-chave** em arquiteturas de software populares, oferecendo uma visão clara de suas diferenças e semelhanças. As principais contribuições são:

- A **análise conceitual e prática** das Ports na Arquitetura Hexagonal e Interfaces na Arquitetura Limpa.
- A **comparação das características** de cada abordagem no contexto de projetos de software.

- A **discussão sobre como cada arquitetura promove a flexibilidade e a testabilidade** do código.
- O uso de **exemplos práticos** em C# para ilustrar os conceitos.

O trabalho tem potencial impacto ao **auxiliar desenvolvedores e arquitetos de software na escolha da arquitetura mais adequada** para seus projetos, considerando os princípios e práticas de cada abordagem.

Conclusões e Trabalhos Futuros

Os autores concluem que ambas as arquiteturas oferecem **abordagens valiosas para construir sistemas flexíveis e modulares**. A escolha entre elas depende das necessidades específicas do projeto e das preferências da equipe de desenvolvimento. Em alguns casos, pode ser interessante **combinar aspectos de ambas**. É essencial que a arquitetura suporte a evolução contínua do sistema. O artigo não especifica trabalhos futuros, mas o estudo pode servir de base para análises mais aprofundadas e aplicações em outros contextos. As limitações incluem o fato de se concentrar em uma análise conceitual, sem abordar a implementação em cenários mais complexos.

Análise Crítica

- **Pontos Fortes**:
  - **Análise comparativa** clara e concisa.
  - **Exemplos de código** para facilitar a compreensão dos conceitos.
  - **Discussão sobre as implicações práticas** de cada arquitetura.
- **Limitações**:
  - **Análise mais conceitual**, sem aprofundar em implementações complexas.
  - **Não aborda o uso combinado de ambas as arquiteturas**.
- **Relevância para a área**:
  - O trabalho contribui para a **compreensão das diferenças e semelhanças** entre arquiteturas populares.
  - Fornece **insights para a tomada de decisão** na escolha da arquitetura.
  - Promove a **conscientização sobre a importância da flexibilidade e modularidade** no desenvolvimento de software.

Palavras-chave

Arquitetura Hexagonal, Arquitetura Limpa, Ports, Interfaces, Design de Software.

Artigo 8 - Desenvolvimento Cross-Platform com React Native: Um Estudo de Caso do Aplicativo NaVeg

Dados do Trabalho:

- Título: Desenvolvimento Cross-Platform com React Native: Um Estudo de Caso do Aplicativo NaVeg
- Ano de Publicação: 2019

- Autoria: Gabriel Rodrigues de Araujo
- Fonte: Artigo disponível no Repositório Institucional da Universidade Federal do Ceará (UFC). Link para acesso

Resumo do Trabalho: Desenvolvimento Cross-Platform com React Native: Um Estudo de Caso do Aplicativo NaVeg

Introdução

Este trabalho, **Desenvolvimento Cross-Platform com React Native: Um Estudo de Caso do Aplicativo NaVeg**, publicado em **2019**, foi desenvolvido por **Gabriel Rodrigues de Araujo** e aborda o **desenvolvimento de um aplicativo cross-platform para auxiliar veganos e vegetarianos de Fortaleza**, sendo relevante devido ao **crescimento do mercado vegetariano e do uso de smartphones no Brasil**. A investigação busca **relatar o processo de criação do aplicativo NaVeg com uso do framework React Native**, tendo como objetivos específicos:

- Compreender o React Native e suas tecnologias adjacentes.
- Descrever as boas práticas de criação, organização e desenvolvimento de um aplicativo com React Native.
- Relatar as dificuldades e desvantagens do uso do React Native.
- Avaliar o esforço necessário para gerar o mesmo aplicativo em Android e iOS e comparar ambas as versões.
  O estudo se propõe a responder como seria viável desenvolver um aplicativo para o público vegetariano que pudesse abranger os sistemas operacionais Android e iOS.

Fundamentação Teórica

A base teórica inclui **conceitos de desenvolvimento cross-platform, a biblioteca React e o framework React Native**. O trabalho se apoia em autores que discutem a manipulação do DOM, o uso de JSX e componentes, e o funcionamento do React Native para criar interfaces de aplicativos. O trabalho se relaciona com a **área de Sistemas de Informação Multimídia** e utiliza conceitos de **desenvolvimento de software, interfaces de usuário e tecnologias web**.

Metodologia

Foi utilizada uma abordagem **de estudo de caso**, empregando **metodologia de Design Thinking e princípios do Scrum**. Os dados foram coletados através de **pesquisa de mercado com questionário online, análise de aplicativos concorrentes e testes informais com usuários**. A análise envolveu a **descrição do processo de desenvolvimento, comparação entre plataformas e avaliação da interface**.

Desenvolvimento

O trabalho foi estruturado em **etapas de conceituação do aplicativo, configuração do ambiente de desenvolvimento, desenvolvimento das telas, integração com o banco de dados Firebase e deploy para Android e iOS**. Os autores implementaram o **aplicativo NaVeg utilizando React Native, com componentes reutilizáveis, gerenciamento de estado e navegação com React Navigation**. O desenvolvimento considerou **boas práticas de organização de código, separação de responsabilidades e otimização do uso da ferramenta**.

Resultados

As principais descobertas incluem a **viabilidade do uso de React Native para desenvolvimento cross-platform**, a **facilidade de criar interfaces reativas e reutilizáveis** e o **desempenho satisfatório do aplicativo em Android**. Os autores identificaram **desafios na geração de builds para iOS devido a restrições da Apple**, a **necessidade de ajustes na interface para o iPhoneX** e **pontos de melhoria na usabilidade da aplicação**. A análise revelou **diferenças no tamanho dos arquivos gerados para Android e iOS**.

Contribuições

O trabalho inova ao **apresentar um estudo de caso sobre o desenvolvimento de um aplicativo cross-platform para um nicho de mercado específico**, diferenciando-se pela **aplicação prática dos conceitos teóricos e pela comparação entre as plataformas**. As principais contribuições são:

- Relato detalhado do processo de desenvolvimento com React Native.
- Análise das dificuldades e desvantagens do uso do React Native.
- Comparação entre as versões do aplicativo em Android e iOS.
  com potencial impacto em **projetos de desenvolvimento de aplicativos móveis, estudos sobre tecnologias cross-platform e empreendedorismo no mercado vegetariano**.

Conclusões e Trabalhos Futuros

Os autores concluem que o **React Native é uma ferramenta eficiente para desenvolvimento cross-platform, mas apresenta desafios como a necessidade de dependências e a dificuldade na exportação para iOS**. São sugeridos como trabalhos futuros a **implementação das funcionalidades restantes do aplicativo, testes com usuários, busca de soluções para exportar para iOS e otimização do aplicativo**. As limitações identificadas incluem a **dificuldade na exportação para iOS, a falta de testes completos com usuários e o escopo limitado do projeto**.

Análise Crítica

- **Pontos Fortes**: Clareza na apresentação do processo de desenvolvimento, detalhamento das tecnologias utilizadas, avaliação comparativa entre plataformas, e o uso de metodologias de desenvolvimento.
- **Limitações**: Dificuldade na exportação para iOS, testes com usuários limitados a plataforma Android, e o escopo do projeto restrito a funcionalidades básicas.
- **Relevância para a área**: O trabalho é relevante por fornecer um estudo de caso prático sobre o uso de React Native no desenvolvimento de um aplicativo real, além de trazer informações sobre o mercado vegetariano e desafios encontrados no desenvolvimento cross-platform.

Palavras-chave

Cross-platform, React Native, Vegetarianismo

Artigo 9 - Mobile development in Swift, Java and React Native: an experimental evaluation in audioguides

Dados do Trabalho:

- Título: Mobile development in Swift, Java and React Native: an experimental evaluation in audioguides
- Ano de Publicação: 2019
- Autoria: Hugo Brito, Álvaro Santos, Jorge Bernardino e Anabela Gomes
- Fonte: Artigo disponível no IEEE Xplore Digital Library. Link para acesso

Resumo do Trabalho: "Mobile development in Swift, Java and React Native: an experimental evaluation in audioguides"

Introdução

- Este estudo, publicado em **2019** por **Hugo Brito, Álvaro Santos, Jorge Bernardino e Anabela Gomes**, explora o **desenvolvimento de aplicações móveis** com foco na comparação de três tecnologias: **Swift (para iOS), Java (para Android) e React Native (para ambas as plataformas)**.
- A pesquisa aborda o problema da necessidade de **aplicações de alto desempenho, com desenvolvimento rápido e custos reduzidos**, além da procura por **soluções híbridas** que permitam o desenvolvimento multiplataforma com um único código.
- O **objetivo principal** do estudo é **comparar o tempo de desenvolvimento** das funcionalidades essenciais de uma **aplicação de áudioguia** usando as três abordagens.
- O estudo busca entender qual a melhor abordagem para o desenvolvimento de aplicações móveis, devido ao crescimento do mercado de aplicações móveis, onde as empresas procuram construir aplicações eficientes, com mais funcionalidades e com o mesmo desempenho.

Fundamentação Teórica

- A pesquisa se baseia nos conceitos de **desenvolvimento móvel**, **frameworks híbridas** e, especificamente, o **React Native**.
- O **React Native** surge como uma solução para superar as limitações de abordagens anteriores, combinando o desenvolvimento em **JavaScript**, que é considerado uma das melhores soluções para frontend.
- O artigo também menciona outras soluções híbridas como **PhoneGap, Appcelerator e Ionic**, mas enfatiza o **React Native** como uma solução mais vantajosa para o desenvolvimento de aplicações móveis.
- A **criação do React Native** veio após o sucesso do **ReactJS**, com o objetivo de reduzir o tempo de desenvolvimento através da reutilização de código.

Metodologia

- O estudo adota uma **abordagem comparativa experimental**, analisando o tempo de desenvolvimento de uma **aplicação de áudioguia** em **Swift, Java e React Native**.
- A escolha da **aplicação de áudioguia** justifica-se pela sua relevância no contexto empresarial e pela mudança de paradigma no desenvolvimento de aplicações, de soluções nativas para híbridas.
- As funcionalidades analisadas incluem: **galeria e *swipe* de imagens, leitura de QRCodes, gestão de favoritos, carregamento e execução de áudio, listas dinâmicas de dados e obtenção e armazenamento de dados**.
- O desenvolvimento seguiu a **metodologia SCRUM**, registrando o tempo gasto em cada tarefa.

Resultados

- O **Swift** apresentou o **menor tempo de desenvolvimento total**, seguido de perto pelo React Native e Java.
- O **React Native** demonstrou vantagens na obtenção de dados via **REST API**, devido à sua compatibilidade com o formato **JSON**, mas necessitou de uma biblioteca externa para realizar o *caching* de imagens.
- O **Java** foi consistentemente mais lento em algumas tarefas, como o carregamento de áudio, enquanto o **Swift** mostrou melhor desempenho em manipulação de áudio.
- No desenvolvimento de **listas dinâmicas**, o **React Native** apresentou tempos mais rápidos devido ao uso do componente *flatlist*, que permite a reutilização de código.
- Apesar das vantagens do React Native, na configuração de permissões para a câmara, o desenvolvimento nativo é mais vantajoso, devido a particularidades de cada sistema operativo.
- O desenvolvimento em **Java** para Android pode ser de 20% a 30% mais lento em relação ao **Swift** para iOS. No estudo apresentado o Java foi apenas 11.1% mais lento que o Swift.

Contribuições

- O estudo fornece uma análise detalhada e comparativa do **tempo de desenvolvimento** entre **Swift, Java e React Native** num cenário real de aplicação de áudioguias.
- O trabalho identifica os **pontos fortes e fracos de cada tecnologia**, auxiliando na tomada de decisão sobre a melhor opção para cada projeto.

Conclusões e Trabalhos Futuros

- O **React Native** demonstrou um **tempo de desenvolvimento semelhante ao Java**, mas com a vantagem de ser uma solução multiplataforma.
- A pesquisa sugere que, para **pequenas e médias empresas**, o **React Native** pode ser uma opção vantajosa devido ao desenvolvimento para ambas as plataformas com uma única equipe, o que pode reduzir os custos e tempo de desenvolvimento.
- Como **trabalho futuro**, os autores propõem uma **avaliação experimental com usuários**, comparando a satisfação com as versões nativas e híbridas da mesma aplicação.

Análise Crítica

- **Pontos Fortes:** Análise prática e detalhada do tempo de desenvolvimento, identificação de vantagens e desvantagens de cada tecnologia, foco em um caso de uso relevante.
- **Limitações:** O estudo não avaliou a experiência do usuário final.
- **Relevância para a área:** O estudo é relevante para desenvolvedores e empresas que precisam escolher entre o desenvolvimento nativo e híbrido para aplicações móveis.

Palavras-chave

JavaScript, Android, iOS, React Native.

Artigo 10 - Desenvolvimento de um Aplicativo Utilizando o Framework Flutter e Arquitetura Limpa

Dados do Trabalho:

- Título: Desenvolvimento de um Aplicativo Utilizando o Framework Flutter e Arquitetura Limpa
- Ano de Publicação: 2021
- Autoria: Carlos Eduardo de Oliveira Bueno
- Fonte: Artigo disponível no Repositório Institucional da PUC Goiás. Link para acesso

Resumo do Trabalho: "Desenvolvimento de um Aplicativo Utilizando o Framework Flutter e Arquitetura Limpa"

Introdução

- O trabalho, publicado em **2021** por **Carlos Eduardo de Oliveira Bueno**, aborda o **desenvolvimento de um aplicativo móvel** utilizando o **framework Flutter** e os princípios da **Arquitetura Limpa**.
- A pesquisa é relevante devido à crescente adoção de smartphones e à necessidade de **arquiteturas de software bem estruturadas** para facilitar a manutenção e evolução das aplicações.
- O **objetivo geral** é **desenvolver um aplicativo** que realize a autenticação do usuário e, em caso de sucesso, o redirecione para a página inicial.
- Os **objetivos específicos** incluem: **estudar o framework Flutter e a linguagem Dart, estudar arquitetura em camadas, coesão e acoplamento, aplicar a Arquitetura Limpa, e utilizar o Android Studio como ambiente de desenvolvimento**.

Fundamentação Teórica

- O trabalho explora os conceitos de **aplicações híbridas e nativas**, destacando as vantagens do Flutter para o desenvolvimento de aplicativos híbridos.
- O estudo aborda a **arquitetura do Flutter**, incluindo seus componentes como **Embedder, Engine e Framework**, e destaca o uso de uma **interface de usuário declarativa**.
- São discutidos os conceitos de **acoplamento e coesão**, essenciais para a criação de software de fácil manutenção, e o padrão de **arquitetura em camadas**.

- A **Arquitetura Limpa**, proposta por Robert C. Martin, é apresentada como um modelo para facilitar a manutenção, testes e evolução do software, através de um sistema de camadas e baixo grau de dependência.

Metodologia

- A metodologia envolveu **pesquisa bibliográfica** sobre o Flutter, a linguagem Dart e a Arquitetura Limpa.
- Foram estudados os conceitos de **arquitetura em camadas, coesão e acoplamento**, e a implementação da aplicação envolveu a criação de protótipos.
- O desenvolvimento da aplicação prática seguiu os princípios da Arquitetura Limpa.

Desenvolvimento

- A aplicação foi estruturada em **seis camadas principais**: **Domain, Data, Infra, UI, Presentation e Validation**.
- A camada **Domain** define as **entidades e contratos de casos de uso**, a camada **Data** implementa os **casos de uso e adaptadores**, e a camada **Infra** implementa os **contratos de repositórios**, usando bibliotecas externas.
- A camada **UI** é responsável pela criação da interface do usuário, a camada **Presentation** faz a conexão entre a camada de dados e a interface do usuário, e a camada **Validation** valida informações inseridas na camada de Presentation.
- O projeto utilizou o padrão de projeto **factory** para instanciar classes e injetar dependências.

