UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

OTÁVIO RODRIGUES BAMBANS

Relatório final:
Gerador automático de código (programas) baseado em grafos de conhecimento

São Paulo
2023

OTÁVIO RODRIGUES BAMBANS

Relatório final:
Gerador automático de código (programas) baseado em grafos de conhecimento

Relatório final de atividades de Iniciação Científica apresentado à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo.

Orientador: Prof. Dr. Fábio Nakano

Coorientador: Prof. Dr. José de Jesús Pérez Alcázar

São Paulo

2023

Resumo

Esta pesquisa de iniciação científica[1] a explorou a automação da parametrização e construção de código através do uso de grafos de conhecimento. O objetivo principal é desenvolver um gerador de código baseado em grafos de conhecimento, que pode ser considerado uma extensão dos tradicionais geradores de código baseados em modelos. Neste novo enfoque, modelos (fragmentos de programas) e dados são armazenados em um banco de dados orientado a grafos, possibilitando o uso, gerenciamento e interoperabilidade dos dados com base em conceitos de IoT e padrões da Web Semântica.

A pesquisa busca contribuir para a agilização da programação e implantação de serviços em Cidades Inteligentes. Ao automatizar a parametrização e a construção de código, os desenvolvedores podem se concentrar em tarefas mais criativas e de alto nível, tornando o processo de desenvolvimento mais eficiente e eficaz.

O uso de grafos de conhecimento oferece vantagens significativas, permitindo que o sistema seja mais adaptável e flexível. Além disso, o armazenamento de modelos e dados em um formato de grafo possibilita a exploração e reutilização de conhecimento de maneira mais intuitiva e poderosa.

Ao final do estudo, o gerador de código desenvolvido proporciona melhorias tangíveis na forma como os serviços são concebidos e implementados em Cidades Inteligentes, levando a uma maior eficiência e escalabilidade do ecossistema tecnológico urbano. A pesquisa representa um passo importante em direção a um futuro onde a automação e a inteligência artificial são elementos-chave na criação de cidades cada vez mais conectadas e inteligentes.

Palavras chave: Web Semântica, Internet das Coisas, Gerador Automático de Programas, Grafo de Conhecimento, W3C RDF, W3C SPARQL.

[1] Com financiamento PUB-USP 2022-2023: projeto 906/2022.

Abstract

This scientific research[2] explored the automation of parameterization and code generation through the use of knowledge graphs. The main objective is to develop a code generator based on knowledge graphs, which can be considered an extension of traditional model-based code generators. In this new approach, models (program fragments) and data are stored in a graph-oriented database, enabling the use, management, and interoperability of data based on IoT concepts and Semantic Web standards.

The research aims to contribute to streamlining programming and service deployment in Smart Cities. By automating parameterization and code generation, developers can focus on more creative and high-level tasks, thereby making the development process more efficient and effective.

The use of knowledge graphs offers significant advantages, allowing the system to be more adaptable and flexible. Additionally, storing models and data in a graph format enables the exploration and reuse of knowledge in a more intuitive and powerful manner.

At the end of the study, the developed code generator provides tangible improvements in how services are conceived and implemented in Smart Cities, leading to greater efficiency and scalability of the urban technological ecosystem. The research represents a significant step towards a future where automation and artificial intelligence play key roles in creating increasingly connected and intelligent cities.

Keywords: Semantic Web, Internet of Things, Automatic Program Generator, Knowledge Graph, W3C RDF, W3C SPARQL.

[2] With funding from PUB-USP 2022-2023: project 906/2022.

Sumário

Gerador automático de código (programas) baseado em grafos de conhecimento

1 Introdução

1.1 Cidades Inteligentes e Internet das Coisas

Com o crescimento das demandas admistrativas, surgiu nos dias de hoje o conceito de "Cidades Inteligentes" (BELLINI; NESI; PANTALEO, 2022). Esta pode ser definida como uma concentração urbana (*id est*, cidade) na qual métodos computacionais são usados para que exista algum nível de automação e automatização sobre a tomada de decisões, dada uma coleta de dados através de sensores e alguma ação no ambiente através de atuadores. Não obstante, em se tratando de Inteligência de Sistemas, é possível que sejam usadas aplicações com Inteligência Artificial e métodos estatísticos para previsão de cenários futuros, predição de comportamentos percebidos no ambiente (*exempli gratia*, reconhecimento de padrões) e tomadas de decição em tempo real - todos como tentativas de explicação de situações reais ocorridas no ambiente o qual está inserido o sistema.

Considerando uma grande automatização de coleta de dados, processamento de informações e, portanto, tomada de decisões, sistemas computacionais mais complexos e redes de comunicação mais robustas tornaram-se imprescindíveis.

No entanto, o desenvolvimento de aplicações nesse contexto torna-se substancialmente mais difícil, ao passo que o ciclo de vida dos programas passa a ser prejudicado pela alta dinamicidade que os sistemas podem apresentar.

Sendo assim, torna-se evidente o que hoje é conhecido como Internet das Coisas - IoT - (ORACLE, 2022). Nela, considerando uma grande quantidade de dispositivos (*i.e.*, microcomputadores, sensores, atuadores, dispositivos de comunicação *etc.*) a serem programados e coordenados, crescendo em um ritmo extremamente acelerado, a gestão individual deles torna-se quase impossível. Para tanto, a construção de sistemas inteligentes (centralizados ou não) passou a ser imprescindível, uma vez que:

- A automação se torna necessária na distribuição dos serviços desempenhados pelos nós e na forma de tratamento da coleção de dados percebida pelo sistema;
- A implantação e manutenção do sistema, considerando um cenário como uma atualização de software dos nós, precisa de métodos automáticos de implantação, uma vez que a implantação individual é, por diversos motivos como geográficos, quantitativos *et cetera*, impossibilitada de ser desempenhada;

- A heterogeneidade dos componentes do sistema pode, por vezes, necessitar de interfaces de homogeneização, que reduzam a dificuldade de implementação dos serviços e o esforço empregado na implantação dos componentes.

Por fim, há também uma grande demanda computacional no que concerne o processamento de uma grande quantidade de dados e de protocolos eficientes de distribuição e roteamento dos dados coletados. Não só, estes diversos sistemas necessitam de ferramentas robustas para persistência e recuperação dos dados coletados. Para tanto, considerando os devidos propósitos deste trabalho, foram usadas aquelas baseadas em recursos semânticos, *id est*, que fazem uso do significado que as informações inferidas e coletadas trazem consigo para que fossem extraídas características dos serviços e, assim, agilizar de alguma maneira alguma(s) implantação(ões).

