UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Alexandre Antônio Lima da Silva

Reestruturação do Mosaicode para Compatibilidade com Python Moderno e Arquivos JSON

São João del-Rei

2025

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL-REI

Alexandre Antônio Lima da Silva

Reestruturação do Mosaicode para Compatibilidade com Python Moderno e Arquivos JSON

Monografia apresentada como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFSJ.

Orientador: Flávio Luiz Schiavoni

Universidade Federal de São João del-Rei – UFSJ

Bacharelado em Ciência da Computação

São João del-Rei

2025

Alexandre Antônio Lima da Silva

Reestruturação do Mosaicode para Compatibilidade com Python Moderno e Arquivos JSON

Monografia apresentada como requisito da disciplina de Projeto Orientado em Computação II do Curso de Bacharelado em Ciência da Computação da UFSJ.

Flávio Luiz Schiavoni
Orientador

Elder José Reioli Cirilo
Convidado 1

Jesuliana Nascimento Ulysses
Convidado 2

São João del-Rei
2025

Agradecimentos

Agradeço, em primeiro lugar, aos meus pais e ao meu irmão, pelo apoio constante ao longo de toda a minha trajetória. À minha namorada, por me incentivar diariamente e estar sempre ao meu lado. Ao Flávio, pelo suporte e pela dedicação durante a realização deste trabalho, e a todos os professores do departamento, pelo conhecimento compartilhado ao longo da graduação.

"And then one day you find
Ten years have got behind you
No one told you when to run
You missed the starting gun"
Pink Floyd

Resumo

O Mosaicode é um ambiente de programação visual que permite a criação de programas por meio de blocos conectáveis, voltado especialmente para aplicações em arte digital. Este trabalho apresenta a modernização estrutural e sintática da ferramenta, originalmente escrita em Python, com o objetivo de atualizar sua base de código e torná-la mais modular e manutenível. As melhorias incluíram a substituição de práticas obsoletas por recursos modernos, além disso, arquivos de extensões e configuração em Python e XML foram migrados para o formato JSON. A arquitetura do sistema foi reorganizada em camadas separadas de controle, modelo, persistência e interface gráfica. Testes foram conduzidos para avaliar a reestruturação da ferramenta, resultando em um código moderno e manutenível. Os resultados demonstram a viabilidade da abordagem, abrindo caminho para manutenções futuras e extensibilidade do código fonte.

Palavras-chaves: Mosaicode, Python, Manutenção, Código legado.

Abstract

Mosaicode is a visual programming environment that allows the creation of programs through connectable blocks, aimed especially at applications in digital art. This work presents the structural and syntactic modernization of the tool, originally written in Python, with the objective of updating its code base and making it more modular and maintainable. The improvements included the replacement of obsolete practices by modern resources in addition, Python and XML configuration and extension files have been migrated to JSON format. The system architecture has been reorganized into separate layers of control, model, persistence, and graphical interface. Tests were conducted to evaluate the restructuring of the tool, resulting in a modern and maintainable code. The results demonstrate the feasibility of the approach, paving the way for future maintenance and extensibility of the source code.

Key-words: Mosaicode, Python, Maintenance, Legacy Code.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A arte digital se desenvolve como uma manifestação artística dos tempos contemporâneos, que envolve a criação e apresentação de uma obra de arte através do uso de tecnologia, bem como de software. Ela expande os limites tradicionais da arte e permite ao público interagir com a obra em diferentes níveis, criando expressões visuais, sonoras e interativas. A arte digital pode ser criada apenas em ambientes computacionais. Além disso, é importante mencionar que a interseção entre software e arte computacional forma um espaço interdisciplinar que renova paradigmas culturais e amplia o alcance da arte em contextos sociais mediados pela tecnologia (AHMED, 2009).

Nesse sentido, há um foco crescente em novos paradigmas que permitem o desenvolvimento e a criação de obras interativas e multimídia sem a necessidade de um entendimento profundo de linguagens de programação (TRIBE, 2006).

Entre estas, a programação visual se destaca como uma abordagem pedagógica inovadora que facilita o desenvolvimento de soluções computacionais por meio de interfaces gráficas de usuário. Programação visual refere-se ao desenvolvimento de programas de computador através de gráficos em vez de representação puramente textual. Um dos modelos mais comumente utilizados nesse paradigma é o modelo de fluxo de dados, onde os programas são expressos na forma de grafos direcionados. Aqui, os nós do grafo representam funções, blocos ou operações, enquanto as arestas (ou arcos) representam os caminhos através dos quais os dados são transmitidos e processados (HILS, 1992).