Resultados

- O trabalho resultou no desenvolvimento de um aplicativo de **autenticação de usuário** que segue os princípios da **Arquitetura Limpa**, permitindo que o usuário insira suas credenciais e seja redirecionado para a página inicial em caso de sucesso.
- A aplicação demonstra como **organizar as camadas**, definir **responsabilidades** e **minimizar dependências** entre componentes.

Contribuições

- O trabalho apresenta uma aplicação prática da **Arquitetura Limpa** em um projeto Flutter, mostrando como criar um sistema com **camadas bem definidas** e **fácil manutenibilidade**.
- O estudo auxilia no entendimento dos conceitos de **coesão, acoplamento e separação de responsabilidades** no desenvolvimento de software.

Conclusões e Trabalhos Futuros

- O trabalho conclui que a **Arquitetura Limpa é viável** para o desenvolvimento de aplicações móveis utilizando o framework Flutter, permitindo que o código seja estruturado de acordo com as necessidades do desenvolvedor, respeitando os princípios arquiteturais.
- É sugerido como **trabalho futuro** a avaliação de outras arquiteturas, como a Arquitetura de Cebola e a Arquitetura Hexagonal.

Análise Crítica

- **Pontos Fortes**: Aplicação prática da Arquitetura Limpa em um projeto Flutter, foco em conceitos de arquitetura e organização de código.
- **Limitações**: Necessidade de avaliação comparativa com outras arquiteturas.
- **Relevância para a área**: O trabalho contribui para a área de desenvolvimento de aplicações móveis, fornecendo um exemplo prático da aplicação da Arquitetura Limpa.

Palavras-chave

Arquitetura, Arquitetura Limpa, Flutter, Desenvolvimento Mobile.

Artigo 11 - Desenvolvimento e análise de duas arquiteturas de software para a operação de um serviço em nuvem destinado à contratação de profissionais de limpeza doméstica

Dados do Trabalho:

- Título: Desenvolvimento e análise de duas arquiteturas de software para a operação de um serviço em nuvem destinado à contratação de profissionais de limpeza doméstica
- Ano de Publicação: 2024
- Autoria: Nicolas Vasca Galindo
- Fonte: Artigo disponível no Repositório Institucional da UFOP. Link para acesso

Resumo do Trabalho: "Desenvolvimento e análise de duas arquiteturas de software para a operação de um serviço em nuvem destinado à contratação de profissionais de limpeza doméstica"

Introdução

- O trabalho, publicado em **2024** por **Nicolas Vasca Galindo**, aborda o **desenvolvimento e análise de desempenho de duas arquiteturas de software** para um **serviço em nuvem** destinado à **contratação de profissionais de limpeza doméstica**.
- A pesquisa é relevante devido ao **crescimento do uso de aplicativos** para serviços e à necessidade de **melhorar a busca por diaristas qualificadas**, além de aumentar a oferta de serviços para esses profissionais.
- O **objetivo principal** é **desenvolver e analisar duas arquiteturas** (monolítica e microserviços) para um aplicativo de serviços de limpeza, comparando seu desempenho.
- Os **objetivos específicos** incluem: **desenvolver o backend de um aplicativo de contratação de serviços de limpeza**, **implementar e investigar as diferenças entre as arquiteturas monolítica e de microserviços**, e **realizar análise de desempenho para avaliar o tempo de resposta de cada arquitetura**.

Fundamentação Teórica

- O trabalho se baseia em conceitos como **economia compartilhada**, **arquitetura de software** (monolítica e microserviços), **linguagens de programação** (JavaScript, Node.js), **banco de dados NoSQL** (MongoDB) e **análise de desempenho** (Apache JMeter).
- São mencionados autores e conceitos relacionados a **arquitetura de software**, **economia compartilhada**, e **análise de desempenho**.
- O trabalho se relaciona com a **área de ciência da computação**, especificamente com o **desenvolvimento de aplicações web e análise de desempenho**.

Metodologia

- Foi utilizada uma abordagem de **desenvolvimento e análise comparativa de arquiteturas de software**, com **prototipação** dos aplicativos e testes de desempenho.
- A **coleta de dados** envolveu a simulação de diferentes cargas de trabalho utilizando a ferramenta **Apache JMeter**, com variação no número de *threads* (10, 100, 1000, 2000 e 4000).
- A **análise de dados** se concentrou no **tempo de resposta** e na **porcentagem de erros** de cada arquitetura sob diferentes cargas.

Desenvolvimento

- O trabalho foi estruturado no **desenvolvimento de duas APIs**: uma com **arquitetura monolítica** e outra com **arquitetura de microserviços**.
- As APIs foram desenvolvidas em **JavaScript (Node.js)**, utilizando o **framework Express.js** e o banco de dados **NoSQL MongoDB**.
- A **prototipação** das telas foi feita no **Figma**.
- O desenvolvimento incluiu a criação de **serviços de autenticação, usuários, prestadores de serviço, endereços e agendamentos**, com funções para criar, buscar, atualizar e deletar dados.
- Para a arquitetura de microserviços, a distribuição foi feita de acordo com a demanda, atribuindo maior capacidade de processamento aos serviços mais chamados. Para a arquitetura monolítica, foi utilizada uma única máquina.
- A **infraestrutura** utilizada foi a **Amazon Web Services (AWS)**.

Resultados

- Os resultados indicam que ambas as arquiteturas possuem **tempos de resposta similares com até 100 *threads***.
- Em testes com **1000 *threads***, a **arquitetura de microserviços** apresentou **tempos de resposta menores e menor porcentagem de erros**, demonstrando sua superioridade em relação à arquitetura monolítica em termos de escalabilidade.
- As arquiteturas atingiram seus limites ao serem testadas com **2000 e 4000 *threads***, demonstrando que os 16GB de RAM foram um fator limitante.

Contribuições

- O trabalho apresenta uma **comparação prática entre duas arquiteturas de software** para um serviço em nuvem.
- O estudo demonstra a **superioridade da arquitetura de microserviços em cenários de alta carga** e fornece um guia para o desenvolvimento de sistemas com requisitos de escalabilidade.

Conclusões e Trabalhos Futuros

- O trabalho conclui que a **arquitetura de microserviços é mais adequada** para o cenário apresentado, devido ao seu melhor desempenho em situações de alta carga.
- São sugeridos como **trabalhos futuros** a implementação completa dos aplicativos utilizando a API desenvolvida, testes em maior escala, e análise de custos de cada arquitetura.

Análise Crítica

:

- **Pontos Fortes**: Abordagem prática com prototipação e testes de desempenho; comparação clara entre as arquiteturas; uso de tecnologias atuais; identificação da superioridade da arquitetura de microserviços em alta carga.
- **Limitações**: Testes limitados a 4000 *threads*; falta de análise de custos comparativa; não implementação dos aplicativos.
- **Relevância para a área**: O trabalho contribui para o desenvolvimento de aplicações web escaláveis, com foco em serviços de contratação de profissionais.

Palavras-chave

API, Arquitetura de Software, Análise de Desempenho.

Artigo 12 - Engenharia de Software Moderna

Dados do Trabalho:

- Título: Engenharia de Software Moderna
- Ano de Publicação: 2020
- Autoria: Marco Tulio Valente
- Fonte: Artigo disponível no site do autor. Link para acesso

Resumo do Trabalho:

Introdução

O livro "Engenharia de Software Moderna", de Marco Tulio Valente, publicado em 2020, aborda os principais tópicos e técnicas da engenharia de software moderna, com foco em métodos ágeis e práticas atuais de desenvolvimento de software. O livro busca suprir a lacuna de material didático atualizado na área, oferecendo um conteúdo abrangente e prático para alunos de graduação e profissionais. O objetivo principal do livro é fornecer um **texto conciso e duradouro** que cubra tanto os conceitos tradicionais quanto as técnicas modernas de engenharia de software.

Fundamentação Teórica

O livro abrange uma variedade de conceitos e teorias fundamentais da engenharia de software, incluindo:

- **Processos de desenvolvimento de software**, com ênfase em métodos ágeis como **Extreme Programming (XP), Scrum e Kanban**.
- **Engenharia de Requisitos**, incluindo técnicas como **histórias de usuários, casos de uso e Produto Mínimo Viável (MVP)**.
- **Modelagem de software**, com foco na **Unified Modeling Language (UML)**.
- **Princípios de projeto de software**, como **integridade conceitual, ocultamento de informação, coesão e acoplamento**, além de princípios **SOLID**.
- **Padrões de projeto**, incluindo **Fábrica, Singleton, Proxy, Adaptador, Fachada, Decorador, Strategy, Observador, Template Method e Visitor**.
- **Arquitetura de software**, como **arquitetura em camadas, MVC, microsserviços e arquiteturas orientadas a mensagens**.
- **Testes de software**, com discussões sobre **testes de unidade, integração e sistema**, além de testes funcionais e estruturais.
- **Refatoração**, como uma prática para melhorar a qualidade interna do código.
- **DevOps**, incluindo **controle de versões, integração contínua e deployment contínuo**.
  O livro também aborda o **Software Engineering Body of Knowledge (SWEBOK)**, que define 12 áreas de conhecimento em Engenharia de Software.

Metodologia

O livro utiliza uma **abordagem prática**, com exemplos de código e discussões de casos reais. O livro foi escrito seguindo **princípios ágeis**, com cada capítulo sendo tratado como um sprint, e disponibilizado para feedback após sua conclusão. O livro não está acoplado a nenhuma linguagem de programação específica, utilizando uma sintaxe de código neutra. Os exercícios de fixação no final de cada capítulo incentivam a aplicação prática dos conceitos. A obra também possui um site com material complementar, incluindo exercícios e roteiros de aula prática.

Desenvolvimento

O livro é estruturado em 11 capítulos e um apêndice, que abrangem os principais tópicos da engenharia de software moderna, incluindo:

- **Introdução:** Aborda definições, contexto histórico e os principais assuntos estudados na área.
- **Processos:** Apresenta métodos ágeis (XP, Scrum, Kanban) e tradicionais (Waterfall, Processo Unificado).
- **Requisitos:** Discute a engenharia de requisitos, incluindo histórias de usuários, casos de uso, MVP e testes A/B.
- **Modelos:** Explora modelos de software, com foco em diagramas UML.
- **Princípios de Projeto:** Apresenta princípios como integridade conceitual, ocultamento de informação, coesão, acoplamento e princípios SOLID.
- **Padrões de Projeto:** Apresenta diversos padrões de projeto, como Fábrica, Singleton, Proxy, Adaptador, etc

- **Arquitetura:** Discute arquiteturas como em camadas, MVC, microsserviços e orientadas a mensagens.
- **Testes:** Aborda testes de unidade, integração e sistema, além de testes funcionais e não-funcionais.
- **Refatoração:** Apresenta técnicas de refatoração para melhorar a qualidade interna do código.
- **DevOps:** Introduz o conceito de DevOps e práticas como controle de versões, integração contínua e entrega contínua.
- **Apêndice Git:** Apresenta os principais comandos do sistema de controle de versões Git.

Resultados

Os principais resultados práticos do livro incluem:

- Uma compreensão abrangente dos processos, técnicas e princípios da engenharia de software moderna.
- Conhecimento sobre métodos ágeis e como aplicá-los em projetos de software.
- Habilidade de modelar sistemas utilizando UML.
- Compreensão de princípios de projeto para criar software com alta qualidade.
- Conhecimento de padrões de projeto para resolver problemas comuns.
- Familiaridade com diferentes arquiteturas de software.
- Conhecimento sobre testes de software, refatoração e DevOps.
- Habilidade de usar o sistema de controle de versões Git.

Contribuições

As principais contribuições do livro são:

- Oferecer um material didático moderno e atualizado para cursos de engenharia de software.
- Apresentar uma visão geral da área, cobrindo desde os conceitos tradicionais até as práticas mais recentes.
- Fornecer exemplos práticos e discussões de casos reais para ilustrar os conceitos.
- Apresentar técnicas e princípios para construção de software com melhor qualidade.
- Contribuir com o ensino de engenharia de software no contexto brasileiro.
- Enfatizar a importância de práticas ágeis e de desenvolvimento de software de alta qualidade.

Conclusões e Trabalhos Futuros

O livro conclui que a engenharia de software é uma área em constante evolução, que exige aprendizado e atualização contínua. O livro aborda as principais técnicas e princípios da área, preparando os leitores para enfrentar os desafios do desenvolvimento de software moderno. O livro encoraja o uso de práticas ágeis e a importância de criar sistemas de alta qualidade. Trabalhos futuros poderiam explorar tópicos mais avançados e específicos da engenharia de software.

Análise Crítica

- **Pontos Fortes:** O livro oferece uma cobertura abrangente dos tópicos de engenharia de software, com foco em técnicas e práticas modernas. A linguagem clara e acessível facilita o aprendizado, e os exemplos práticos tornam o conteúdo mais relevante. O livro é bem estruturado, com exercícios ao final de cada capítulo, e possui um site com material complementar.
- **Limitações:** O livro não aborda em detalhes algumas áreas mais específicas, como métodos formais e aspectos econômicos da engenharia de software, que são apenas mencionados. Além disso, o livro não se aprofunda em nenhuma linguagem de programação específica.
- **Relevância para a área:** O livro é altamente relevante para estudantes de graduação e profissionais da área de engenharia de software, fornecendo uma base sólida para o desenvolvimento de sistemas de alta qualidade. A abordagem prática e a ênfase em métodos ágeis tornam o livro útil para o desenvolvimento de software moderno.

Palavras-chave

Engenharia de Software, Métodos Ágeis, UML, Padrões de Projeto, Arquitetura de Software, Testes de Software, Refatoração, DevOps, Git.

Artigo 13 - Enterprise service application architecture based on Domain Driven Model Design

Dados do Trabalho:

- Título: Enterprise service application architecture based on Domain Driven Model Design
- Ano de Publicação: 2020
- Autoria: Yu Ding, LiLing Wang, Shaokun Li, XuTong Wang e JianWei Zhang
- Fonte: Artigo disponível no IEEE Xplore Digital Library. Link para acesso

Resumo do Trabalho: Enterprise service application architecture based on Domain Driven Model Design

Introdução

- O artigo, publicado em **2020** por **Yu Ding, LiLing Wang, Shaokun Li, XuTong Wang e JianWei Zhang**, aborda a **arquitetura de aplicações de serviços empresariais baseada em Domain Driven Design (DDD)**.
- A pesquisa é relevante devido à necessidade de resolver problemas como a dificuldade em expressar as necessidades de negócios das empresas, a dificuldade em usar modelos de funcionalidades e a dificuldade em integrar sistemas de larga escala no desenvolvimento de software.
- O **objetivo principal** é propor um **esquema de design de plataforma web baseado em DDD** para projetos de integração de larga escala.
- O artigo visa fornecer um **modo geral de arquitetura de desenvolvimento de software** para o design de plataformas integradas de grande escala.

Fundamentação Teórica

- O trabalho se baseia em conceitos como **Domain Driven Design**, arquitetura em camadas, microserviços, e modelagem de domínio.
- O artigo menciona a evolução dos modelos de desenvolvimento de software, passando por modelos orientados a processos, objetos e serviços, até chegar ao desenvolvimento ágil e microserviços.
- O trabalho se relaciona com a **área de engenharia de software**, especificamente com o **design de arquiteturas de software para sistemas empresariais**.

Metodologia

- Foi utilizada uma abordagem de **análise e design de modelo de domínio**, com foco na arquitetura em camadas e no uso do .Net Entity Framework.
- O artigo propõe um esquema de design baseado em DDD, com foco na análise de flexibilidade de negócios, modelo estratégico de nível empresarial e design de base em larga escala.