1.2 Domínios de aplicações em IoT

Em revisões da literatura sobre a composição de aplicações baseadas em Internet das Coisas são tratados os domínios sobre os quais usualmente são criadas tais aplicações.

Na revisão sistemática (BELLINI; NESI; PANTALEO, 2022), é mostrado que o desenvolvimento de novas tecnologias não deve ser redundante, alinhando, se possível, com a Agenda 2030 das Nações Unidas, permitindo que tecnologia e meio ambiente andem lado a lado. Para tal, foram extraídos 8 domínios de problemas nos quais, em seus contextos, é possível que sejam desenvolvidas soluções baseadas em IoT, sendo eles:

- Governança: como Governo Eletrônico, políticas de decisão colaborativas e engajamento cívico;
- Vida e infraestrutura: edificações, lares, educação, turismo, atividades culturais *etc.* inteligentes;
- Mobilidade e transporte: gestão de tráfego, roteamento dinâmico, estacionamentos inteligentes, mobilidade sustentável, veículos e mobilidade compartilhada;
- Economia: negócios inteligentes, comércio eletrônico, atacado e varejo inteligentes (vendas), *marketplaces*, plataformas de serviços;
- Indústria e produção: indústria 4.0, manutenção preventiva, manufatura e agropecuária inteligente;
- Energia: geração e manejo sustentável de energia;

- Meio ambiente: monitoramento inteligente;
- Saúde: hospitais inteligentes, monitoramento da saúde, telemedicina *etc.*

O domínio dos ambientes inteligentes inclui coleção de dados, monitoramento e análise de eventos do tempo e climáticos, sendo estes, portanto, os dados para possíveis tomadas de decisão ou análise do comportamento do ambiente no qual o sistema está inserido.

Entretanto, um dos maiores desafios encontrados por desenvolvedores é compreender o domínio do problema sobre o qual se está atuando. Por exemplo, na área da saúde há uma demanda por softwares que agilizem o diagnóstico médico, porém o domínio de conhecimento sobre tais diagnósticos cabe ao médico e não necessariamente ao desenvolvedor daquele determinado sistema. Sendo assim, uma das proposições feitas é a de se apresentar alguma parametrização a algum gerador de código para que um ou alguns de seus modelos se ajustem ao domínio exigido.

1.3 Arquiteturas comuns em aplicações em IoT

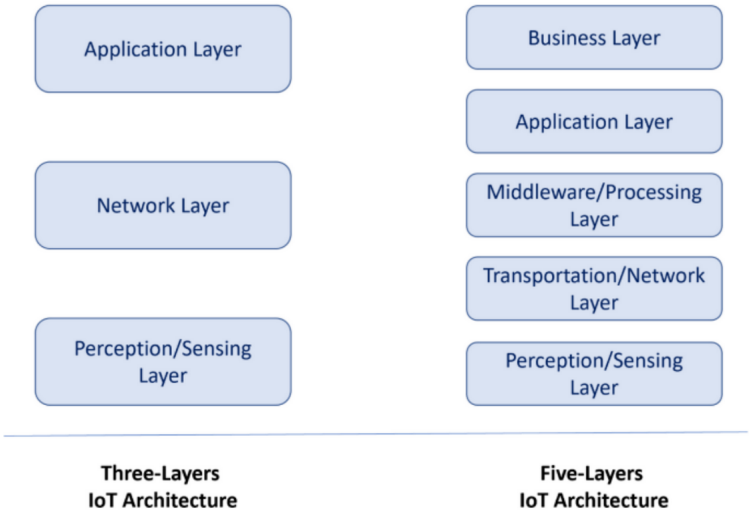
A internet das coisas trouxe um novo paradigma de comunicação digital, no qual há um sistema inteligente (*i.e.*, um sistema que contém ao menos um elemento computacional, coletando informações ou atuando sobre o ambiente, conectado a alguma rede de comunicação) na interação entre dispositivos, na relação entre o mundo físico com os humanos ou sobre interações sobre o meio-ambiente *etc.*

Muito se vê na literatura a respeito da arquitetura usada na construção de sistemas baseados em Internet das Coisas. Duas representações diferentes da organização e transporte dos dados, em modelos que versam sobre 3 e 5 camadas de abstração são apresentados na Figura 1.

Como o modelo de 3 camadas é uma representação simplificada do modelo de 5 camadas, versando sobre o segundo, tem-se (da camada de mais baixo nível para a camada de mais alto nível):

- Camada de percepção/sensorial: é a camada mais perto do mundo físico. Aqui, é onde o sensor ou atuador é capaz de atuar com outros agentes de sua natureza ou de natureza diferente, recebendo ou enviando dados desta interação, explorando redes sem fio;

Figura 1 – Arquiteturas comuns em IoT.



Fonte – (BELLINI; NESI; PANTALEO, 2022)

- Camada de transporte e rede: é a camada que trata do roteamento e transmissão dos dados dos sensores/atuadores. É aqui que se permite a comunicação entre todos os nós e um servidor, por exemplo;
- Middleware/Camada de processamento: camada de agregamento, processamento e tratamento dos dados dos sensores/atuadores, transportados pela camada de transporte, uma vez que são vindouros de dispositivos heterogêneos, com diferentes protocolos, originalmente não projetados para comunicação entre si. Sua função é permitir a interoperabilidade entre os dispositivos conectados. A interoperabilidade possui suas próprias camadas de abstração:
  - Nível técnico: alcança a eficiência máxima na comunicação end-to-end;
  - Nível sintático: gerenciamento de diversos protocolos e formatos;
  - Nível semântico: ou nível Web Semântico, busca a representação não ambígua dos dados, aumentando a expressividade do sistema.
- Camada de aplicação: é a camada que constitui os formatos de saída, aplicações e serviços do sistema para os usuários finais - Explorando os pushing protocols, há uma grande adoção das aplicações orientadas a eventos;
- Camada de negócios: é a camada final, a qual trata das ferramentas front-end, consumindo os dados vindos da camada de aplicação, produzindo as aplicações de big-data, serviços de visualização, CRMs etc., suportando as ferramentas de

análise de risco, decisão, entre outras, específicas para os usuários que consumirão os resultados de todas as camadas anteriores.

Não só, existem ainda diferentes tipos de computação, cada qual com suas particularidades. Desses modelos de computação distribuída, temos: Edge, Fog e Cloud Computing. Internet das Coisas está mais para Fog Computing, uma vez que o poder de processamento está individualizado, em geral, em cada dispositivo conectado à rede - de fato isto é importante, uma vez que é possível que o tráfego na rede seja substancialmente reduzido.