Esse paradigma alivia os desafios associados às linguagens de programação textual tradicionais, permitindo que usuários com diferentes formações compreendam os conceitos fundamentais de lógica e programação. Como resultado, indivíduos com experiência prévia mínima ou nenhuma em programação conseguem construir aplicações funcionais montando blocos de código. Isso amplia efetivamente as oportunidades para a produção digital engajada e integra o conhecimento tecnológico nas práticas artísticas e educacionais (RIBAS, 2016).

O Mosaicode se destaca entre essas ferramentas com seus recursos distintos, pois é um ambiente de programação visual destinado tanto a artistas quanto a desenvolvedores que desejam criar aplicações usando diagramas gráficos que são automaticamente transformados em código-fonte. A interface do Mosaicode encapsula recursos utilizando blocos que podem ser reutilizados e interconectados, servindo assim como uma interface intuitiva para a criação de obras artísticas (SCHIAVONI, 2018). A ferramenta permite adicionar novos conjuntos de blocos sob a forma de extensões, cada extensão define seus blocos, portas e templates de código. Vale destacar que se não houver nenhuma extensão insta-

lada no Mosaicode, o software não possuirá nenhuma funcionalidade inicial (RESENDE, 2019).

Suas principais características estão dentro da aplicação que utiliza blocos gráficos interligados como componentes do programa diagrama final executável. A ferramenta interpreta o diagrama fornecido, criando automaticamente o código-fonte correspondente que é então executado na linguagem de programação apropriada (SCHIAVONI, 2018). Com o Mosaicode, os usuários alcançam uma abstração parcial do problema, pois podem construir estruturas para aplicações digitais artísticas sem precisar se envolver em programação textual. É, assim, uma abordagem única que combina geração automática de código e programação visual, uma vez que a solução melhora significativamente as oportunidades oferecidas para engajar com computação criativa e desenvolver aplicações de software interativas (COSTA; SCHIAVONI, 2019).

O Mosaicode tem o potencial de auxiliar o processo do usuário de criação de arte digital. No entanto, neste momento, a ferramenta possui um código fonte que está desatualizado e sem suporte. Esse código obsoleto e desatualizado acumula problemas estruturais que dificultam sua manutenção e uso em contextos contemporâneos.

O código fonte não recebe atualizações há 5 anos, e está escrito na versão 2 da linguagem Python, anterior ao Python 3.6, que não são mais suportadas [1]. Isso resulta em problemas de incompatibilidade do sistema com ambientes de execução modernos, bibliotecas recentes e até mesmo sistemas operacionais atualizados.

Outra questão em relação a versão atual da ferramenta se dá em relação a sua persistência. O software usa XML para definir os elementos que compõem o código, como blocos, portas, preferências e templates. Embora o XML ainda sirva para estas funcionalidades, sua estrutura é antiquada e excessivamente verboso para sistemas modernos. O formato de marcação JSON vem sendo amplamente utilizado como padrão para descrever estruturas de dados em aplicações web, APIs e configurações por sua sintaxe mais simples, integração com bibliotecas contemporâneas e suporte nativo em diversas linguagens (BREJE, 2018).

Além disso, o Mosaicode, que é baseado em extensões, não permite baixar e instalar extensões através da interface gráfica. Para adicionar novos blocos, é necessário ter algum conhecimento prévio sobre a estrutura interna do sistema e realizar operações manuais, como copiar arquivos Python para certos diretórios.

Essas questões trazem a necessidade de modernizar o código-fonte, substituir XML por arquivos JSON, usar as melhores práticas de engenharia de software e adicionar um recurso de auto-instalação de extensões.

[1] Status of Python versions. Disponível em: <https://devguide.python.org/versions/#versions>. Acesso em: 15 jul. 2025.

1.1 Objetivos Gerais

O Mosaicode enfrenta um processo de obsolescência tecnológica devido à falta de manutenção em seu código-fonte. Este problema resulta na estagnação de suas aplicações, tanto do ponto de vista funcional quanto em sua manutenção, devido à ausência de atualizações. Assim, este trabalho tem como objetivo central da proposta reescrever a base de código do Mosaicode aplicando conceitos de Engenharia de Software e utilizando os recursos oferecidos pelas versões mais recentes do Python.