Desenvolvimento

- O trabalho é estruturado na **proposta de um modelo de arquitetura em camadas baseado em DDD**, com separação de responsabilidades entre as camadas.
- As camadas incluem: **interface do usuário**, **aplicação**, **domínio** e **infraestrutura**, cada uma com responsabilidades bem definidas.
    - A **camada de domínio** é central, onde as regras de negócio são transformadas em código.
    - A **camada de infraestrutura** fornece mecanismos de persistência de dados.
- O artigo enfatiza a importância da **consistência estratégica** na construção do modelo, definindo claramente os limites de contexto, mapas de contexto e compartilhamento de núcleo.
- O design flexível é apresentado como fundamental para atender aos requisitos do usuário, com uma abordagem visual e flexível para análise de negócios.
- O artigo propõe a utilização do padrão Mediator para desacoplar a camada de aplicação da camada de API, facilitando a manutenção e o desenvolvimento.
- A implementação da plataforma inclui o uso de **CQRS (Command Query Responsibility Segregation)** para separar as operações de leitura e escrita de dados.
- O padrão **RESTful** é introduzido para padronizar a definição de interfaces.

Resultados

- O artigo apresenta um **modelo de arquitetura** que visa a melhoria na comunicação entre as equipes de desenvolvimento, melhor compreensão do negócio e maior flexibilidade e reusabilidade do código.
- A aplicação do DDD promove um entendimento claro das necessidades do usuário.
- O uso de camadas bem definidas e o desacoplamento proporcionado pelo Mediator ajudam a controlar a complexidade do software.

Contribuições

- O artigo oferece um **modelo de arquitetura geral para o desenvolvimento de plataformas integradas de grande escala**.
- A aplicação do DDD contribui para o desenvolvimento de software que atende às necessidades reais dos usuários e promove a colaboração entre desenvolvedores e designers de requisitos.
- A arquitetura proposta visa melhorar a **reusabilidade, escalabilidade e manutenção** de sistemas empresariais.

Conclusões e Trabalhos Futuros

- O estudo conclui que o **DDD é uma abordagem eficaz para lidar com a complexidade no design de sistemas empresariais**.
- A pesquisa destaca a importância de definir claramente os limites de contexto e o compartilhamento de núcleos para promover a reutilização e a escalabilidade em sistemas de larga escala.
- A arquitetura proposta é **flexível e adaptável** a diferentes cenários de negócio, e os princípios podem ser aplicados em várias plataformas.

Análise Crítica

- **Pontos Fortes**: Abordagem prática com exemplos de implementação; foco na clareza da comunicação e na compreensão do negócio; proposta de arquitetura para sistemas de larga escala; uso de padrões de design como Mediator e CQRS.
- **Limitações**: O artigo não detalha a implementação completa de um sistema real, mas fornece um modelo arquitetônico.
- **Relevância para a área**: O trabalho contribui para o desenvolvimento de aplicações empresariais mais flexíveis, escaláveis e fáceis de manter, com foco no uso de DDD.

Palavras-chave

Domain-Driven Design, Arquitetura de Software, Micro-serviços, Modelagem de Domínio, Plataforma Integrada.

Artigo 14 - Estado da arte da pesquisa em: Clean Architecture e princípios de SOLID

Dados do Trabalho:

- Título: Estado da arte da pesquisa em: Clean Architecture e princípios de SOLID
- Ano de Publicação: 2022
- Autoria: Vinicius Barros Silva Ferreira, Carlos Antônio Ferreira e Eliana Tiba Gomes Grande
- Fonte: Artigo disponível no ResearchGate. Link para acesso

Resumo do Trabalho: Estado da arte da pesquisa em: Clean Architecture e princípios de SOLID

Introdução:

- O artigo, publicado em **2022**, desenvolvido por **Vinicius Barros Silva Ferreira, Carlos Antônio Ferreira e Eliana Tiba Gomes Grande**, aborda o estado da arte da pesquisa em **Clean Architecture e princípios SOLID**, no contexto do desenvolvimento de aplicações mobile.
- A pesquisa é relevante devido à **alta demanda por serviços mobile** e à necessidade de adotar abordagens de engenharia de software que atendam aos requisitos específicos de aplicações móveis, como separação de responsabilidades em camadas e adaptação a mudanças e correções.
- O **objetivo principal** é elaborar uma **revisão bibliométrica da produção científica** sobre Clean Architecture e SOLID, disponível na base de dados Periódicos Capes, entre 2012 e 2022.
- A investigação busca identificar **lacunas de pesquisa** e contribuir com reflexões sobre a temática.
- O estudo se propõe a analisar como as abordagens de Clean Architecture e SOLID podem ser aplicadas no desenvolvimento de aplicações mobile.

Fundamentação Teórica:

- O trabalho se baseia nos princípios de **Clean Architecture** e **SOLID**, que orientam a organização de funções e estruturas de dados em classes e suas interconexões.
- O artigo aborda a importância da **arquitetura de software** para a organização, modularização e documentação do código, facilitando a compreensão dos processos, o alinhamento das soluções aos requisitos e a automação de testes.
- Os princípios **SOLID** (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) são detalhados e explicados.
- A **arquitetura em camadas** é apresentada como um padrão compatível com os princípios SOLID, que suporta o desenvolvimento incremental, a modificação e a portabilidade de sistemas.
- O **Clean Architecture** é descrito como um modelo de organização de sistemas que facilita o desenvolvimento, implantação, manutenção e integração de regras de negócio, separando as camadas do software para torná-lo mais testável, escalável e manutenível.
- A **regra da dependência**, um conceito fundamental da Clean Architecture, é abordada, destacando que as dependências do código-fonte devem sempre apontar para dentro, garantindo o desacoplamento entre as camadas.
- O artigo também menciona a **arquitetura mobile**, enfatizando a necessidade de flexibilidade, testabilidade e adaptação a diferentes tamanhos de tela, sensores e ambientes de conexão.

Metodologia:

- Foi utilizada uma abordagem **bibliométrica** com caráter **exploratório-descritivo**, e abordagem **quantitativa e qualitativa** (método misto).
- A coleta de dados foi realizada na base de dados **Periódicos Capes**, com um recorte temporal de 2012 a 2022.
- Os termos de busca utilizados foram: **clean architecture, mobile e software**.
- O processo metodológico incluiu: definição do escopo da pesquisa, definição da base de dados, aplicação dos termos de busca, análise prévia do material encontrado, exclusão do material não enquadrado, análise do material alinhado e discussão dos resultados.

Desenvolvimento:

- A pesquisa identificou **27 artigos científicos** a partir dos termos de busca, dos quais apenas **quatro** estavam alinhados ao escopo do estudo.
- Os artigos selecionados foram publicados em revistas com Qualis Capes A2 a B2 na área de pesquisa em Ciências da Computação.
- O artigo detalha os 4 artigos que se alinham ao escopo do estudo, que abordam arquiteturas de software voltadas para computação em nuvem e mobile, com foco em:
    - Arquiteturas de software para armazenamento e processamento de dados em nuvem, seguindo os princípios SOLID.
    - Abordagem arquitetônica baseada no cyber-foraging para aumentar o processamento e armazenamento de dispositivos móveis com recursos limitados.
    - Revisão de arquiteturas em cloud que suportam software como serviço para dispositivos móveis, transferência de dados para servidores em nuvem, internet das coisas e computação com foco na privacidade e segurança de dados.
    - Criação de um padrão para geração de código automática em Android com linguagem Kotlin, alinhado com as regras da Clean Architecture.
- O trabalho também descreve o padrão **Clean Dart**, uma proposta de arquitetura limpa para Flutter, que divide o código em camadas como **Presenter, Domain, Infra e External**.

Resultados:

- A análise bibliométrica revelou uma **carência de produção científica** sobre a aplicação de Clean Architecture em plataformas mobile, com foco em aspectos práticos.
- Os resultados indicam que as pesquisas sobre Clean Architecture e SOLID estão mais focadas em arquiteturas de sistemas para ambientes de nuvem e na sua adaptação para dispositivos móveis.
- Os artigos analisados destacam a importância de arquiteturas em camadas para o desenvolvimento de aplicações mobile, com separação de responsabilidades e flexibilidade para mudanças.

Contribuições:

- O artigo contribui para a compreensão do estado da arte da pesquisa em Clean Architecture e princípios SOLID, especialmente no contexto de aplicações mobile.

- A pesquisa identifica lacunas de conhecimento e sugere novas direções de pesquisa, como a aplicação de Clean Architecture em ambientes mobile com grande demanda e consumo de dados.
- O estudo destaca a importância da combinação dos princípios SOLID com a Clean Architecture para o desenvolvimento de aplicações mobile mais eficazes.

Conclusões e Trabalhos Futuros:

- O estudo conclui que há uma carência de produções científicas que contemplem o uso de abordagens arquitetônicas baseadas em Clean Architecture em plataformas mobile em um contexto prático.
- É sugerido que haja mais estudos sobre a aplicação da arquitetura em ambientes mobile, considerando a grande demanda e consumo de dados.
- O estudo reconhece suas limitações por se tratar de uma pesquisa em apenas uma base de dados, e sugere a exploração em outras bases e periódicos da área.

Análise Crítica:

- **Pontos Fortes**: Revisão bibliométrica abrangente e bem estruturada; identificação de lacunas de pesquisa; apresentação clara dos conceitos de Clean Architecture e SOLID; discussão relevante sobre arquitetura mobile.
- **Limitações**: Pesquisa limitada a uma única base de dados; poucos artigos encontrados que abordassem o tema central do estudo.
- **Relevância para a área**: O trabalho é relevante para pesquisadores e desenvolvedores interessados em arquitetura de software, especialmente no contexto de aplicações mobile, e para entender a importância da combinação dos princípios SOLID com a Clean Architecture.

Palavras-chave

Dispositivos móveis; Aplicativo; Arquitetura; SOLID; Software design.

Artigo 15 - Estudo Exploratório sobre o Design da Usabilidade Integrado ao Design da Arquitetura do Software

Dados do Trabalho:

- Título: Estudo Exploratório sobre o Design da Usabilidade Integrado ao Design da Arquitetura do Software
- Ano de Publicação: 2018
- Autoria: Alex Felipe Ferreira Costa
- Fonte: Artigo disponível no Repositório Institucional da Universidade Federal do Ceará (UFC). Link para acesso

Resumo do Trabalho: "Estudo Exploratório sobre o Design da Usabilidade Integrado ao Design da Arquitetura do Software"

Introdução:

O trabalho, intitulado **"Estudo Exploratório sobre o Design da Usabilidade Integrado ao Design da Arquitetura do Software"**, publicado em **2018** por **Alex Felipe Ferreira Costa**, explora a problemática de como a usabilidade pode ser integrada ao design da arquitetura de software, um desafio relevante devido ao impacto da usabilidade na qualidade do software. O estudo busca **investigar como os mecanismos de usabilidade podem ser modelados no design da arquitetura do software**. Os objetivos específicos incluem: analisar a adoção de diretrizes de usabilidade no design arquitetural, como essas diretrizes influenciam as decisões arquiteturais e investigar soluções alternativas. A pesquisa busca responder como os mecanismos de usabilidade podem ser representados no design da arquitetura.

Fundamentação Teórica:

A base teórica do estudo inclui conceitos de **usabilidade**, **arquitetura de software** e **Functional Usability Features (FUFs)**. A pesquisa apoia-se em autores como Shackel, Nielsen, e nas normas ISO 9241-11 e ISO 9126 para definir usabilidade. O trabalho se relaciona com a área da **Engenharia de Software**, utilizando conceitos de arquitetura de software, modelo de visão 4+1, e UML.

Metodologia:

Foi utilizada uma abordagem de **estudo exploratório**, empregando **revisão bibliográfica**, **estudo empírico** em ambiente acadêmico e **análise de dados**. Os dados foram coletados por meio de **projetos de design arquitetural** realizados por 10 equipes de estudantes e questionários sobre a percepção dos participantes. A análise envolveu **análise quantitativa** da representação dos mecanismos de usabilidade, **análise de soluções alternativas** e **análise qualitativa** dos relatos dos participantes.

Desenvolvimento:

O trabalho foi estruturado em **quatro etapas principais**: seleção de padrões de usabilidade, treinamento dos participantes, execução dos projetos e análise de resultados. Os autores adaptaram e usaram meta-modelos para representar os mecanismos de usabilidade em diagramas UML.

Resultados:

As principais descobertas incluem a identificação da **necessidade de melhorias nas diretrizes de usabilidade** e maior direcionamento para a integração da usabilidade na arquitetura de software. O estudo explorou soluções alternativas para representar a usabilidade e identificou as percepções das equipes sobre os desafios e motivações ao integrar a usabilidade na arquitetura de software. Os resultados quantitativos indicaram que os **mecanismos de usabilidade mais especificados foram alerta, abortar e ajuda multinível**, enquanto outros como agregação de comandos e passo a passo não foram especificados por nenhuma equipe.

Contribuições:

O trabalho inova ao fornecer **evidências empíricas** sobre como a usabilidade pode ser integrada no design da arquitetura, **explorando soluções alternativas** e **percepções dos participantes**. As principais contribuições são:

- identificação de dificuldades em incorporar requisitos de usabilidade.
- avaliação do uso de meta-modelos de usabilidade.
- identificação da necessidade de maior direcionamento na integração da usabilidade na arquitetura.

O estudo tem potencial impacto na área de desenvolvimento de software ao promover a criação de sistemas com melhor qualidade de uso.

Conclusões e Trabalhos Futuros:

Os autores concluem que a **incorporação dos requisitos funcionais de usabilidade ainda não é facilmente integrada ao design da arquitetura** e que há uma necessidade de maior direcionamento para a tomada de decisões na integração da usabilidade na arquitetura. São sugeridos como trabalhos futuros investigar novas soluções alternativas e aplicar o estudo na indústria. As limitações identificadas incluem o uso opcional dos meta-modelos e a experiência dos participantes.

Análise Crítica:

- **Pontos Fortes:** O estudo é bem estruturado, com uma abordagem metodológica clara. A análise dos resultados é abrangente, incluindo aspectos quantitativos e qualitativos.
- **Limitações:** O estudo é realizado em ambiente acadêmico, o que pode limitar a generalização dos resultados para a indústria. O uso dos meta-modelos foi opcional, o que pode ter influenciado as decisões das equipes.
- **Relevância para a área:** O trabalho é relevante para a área de Engenharia de Software ao fornecer evidências empíricas e um ponto de partida para futuras pesquisas sobre a integração da usabilidade no design da arquitetura.

Palavras-chave

Usabilidade, Arquitetura de Software, Design, Estudo Exploratório.

Artigo 16 - Estudo Teórico da Arquitetura de Software Model View Controller

Dados do Trabalho:

- Título: Estudo Teórico da Arquitetura de Software Model View Controller
- Ano de Publicação: 2021
- Autoria: Alice de Fátima da Fonseca Mantovani
- Fonte: Artigo disponível no Repositório da UNICAMP. Link para acesso

Resumo do Trabalho: "Estudo Teórico da Arquitetura de Software Model View Controller"

Introdução:

O trabalho, intitulado **"Estudo Teórico da Arquitetura de Software Model View Controller"**, publicado em **2021** por **Alice de Fátima da Fonseca Mantovani**, aborda a crescente complexidade dos softwares e a necessidade de padronização nos processos de criação de código. A pesquisa é relevante devido à importância de mitigar erros e produzir sistemas mais eficientes, usáveis e extensíveis. O objetivo principal é **consolidar estudos, pesquisas e projetos sobre a arquitetura de software MVC (Model-View-Controller)** e suas diversas variações. Os objetivos específicos incluem identificar, analisar e selecionar estudos sobre o padrão MVC, com ênfase em suas variações e benefícios, além de apresentar as linguagens e frameworks mais adequados ao padrão. O estudo se propõe a responder como o padrão MVC e suas variações podem ser aplicados para melhorar o desenvolvimento de software.