Ainda mais, do objetivo posteriormente discutido neste trabalho, observa-se uma atuação no nível da camada de processamento e de aplicação, uma vez que o nível semântico e os formatos de saída estão definidos nestas duas camadas chaves para o desenvolvimento do gerador de código especificado.

1.4 Entraves no desenvolvimento de aplicações em IoT

Como apresentado por (PATEL; CASSOU, 2015), em um ponto de vista sobre a cooperação de objetos ditos inteligentes - *id est*, objetos que se comunicam entre si, adquirindo ou enviando informações - na construção de sistemas baseados em IoT, tem-se grandes dificuldades para tal tipo de desenvolvimento. Dentre eles, tem-se:

- Separação de regras: as aplicações em IoT demandam uma certa variedade de conhecimento para serem desenvolvidas. No entanto, não é verdade que todos os desenvolvedores possuem todo o conhecimento do domínio e, também, não se pode afirmar que todos possuem as mesmas habilidades para a apresentação da solução.
- Heterogeneidade[1]: a portabilidade de código é algo dificultoso, haja vista que os diferentes tipos de dispositivos, modos de operação, plataformas etc. pedem soluções específicas;
- Escalabilidade: a arquitetura de sistemas em IoT modernos são parte da área de conhecimento de sistemas distribuídos. A escalabilidade é dificultada por dois fatores principais, no que concerne a orquestração:

  1 Heterogeneidade: naturezas diversas dos dispositivos;

  2 Quantidade: coordenação de uma grande quantidade de dispositivos.

[1] Não só de dispositivos, mas também de protocolos e outros níveis de interoperabilidade.

- Diferentes fases do ciclo de vida: os problemas existentes aparecem e são atribuídos às diferentes fases do ciclo de vida (desenvolvimento, implantação e manutenção). Em geral, são nestes momentos em se exige um maior consumo de tempo e, devido à natureza da situação, maior propensão a existência de erros.

Algumas soluções já foram apresentadas, *e.g., node-centric programming*, (devido à grande quantidade de dispositivos e a heterogeneidade deles, esta não é uma boa abordagem). Uma outra solução está presente na macroprogramação - capacidade de abstrair detalhes de baixo nível de dispositivos em variedade -, no entanto, a maioria das soluções deste tipo têm um foco muito grande na fase de desenvolvimento, sendo que esta é apenas uma fração do ciclo de vida [2]. Para tentar solucionar os problemas de abordagem da macroprogramação, o modelo MDD (*Model-Driven Design*) foi proposto. Neste, por fim, os problemas são abstraídos até determinado nível, provendo ferramentas que transformem estas abstrações em código, melhorando a produtividade do desenvolvimento da solução.

Este trabalho tem por proposta facilitar e agilizar o desenvolvimento de sistemas (mais especificamente programas e/ou código) para Internet das Coisas de tal maneira a permitir uma redução dos problemas entravados. A parametrização ou composição seria usada para, por exemplo, reduzir o esforço empregrado no desenvolvido dada uma alta heterogeneidade. Ainda mais, os recursos semânticos usados têm por premissa permitir alguma explicabilidade sobre o processo de geração e possibilitando que inferência seja usada na construção de novas informações.

1.5 Uma breve revisão sobre grafos de conhecimento

Grafos de Conhecimento são usados, por sua generalidade, para armazenar dados e informações as quais, por exemplo, podem ser aproveitadas por ferramentas CASE (*Computer-Aided Software Engineering*) na criação de ferramentas de apoio da Engenharia de Software. Em geral, sua estrutura é tal que o grafo é a representação de conceitos sobre alguma entidade ou domínio, sendo seus vértices os valores de suas propriedades e as arestas direcionadas as relações (comumente nomeadas) entre as propriedades.

Com o advento da Web Semântica e sua padronização pela W3C, surgiram métodos de codificação e estandardização da topologia estruturada pelos grafos de conhecimento.

[2] Macroprogramming in the Internet of Things: A Systematic Mapping Study
Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling

Sendo assim, adotou-se semanticamente[3](CARROLL; KLYNE, 2004) o uso de triplas - (S, P, O) - ou de operações lógicas - P(S, O) - para organizar os dados de tal forma que, para cada dado S (sujeito) - vértice -, há um dado P (predicado) - aresta - que o conecta a outro dado O (objeto) - vértice.

Ainda mais, é importante que o domínio dos elementos codificados no grafo - *i.e.*, sua interface com o sistema - seja especificada. Portanto, há a necessidade de se construir um vocabulário/ontologia (dada a definição de T. R. Gruber e R. Studer, uma ontologia é uma especificação explícita e formal de uma concepção compartilhada (STAAB; STUDER, 2009)), de forma a se estandardizar as interações entre os nodos. Não só, as informações são armazenadas de forma fragmentada - com baixa granularidade, possibilitando que as análises, partindo das relações, gerem conhecimentos mais precisos, formando soluções completas e estruturadas, o que facilita a gestão do conhecimento para uso e compreensão.

Por fim, a W3C especifica uma recomendação de como recuparar estes dados. Considerando que há um banco de dados capaz de guardar um grafo de conhecimento - *i.e.*, RDF -, a linguagem usada na manipulação e uso dos dados deve ser a SPARQL[4] (SPARQL Protocol and RDF Query Language). Nela, há uma interface muito semelhante ao que é provido pelo SQL padrão (*Structured Query Language*), para bancos de dados relacionais, no entanto, para um banco de dados não-relacional estruturado sobre grafos de conhecimento RDF. Com ela, seguindo o conceito das triplas ordenadas, é possível que sejam recuperados os sujeitos, predicados e objetos correspondentes aos conceitos tidos previamente.

1.6 Engenharia de Software, ferramentas CASE e geradores de programas

Note-se que a aplicação da Web Semântica no armazenamento e análise dos dados gerados pelos sensores em uma cidade inteligente é assunto razoavelmente estudado, que já conta com arcabouços comerciais como NEO4J (NEO4J, 2021). Por outro lado, arcabouços baseados em Web Semântica para prototipação e implantação de programas visando acelerar essas etapas, logo, armazenando módulos reusáveis do código-fonte em um grafo de conhecimento para subsequente recuperação, parametrização e composição, seja automática ou semi-automática, ainda estão por chegar a esse grau de maturidade. Por

[3] *id est*, W3C *Resource Description Framework* (RDF)
[4] Especificação do SPARQL 1.1 da W3C: ⟨https://www.w3.org/TR/sparql11-overview/⟩

exemplo, *SWoTSuite* (PATEL *et al.*, 2017) , apresenta um arcabouço para prototipação de aplicações em Internet das Coisas, mas apresenta mais detalhes sobre a arquitetura geral da aplicação que à parametrização e construção [5] dos programas que são executados nos elementos dessa arquitetura.