1.2 Objetivos Específicos

Além do objetivo geral, o projeto busca alcançar os seguintes objetivos específicos:

- Revisar a estrutura do Mosaicode original e acompanhar a evolução na identificação de componentes legados, padrões obsoletos e limitações técnicas que envolvem a linguagem, arquitetura e até mesmo a organização dos arquivos.
- Atualizar o código-fonte da aplicação para garantir sua funcionalidade em versões mais recentes do Python, implementando tipagem estática, organização modular e bibliotecas nativas atualizadas.
- Mover os arquivos de configuração de XML para JSON focando na simplificação, legibilidade e validação estrutural em "blocos, portas, templates e preferências".
- Reestruturar a arquitetura interna da aplicação visando maior coesão, separação de responsabilidades, modularização e clareza na organização do código-fonte.
- Permitir o download e instalação de extensões de forma dinâmica e segura ao usuário, por meio de interface gráfica.
- Garantia da integridade da aplicação após a modernização e a estabilidade frente a futuras atualizações.
- Avaliar os ganhos de desempenho e qualidade com base em métricas objetivas.
- Desenvolver testes automatizados e avaliar a cobertura de código da aplicação antes e depois da modernização.

1.3 Justificativa

A atualização de sistemas legados é um tópico comum em Engenharia de Software, considerando a evolução rápida no desenvolvimento e nas linguagens de programação.

Softwares legados mantêm tecnologias arcaicas, e sua arquitetura pode comprometer a eficiência, segurança e competitividade organizacional (GATTO, 2007).

Práticas de modernização permitem a aplicação de conhecimentos teóricos, consolidando habilidades como refatoração de código, a construção de testes automatizados, design modular e a validação de dados. O objeto em estudo neste trabalho é o Mosaicode,sua base de código original possui trechos que foram criados com recursos de versões legadas da linguagem Python, utilizando estruturas e bibliotecas que não possuem mais suporte. Avaliar as versões original e reescrita do Mosaicode demonstrará os benefícios práticos da modernização de padrões e a verificação do desempenho, legibilidade e confiabilidade do sistema.

Do ponto de vista acadêmico, este projeto agrega valor ao discurso sobre a sustentabilidade de longo prazo do software. A sustentabilidade do software diz respeito à capacidade de um sistema de perdurar e ser mantido ao longo do tempo, o que inclui tanto sua longevidade quanto a facilidade de manutenção. Assim como na modernização e manutenção do Mosaicode, é possível reduzir o envelhecimento do software e a dívida técnica. Isso aumenta a relevância do software tanto na formação profissional de estudantes quanto no desenvolvimento profissional contínuo e na melhoria dos sistemas. Avaliar as versões original e modernizada do Mosaicode demonstrará os benefícios práticos da modernização de padrões e a confirmação do desempenho, legibilidade e confiabilidade do sistema (BARICHELLO, 2022).

1.4 Ferramentas Relacionadas

Discutiremos ferramentas que estão relacionadas tanto ao Mosaicode quanto à programação visual nesta seção.

1.4.1 KNIME

KNIME[2] é uma das primeiras e mais conhecidas ferramentas analíticas desenvolvidas na Universidade de Konstanz em 2004 e é amplamente utilizada para pesquisa, ensino e colaboração devido à sua interface amigável que não requer codificação. O KNIME é uma plataforma de programação visual de código aberto para ciência de dados e mineração de informações. É um ambiente modular que permite a construção interativa de fluxos de trabalho (fluxogramas visuais) para processamento de dados de maneira passo a passo, conectando componentes chamados nós. (ACITO, 2023)

Cada passo de manipulação ou análise de dados é representado por um ícone (nó) que é controlado em um editor gráfico que realiza operações como leitura de dados, várias

[2] KNIME. Disponível em: <http://knime.com/>. Acesso em: 15 jul. 2025.

transformações, treinamento de modelos ou visualização de resultados. Em termos de recursos, o KNIME possui uma biblioteca abrangente de nós para integração de dados (com vários conectores de fontes), limpeza e preparação de dados, algoritmos de aprendizado de máquina, mineração de texto, análise química e biológica, e visualização de dados. A arquitetura é extensível: novos algoritmos ou métodos podem ser facilmente integrados como nós personalizados ou como extensões (BERTHOLD, 2009). A Figura 1 mostra um exemplo de um fluxo de trabalho com manipulação e visualização de dados.