Fundamentação Teórica:

A base teórica do estudo inclui conceitos de **padrões de projeto**, **arquiteturas de software**, e especificamente o padrão **MVC**. A pesquisa apoia-se em autores como Trygve Reenskaug, o criador do padrão MVC, e nos conceitos de padrões de projeto de Gamma et al.. O trabalho se relaciona com a área da **Engenharia de Software**, utilizando conceitos de padrões de projeto, arquitetura de software e o modelo MVC.

Metodologia:

Foi utilizada uma abordagem de **revisão bibliográfica sistemática (RBS)**, que busca analisar artigos de uma determinada área para aprimorar ideias e construir uma base sólida de conhecimento. A metodologia emprega três fases principais: **entrada**, **processamento** e **saída**. Os dados foram coletados através de **busca em bases de dados**, como Google Acadêmico, IEEE Xplore, e bibliotecas universitárias, utilizando strings de busca específicas. A análise envolveu **aplicação de filtros** para inclusão e exclusão de artigos, leitura e análise dos artigos selecionados.

Desenvolvimento:

O trabalho foi estruturado em **seis capítulos**: introdução, metodologia, referencial teórico, ferramentas de desenvolvimento, conclusão e referências. Os autores realizaram uma revisão sistemática da literatura, analisando artigos, livros e teses sobre o padrão MVC e suas variações. A pesquisa considerou **critérios de inclusão e exclusão** para selecionar os estudos mais relevantes.

Resultados:

As principais descobertas incluem a **consolidação de informações sobre o padrão MVC e suas variações**, bem como a identificação de **linguagens e frameworks mais utilizados** que empregam o padrão. A análise revelou a importância do MVC para o desenvolvimento de software mais eficiente e fácil de manter. O estudo também catalogou diferentes interpretações do padrão MVC e suas variações, como **HMVC, MVP, MVA, MVVM e PAC**.

Contribuições:

O trabalho inova ao **reunir e analisar diversas fontes sobre o padrão MVC**, consolidando a pesquisa em um material único de consulta. As principais contribuições são:

- Apresentação detalhada do padrão MVC e suas variações.
- Identificação de frameworks e linguagens de programação que utilizam o padrão.
- Discussão sobre os benefícios e a aplicabilidade do padrão MVC.
  O estudo tem potencial impacto na área de desenvolvimento de software, ao facilitar o acesso a informações sobre o padrão MVC e suas variações.

Conclusões e Trabalhos Futuros:

Os autores concluem que o padrão MVC é fundamental para o desenvolvimento de softwares complexos, e que existem diversas variações e interpretações do padrão, que dificultam o acesso a informações claras sobre a arquitetura. São sugeridos como trabalhos futuros aprofundar a pesquisa sobre as variações do MVC e suas aplicações em diferentes contextos. As limitações identificadas incluem a variabilidade de aplicações relacionadas ao MVC e a abrangência do espaço amostral.

Análise Crítica:

- **Pontos Fortes:** O estudo utiliza uma metodologia sistemática, com critérios claros de inclusão e exclusão de artigos, e apresenta uma análise detalhada das fontes encontradas.
- **Limitações:** A pesquisa pode ser limitada pela dificuldade em acessar todas as variações do padrão MVC e pelas diferentes interpretações e aplicações do padrão.
- **Relevância para a área:** O trabalho é relevante para a área de Engenharia de Software ao fornecer uma visão geral do padrão MVC, suas variações e aplicações, e ao contribuir para a consolidação de conhecimento na área.

Palavras-chave

Padrões de projeto, MVC, arquiteturas de software.

Artigo 17 - Explorando Estilos Arquiteturais para Servir Sistemas Inteligentes

Dados do Trabalho:

- Título: Explorando Estilos Arquiteturais para Servir Sistemas Inteligentes
- Ano de Publicação: 2022
- Autoria: Washington Luiz Meireles de Lima e Ygor Tavela Alves da Silva
- Fonte: Artigo disponível no Linux IME USP. Link para acesso

Resumo do Trabalho: "Explorando Estilos Arquiteturais para Servir Sistemas Inteligentes"

Introdução:

O trabalho, intitulado **"Explorando Estilos Arquiteturais para Servir Sistemas Inteligentes"**, publicado em **2022** por **Washington Luiz Meireles de Lima** e **Ygor Tavela Alves da Silva**, aborda a questão de como construir uma arquitetura de software adequada para sistemas inteligentes, que são sistemas que evoluem e melhoram com o tempo através do aprendizado de máquina. A pesquisa é relevante devido à crescente complexidade desses sistemas e a necessidade de garantir que sejam robustos, tenham um longo tempo de vida útil e sejam de baixo custo. O objetivo principal é **explorar padrões arquiteturais para servir um sistema inteligente**, avaliando diferentes cenários e os trade-offs de cada padrão adotado. Os objetivos específicos incluem:

- Criar um sistema de benchmark.
- Realizar um estudo comparativo entre diferentes padrões de comunicação.
- Analisar os prós e contras de cada abordagem.
- Discutir a aplicabilidade de cada padrão.

O estudo se propõe a responder qual a forma mais adequada para servir um sistema inteligente, considerando a questão da escalabilidade e a comunicação entre quem serve e quem consome o modelo.

Fundamentação Teórica:

A base teórica do estudo inclui conceitos de **sistemas inteligentes**, **aprendizado de máquina**, **arquitetura de software**, **padrões de design**, **estilos arquiteturais**, e **padrões de comunicação**. O trabalho se apoia em autores como Geoff Hulten para sistemas inteligentes, Robert C. Martin para arquitetura de software e Gregor Hohpe e Bobby Woolf para padrões de comunicação. O trabalho se relaciona com a área da **Engenharia de Software** e **Inteligência Artificial**, utilizando conceitos de aprendizado de máquina para sistemas inteligentes e padrões de arquitetura de software para a construção de sistemas robustos e escaláveis.

Metodologia:

A metodologia utilizada é de natureza **exploratória e comparativa**, com foco em criar um sistema de benchmark para testar diferentes estilos arquiteturais e padrões de comunicação. O estudo envolve as seguintes etapas:

- Elaboração da proposta de trabalho.
- Revisão da literatura.
- Planejamento e implementação de um sistema de benchmark.
- Realização de um estudo comparativo.
- Elaboração da monografia.

O sistema de benchmark será desenvolvido utilizando o estilo arquitetural hexagonal, para diminuir o acoplamento entre os componentes, utilizando um modelo mock para simular modelos de aprendizado de máquina.

Desenvolvimento:

O trabalho foi estruturado em **quatro capítulos**: introdução, revisão de literatura, proposta e plano de projeto. O desenvolvimento inclui a criação de um sistema de benchmark que simula um sistema inteligente, utilizando um modelo mock para testes. Os autores planejam comparar diferentes padrões de comunicação (API REST e mensageria) para analisar como se comportam em termos de escalabilidade e desempenho.

Resultados:

Os resultados esperados incluem a **identificação dos prós e contras de cada padrão de comunicação** ao servir um modelo de aprendizado de máquina, com foco na escalabilidade. A análise comparativa deverá revelar qual padrão de comunicação é mais adequado em diferentes cenários.

Contribuições:

O trabalho inova ao **explorar padrões arquiteturais para sistemas inteligentes**, com foco em como servir modelos de aprendizado de máquina de forma escalável e eficiente. As principais contribuições são:

- Um sistema de benchmark para avaliar diferentes estilos arquiteturais e padrões de comunicação.
- Uma análise comparativa dos prós e contras de diferentes padrões de comunicação (API REST e mensageria).
- Uma discussão sobre a aplicabilidade de cada padrão em diferentes cenários.

O estudo tem potencial impacto na área de desenvolvimento de software, ao fornecer *insights* sobre como projetar sistemas inteligentes que sejam escaláveis e de fácil manutenção.

Conclusões e Trabalhos Futuros:

Os autores pretendem concluir que a **escolha da arquitetura e dos padrões de comunicação** é crucial para o desempenho e a escalabilidade de sistemas inteligentes. São sugeridos como trabalhos futuros aprofundar a análise de outros padrões de comunicação e cenários de uso, bem como avaliar o impacto de diferentes modelos de aprendizado de máquina na arquitetura do sistema.

Análise Crítica:

- **Pontos Fortes:** O trabalho aborda um tema relevante, a arquitetura de software para sistemas inteligentes, utilizando uma abordagem prática com a criação de um sistema de benchmark e a comparação de diferentes padrões de comunicação.
- **Limitações:** A utilização de um modelo mock pode não capturar todas as complexidades de um modelo real, e o estudo está limitado a uma comparação de dois padrões de comunicação (API REST e mensageria).
- **Relevância para a área:** O trabalho é relevante para a área de Engenharia de Software e Inteligência Artificial, ao fornecer informações sobre como construir sistemas inteligentes escaláveis e de fácil manutenção.

Palavras-chave

Sistemas inteligentes, arquitetura de software, aprendizado de máquina, padrões de comunicação, API REST, mensageria, escalabilidade.

Artigo 18 - Get Your Hands Dirty on Clean Architecture

Dados do Trabalho:

- Título: Get Your Hands Dirty on Clean Architecture
- Ano de Publicação: 2023
- Autoria: Tom Hombergs
- Fonte: Artigo disponível na Amazon. Link para acesso

Resumo do Livro: "Get Your Hands Dirty on Clean Architecture"

Introdução

O livro "Get Your Hands Dirty on Clean Architecture", de Tom Hombergs, publicado em 2023, aborda a construção de aplicações "limpas" usando exemplos de código em Java. O livro foca em arquitetura de software, enfatizando a **manutenibilidade** como um atributo de qualidade mais importante do que a funcionalidade, pois um sistema manutenível é fácil de mudar e adaptar. O objetivo principal é orientar os desenvolvedores na criação de software flexível e adaptável, capaz de responder a mudanças e requisitos inesperados, através da arquitetura limpa e hexagonal.

Fundamentação Teórica

O livro aborda os conceitos de **Arquitetura Limpa** (Clean Architecture) e **Arquitetura Hexagonal** (Hexagonal Architecture), também conhecida como "Ports and Adapters". Apresenta os princípios **SOLID**, incluindo o **Princípio da Responsabilidade Única** (Single Responsibility Principle - SRP) e o **Princípio da Inversão de Dependência** (Dependency Inversion Principle - DIP). Também explora como a inversão de dependências pode desacoplar a lógica de domínio das camadas de web e persistência. A obra discute padrões de design e organização de código para promover a manutenibilidade, adaptabilidade e flexibilidade dos sistemas.

Metodologia

O livro adota uma abordagem prática, utilizando um exemplo de aplicação web chamada "BuckPal" para demonstrar os conceitos de arquitetura limpa e hexagonal. A metodologia inclui a implementação de casos de uso, adaptadores web e de persistência, além de abordar testes de unidade, integração e sistema. O livro enfatiza a importância de organizar o código de forma a refletir a arquitetura do sistema, utilizando uma estrutura de pacotes expressiva e técnicas de injeção de dependência. São também discutidas estratégias de mapeamento de modelos entre diferentes camadas.

Desenvolvimento

O livro é estruturado em 15 capítulos que cobrem diversos tópicos relacionados à arquitetura de software:

- **Introdução à Manutenibilidade**: O livro começa discutindo a importância da manutenibilidade e como ela influencia a qualidade do software.
- **Problemas com Arquitetura em Camadas**: Apresenta as desvantagens de uma arquitetura em camadas tradicional, como design guiado por banco de dados e a tendência a atalhos.
- **Inversão de Dependências**: Introduz os princípios SOLID, com foco no SRP e no DIP, explicando como aplicar esses princípios para criar uma arquitetura mais flexível.
- **Organização do Código**: Aborda diferentes formas de organizar o código, incluindo a organização por camadas, por funcionalidades e uma estrutura de pacotes expressiva que reflete a arquitetura hexagonal.
- **Implementação de Casos de Uso**: Explica como implementar casos de uso, incluindo validação de entrada e regras de negócio, utilizando modelos de entrada e saída específicos.
- **Implementação de Adaptadores Web e de Persistência**: Detalha a implementação de adaptadores web e de persistência, ressaltando o uso da inversão de dependência e a separação de responsabilidades.
- **Testes de Elementos da Arquitetura**: Discute a estratégia de testes para os diferentes elementos da arquitetura, incluindo testes de unidade, integração e sistema.
- **Mapeamento entre Limites**: Explora estratégias para o mapeamento de modelos entre as diversas camadas da aplicação, incluindo estratégias como "No Mapping", "Two-Way", "Full" e "One-Way".
- **Montagem da Aplicação**: Demonstra como montar uma aplicação utilizando código simples, o classpath scanning do Spring e o Java Config do Spring.
- **Atalhos Conscientes**: Apresenta o conceito de atalhos, os problemas que eles acarretam e quando tomá-los conscientemente.
- **Reforçando os Limites da Arquitetura**: Explora como aplicar os modificadores de visibilidade, funções de aptidão pós-compilação e artefatos de construção para fazer valer os limites da arquitetura.
- **Gerenciamento de Múltiplos Contextos Delimitados**: Apresenta diferentes formas de como gerenciar múltiplos contextos delimitados na mesma aplicação.
- **Abordagem Baseada em Componentes para Arquitetura de Software**: Apresenta uma arquitetura baseada em componentes como uma alternativa, destacando a modularidade e a importância da definição de interfaces claras.
- **Decidindo sobre um Estilo de Arquitetura**: Conclui o livro abordando a questão de quando escolher um estilo de arquitetura e como evoluir o código ao longo do tempo.

Resultados

Os principais resultados práticos do livro incluem:

- Uma compreensão clara de como aplicar a arquitetura limpa e hexagonal em aplicações web.
- Uma estrutura de pacotes expressiva que reflete a arquitetura do sistema, facilitando a comunicação, o desenvolvimento e a manutenção.

- O uso de modelos de entrada e saída específicos para cada caso de uso, promovendo o desacoplamento e evitando efeitos colaterais.
- O uso da inversão de dependência para desacoplar a lógica de domínio das camadas de web e persistência.
- Uma estratégia de testes que cobre os diferentes elementos da arquitetura, incluindo testes de unidade, integração e sistema.

Contribuições

As principais contribuições do livro são:

- Uma abordagem prática e detalhada para a construção de aplicações web utilizando a arquitetura limpa e hexagonal.
- A ênfase na manutenibilidade como um atributo de qualidade essencial, influenciando a flexibilidade, a adaptabilidade e a produtividade do desenvolvedor.
- A demonstração de como aplicar o DIP para desacoplar componentes e criar sistemas mais flexíveis e adaptáveis.
- Um guia para decidir quando e como tomar atalhos conscientemente, documentando essas decisões para que possam ser reavaliadas no futuro.
- Uma discussão sobre como gerenciar múltiplos contextos delimitados em uma mesma aplicação, mantendo as fronteiras claras.

Conclusões e Trabalhos Futuros

O livro conclui que a escolha da arquitetura de software é um processo contínuo, que deve evoluir junto com as necessidades do projeto. A arquitetura hexagonal e limpa são boas opções para aplicações que precisam evoluir e se adaptar a mudanças, mas outras opções podem ser mais adequadas em certos casos. É importante que a equipe compreenda os trade-offs de cada estilo e tome decisões conscientes para criar sistemas sustentáveis e fáceis de manter.

Análise Crítica

- **Pontos Fortes:** O livro fornece uma abordagem prática e detalhada, usando exemplos de código em Java para ilustrar os conceitos de arquitetura limpa e hexagonal. A ênfase na manutenibilidade e no desacoplamento de componentes são pontos fortes.
- **Limitações:** Embora o livro explore vários aspectos da arquitetura de software, ele se concentra na arquitetura limpa e hexagonal, sem explorar outras opções em detalhes. O uso de um exemplo de aplicação web pode não abranger todos os cenários possíveis.
- **Relevância para a área:** O livro é relevante para desenvolvedores de software de todos os níveis de experiência, interessados em criar aplicações web manuteníveis, adaptáveis e flexíveis. É uma ótima fonte para quem deseja compreender a arquitetura limpa e hexagonal e suas aplicações práticas.