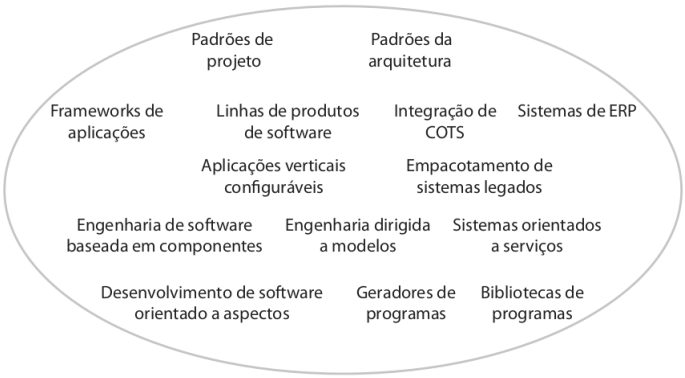
A fim de direcionar este projeto para uma primeira exploração sobre parametrização e construção automática ou semi-automática de programas baseados em grafos de conhecimento, há necessidade de especificar alguns aspectos do desenvolvimento de programas.

De acordo com o que há na Engenharia de Software, existem diversas ferramentas de apoio (Ferramentas CASE), dentre as quais há um grupo conhecido como Ferramentas de Geração de Código e, de forma mais específica, usar-se-á inicialmente no projeto um grupo determinado como "Geradores de código baseados em modelos". Estes geradores têm por característica gerar de forma dinâmica códigos para fins especificados no projetos, preenchendo modelos estáticos previamente formulados - os modelos podem variar, neste caso, a depender dos parâmetros de entrada que o gerador recebe.

Das mais variadas metologias ágeis de desenvolvimento, SCRUM é a que mais se adéqua ao trabalho a ser desenvolvido, uma vez que é uma metodologia iterativa e não prescreve métodos de programação específicos, se encaixando muito bem em abordagens mais técnicas (SOMMERVILLE, 2022). Não só, parte crucial do desenvolvimento que se está construindo galga-se no reúso de software, a partir de modelos de software, assim como no panorama exibido na Figura 3, preenchidos com informações sobre o domínio do problema a partir de um Grafo de Conhecimento. Assim, esta combinação torna o projeto possível, uma vez que SCRUM é flexível e o reúso de software é um método concordante com a metodologia apresentada.

[5] em Web Semântica, composição poderia ser aplicado

Figura 2 – O panorama de reúso.



Fonte – (SOMMERVILLE, 2022)

1.7 Gerador automático de código (programas) baseado em grafos de conhecimento

Dados alguns possíveis domínios de aplicação de sistemas inteligentes baseados em Internet das Coisas, um panorama geral sobre sua estrutura arquitetural e alguns entraves - e suas motivações - no desenvolvimento, abre-se espaço para a apresentação de algumas soluções que viabilizem uma redução no esforço empregado no desenvolvimento, implementação, gerenciamento, implantação e manutenção de tais sistemas.

Dada uma alta demanda para se desenvolver sistemas que integrem diversas soluções de IoT, surgiu a necessidade de que fossem criadas pesquisas na área, discorrendo sobre como aumentar a eficiência nos aspectos gerenciais e computacionais. Dessa forma, foi percebido que todo o trabalho inteligente é feito ainda por pessoas e o trabalho mecanizado, pelos computadores. Assim, nasceu um novo campo de estudo, hoje tratado como "Web Semântica"(BERNERS-LEE *et al.*, 2001). Este novo campo de estudo trata das relações entre pessoas e computadores, estabelecendo padrões semânticos na criação de vocabulários e regras para interoperação dos dados, *id est*, as informações possuem significado explícito, permitindo que os computadores entendam as informações tanto quanto os humanos que os operam (ANTONIOU, 2012). Não só, os Grafos de Conhecimento são usados para registrar informações aproveitadas na criação de ferramentas de apoio da Engenharia de Software.

O objetivo desta iniciação científica foi explorar a automatização da parametrização e construção de código usando grafos de conhecimento. Com um pouco mais de detalhes, pretendeu-se desenvolver um gerador de código baseado em grafos de conhecimento, de certa forma, uma extensão do conceito de geradores de código baseados em modelos, uma

vez que os modelos (fragmentos de programas) e dados usados pelo sistema desenvolvido são armazenados em um banco de dados orientado a grafos e o uso, gerenciamento e interoperabilidade dos dados está galgada nos conceitos de IoT e nos padrões da Web Semântica. De maneira mais ampla, espera-se que, com o programa desenvolvido, seja possível agilizar a programação e implantação de serviços em Cidades Inteligentes.

2 Metodologia de desenvolvimento

2.1 Metodologias ágeis

Desde a década de 1950, métodos de desenvolvimento incrementais e iterativos existem, dada uma evolução na gestão de projetos e no desenvolvimento adaptativo de software - sendo este último mais em voga a partir da década de 1970.

Entre as décadas de 1980 e 1990 (início) era comum o pensamento de que o desenvolvimento de software deveria ser baseado em um planejamento cuidadoso de todo o processo - *heavyweight methodologies* - (SOMMERVILLE, 2022). Sendo assim, nos meados da década de 1990, desenvolvedores de software insatisfeitos com a abordagem usual da época proporam metodologias mais leves - *lightweight methodologies* -, as quais, a partir do manifesto por eles proposto em 2001 (Manifesto para o Desenvolvimento Ágil de Software), passaram a ser formalmente conhecidas como "Metodologias Ágeis".

Dos princípios básicos das metodologias ágeis, pode-se citar:

Princípios	**Descrição**
Envolvimento do cliente	Os clientes devem participar do processo de desenvolvimento. Seus papéis incluem fornecer os requisitos e avaliar as iterações do sistema.
Entrega incremental	O software é desenvolvido aos poucos, tendo o foco mantidos nos requisitos um a um.
Pessoas, não processos	As habilidades individuais dos membros da equipe devem ser exaltadas. Sem precrição, cada um deve realizar as tarefes à sua maneira.
Aceitar as mudanças	Os requisitos do sistema são variáveis, por isso o projeto deve acomodar a possibilidade de mudança.
Manter a simplicidade	Sempre que possível, trabalhe a evitar a complexidade. Tanto o produto quanto o processo devem ser simples.