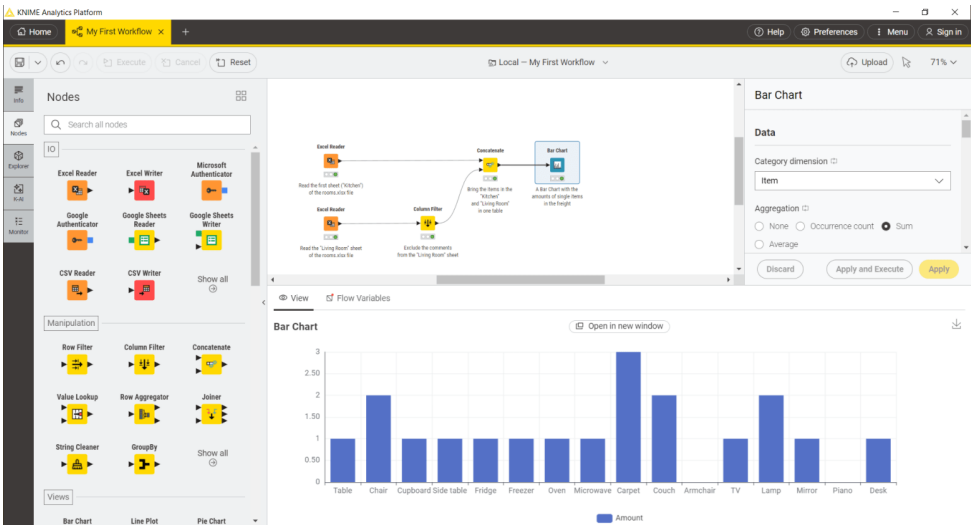

Figura 1 – Exemplo de fluxo de trabalho no KNIME

1.4.2 Node-RED

Node-RED[3] é um ambiente de desenvolvimento centrado em fluxos de dados com uma interface visual, feito em JavaScript e rodando em cima do Node.js. Ele foca em aplicações de IoT e Indústria 4.0. A interface do Node-RED possui três áreas fundamentais: um painel com nós que fornece elementos funcionais, o painel de fluxos que conecta nós e fornece lógica à aplicação e o painel de informações que fornece um monitoramento para o sistema para que problemas possam ser solucionados. Os fluxos são armazenados em JSON, o que torna versionamento, compartilhamento e reutilização mais eficientes.(FERENCZ; DOMOKOS, 2019)

A ferramenta foi criada para facilitar a integração de hardware com APIs e serviços web, porém agora ocorre uma expansão ao seu uso na robótica, onde se pode agilmente desenvolver softwares de controle para sistemas robóticos como braços SCARA e robôs cartesianos de 6 eixos. Dessa forma, o Node-RED ajuda a simplificar o processo de desenvolvimento de software de automação, aumentando a produtividade, a reutilização de código e proporcionando melhor clareza visual ao lidar com designs de sistemas complexos. Sua aplicação em ambientes de robótica é favorecida pelo fato de poder ser executado

[3] Node-RED. Disponível em: <http://https://nodered.org/>. Acesso em: 15 jul. 2025.

em plataformas modestas como dispositivos embutidos e computadores pessoais, além de ser baseado em Node.js.(GARBEV, 2022) A figura 2 ilustra um fluxo de mensagens no Node-RED para controle visual e em tempo real de um robô.

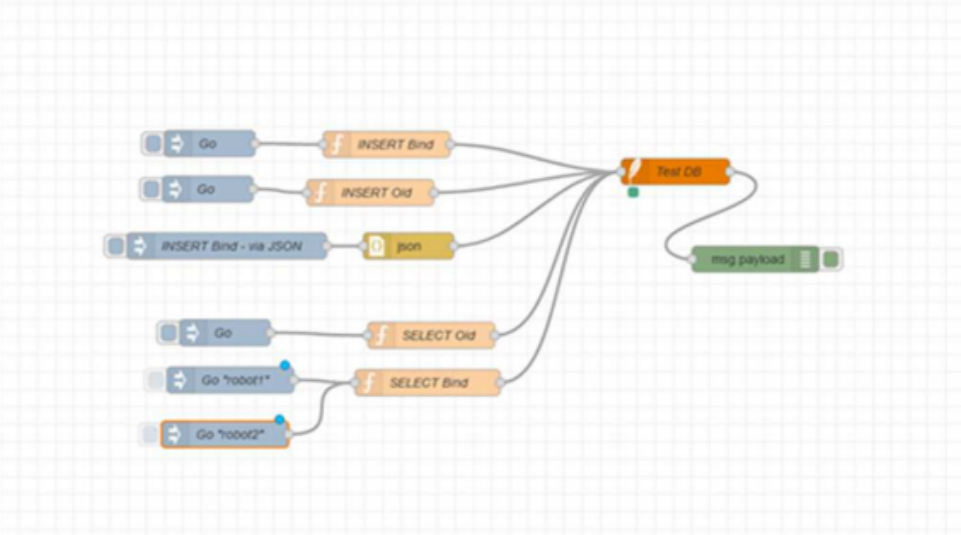

Figura 2 – Exemplo de fluxo

Fonte: (GARBEV, 2022)