Palavras-chave

Arquitetura Limpa, Arquitetura Hexagonal, Ports and Adapters, Manutenibilidade, Inversão de Dependência, SOLID, Java, Testes, Modelos, Componentes.

Artigo 19 - Migração de arquitetura: Monolítico para Microsserviços usando Domain-Driven Design

Dados do Trabalho:

- Título: Migração de arquitetura: Monolítico para Microsserviços usando Domain-Driven Design
- Ano de Publicação: 2021
- Autoria: Henrique Fernando Santos Marques
- Fonte: Artigo disponível no Repositório Científico do Instituto Politécnico do Porto (recipp). Link para acesso

Resumo do Trabalho: "Migração de arquitetura: Monolítico para Microsserviços usando Domain-Driven Design"

Introdução

O trabalho "Migração de arquitetura: Monolítico para Microsserviços usando Domain-Driven Design", de Henrique Fernando Santos Marques, publicado em **Outubro de 2021**, aborda o problema da **dificuldade de gestão e compreensão da complexidade de um sistema monolítico** na empresa SPMS (Serviços Partilhados do Ministério da Saúde). A pesquisa é relevante devido à crescente evolução dos projetos da empresa, resultando em **problemas de agilidade no desenvolvimento e manutenção**. O objetivo principal é **apresentar e avaliar a migração parcial de uma arquitetura monolítica para uma baseada em microsserviços, utilizando Domain-Driven Design (DDD)**. Os objetivos específicos incluem a investigação de processos de migração e a adequação da construção de microsserviços ao design de software DDD.

Fundamentação Teórica

A base teórica do trabalho inclui:

- **Microsserviços**: Apresenta as características, desafios e padrões da arquitetura de microsserviços, como **Strangler Fig, Sidecar, camada de anticorrupção e agregação de gateway**. A comparação de desempenho entre arquiteturas monolíticas e de microsserviços é abordada, assim como o uso de microsserviços na indústria e casos de estudo.
- **Domain-Driven Design (DDD)**: Explica os conceitos de domínio, subdomínio, contextos vinculados, linguagem ubíqua, além de limitações, vantagens e desvantagens da implementação do DDD.

Metodologia

Foi utilizada uma **abordagem de estudo de caso**, focada na migração parcial da arquitetura da SPMS. A metodologia empregou:

- **Análise de valor** utilizando o modelo **New Concept Development (NCD)**.
- **Processo de seleção de ideias** através do método **Analytic Hierarchy Process (AHP)**.
- **Desenvolvimento de linguagem ubíqua** através da prática de **Event Storming**.
- **Implementação de microsserviços** com base nos princípios do DDD.
- **Avaliação** utilizando o modelo **Quantitative Evaluation Framework (QEF)**.

Desenvolvimento

O trabalho foi estruturado em sete capítulos:

1. **Introdução: Apresenta o contexto, problema e objetivos do projeto.**
2. **Estado da Arte**: Explora conceitos relacionados a microsserviços e DDD.
3. **Análise de Valor**: Apresenta a análise e proposta de valor do projeto.
4. **Design**: Aborda a estrutura atual e a nova arquitetura de microsserviços com DDD.
5. **Construção**: Descreve o processo de migração e implementação da solução.
6. **Experimentação e Avaliação**: Apresenta os indicadores e metodologias de avaliação.
7. **Conclusão**: Apresenta as conclusões, limitações e trabalhos futuros.

Os autores implementaram uma **migração parcial** de um sistema monolítico para microsserviços, utilizando o padrão **Strangler Fig**. A comunicação entre microsserviços foi estabelecida através de uma **API Gateway**. A estrutura interna de cada microsserviço foi organizada em **camadas (aplicacional, domínio e infraestrutura)**. Padrões de desenvolvimento como **injeção de dependências** e **objeto de transferência de dados (DTO)** foram aplicados.

Resultados

As principais descobertas incluem:

- A migração para uma arquitetura de microsserviços com DDD **resolveu problemas de escalabilidade e agilidade**.
- A abordagem **facilitou a compreensão e gestão das regras de negócio**.
- O uso de **Event Storming** auxiliou na criação de uma linguagem ubíqua.
- A **estrutura de camadas** nos microsserviços promoveu o desacoplamento.
- A aplicação dos princípios de DDD resultou em **código mais limpo e fácil de manter**.
- O modelo QEF confirmou que os objetivos de escalabilidade, manutenção e facilidade de gestão do domínio foram atingidos.

Contribuições

O trabalho inova ao:

- Apresentar um estudo de caso prático da migração de uma arquitetura monolítica para microsserviços usando DDD.
- Demonstrar a aplicação de padrões de desenvolvimento de microsserviços e DDD em um contexto real.
- Avaliar quantitativamente os resultados da migração através do modelo QEF.
- Evidenciar a importância da linguagem ubíqua e do Event Storming para o entendimento do domínio.
- Contribuir com o conhecimento sobre migração de sistemas monolíticos para microsserviços.

Conclusões e Trabalhos Futuros

Os autores concluem que a migração para microsserviços com DDD melhorou a escalabilidade, manutenção e gestão do domínio do projeto. São sugeridos como trabalhos futuros:

- A **continuidade da migração** para completar a transição do sistema monolítico para microsserviços.
- A exploração de **ambientes de implementação com containers (Docker, Kubernetes)**.
- Aprofundamento na **gestão do domínio** para instigar a compreensão da sua necessidade.

As limitações identificadas incluem dificuldades na transição para DDD devido à necessidade de manter o sistema monolítico e a dificuldade inicial de gestão da documentação.

Análise Crítica

- **Pontos Fortes**: O trabalho apresenta uma abordagem prática e bem estruturada para a migração de sistemas monolíticos para microsserviços. A utilização de DDD e padrões de desenvolvimento promove a qualidade e a manutenibilidade do código. A avaliação quantitativa dos resultados fortalece a validade da pesquisa.
- **Limitações**: A migração foi parcial, e a implementação de DDD teve desafios devido à necessidade de manter o sistema monolítico. A gestão da documentação foi um ponto de dificuldade inicial.
- **Relevância para a área**: O trabalho é relevante para a área de engenharia de software, pois apresenta um estudo de caso prático e avaliado sobre a migração de sistemas monolíticos para microsserviços usando DDD. A pesquisa demonstra a importância da adoção de boas práticas e arquiteturas modernas para o desenvolvimento de software.

Palavras-chave

Microsserviços, Domain-Driven Design, Escalabilidade, Domínio.

Artigo 20 - Repercusión de arquitectura limpia y la norma ISO/IEC 25010 en la mantenibilidad de aplicativos Android

Dados do Trabalho:

- Título: Repercusión de arquitectura limpia y la norma ISO/IEC 25010 en la mantenibilidad de aplicativos Android
- Ano de Publicação: 2021
- Autoria: José Francisco Arias-Orezano, Benjamín David Reyna-Barreto e Guillermo Mamani-Apaza
- Fonte: Artigo disponível no SciELO. Link para acesso

Resumo do Trabalho: "Repercusión de arquitectura limpia y la norma ISO/IEC 25010 en la mantenibilidad de aplicativos Android"

Introdução

O trabalho "Repercussão de arquitetura limpa e a norma ISO/IEC 25010 na mantenibilidade de aplicativos Android", publicado em **2021**, desenvolvido por José Francisco Arias-Orezano, Benjamín David Reyna-Barreto e Guillermo Mamani-Apaza, aborda o problema da **dificuldade em garantir a estabilidade de aplicativos móveis** durante atualizações e adições de funcionalidades. A pesquisa é relevante devido à constante evolução dos aplicativos móveis, impulsionada pelas necessidades dos usuários, tecnologia e novos dispositivos, o que torna o **mantenimento complexo**. O objetivo principal foi **estabelecer o impacto da implementação da arquitetura limpa e da norma ISO/IEC 25010 na mantenibilidade do aplicativo móvel Educar Teacher**.

Fundamentação Teórica

A base teórica do trabalho inclui:

- **Mantenibilidade:** A mantenibilidade é uma característica da qualidade interna do software, definida pela norma ISO/IEC 25010 como a **capacidade de uma aplicação de software ser modificada**. Ela é avaliada por critérios como **analisabilidade, cambiabilidade, estabilidade e testabilidade**.
- **Arquitetura Limpa:** Proposta por Robert Cecil Martin em 2012, é uma arquitetura que visa a **separação de preocupações** através de camadas de negócio e interfaces, buscando o suporte à evolução contínua de aplicações.
- **ISO/IEC 25010**: Norma que descreve a característica da mantenibilidade como a capacidade de uma aplicação de software ser modificada.

Metodologia

Foi utilizada uma **abordagem ex post facto quase experimental de corte transversal**, onde a mantenibilidade do aplicativo Educar Teacher (grupo experimental) foi comparada com a do aplicativo CRM Distribuição (grupo controle). A avaliação da mantenibilidade considerou:

- **Unidades de análise:** pacotes, classes e linhas de código.
- **Variáveis:** A variável independente foi a arquitetura limpa e a norma ISO/IEC 25010, e a variável dependente foi a mantenibilidade.
- **Métricas:** Os critérios da mantenibilidade foram medidos com base em métricas definidas pela norma ISO/IEC 25010 e pela ferramenta LogisCope.
- **Ferramentas:** A extensão MetricsReloaded para o ambiente de desenvolvimento Android Studio foi usada para coletar os resultados das métricas.

Desenvolvimento

O trabalho foi estruturado em:

- **Construção do aplicativo Educar Teacher:** Baseada na implementação "Architecture Blueprints [beta] - MVP + Clean Architecture" do repositório GitHub da Google. Foram utilizados os princípios SOLID no design orientado a objetos.

- **Avaliação da Mantenibilidade:** Avaliação das métricas de mantenibilidade em ambos os aplicativos, usando os critérios de analisabilidade, cambiabilidade, estabilidade e testabilidade, através do uso da ferramenta MetricsReloaded e análise dos valores em relação a um intervalo mínimo e máximo aceitável.
- **Cálculo dos resultados:** Cálculo das fórmulas de cada critério de mantenibilidade com base nas métricas avaliadas.

O aplicativo Educar Teacher foi desenvolvido em Java para Android, utilizando serviços Rest em C# e um banco de dados SQLite. A arquitetura limpa foi implementada com camadas de apresentação (MVP), domínio (lógica de negócio) e dados.

Resultados

As principais descobertas incluem:

- A implementação da arquitetura limpa e da norma ISO/IEC 25010 resultou em uma melhoria de **7% na analisabilidade**, com uma redução na complexidade do código.
- Houve um aumento de **56% na estabilidade**, indicando uma maior capacidade de evitar efeitos inesperados após modificações.
- A testabilidade melhorou em **0.7%**, com uma pequena diferença entre os aplicativos.
- A cambiabilidade aumentou em **0.9%**, mostrando uma ligeira melhoria na facilidade de realizar modificações.
- A métrica (CL_WMC) apresentou uma redução na complexidade, onde o aplicativo Educar Teacher teve um valor de 24.58 em comparação com o aplicativo CRM Distribuição com 26.2.
- A métrica (CL_CMOF) demonstrou que o aplicativo Educar Teacher teve um melhor grau de relação com 6.06% em comparação com 9.83% do aplicativo CRM Distribuição.
- A métrica (CL_DATA_PUBLIC) demonstrou que o aplicativo Educar Teacher teve um valor de 6.22 em comparação com o valor de 1.36 do aplicativo CRM Distribuição, indicando um maior número de atributos para facilitar a manutenção.
- A métrica (CL_FUNC_PUBLIC) demonstrou que o aplicativo Educar Teacher teve um valor de 5.59 em comparação com o valor de 2.29 do aplicativo CRM Distribuição, indicando um maior número de funções públicas para facilitar a manutenção.

Contribuições

O trabalho inova ao:

- Apresentar um estudo empírico sobre o impacto da arquitetura limpa e da norma ISO/IEC 25010 na mantenibilidade de aplicativos Android.
- Utilizar um modelo de avaliação baseado em métricas e critérios de mantenibilidade.
- Demonstrar que a arquitetura limpa pode melhorar a qualidade interna de aplicativos móveis.
- Fornecer evidências quantitativas do impacto positivo da arquitetura limpa na mantenibilidade de aplicativos.

Conclusões e Trabalhos Futuros

Os autores concluem que a implementação da arquitetura limpa e da norma ISO/IEC 25010 no aplicativo Educar Teacher teve uma **repercussão positiva na mantenibilidade**, com melhorias nos critérios de analisabilidade, estabilidade, testabilidade e cambiabilidade. Não há sugestões de trabalhos futuros específicas apresentadas no texto fornecido. As limitações identificadas incluem que as aplicações em estudo são difíceis de entender e apresentam uma baixa coesão, visto que os valores de (CL_WMC) estão fora do intervalo aceitável definido pela norma ISO/IEC 25010.

Análise Crítica

- **Pontos Fortes:** O estudo apresenta uma abordagem sistemática e rigorosa para avaliar o impacto da arquitetura limpa na mantenibilidade de aplicativos Android. A utilização de métricas e um grupo de controle fortalece a validade dos resultados.
- **Limitações:** Os valores de complexidade (CL_WMC) das aplicações estão fora do intervalo aceitável, indicando baixa coesão.
- **Relevância para a área:** O trabalho é relevante para a área de desenvolvimento de aplicativos móveis, pois demonstra o impacto da adoção de arquiteturas de software modernas e normas de qualidade na manutenibilidade e estabilidade dos aplicativos.

Palavras-chave

Aplicativos móveis, Android, Arquitetura de software, Arquitetura limpa, Qualidade de software.

Artigo 21 - Refactoring the Whitby Intelligent Tutoring System for Clean Architecture

Dados do Trabalho:

- Título: Refactoring the Whitby Intelligent Tutoring System for Clean Architecture
- Ano de Publicação: 2021
- Autoria: Paul S. Brown, Vania Dimitrova, Glen Hart, Anthony G. Cohn e Paulo Moura
- Fonte: Artigo disponível no arXiv. Link para acesso

Resumo do Trabalho: "Refactoring the Whitby Intelligent Tutoring System for Clean Architecture"

Introdução

O trabalho "**Refactoring the Whitby Intelligent Tutoring System for Clean Architecture**", publicado em 2021, desenvolvido por Paul S. Brown, Vania Dimitrova, Glen Hart, Anthony G. Cohn e Paulo Moura, aborda o **problema da arquitetura de software complexa e pouco reutilizável em um sistema de tutoria inteligente**, sendo relevante devido à necessidade de melhorar a manutenibilidade, extensibilidade e reutilização de código em projetos de software. A investigação busca **refatorar o sistema Whitby para uma arquitetura mais limpa**, tendo como objetivos específicos:

- **Desacoplar o raciocinador do Cálculo de Situação**, a estrutura de Autoria de Ontologia e a estrutura de Scaffolding Contingente em bibliotecas de terceiros.
- **Inverter as dependências** através de protocolos e categorias Logtalk.

- **Avaliar as arquiteturas** de duas iterações do Whitby em relação às motivações da refatoração.

O estudo se propõe a responder como aplicar os princípios de arquitetura limpa para melhorar a arquitetura de um sistema de tutoria inteligente.

Fundamentação Teórica

A base teórica inclui os **princípios SOLID de arquitetura limpa**, que são o resultado de debates entre desenvolvedores sobre o que torna o software mais manutenível e extensível. Os princípios SOLID incluem:

- **Princípio da Responsabilidade Única**: cada parte do código deve ter apenas uma razão para mudar.
- **Princípio Aberto-Fechado**: o código deve ser aberto para extensão e fechado para modificação.
- **Princípio da Substituição de Liskov**: partes do código devem ser intercambiáveis.
- **Princípio da Segregação de Interface**: não depender de coisas que não são usadas.
- **Princípio da Inversão de Dependência**: políticas de alto nível não devem depender de detalhes de baixo nível.