Tabela 1 – Sommerville 3.1 adaptado: Os princípios dos métodos ágeis

Diversas metodologias foram criadas a partir dos princípios anteriormente citados. Dentre elas, as de maior sucesso são:

- *Extreme programming*;
- *Scrum* (que será abordada posteriormente);
- *Crystal*;
- *Adaptative Software Development*;
- *Feature Driven Development*;

- *Kanban*;
- *Lean*;
- *Smart*;
- Entre outras...

Ainda hoje, existem discussões acerca da eficácia e eficiências de cada uma das metologias citadas. Algumas se apresentam críticas quanto à velocidade de desenvolvimento, enquanto outras têm foco nas questões socio-técnicas dos sistemas criados, por exemplo. No entanto, para o desenvolvimento desta pesquisa Scrum foi a metodologia ágil adotada, citada adiante.

2.2 Scrum

Scrum é uma das metodologias ágeis de desenvolvimento de software baseado nos princípios *lightweight* de maior sucesso atualmente.

Dentre as mais variadas metodologias ágeis de desenvolvimento é a que mais se adequada ao trabalho desenvolvido, uma vez que é uma metodologia com diversas características - como foi e ao que foi desenvolvido. Por definição, são elas:

- Auto-organizabilidade e auto-gerenciabilidade;
- Multidisciplinarismo e interfuncionalidade;
- Localização conjunta;
- Comprometimento;

Por sua iteratividade, não prescreve métodos de programação específicos, se encaixando muito bem em abordagens mais técnicas (SOMMERVILLE, 2022). Como parte crucial do desenvolvimento construído galga-se no reúso de software, a partir de modelos de software, assim como no panorama exibido na Figura 3, preenchidos com informações sobre o domínio do problema a partir de um Grafo de Conhecimento. Assim, esta combinação torna o projeto possível, uma vez que SCRUM é flexível e o reúso de software é um método concordante com a metodologia apresentada.

2.3 Procedimento implantado

Considerando a metodologia ágil apresentada, o Srum, este presente trabalho foi construído a partir de *sprints* semanais, nos quais, uma vez por semana, ocorreu um encontro entre o Orientador (Prof. Dr. Fábio Nakano) e o Orientado (bacharelando Otávio Rodrigues Bambans).

Não só, sempre que possível - isto é, com informações suficientes -, foram construídas atas para construção de uma base de conhecimento em relação à progressão do projeto, evolução e conclusão dos objetivos e novas descobertas sobre os possíveis caminhos, por exemplo, que o projeto poderia vir a ter.

De forma mais específica, tem-se o seguinte:

- Reuniões semanais com orientadores para avaliação de andamento, com citações, e ajuste de atividades;
- Documentação do andamento em formato de semanário;
- Armazenamento de código em um repositório de código (Github) público: Github: https://github.com/otaviobambans/CodeGenFromKnowledgeGraphs;
- Documentação do código junto à ele no repositório do projeto;
- Consolidação da documentação na ferramenta Overleaf.

3 Tecnologias

3.1 Ambiente de desenvolvimento e execução

A máquina usada para o desenvolvimento do software *CGFKG* tem as seguintes especificações:

- Modelo: Dell Vostro 3510 - 2021;
- Processador: i3-1005G1;
- Memória: 16Gb DDR4 - 2667 MT/s;
- Sistema Operacional : Fedora 38;
- Kernel Linux: 6.4.7-200.fc38.86_64.

Em relação aos softwares e bibliotecas usadas, suas versões são tais que:

- Python: 3.11.4;
- *rdflib*: 6.3.2;
- *asdl*: 0.1.5.

3.2 Mapping Programming Languages to Knowledge Description Languages - MPL2KDL

O programa MPL2KDL (*Mapping Programming Languages to Knowledge Description Languages*)[1], desenvolvido pelo Prof. Dr. André Luis Meneses Silva da Universidade Federal de Sergipe, tem por premissa receber um código em Python e exportar um dos seguintes formatos de arquivos a partir do reconhecimento de um programa Python a partir de sua árvores de sintaxe abstrata:

- RDF/N3;
- RDF/N-triples;
- RDF/XML (padrão);
- RDF/turtle.

Sendo assim, é possível construir um grafo a partir de qualquer programa escrito em Python.

[1] Link para o reposoritório público: ⟨https://github.com/andreluisms/MPL2KDL⟩

3.3.1 Python

Python é uma linguagem de programação criada por Guido van Rossum e teve sua primeira versão lançada em 1991. De propósito geral, multi-paradigma - *i.e.*, permite que seus programas sejam escritos orientados à objetos, proceduralmente ou funcionalmente, entre outros -, dinamicamente tipada e com coletor de lixo.

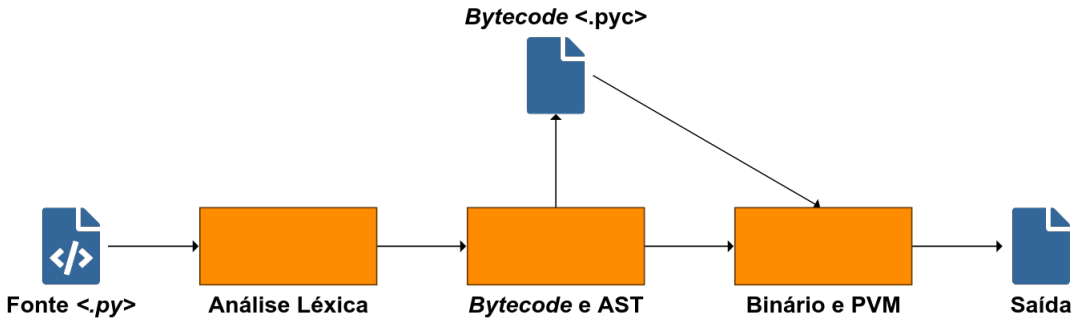
Sua fama nos dias de hoje é dada por conta de sua facilidade de programação - considerando o grande suporte da comunidade sobre a linguagem -, seu suporte à bibliotecas e modularização de programas - aumentando o reúso de software e acelerando o desenvolvimento -, e por sua versatilidade que a torna adequada para uma ampla gama de aplicações, desde desenvolvimento web e científico até automação de tarefas e inteligência artificial.

A natureza interpretada do Python, juntamente com sua capacidade de ser executado em diversas plataformas, contribui para sua portabilidade, o que significa que um programa Python pode ser executado em diferentes sistemas operacionais sem grandes modificações.

A implementação padrão do interpretador de Python é escrita na linguagem de programação C, sendo este conhecido como *CPython*. Primeiramente, ocorre uma análise lexical no código escrito, verificando se há algum erro de digitação ou marcação de bloco e alguns outros erros que podem ser detectados nessa fase de *pré-execução*. Em seguida, ocorre a geração do *bytecode*, na qual é gerada a Árvore de Sintaxe Abstrata (que será abordada mais adiante) pelo *parser* e, posteriormente, sua conversão para um formato *machine-readable*, com extensão `.pyc`. Por fim, o interpretador, sua máquina virtual PVM (*Python Virtual Machine*), recebe o *bytecode* e, em tempo de execução, o converte para um formato binário que de fato é executado - caso ocorra algum erro, a PVM para imediatamente a execução para mostrar os registros de erro.

De forma simplificada, tem-se o seguinte:

Figura 3 – Execução de um programa Python



Fonte – Autoria própria.

Nota-se que o *bytecode* Python (`.pyc`) não é portável, uma vez que depende da versão usada do ambiente Python e de questões relacionadas ao Sistema Operacional e máquina usadas na construção deste código intermediário.

3.3.2 Python *Abstract Syntax Tree*

Abstract Syntax Tree

A árvore de sintaxe abstrata (do inglês *Abstract Sintax Tree* - AST) é uma estrutura de dados abstrata que tem dois propósitos principais:

- Representar semanticamente a estrutura do programa desenvolvido;
- Permitir otmizações e anotações por parte do compilador sobre o código desenvolvido.

Sua construção é intimamente ligada ao compilador que está sendo desenvolvido e suas finalidades. No entanto, alguns dos seus requerimentos mínimos são tais que:

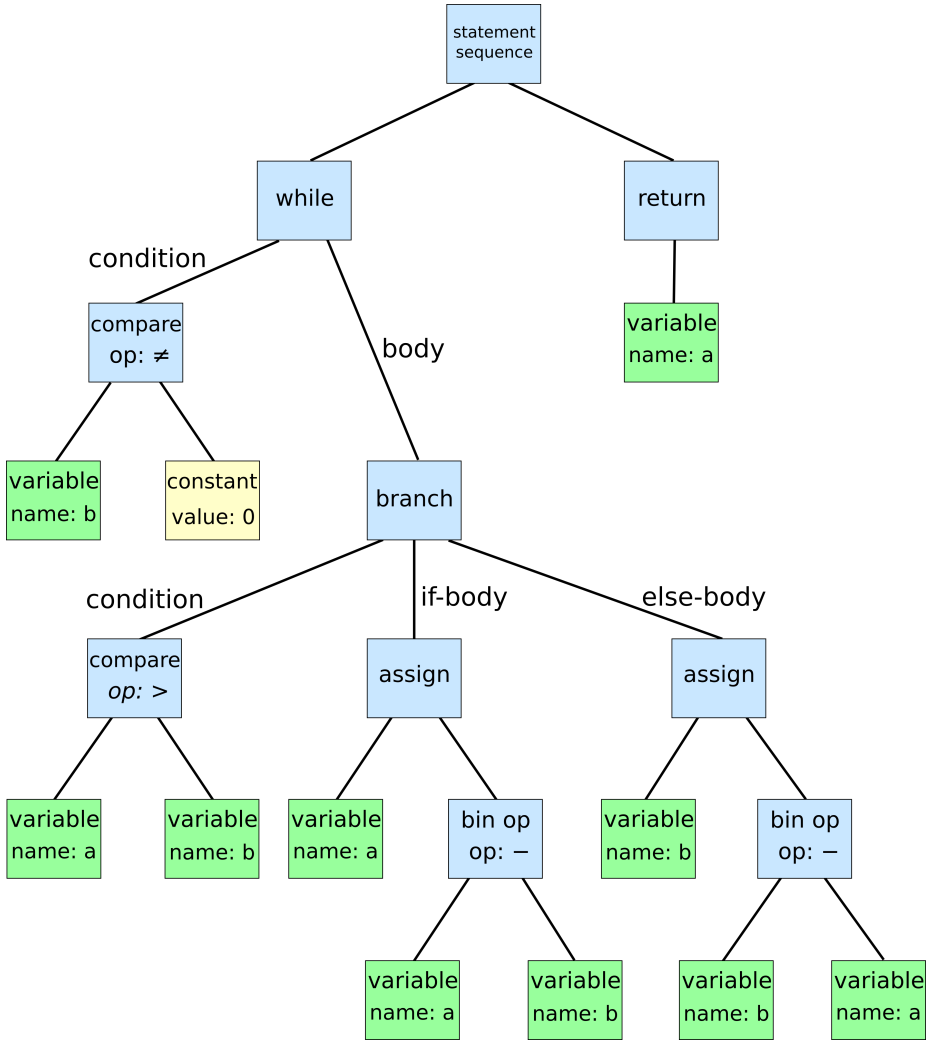
- Os tipos das variáveis devem ser preservados, assim como suas declarações relativas;
- A ordem de execução deve ser bem definida e representada;
- Os componentes da esquerda e direita de operações binárias devem ser corretamente identificados;
- Identificadores e anotações devem ser guardados em declarações.