O trabalho se relaciona com a **Engenharia de Software** e utiliza conceitos de **Cálculo de Situação**, **Autoria de Ontologia**, **Scaffolding Contingente** e **Programação Lógica**.

Metodologia

Foi utilizada uma abordagem de **refatoração de software**, empregando **Logtalk** como linguagem de programação e utilizando os princípios de arquitetura limpa SOLID. Os dados foram coletados através da **análise comparativa da arquitetura** do Whitby antes e depois da refatoração. A análise envolveu a avaliação das dependências entre os componentes do sistema e a aplicação dos princípios SOLID.

Desenvolvimento

O trabalho foi estruturado em duas etapas principais:

1. **Desenvolvimento do OWLSAI**, a primeira versão do Whitby, utilizando Prolog e seu sistema de módulos.
2. **Refatoração do OWLSAI para Whitby**, utilizando Logtalk, protocolos e categorias para desacoplar os componentes e inverter as dependências.

Os autores implementaram três bibliotecas de terceiros: **SitCalc** (Cálculo de Situação), **OntologyAuthoring** (Autoria de Ontologia) e **Scaffolding** (Scaffolding Contingente). A refatoração foi guiada pelo **Princípio da Inversão de Dependência**, usando protocolos Logtalk para definir interfaces. Foi dada atenção ao **acoplamento fraco e alta coesão** dos componentes do sistema.

Resultados

As principais descobertas incluem:

- A refatoração do Whitby para Logtalk melhorou significativamente a **reutilização do código e a extensibilidade**.

- A aplicação do **Princípio da Inversão de Dependência** através de protocolos Logtalk permitiu desacoplar os componentes do sistema e inverter as dependências, tornando o código mais flexível e fácil de manter.
- O uso de protocolos permitiu criar uma **arquitetura de plugin**, onde terceiros podem estender o comportamento do sistema sem modificar o código principal.
- A refatoração resolveu problemas de acoplamento e dependências circulares presentes na versão original do Whitby (OWLSAI).
- **A versão refatorada do Whitby é mais limpa, organizada e fácil de entender do que a versão original do OWLSAI**.

Contribuições

O trabalho inova ao aplicar os princípios de arquitetura limpa SOLID em um sistema de tutoria inteligente, utilizando Logtalk para alcançar a inversão de dependências. As principais contribuições são:

- A extração de **bibliotecas reutilizáveis** (SitCalc, OntologyAuthoring e Scaffolding) para outras aplicações.
- A demonstração da aplicação do **Princípio da Inversão de Dependência** utilizando protocolos Logtalk.
- A criação de uma **arquitetura de plugin** que permite a extensão do sistema sem modificar seu código principal.
- A melhoria da **manutenibilidade, extensibilidade e reutilização** do código do Whitby.

O trabalho tem potencial impacto em áreas como **Engenharia de Software**, **Sistemas de Tutoria Inteligente** e **Programação Lógica**.

Conclusões e Trabalhos Futuros

Os autores concluem que a refatoração do Whitby para Logtalk usando os princípios SOLID resultou em uma arquitetura mais limpa, extensível e reutilizável. São sugeridos como trabalhos futuros:

- Continuar a abordar as questões arquitetônicas restantes, como a dependência do `bede` em `id generator`.
- Dividir o Whitby em microsserviços para escalabilidade.
- Explorar a reutilização das bibliotecas extraídas em outros projetos.

As limitações identificadas incluem a necessidade de algumas alterações na interface do usuário para adaptar o Whitby a novos projetos.

Análise Crítica

- **Pontos Fortes**:
    - A aplicação dos princípios SOLID e a utilização de Logtalk para alcançar a inversão de dependências resultaram em uma arquitetura bem projetada.
    - A extração de bibliotecas reutilizáveis aumenta o valor do trabalho.
    - A criação de uma arquitetura de plugin permite a extensão do sistema sem modificar seu código principal.
- **Limitações**:

    - A necessidade de algumas alterações na interface do usuário para adaptar o Whitby a novos projetos.
- **Relevância para a área**:
    - O trabalho oferece um exemplo prático de como aplicar os princípios de arquitetura limpa em um sistema de tutoria inteligente, contribuindo para a área de Engenharia de Software e Sistemas de Tutoria Inteligente.

Palavras-chave

Arquitetura Limpa, Inversão de Dependência, Logtalk, Refatoração, Sistemas de Tutoria Inteligente

Artigo 22 - Research on Software Evolution Reconstruction Based on Architecture Recovery

Dados do Trabalho:

- Título: Research on Software Evolution Reconstruction Based on Architecture Recovery
- Ano de Publicação: 2018
- Autoria: Linhui Zhong, Haitao Ye e Jing Xia
- Fonte: Artigo disponível no ResearchGate. Link para acesso

Resumo do Trabalho: "Research on Software Evolution Reconstruction Based on Architecture Recovery"

Introdução

O trabalho "**Research on Software Evolution Reconstruction Based on Architecture Recovery**", publicado em 2018, desenvolvido por Linhui Zhong, Haitao Ye e Jing Xia, aborda o **problema da recuperação da história de evolução de software**, sendo relevante devido à dificuldade em manter e reutilizar sistemas legados. A investigação busca **propor um método para recuperar a história de evolução de software** através da tecnologia de engenharia reversa da arquitetura de software, utilizando uma árvore binária evolutiva. Os objetivos específicos incluem o desenvolvimento de um algoritmo de construção de árvore binária evolutiva livre de binários e um sistema para sua construção. O estudo se propõe a responder como a arquitetura de software pode ser utilizada para recuperar a relação de evolução entre múltiplas versões de um sistema.

Fundamentação Teórica

A base teórica inclui a **arquitetura de software** como uma descrição abstrata de um sistema de software e a **engenharia reversa de software** como o processo de recuperar informações de alto nível a partir de descrições de baixo nível (código fonte). O trabalho se relaciona com a **Engenharia de Software** e utiliza conceitos de **recuperação de arquitetura de software**, **árvore evolutiva** e **árvore binária evolutiva**. O trabalho se baseia em autores como Tzerpos, Mitchell, Cai, Wang, Garcia, Medvidovic, Tonella, Sartipi, Harris, Antoniol, Jansen, Li, Servant, Aghajani e Mokni, que contribuíram com métodos e técnicas para recuperação de arquitetura e análise da evolução de software.

Metodologia

Foi utilizada uma abordagem de **engenharia reversa**, empregando o **algoritmo de agrupamento Bunch** para restaurar a visão da arquitetura de software e um **algoritmo de árvore binária evolutiva** para rastrear a evolução. Os dados foram coletados através da análise de **códigos-fonte de três sistemas open source**: Cassandra, Hbase e Hive. A análise envolveu a medição de atributos do sistema e a comparação da similaridade entre árvores binárias evolutivas.

Desenvolvimento

O trabalho foi estruturado em cinco módulos principais:

- **Módulo de leitura de código fonte**: responsável por ler o código fonte para análise.
- **Módulo de engenharia reversa de arquitetura**: utiliza o Bunch para restaurar a arquitetura de software.
- **Módulo de medição de atributos**: mede vários atributos do sistema, incluindo informações sobre componentes atômicos, número e tamanho de componentes e tamanho da arquitetura.
- **Módulo de construção de árvore binária evolutiva**: o núcleo do sistema, que utiliza os atributos medidos para construir a árvore.
- **Módulo de comparação de similaridade entre árvores binárias evolutivas**: compara a similaridade entre árvores usando um algoritmo de distância de edição de árvores.

Os autores implementaram um sistema para construir árvores binárias evolutivas, com base em atributos dos componentes e suas relações. A árvore binária evolutiva é construída a partir de um nó raiz, com ramos principais (nós esquerdos) e ramos laterais (nós direitos), e inclui nós virtuais (cópias de nós pais) e nós reais.

Resultados

As principais descobertas incluem:

- A **eficácia do método proposto** para reconstruir a história de evolução entre versões de sistemas legados.
- A identificação de **combinações de atributos** que geram árvores binárias evolutivas mais similares às árvores reais.
- Para **Cassandra**, a combinação de atributos `a;c;d;e;ab;ac;ad;bd;de;abd;ade;bde;abde` resultou na árvore mais similar com uma distância de edição de 3.
- Para **Hbase**, a combinação de atributos `a;b;c;d;e;ab;ac;ad;bd;de;abd;ade;bde;abde` resultou na árvore mais similar com uma distância de edição de 12.
- Para **Hive**, o atributo `b` resultou na árvore mais similar com uma distância de edição de 2.

Contribuições

O trabalho inova ao propor um método para recuperar a história de evolução de software baseado na arquitetura de software. As principais contribuições são:

- O desenvolvimento de um **método para recuperar a relação de evolução** entre múltiplas versões de software.
- A **criação de um algoritmo** para construção de árvores binárias evolutivas.

- A **implementação de um sistema** para realizar a recuperação da história de evolução.
- A **validação do método** através de experimentos com sistemas open source.

O trabalho tem potencial impacto na **manutenção e evolução de software**, na **engenharia reversa** e na **compreensão de sistemas legados**.

Conclusões e Trabalhos Futuros

Os autores concluem que o método proposto para construção de árvores binárias evolutivas é eficaz na recuperação da história de evolução entre versões de sistemas legados. Como trabalhos futuros, sugere-se a exploração de outros métodos de medição de similaridade e a aplicação do método em outros tipos de sistemas. As limitações identificadas incluem a dependência da qualidade do código fonte e a dificuldade em lidar com sistemas com grandes mudanças em sua arquitetura.

Análise Crítica

- **Pontos Fortes**:
  - A abordagem inovadora de usar a arquitetura de software para recuperar a história de evolução.
  - O desenvolvimento e implementação de um sistema para aplicar o método proposto.
  - A validação do método através de experimentos com sistemas open source.
- **Limitações**:
  - A dependência da qualidade do código fonte para a precisão da recuperação.
  - A dificuldade em lidar com mudanças drásticas na arquitetura de sistemas.
- **Relevância para a área**:
  - O trabalho oferece um método eficaz para a compreensão da evolução de sistemas legados, auxiliando na manutenção e reutilização de código.

Palavras-chave

Arquitetura de Software, Engenharia Reversa, Árvore Evolutiva, Árvore Binária Evolutiva, Evolução de Software

Artigo 23 - The History of Software Engineering

Dados do Trabalho:

- Título: The History of Software Engineering
- Ano de Publicação: 2019
- Autoria: Grady Booch
- Fonte: Artigo disponível no Semantic Scholar. Link para acesso
- Palestra no YouTube: Link para acesso

Resumo do Trabalho: "The History of Software Engineering"

Introdução

O texto "A História da Engenharia de Software", publicado em 2019 por Grady Booch, aborda a **evolução da engenharia de software** desde seus primórdios até os desafios contemporâneos. O artigo destaca a importância da engenharia de software como disciplina, explorando as **forças que moldaram sua história e os principais marcos de seu desenvolvimento**. O estudo busca **compreender como a engenharia de software evoluiu** e quais foram as contribuições de indivíduos e eventos.

Fundamentação Teórica

A base teórica inclui a **distinção entre ciência da computação e engenharia de software**, comparando-as às diferenças entre química e engenharia química. O texto se relaciona com a **história da computação** e com os **princípios da engenharia**, utilizando conceitos como **abstração, modularidade, e resolução de forças**. O trabalho discute as contribuições de figuras como Ada Lovelace, George Boole, Grace Hopper, e Margaret Hamilton, entre outros.

Metodologia

O texto adota uma abordagem **histórica e analítica**, examinando a evolução da engenharia de software através de exemplos e eventos. A análise envolve a **identificação de momentos cruciais e inovações** que moldaram a disciplina, desde o trabalho das primeiras "computadoras humanas" até a era da computação pessoal e da inteligência artificial.

Desenvolvimento

O trabalho foi estruturado em **três grandes períodos**:

- **Do século XIX ao século XX**: destacando a importância das "computadoras humanas" e o desenvolvimento dos primeiros computadores eletrônicos.
- **Pós-Segunda Guerra Mundial**: abordando a ascensão da computação, o nascimento da engenharia de software e o desenvolvimento de novas ferramentas e técnicas.
- **Dos anos 60 até os dias atuais**: focando na maturação da engenharia de software, incluindo o surgimento de metodologias e abordagens como programação estruturada, programação orientada a objetos, e metodologias ágeis.

Os autores descrevem a evolução das técnicas de programação e a influência de figuras-chave como Grace Hopper e Margaret Hamilton. O desenvolvimento de linguagens de programação como Fortran e Cobol, a invenção do compilador e a criação de sistemas operacionais e frameworks também são abordados.

Resultados

As principais descobertas incluem:

- A **evolução da engenharia de software** como uma disciplina distinta da ciência da computação.
- A **importância de figuras históricas**, especialmente mulheres, no desenvolvimento da computação e da engenharia de software.
- A **influência de eventos históricos** como a Grande Depressão, a Segunda Guerra Mundial, e a Guerra Fria no desenvolvimento da engenharia de software.

- A **transformação da engenharia de software** com a ascensão da computação pessoal, da internet e da inteligência artificial.
- A **importância da abstração e da modularidade** no desenvolvimento de software.

Contribuições

O trabalho inova ao **apresentar uma visão abrangente e cronológica da história da engenharia de software**, destacando a complexidade de sua evolução e as forças que a moldaram. As principais contribuições são:

- Uma **narrativa detalhada da evolução** da engenharia de software desde seus primórdios até a atualidade.
- A **identificação de figuras-chave** e seus impactos no campo.
- A **análise das forças econômicas e tecnológicas** que impulsionaram o desenvolvimento da engenharia de software.
- A **discussão dos desafios atuais e futuros** da área, especialmente no contexto da inteligência artificial.

O trabalho tem potencial impacto na **compreensão da evolução do campo**, na **valorização de contribuições históricas** e na **inspiração de futuras gerações de engenheiros de software**.

Conclusões e Trabalhos Futuros

Os autores concluem que a engenharia de software é uma disciplina complexa e em constante evolução. O texto ressalta que, apesar das mudanças tecnológicas, os **fundamentos da engenharia de software permanecem relevantes**. O artigo sugere que o futuro da engenharia de software será moldado pela inteligência artificial, computação quântica, e pela disseminação da computação em todos os aspectos da vida. O texto também enfatiza que a **qualidade do código é essencial**, independentemente da tecnologia ou abordagem.

Análise Crítica

- **Pontos Fortes**:
  - A **narrativa abrangente e detalhada** da evolução da engenharia de software.
  - O **destaque da contribuição de mulheres** na história da computação.
  - A **análise da influência de eventos históricos** e forças econômicas.
  - A **discussão dos desafios e tendências futuras** da área.
- **Limitações**:
  - O texto não aprofunda em alguns aspectos técnicos, focando mais na evolução histórica e conceitual.
  - A falta de análises específicas sobre algumas metodologias.
- **Relevância para a área**:
  - O artigo oferece uma **compreensão da evolução da engenharia de software**, auxiliando no entendimento das abordagens e práticas atuais.
  - O trabalho inspira a **inovação e a reflexão** sobre os desafios futuros da área.

Palavras-chave

Engenharia de Software, História da Computação, Abstração, Metodologias Ágeis, Inteligência Artificial

Artigo 24 - Towards a Clean Architecture for Android Apps using Model Transformations

Dados do Trabalho:

- Título: Towards a Clean Architecture for Android Apps using Model Transformations
- Ano de Publicação: 2022
- Autoria: Daniel Sanchez, Alix E. Rojas e Hector Florez
- Fonte: Artigo disponível no ResearchGate. Link para acesso

Resumo do Trabalho: "Towards a Clean Architecture for Android Apps using Model Transformations"

Introdução

O artigo "Towards a Clean Architecture for Android Apps using Model Transformations", publicado em 2022, desenvolvido por Daniel Sanchez, Alix E. Rojas e Hector Florez, aborda a **necessidade de estratégias para produzir aplicativos móveis que sejam extensíveis, escaláveis, testáveis e implantáveis de forma eficaz**. O trabalho propõe uma **abordagem que utiliza conceitos de engenharia dirigida por modelos (MDE) e arquitetura dirigida por modelos (MDA) para construir uma transformação de modelo para texto que permite gerar aplicativos Android usando Clean Architecture**. A investigação busca **automatizar a geração de código para aplicativos Android que implementam a Clean Architecture**, utilizando transformações de modelos.