Um exemplo pode ser dado a partir do algoritmo de Euclides. Considere o seguinte código:

```
while b != 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

Uma possível árvore de sintaxe abstrata para o código considerado é:

Figura 4 – AST do algoritmo de Euclides.



Fonte – ⟨https://commons.wikimedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg#/media/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg⟩

Módulo `ast`

O módulo `ast`[2] é um módulo Python que permite a geração e manipulação da Árvore de Sintaxe Abstrata de qualquer programa em Python antes do passo de compilação para execução na PVM.

Sua gramática é bem definida e, para os nós de classe, existem as seguintes propriedades:

- `_fields`: é uma propriedade capaz de representar os nós filhos;
- `lineno`: representa a linha na qual determinada informação (*token*, nome, valor, bloco etc.) começa;
- `col_offset`: representa a posição dentro da linha onde determinada informação se inicia;
- `end_lineno`: representa a linha na qual determinada informação termina;
- `end_col_offset`: representa a posição dentro da linha onde determinada informação termina.

3.4 W3C Resource Description Framework (RDF) e rdflib

3.4.1 RDF

O Resource Description Framework (RDF)[3], desenvolvido pelo World Wide Web Consortium (W3C), é uma estrutura semântica para representar informações na forma de triplas, que consistem em sujeito, predicado e objeto. O RDF é uma parte fundamental da Web Semântica, um movimento que visa adicionar significado e contexto aos dados da web para que máquinas possam entender e processar informações de maneira mais eficaz.

Aqui está uma descrição mais detalhada do RDF:

- **Triplas RDF:** O RDF organiza informações em triplas, que são declarações básicas compostas por três componentes principais:
  - **Sujeito (Subject):** O recurso ou entidade que está sendo descrito. É representado por um URI (Identificador de Recurso Universal), que é um identificador único na web.

[2] ⟨https://docs.python.org/3/library/ast.html⟩
[3] ⟨https://www.w3.org/TR/rdf11-concepts/⟩

  - **Predicado (Predicate):** A propriedade ou relação que liga o sujeito ao objeto. Também é representado por um URI.
  - **Objeto (Object):** O valor ou recurso associado ao sujeito por meio do predicado. Pode ser um literal (valor de texto) ou outro URI.

- **Sintaxe RDF:** O RDF pode ser representado em diferentes formatos de serialização, como RDF/XML, Turtle, N-Triples e JSON-LD. Cada um desses formatos permite expressar as mesmas informações de maneiras diferentes, mas equivalente, para atender às necessidades específicas.
- **Ontologias RDF:** As ontologias são esquemas que definem os termos, relações e restrições para descrever domínios específicos de conhecimento. Uma ontologia em RDF é expressa por meio de um conjunto de classes, propriedades e restrições, permitindo uma descrição mais rica e estruturada dos dados.
- **RDFS (RDF Schema):** É uma extensão do RDF que fornece classes e propriedades básicas para definir ontologias. Isso permite a criação de hierarquias de classes e relações entre propriedades.
- **OWL (Web Ontology Language):** O OWL é uma linguagem mais avançada para definir ontologias em RDF. Ele oferece diferentes níveis de expressividade, desde OWL Lite até OWL Full, permitindo a criação de ontologias complexas com regras de inferência.
- **Inferência:** Uma das características poderosas do RDF é a capacidade de inferir novas informações com base nas declarações existentes. Isso ocorre por meio de raciocínio lógico, onde a estrutura e as relações definidas nas ontologias permitem que sistemas computacionais deduzam informações implícitas.
- **Aplicações do RDF:** O RDF é amplamente utilizado em diversas áreas, como pesquisa acadêmica, gerenciamento de metadados, integração de dados, pesquisa científica, indústria da informação e muito mais. Ele permite que sistemas compreendam a semântica dos dados, tornando-os interoperáveis e significativos.

Em resumo, o RDF é uma estrutura que permite a representação de dados na forma de triplas, capacitando a Web Semântica ao adicionar significado aos dados da web. Isso promove a interoperabilidade e o compartilhamento de informações entre diferentes sistemas, ajudando a construir uma web mais inteligente e conectada.

3.4.2 `rdflib`

É uma biblioteca Python especializada em manipulação de grafos RDF[4]. Foi concebida para a manipulação e interrogação de dados estruturados no formato RDF (Resource Description Framework). O RDF, que fundamenta a Web Semântica, viabiliza a representação de informações e suas relações de forma semântica, transcendendo as limitações de um modelo puramente sintático. Facilita a criação de grafos RDF por meio de uma rica API, possibilitando a construção e armazenamento de conhecimento estruturado.

Não só, se destaca como uma ferramenta essencial para pesquisadores e desenvolvedores que buscam explorar a Web Semântica e a interoperabilidade de dados estruturados. Através dessa biblioteca, é viável criar, expandir e consultar grafos RDF, permitindo análises semânticas sofisticadas. A utilização da RDFLib abre portas para a realização de pesquisas interdisciplinares, ao possibilitar a integração e análise de dados provenientes de diversas fontes, tais como publicações acadêmicas, dados científicos e bases de conhecimento.

Além disso, a RDFLib é uma ponte para a adoção de consultas SPARQL, a linguagem padrão para a interrogação de dados RDF. Isso possibilita a exploração profunda e personalizada dos dados, revelando insights que podem informar a pesquisa acadêmica em uma variedade de campos. Em suma, o RDFLib se erige como uma valiosa ferramenta no arsenal dos acadêmicos, capacitando-os a mergulhar na complexidade da Web Semântica e a extrair significados aprofundados de dados estruturados.

3.5 Arquitetura e fluxo do sistema

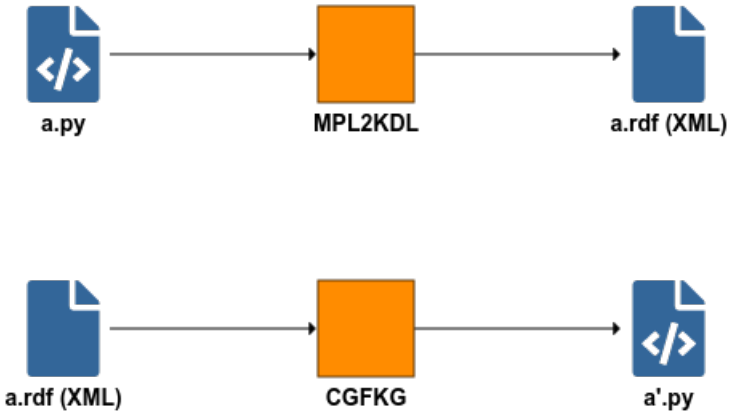
3.5.1 Arquitetura geral: alto nível do sistema MPL2KDL-CGFKG

Imagine um programa Python, o qual chamaremos genericamente de `a.py`. O programa MPL2KDL é capaz de, a partir da AST construída no estágio de compilação do *Bytecode*, gerar um grafo de conhecimento em RDF/XML - o qual chamaremos de `a.rdf` - que representa o programa em nível semântico, isto é, permite que seja inferido conhecimento sobre o programa escrito.

[4] ⟨https://rdflib.readthedocs.io/en/stable/⟩

O grafo de conhecimento em questão, dada sua natureza de construção, pode ser percorrido através da estrtutura determinada pela AST do programa Python. Sendo assim, o programa `CGFKG` - "bambans" - assim o faz. Através da AST do Python, segue a estrutura lá definida para que, recursivamente, dispare consultas sobre o grafo com SPARQL e, para cada valor retornado, é realizada uma análise quanto a classe que pertece, *id est*, é visto se há e qual deve ser o token que o antecede ou sucede e é analisado qual o escopo usado, portanto, infere-se os delimitadores de bloco - em Python, a quantidade de tabulações (Figura 5).

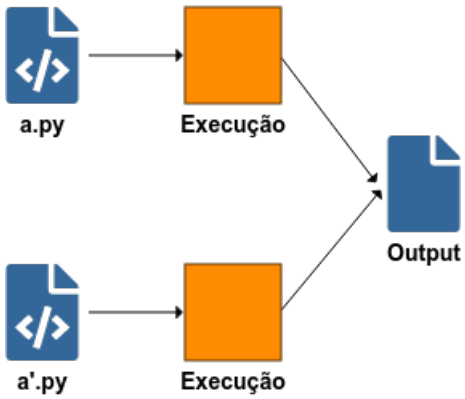
Figura 5 – Arquiterura geral.



Fonte – Autoria própria.

A saída esperada da execução do `CGFKG` sobre o grafo `a.rdf` é um programa `a'.py` que, eventualmente, pode estar com diferentes declarações do programa original, entretanto, com execução equivalente - Figura 6.

Figura 6 – As saídas dos programas escritos e gerados são equivalentes.



Fonte – Autoria própria.

3.5.2 Arquitetura do `CGFKG`

A ordem de execução dos métodos do CGFKG executados sobre o grafo é a seguinte:

Figura 7 – Arquitetura do CGFKG.

Fonte – Autoria própria.

Como na Figura 7, de forma mais detalhada, os métodos têm as seguintes funcionalidades:

- `Recursive`: gera um vetor que mapeia `‘‘Classe’’ -> ast.Classe`, consulta do grafo em busca do nome do arquivo original e dispara o método `ASTRecursive`;
- `ASTRecursive`: pega as subclasses do nó de entrada do grafo (Module) e chama o método `GraphRecursive` para o *stmt*, primeiro filho de *Module*;
- `GraphRecursive`: faz consultas recursivas a partir do nó *stmt* - basicamente, são caminhos até as folhas de uma árvore AST -, analisa se há necessidade e qual token deve acompanhar determinados valores a partir de sua classe, calcula o escopo de bloco - nível de tabulação - da posição dos valores e os exibe na saída padrão.

Portanto, o CGFKG tem por premissa realizar o oposto do desempenhado por MPL2KDL, uma vez que um se propõe a gerar um grafo com os conhecimentos obtidos do código e o outro - aqui desenvolvido -, a partir do grafo, gera código Python.

4 Conclusão e discussão

O desenvolvimento do gerador apresenta uma abordagem inovadora para simplificar a criação de programas através da utilização de grafos de conhecimento. Ao receber um grafo codificado em RDF/XML que contém informações sobre um programa em Python, o gerador opera por meio de consultas recursivas sobre o grafo, resultando na geração de um equivalente em código Python. A integração desse gerador é direta, bastando importar a biblioteca 'CGFKG' (Code Generator From Knowledge Graphs) e empregar o método 'Recursive', ao qual o grafo deve ser passado como parâmetro. O grafo RDF, a ser previamente carregado em um objeto 'Graph' da biblioteca 'RDFLIB' por meio do método de *parsing* `parse`, habilita esse processo.

O propósito subjacente a essa empreitada é a otimização do desenvolvimento e implementação de serviços de software em Cidades Inteligentes. A automação da geração de código por meio de grafos de conhecimento visa agilizar a parametrização e construção de programas, com foco na capacidade de gerar saídas equivalentes ao programa original. O programa "bambans", parte integral da abordagem CGFKG, se destaca como um facilitador nesse cenário. Ele percorre grafos de conhecimento codificados em RDF/XML, que detêm detalhes sobre programas em Python, e emprega essas informações para criar um código Python funcionalmente equivalente. A funcionalidade do código gerado se equipara à do programa original que serviu de base para a criação do grafo com o MLP2KDL.

As etapas desenvolvidas nesse processo abrangem diversas tarefas cruciais:

- O desenvolvimento do gerador foi concretizado em Python, aderindo aos princípios da Web Semântica, RDF e SPARQL;
- A validação da abordagem pode ser assegurada por meio de testes no protótipo da ferramenta. O gerador foi avaliado ao gerar código destinado à programação de dispositivos IoT voltados para medição de temperatura e umidade do ar ou atuadores como fechadura automatizadas;
- A acessibilidade e colaboração foram fomentadas com a disponibilização do código-fonte do gerador automático no Github[1];

Em síntese, a concepção e realização desse gerador automático representam um avanço notável no âmbito do desenvolvimento de software para Cidades Inteligentes. A

[1] ⟨https://github.com/bambans/CodeGenFromKnowledgeGraphs⟩

integração de conceitos da Web Semântica e tecnologias como RDF e SPARQL demonstra o comprometimento com soluções inovadoras. O programa ‘bambans‘ se destaca como a personificação dessa abordagem, capacitando a geração automatizada de código Python eficaz e funcionalmente equivalente, a partir de grafos de conhecimento que encapsulam informações sobre programas pré-existentes.

Referências

ANTONIOU, G. e. a. *A Semantic Web primer*. 3. ed. [S.l.]: Massachusetts: Massachusetts Institute of Technology, 2012. Citado na página 15.

BELLINI, P.; NESI, P.; PANTALEO, G. IoT-enabled smart cities: A review of concepts, frameworks and key technologies. *Applied Sciences*, MDPI AG, v. 12, n. 3, p. 1607, fev. 2022. Disponível em: ⟨https://doi.org/10.3390/app12031607⟩. Citado 3 vezes nas páginas 7, 8 e 10.

BERNERS-LEE; TIM; HENDLER; JAMES; LASSILA, O. The semantic web. *Scientific american*, JSTOR, v. 284, n. 5, p. 34–43, 2001. Citado na página 15.

CARROLL, J.; KLYNE, G. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. [S.l.], 2004. Https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/. Citado na página 13.

NEO4J. 2021. Disponível em: ⟨https://neo4j.com/company/⟩. Citado na página 13.

ORACLE. *What is IoT?* 2022. Disponível em: ⟨https://www.oracle.com/internet-of-things/what-is-iot/⟩. Citado na página 7.

PATEL, P.; CASSOU, D. Enabling high-level application development for the internet of things. *Journal of Systems and Software*, v. 103, p. 62–84, 2015. ISSN 0164-1212. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S0164121215000187⟩. Citado na página 11.

PATEL, P.; GYRARD, A.; DATTA, S. K.; ALI, M. I. Swotsuite: A toolkit for prototyping end-to-end semantic web of things applications. In: . Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017. (WWW '17 Companion), p. 263–267. ISBN 9781450349147. Disponível em: ⟨https://doi.org/10.1145/3041021.3054736⟩. Citado na página 14.

SOMMERVILLE, I. *Engenharia de software*. [S.l.: s.n.], 2022. ISBN 9788579361081. Citado 4 vezes nas páginas 14, 15, 17 e 18.

STAAB, S.; STUDER, R. (Ed.). *Handbook on Ontologies*. Springer Berlin Heidelberg, 2009. Disponível em: ⟨https://doi.org/10.1007/978-3-540-92673-3⟩. Citado na página 13.