Fundamentação Teórica

A base teórica inclui **conceitos de Model-Driven Engineering (MDE), transformações de modelos e Clean Architecture**. O trabalho se relaciona com a **engenharia de software**, **arquitetura de software e desenvolvimento de aplicativos móveis**. Utiliza conceitos fundamentais como **metamodelos, modelos, transformações de modelos, separação de preocupações e inversão de dependência**. O artigo menciona autores como Robert C. Martin, que propôs a Clean Architecture.

Metodologia

Foi utilizada uma abordagem de **engenharia dirigida por modelos**, empregando **transformações de modelos para gerar código fonte para aplicativos Android**. O trabalho usa uma **metamodelagem para abstrair o domínio do aplicativo**, criando um modelo que descreve as operações CRUD (Create, Read, Update, Delete). Uma **transformação de modelo para texto é usada para gerar o código Kotlin** com base no modelo criado. A metodologia também adota princípios SOLID para a arquitetura do software gerado.

Desenvolvimento

O trabalho foi estruturado em **várias etapas principais**:

- **Criação de um metamodelo** que descreve as entidades, operações e interface do usuário de um aplicativo Android.
- **Desenvolvimento de uma transformação de modelo para texto** que converte o modelo em código fonte Kotlin para Android.
- **Implementação da Clean Architecture** no código gerado, separando a lógica de negócios das preocupações de infraestrutura.
- **Utilização de componentes do Android Jetpack** e do Firebase Firestore como fonte de dados.
- **Aplicação dos princípios SOLID** na arquitetura gerada para promover flexibilidade e manutenibilidade do código.

Os autores implementaram um metamodelo que separa as definições da View e do Model. Usaram o Acceleo para definir a transformação modelo-texto, gerando código Kotlin para uma aplicação Android que se conecta ao Firestore. Foi desenvolvido um estudo de caso de um aplicativo para registro de informações de empregados de uma empresa para exemplificar a abordagem.

Resultados

As principais descobertas incluem:

- A **geração automática de código** para aplicativos Android usando transformações de modelos.
- A **implementação da Clean Architecture** no código gerado, facilitando a separação de preocupações.
- O uso de **componentes do Android Jetpack** e do Firebase Firestore para criar aplicativos funcionais.
- A **aplicação dos princípios SOLID** para melhorar a qualidade do código gerado.
- A **redução do esforço** necessário para implementar a Clean Architecture, por meio da automatização do processo de geração de código.

Contribuições

O trabalho inova ao **utilizar uma abordagem de MDE para gerar automaticamente o código fonte de um aplicativo Android baseado na Clean Architecture**. As principais contribuições são:

- A **proposta de uma metodologia** que combina MDE e Clean Architecture para o desenvolvimento de aplicativos Android.
- A **automatização da geração de código** para aplicativos Android, economizando tempo e esforço para os desenvolvedores.
- A **implementação da Clean Architecture** de forma consistente e eficiente através da geração de código.
- A **integração de boas práticas** de desenvolvimento de Android, como o uso do Kotlin, componentes do Jetpack e o Firestore.

O trabalho tem potencial impacto na **redução da complexidade do desenvolvimento de aplicativos móveis** e na **promoção do uso de boas práticas de arquitetura de software**.

Conclusões e Trabalhos Futuros

Os autores concluem que a **Clean Architecture é adequada para o gerenciamento de periféricos móveis**, pois permite a incorporação fácil de componentes separados para diferentes tipos de periféricos, escalando a solução sem afetar as regras de negócios. O trabalho também ressalta a **economia de tempo e esforço na construção das fronteiras** da Clean Architecture, através da transformação de modelos. Como trabalhos futuros, os autores planejam:

- **Implementar completamente o princípio SRP** (Single Responsibility Principle) dos princípios SOLID.
- **Realizar uma avaliação métrica** dos componentes e classes para definir o nível de responsabilidade e dependência.
- **Permitir que os usuários escolham quais entidades podem ir para diferentes componentes** para uma integração vertical dos componentes.

Análise Crítica

- **Pontos Fortes**:
  - A **abordagem inovadora** que combina MDE e Clean Architecture.
  - A **automatização da geração de código** que economiza tempo e esforço.
  - A **implementação da Clean Architecture** que melhora a manutenibilidade.
  - A **aplicação de boas práticas** de desenvolvimento de Android.
- **Limitações**:
  - Ainda **não implementa completamente o princípio SRP** dos princípios SOLID.
  - A **complexidade de permitir a definição da intenção do usuário no sistema modelado**.
  - A **limitação** que o modelo permite modelar apenas operações CRUD.
- **Relevância para a área**:
  - O artigo oferece uma **abordagem para automatizar o desenvolvimento de aplicativos Android**, reduzindo a complexidade e o tempo de desenvolvimento.
  - O trabalho **promove a adoção de boas práticas de arquitetura de software**, como a Clean Architecture.
  - A **metodologia apresentada pode ser adaptada e estendida** para outros tipos de aplicativos e tecnologias.

Palavras-chave

Model-Driven Engineering, Model Transformation, Clean Architecture, Android, Kotlin

Artigo 25 - Um estudo sobre diferentes arquiteturas para aplicações mobile na nuvem

Dados do Trabalho:

- Título: Um estudo sobre diferentes arquiteturas para aplicações mobile na nuvem
- Ano de Publicação: 2024
- Autoria: João Vitor Fidelis Cardozo
- Fonte: Artigo disponível no Repositório Institucional da UFSCar. Link para acesso

Resumo do Trabalho: "Um estudo sobre diferentes arquiteturas para aplicações mobile na nuvem"

Introdução

O trabalho "Um estudo sobre diferentes arquiteturas para aplicações mobile na nuvem", publicado em **2024**, desenvolvido por **João Vitor Fidelis Cardozo**, aborda a **demanda do mercado de software para o desenvolvimento de aplicações web e mobile, juntamente com o surgimento da computação em nuvem**. O estudo analisa e compara **três diferentes arquiteturas de software** para aplicações móveis hospedadas na nuvem, com foco em critérios específicos de avaliação. O objetivo principal da pesquisa é **analisar e comparar três arquiteturas distintas para aplicações móveis com foco em modificabilidade, confiabilidade e segurança**.

Fundamentação Teórica

A base teórica inclui **conceitos de computação em nuvem, arquiteturas de software, microsserviços, Back-end as a Service (BaaS), Database as a Service (DaaS), e serviços de mensageria**. O trabalho se relaciona com a **engenharia de software**, **desenvolvimento de aplicações móveis** e **arquitetura de sistemas distribuídos**. Utiliza conceitos fundamentais como **modificabilidade, tolerância a falhas, confidencialidade, comunicação síncrona e assíncrona, e arquitetura orientada a eventos (EDA)**. O trabalho também se baseia na norma **ISO/IEC 25010** para avaliação da qualidade de software. Autores como Robert C. Martin são mencionados.

Metodologia

Foi utilizada uma abordagem **comparativa e empírica**, empregando **análise de código-fonte, testes práticos e simulação de cenários de falha**. A coleta de dados foi realizada através da **execução do sistema em ambiente local**, com a **instalação do aplicativo em um dispositivo móvel Android** e a configuração de um **servidor Node.js e Express** para as arquiteturas que o requeriam. A análise envolveu a **avaliação de três arquiteturas distintas**, denominadas FIREBASE, FIREBASE-MYSQL e FIREBASE-MSG, em relação aos critérios de **modificabilidade, tolerância a falhas e confidencialidade**. A metodologia também considerou as **vantagens de um back-end próprio e o uso de serviços de mensageria**.

Desenvolvimento

O trabalho foi estruturado em várias etapas principais:

- **Implementação de um aplicativo móvel** denominado "Caronas Universitárias" com três arquiteturas de software distintas.
- **Arquitetura FIREBASE:** Utiliza **serviços do Google Firebase** (Firestore, Realtime Database, Storage, Autenticação) diretamente no front-end.
- **Arquitetura FIREBASE-MYSQL:** Utiliza **Firebase Realtime Database no front-end e um back-end próprio com Node.js, Express e um banco de dados relacional MySQL**.
- **Arquitetura FIREBASE-MSG:** Mantém a base da segunda arquitetura, mas com a adição do **serviço de mensageria RabbitMQ**.
- **Avaliação comparativa das três arquiteturas** com base nos critérios definidos pela ISO/IEC 25010 e outros critérios.

Os autores implementaram um aplicativo para gerenciamento de caronas com três arquiteturas diferentes, utilizando React Native e JavaScript no front-end, e Node.js e Express no back-end das arquiteturas que o requeriam. As arquiteturas foram construídas para ter o mesmo propósito, a de gerenciar caronas na mesma cidade para universitários.

Resultados

As principais descobertas incluem:

- **A arquitetura FIREBASE (primeira arquitetura)** apresentou **vantagens em termos de modificabilidade, tolerância a falhas e confidencialidade**, devido ao uso direto dos serviços do Google Firebase.
- O **acesso direto ao Firestore** aprimora a modificabilidade e oferece uma gestão mais eficiente para o tratamento de falhas, além de assegurar a confidencialidade.
- **Arquiteturas com back-end próprio (segunda e terceira arquiteturas)** fornecem **maior controle sobre a infraestrutura e o código, além de otimizar os custos a longo prazo**.
- O **uso de serviços de mensageria**, como o RabbitMQ, pode proporcionar **maior escalabilidade e alta disponibilidade de dados**.
- A **arquitetura FIREBASE (primeira arquitetura)** lida melhor com **cenários offline**, armazenando requisições em cache local.

Contribuições

O trabalho inova ao **comparar de forma empírica três arquiteturas de software para aplicações móveis**, identificando vantagens e desvantagens em cada uma. As principais contribuições são:

- **Análise detalhada dos critérios de modificabilidade, tolerância a falhas e confidencialidade** para diferentes arquiteturas de aplicações móveis em nuvem.
- **Avaliação das vantagens do uso de um back-end próprio** em comparação com serviços de nuvem.
- **Discussão sobre o uso de serviços de mensageria** para melhorar a escalabilidade e disponibilidade de aplicações.
- **Resultados empíricos** que podem auxiliar desenvolvedores na escolha da arquitetura mais adequada para seus projetos.

O trabalho tem potencial impacto ao fornecer **insights práticos para o desenvolvimento de aplicações móveis em nuvem**, considerando diferentes necessidades e restrições de projetos de software.

Conclusões e Trabalhos Futuros

Os autores concluem que **a arquitetura FIREBASE (primeira arquitetura) é mais vantajosa em termos de modificabilidade, tolerância a falhas e confidencialidade**, devido ao uso direto dos serviços do Google Firebase. No entanto, **arquiteturas com back-end próprio e serviços de mensageria (segunda e terceira arquiteturas)** oferecem **maior controle, otimização de custos e escalabilidade**. Como trabalhos futuros, sugere-se a **implementação de mecanismos de cache local para as arquiteturas com back-end próprio**, para lidar com cenários offline de forma mais eficiente.

Análise Crítica

- **Pontos Fortes**:
  - **Abordagem empírica** e comparativa das arquiteturas.
  - **Análise detalhada dos critérios de qualidade** de software.
  - **Identificação de vantagens e desvantagens** de cada arquitetura.
  - **Resultados práticos** que auxiliam na tomada de decisão sobre arquitetura de software.
- **Limitações**:
  - A análise dos critérios de comparação não foi exaustiva, considerando apenas casos específicos.
  - A avaliação de outros critérios, como back-end próprio e serviço de mensageria, não foi tão sistemática quanto os critérios principais.
- **Relevância para a área**:
  - O estudo contribui para o conhecimento sobre **arquiteturas de software para aplicações móveis em nuvem**.
  - Os resultados podem **auxiliar desenvolvedores e arquitetos de software** na escolha da arquitetura mais adequada para seus projetos.
  - A análise comparativa pode ser utilizada como **base para futuros estudos e pesquisas** na área.

Palavras-chave

Web, Mobile, Mobile Architecture, Messaging, NoSQL, SQL.

Artigo 26 - Um Método para o Desenvolvimento de Software Baseado em Microsserviços

Dados do Trabalho:

- Título: Um Método para o Desenvolvimento de Software Baseado em Microsserviços
- Ano de Publicação: 2016
- Autoria: Thiago Pereira Rosa
- Fonte: Artigo disponível no Repositório Institucional da UFC. Link para acesso

Resumo do Trabalho:

Introdução

O trabalho "Um Método para o Desenvolvimento de Software Baseado em Microsserviços", publicado em **2016**, desenvolvido por **Thiago Pereira Rosa**, aborda a **complexidade crescente no desenvolvimento de software** devido ao avanço da tecnologia e a necessidade de abordagens mais eficientes do que a arquitetura monolítica para aplicações corporativas grandes. A pesquisa propõe e avalia um método para o desenvolvimento de software baseado em microsserviços como uma alternativa. O objetivo principal é **propor e validar um método para o desenvolvimento de software baseado em microsserviços**. Os objetivos específicos incluem: (i) identificar modelos de desenvolvimento existentes baseados em microsserviços; (ii) identificar características e funcionalidades fundamentais em sistemas baseados em microsserviços; (iii) definir um método para o desenvolvimento de software baseado em microsserviços; e (iv) realizar um estudo de caso utilizando o método proposto.

Fundamentação Teórica

A base teórica inclui **conceitos de microsserviços, arquitetura de software, aplicações monolíticas, REST, computação em nuvem, resiliência e Scrum**. O trabalho se relaciona com a **engenharia de software**, **arquitetura de software** e **sistemas distribuídos**. Utiliza conceitos como **componentização, design orientado a domínio (DDD), API, URL, entrega contínua e virtualização**. Autores como Fowler e Lewis, Newman, Evans, e Martin são mencionados. O trabalho também se baseia em conceitos como o princípio da responsabilidade única e os princípios do The Twelve-Factor App.

Metodologia

Foi utilizada uma abordagem **de pesquisa aplicada**, empregando **revisão bibliográfica**, **desenvolvimento de um método** e **estudo de caso**. A coleta de dados foi realizada através da **análise da literatura existente sobre microsserviços e tecnologias relacionadas**. O desenvolvimento do método envolveu a **estruturação lógica dos artefatos da solução**, a **definição das tecnologias para auxiliar a construção de microsserviços**, o **planejamento da comunicação entre os microsserviços** e a **definição do método para construir microsserviços**. A avaliação do método foi feita através do **desenvolvimento de um estudo de caso**.

Desenvolvimento

O trabalho foi estruturado em várias etapas principais:

- **Revisão bibliográfica** para identificar modelos de desenvolvimento baseados em microsserviços.
- **Estruturação lógica dos artefatos** da solução para organização do projeto.
- **Definição de tecnologias** como Docker para auxiliar na construção dos microsserviços.
- **Planejamento da comunicação** entre os microsserviços.
- **Definição do método** para construir microsserviços, baseado no The Twelve-Factor App.
- **Implantação dos microsserviços** com Docker.
- **Realização de um estudo de caso** para avaliar o método.

Os autores implementaram um sistema para complementar o tratamento medicinal com aplicação de fototerapia, utilizando o framework Spring Boot para o desenvolvimento dos microsserviços e tecnologias como HTML, Javascript e CSS para a interface do usuário. O sistema foi dividido em microsserviços de middleware, autenticação e gerenciamento de clientes.

Resultados

As principais descobertas incluem:

- **O método proposto facilita o desenvolvimento de sistemas complexos** e confiáveis baseados em microsserviços.
- **A abordagem ágil** utilizada no método auxilia no planejamento e implementação.
- **A modularização e granulação de serviços** são pontos fortes do método.
- **O método considera aspectos cruciais** para o desenvolvimento de microsserviços, como separação de interesses, diferenciação de domínios de negócio e comunicação entre serviços.
- O estudo de caso demonstrou a **aplicabilidade e adaptabilidade do método**.
- **O uso do Docker** para implantação dos microsserviços simplifica a escalabilidade e portabilidade.
- A utilização do framework **Spring Boot** para construção dos microsserviços se mostrou uma alternativa viável.

Contribuições

O trabalho inova ao **propor um método para o desenvolvimento de software baseado em microsserviços**, que abrange desde a elicitação de requisitos até a implantação e monitoramento. As principais contribuições são:

- **Um método bem definido com nove passos** para construção de sistemas baseados em microsserviços.
- **A integração de conceitos de alta tecnologia**, computação em nuvem e escala horizontal e vertical.
- **A utilização de componentes de código aberto** para reduzir custos com licenciamento.
- **Um estudo de caso** que valida a aplicação do método.
- **Direcionamento** para a criação de serviços mais complexos, confiáveis e modulares.

O trabalho tem potencial impacto ao **facilitar o desenvolvimento de sistemas de software mais complexos, confiáveis, modulares e escaláveis**.

Conclusões e Trabalhos Futuros

Os autores concluem que o método proposto **auxilia na criação e projeto de sistemas de software complexos e confiáveis baseados em microsserviços**, considerando as fases de desenvolvimento, testes e manutenção. Como trabalhos futuros, sugere-se a **utilização do protocolo AMQP** em lugar do REST para troca de mensagens assíncronas. Além disso, recomenda-se a **exploração da programação reativa** para aplicações de baixa latência e alta disponibilidade.

Análise Crítica

- **Pontos Fortes**:
  - **Metodologia bem definida** e estruturada em passos.
  - **Integração de diversos conceitos** e tecnologias relevantes para microsserviços.
  - **Estudo de caso prático** que demonstra a aplicação do método.
  - **Discussão sobre tecnologias** consolidadas no mercado para microsserviços.
- **Limitações**:
  - O método pode ser considerado complexo e **pesado para aplicações mais simples**.
  - Pode exigir **camadas adicionais de tecnologia** para suportar a execução completa do sistema.
  - A **heterogeneidade tecnológica** pode aumentar a complexidade.
- **Relevância para a área**:
  - O trabalho contribui para o conhecimento sobre **desenvolvimento de software baseado em microsserviços**.
  - O método proposto pode ser **útil para desenvolvedores e arquitetos de software** que desejam adotar essa abordagem.
  - As **sugestões de trabalhos futuros** indicam direções para novas pesquisas na área.

Palavras-chave

Software como Serviço, Desenvolvimento de Software, Sistema Monolítico, Arquitetura orientada a Serviços.

Artigo 27 - Uma Avaliação Qualitativa de Estilos Arquiteturais a partir das Características da Arquitetura Evolutiva

Dados do Trabalho:

- Título: Uma Avaliação Qualitativa de Estilos Arquiteturais a partir das Características da Arquitetura Evolutiva
- Ano de Publicação: 2022
- Autoria: Agner Esteves Ballejo e Wilson Vendramel
- Fonte: Artigo disponível na Revista de Tecnologia da FATEC Ourinhos. Link para acesso

Resumo do Trabalho:

Introdução

O trabalho "Uma Avaliação Qualitativa de Estilos Arquiteturais a partir das Características da Arquitetura Evolutiva", publicado em **2022**, desenvolvido por **Agner Esteves Ballejo e Wilson Vendramel**, aborda a necessidade de **adaptar constantemente os sistemas de software devido à sua evolução contínua**, tornando crucial a adoção de padrões e técnicas que facilitem a manutenção. A pesquisa visa **avaliar qualitativamente diferentes estilos arquiteturais** com base nas características da arquitetura evolutiva, que permite mudanças mais fáceis e sustentáveis. O objetivo principal é **realizar uma avaliação qualitativa de estilos arquiteturais**

a partir das características da arquitetura evolutiva. Para tal, foi realizada uma pesquisa com profissionais de desenvolvimento de software para avaliar a aderência das técnicas estudadas à prática.

Fundamentação Teórica

A base teórica inclui **conceitos de evolução de software, estilos arquiteturais, arquitetura evolutiva, quantum arquitetural, acoplamento apropriado, mudanças incrementais e mudanças guiadas**. O trabalho se relaciona com a **engenharia de software** e **arquitetura de software**, utilizando conceitos como **modificabilidade, requisitos não funcionais, design de software, integração contínua (CI), entrega contínua (CD), e fitness functions**. Autores como Lehman, Sommerville, Bass, Pressman, Maxim, Ford, Fowler, Lewis e Garlan são mencionados. O trabalho também se baseia no conceito de que a arquitetura de software atua como uma ponte entre os objetivos de negócio e o software resultante.

Metodologia

Foi utilizada uma abordagem de **pesquisa qualitativa** com **propósito exploratório**, empregando **pesquisa bibliográfica** e **questionário do tipo Survey**. Os dados foram coletados através de um **questionário com 15 questões**, aplicado a **44 profissionais de desenvolvimento de software**. A análise envolveu a **interpretação das respostas** dos profissionais, com foco na **experiência com os estilos arquiteturais estudados**. Os dados foram analisados com base nos conceitos da arquitetura evolutiva.

Desenvolvimento

O trabalho foi estruturado em **cinco seções principais**:

- **Introdução:** Apresentação do contexto e objetivos da pesquisa.
- **Estilos Arquiteturais**: Apresentação e descrição dos quatro estilos arquiteturais avaliados: monolítico, microsserviços, baseado em eventos e serverless.
- **Características da Arquitetura Evolutiva**: Explicação das três características da arquitetura evolutiva: acoplamento apropriado, mudanças incrementais e mudanças guiadas.
- **Aplicação da Pesquisa**: Apresentação do perfil dos respondentes e análise dos resultados do questionário.
- **Conclusão**: Síntese dos resultados e discussão sobre o impacto da arquitetura evolutiva no desenvolvimento de software.

O trabalho detalha os quatro estilos arquiteturais estudados. O estilo **monolítico** é caracterizado por ter toda a aplicação em um mesmo artefato. O estilo de **microsserviços** envolve pequenos serviços independentes trabalhando juntos. O estilo **baseado em eventos** é uma arquitetura distribuída e assíncrona. O estilo **serverless** possibilita a implantação de aplicações na nuvem sem a necessidade de gerenciar servidores. As características da arquitetura evolutiva incluem: **acoplamento apropriado**, que busca a redução do acoplamento entre módulos, **mudanças incrementais**, que promovem a introdução de novas funcionalidades em pequenos lotes, e **mudanças guiadas**, que utilizam *fitness functions* para guiar o processo de desenvolvimento e garantir a manutenção das características da arquitetura.

Resultados

As principais descobertas incluem:

- **Microsserviços** foram considerados o estilo com maior aderência às características da arquitetura evolutiva.
- **Monolítico** foi o estilo com menor aderência devido ao seu acoplamento e dificuldades em mudanças incrementais.
- Os estilos **baseado em eventos e serverless** mostraram aderência similar, ambos com acoplamento apropriado e facilidade para mudanças incrementais.
- A pesquisa indicou que quanto **menor o quantum do sistema**, maior a possibilidade de um acoplamento apropriado, mudanças incrementais e mecanismos de mudanças guiadas, como as *fitness functions*.
- Os profissionais concordam que é possível criar *fitness functions* para todos os estilos, embora seja mais fácil para microsserviços.

Contribuições

O trabalho inova ao **avaliar estilos arquiteturais com base nas características da arquitetura evolutiva**, fornecendo *insights* sobre a sustentabilidade e a adaptabilidade dos estilos de arquitetura para mudanças. As principais contribuições são:

- Uma **avaliação qualitativa** de quatro estilos arquiteturais (monolítico, microsserviços, baseado em eventos e serverless).
- Uma análise da aderência dos estilos arquiteturais às características da arquitetura evolutiva (acoplamento apropriado, mudanças incrementais e mudanças guiadas).
- *Insights* sobre o **impacto do tamanho do quantum** na evolução do software.
- **Resultados que corroboram** a importância de um acoplamento apropriado, mudanças incrementais e *fitness functions* para a sustentabilidade da arquitetura.

O trabalho tem potencial impacto ao **orientar a escolha de estilos arquiteturais** que promovam a evolução sustentável dos sistemas de software, com maior facilidade de manutenção e custos mais baixos.

Conclusões e Trabalhos Futuros

Os autores concluem que a **combinação adequada entre estilos arquiteturais e técnicas da arquitetura evolutiva** resulta em arquiteturas sustentáveis e abertas a mudanças. Como trabalhos futuros, sugerem **analisar, projetar e implementar arquiteturas** a partir de um ou mais estilos arquiteturais, avaliando qualitativamente e/ou quantitativamente se os conceitos e técnicas da arquitetura evolutiva propiciam a construção e evolução de sistemas de software mais sustentáveis.

Análise Crítica

- **Pontos Fortes**:
    - **Metodologia clara** e bem definida para avaliar os estilos arquiteturais.
    - **Análise qualitativa** baseada na experiência de profissionais da área.
    - **Discussão das características da arquitetura evolutiva** e sua aplicação prática.
    - **Apresentação dos resultados** de forma clara e objetiva.

- **Limitações**:
  - **Amostra de pesquisa** limitada a 44 profissionais, o que pode não representar todos os cenários.
  - A **natureza qualitativa da pesquisa** limita a generalização dos resultados.
- **Relevância para a área**:
  - O trabalho contribui para o conhecimento sobre **arquitetura de software e sua evolução**.
  - Fornece **insights práticos** para profissionais da área sobre a escolha de estilos arquiteturais.
  - Os resultados podem ser **usados para guiar o desenvolvimento** de sistemas de software mais sustentáveis.

Palavras-chave

Evolução de software, Estilos arquiteturais, Arquitetura evolutiva, Quantum arquitetural, Acoplamento apropriado.

Artigo 28 - Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes

Dados do Trabalho:

- Título: Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes
- Ano de Publicação: 2007
- Autoria: Tiago César Moronte
- Fonte: Artigo disponível na Biblioteca Digital Brasileira de Teses e Dissertações. Link para acesso

Resumo do Trabalho:

Com base nas fontes fornecidas e na estrutura definida, o resumo do trabalho "Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes" será apresentado nos seguintes pontos:

Introdução

O trabalho, "Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes", publicado em **2007**, desenvolvido por **Tiago César Moronte**, aborda a **construção de arquiteturas de software baseadas em componentes (DBC)**, um problema relevante devido à falta de consenso entre os conceitos, termos e definições utilizados nas abordagens de arquitetura de software e DBC. A investigação busca **apresentar uma infra-estrutura de software** que apoie a construção de arquiteturas de software baseadas em componentes desde a sua especificação até a sua materialização em forma de código. O objetivo principal é **formalizar e relacionar os conceitos presentes nas abordagens de arquitetura de software e DBC** através de um metamodelo conceitual.

Fundamentação Teórica

A base teórica inclui **arquitetura de software, desenvolvimento baseado em componentes (DBC), estilos arquiteturais, ADLs (Architecture Description Languages), o modelo de componentes COSMOS e o ambiente Eclipse**. O trabalho se relaciona com a **engenharia de software** e utiliza conceitos como **componentes de software, conectores arquiteturais, interfaces providas e requeridas, reutilização de software, e verificação de arquitetura**. Autores como Cheesman e Daniels são mencionados.

Metodologia

Foi utilizada uma abordagem de **desenvolvimento de software**, empregando **modelagem conceitual, projeto arquitetural e implementação de software**. Os dados foram coletados através da **análise de requisitos, casos de uso e especificação de componentes**. A análise envolveu a **validação da infra-estrutura de software** por meio de estudos de caso.

Desenvolvimento

O trabalho foi estruturado em **seis capítulos principais**:

- **Introdução**: Apresentação do contexto e motivação do trabalho.
- **Fundamentos de Arquitetura de Software e DBC**: Definição dos principais termos e conceitos.
- **Um Metamodelo Integrado para Arquitetura de Software e DBC**: Apresentação do metamodelo conceitual.
- **Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Componentes**: Apresentação das ferramentas e tecnologias utilizadas.
- **Estudos de Caso de Uso da Infra-estrutura de Software Proposta**: Avaliação da infra-estrutura através de três estudos de caso.

Conclusões e Trabalhos Futuros: Conclusões e sugestões para trabalhos futuros.

Os autores implementaram uma **infraestrutura de software**, denominada **Bellatrix**, que consiste em um conjunto de ferramentas para modelagem, verificação e geração de código para arquiteturas de componentes. A infraestrutura foi construída sobre o ambiente **Eclipse**. O metamodelo conceitual foi organizado em quatro partes: um metamodelo conceitual de arquitetura de software, um metamodelo conceitual para DBC, um metamodelo para configuração arquitetural de componentes e um metamodelo para estilos arquiteturais. O modelo de componentes utilizado é o **COSMOS**, que materializa os conceitos de arquitetura de software em uma linguagem de programação (Java).

Resultados

As principais descobertas incluem:

- A criação de um **metamodelo conceitual integrado** para arquitetura de software e DBC.
- O desenvolvimento de uma **infraestrutura de software** que apoia a construção de arquiteturas de componentes.
- A implementação de **ferramentas para modelagem, verificação e geração de código**.
- A **validação da infraestrutura através de estudos de caso**.

- A **geração automática de código** seguindo o modelo COSMOS.

A análise revelou a importância de **ferramentas que auxiliem a modelagem de arquiteturas de componentes**, com capacidade para verificar a conformidade da arquitetura com os estilos arquiteturais.

Contribuições

O trabalho inova ao **integrar as abordagens de arquitetura de software e DBC** por meio de um metamodelo conceitual, formalizando e relacionando seus conceitos. As principais contribuições são:

- Um **metamodelo conceitual integrado** para arquitetura de software e DBC.
- Uma **infraestrutura de software** que apoia a construção de arquiteturas de componentes desde a especificação até a geração de código.
- Um **conjunto de ferramentas** para modelagem, verificação e geração de código.
- A **utilização do modelo COSMOS** para a geração de código.
- **Validação da infraestrutura** através de estudos de caso.

O trabalho tem potencial impacto ao **facilitar o desenvolvimento de sistemas de software baseados em componentes**, com maior qualidade e menor custo.

Conclusões e Trabalhos Futuros

Os autores concluem que a infraestrutura de software **auxilia a construção de arquiteturas de software baseadas em componentes**, desde a modelagem até a materialização em código. Como trabalhos futuros, sugerem a implementação de outras tecnologias e modelos de componentes para a geração automática de código, melhorias na exibição da arquitetura de componentes, e integração com ferramentas UML. As limitações identificadas incluem a complexidade do ambiente Eclipse e a falta de documentação da biblioteca AcmeLib.

Análise Crítica

- **Pontos Fortes**:
    - **Metamodelo conceitual** que integra arquitetura de software e DBC.
    - **Infraestrutura de software** com ferramentas para todo o ciclo de vida da arquitetura.
    - **Geração automática de código** em modelo COSMOS.
    - **Validação** por meio de estudos de caso.
- **Limitações**:
    - **Complexidade do ambiente Eclipse**.
    - **Dificuldade na organização e visualização** da arquitetura de componentes.
    - **Limitação na geração de código** para classes Java e exceções.
- **Relevância para a área**:
    - O trabalho contribui para o desenvolvimento de **sistemas de software mais eficientes e reutilizáveis**.
    - Fornece **ferramentas práticas** para arquitetos e desenvolvedores de software.
    - Promove a **integração de conceitos de arquitetura de software e DBC**.

Palavras-chave

Arquitetura de software, Desenvolvimento baseado em componentes, Metamodelo, COSMOS, Eclipse.