1.4.3 Pure Data

Criado em meados da década de 1990 por *Miller Puckette* como uma evolução do programa *Max*, Pure Data (Pd)[4] é um programa que atua como um ambiente de programação gráfica para a música computadorizada em tempo real. O Pd é um sistema aberto multiplataforma que roda em Windows, MacOS e GNU/Linux, podendo, portanto, colocar em prática a criação de patches interativos feitos de objetos que são conectados e interconectados visualmente. Com isso, o software permite o tratamento de estruturas de dados compostas, a integração unificada de sinais diversos como áudio e vídeo, e o processamento gráfico e em tempo real, simplificando assim a duplicação de dados e a cópia de múltiplos patches. (PUCKETTE, 1996)

O Pd pode ser utilizado no ensino do processamento de fala, especialmente por possibilitar demonstrações e exercícios ao vivo, o que torna o aprendizado de conceitos complexos mais simples e acessível a estudantes e ao público em geral. Por exemplo, em uma aula no Departamento de Ciência da Computação da Universidade de Sheffield, patches baseados em Pd são utilizados para ensinar acústica elementar e algoritmos para processar digitalmente a linguagem falada. Além disso, o Pd tem sido usado para exposições interativas de ciência, como "Digital Voice Factory", que permite aos usuários manipular e explorar seus próprios sons vocais em tempo real. (MOORE, 2014) A Figura

[4] Pure Data. Disponível em https://puredata.info/. Acesso em: 15 jul. 2025.

3 demontra um exemplo de um fluxo para a síntese de uma voz individual no sistema de canto formântico.

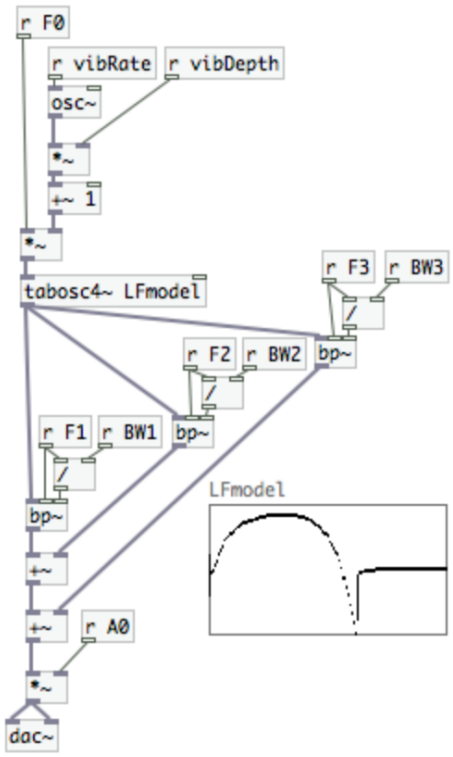

Figura 3 – Exemplo de fluxo de processamento no Pd

Fonte: (HOWARD, 2013)

1.5 Organização do Trabalho

O trabalho é dividido em seis capítulos. No Capítulo 1, o cenário do projeto é apresentado junto com seus objetivos, e também a justificativa da proposta. No Capítulo 2, são expostos os fundamentos teóricos relativos a Mosaicode, linguagem Python e o desenvolvimento de software utilizando boas práticas. A proposta para a realização do trabalho é abordada no Capítulo 3.

O Capítulo 4 inclui tanto o diagnóstico da versão original do sistema como todos os desenvolvimentos que seguem: a modernização da base de código, migração de extensões e arquivos de configuração para JSON, nova funcionalidade de instalação de extensões, e os testes realizados. Os principais resultados alcançados são apresentados no Capítulo 5, enquanto as considerações finais e sugestões para trabalhos futuros estão no Capítulo 6.

2 Referencial Teórico

Os conceitos apresentados neste capítulo fornecem a base teórica necessária para compreender os fundamentos do ambiente Mosaicode, os recursos técnicos e as práticas recomendadas de validação, testagem e manutenção de sistemas. A articulação entre esses elementos sustenta as decisões adotadas no desenvolvimento do trabalho, que busca alinhar uma aplicação visual tradicional a padrões atuais de desenvolvimento de software.

2.1 Mosaicode

O Mosaicode surgiu a partir da necessidade de refatorar e modernizar o ambiente de programação visual Harpia, que originalmente foi criado para o desenvolvimento de aplicações de visão computacional, mas tornou-se obsoleto devido à descontinuidade das suas dependências. Além disso, o ambiente passou a suportar diferentes domínios de aplicação, como Arte Digital, Computação Musical, Computação Gráfica, Realidade Virtual e Aumentada, afastando-se do foco exclusivo em visão computacional (GONÇALVES; SCHIAVONI, 2020). A figura 4 mostra um diagrama feito no ambiente Harpia.

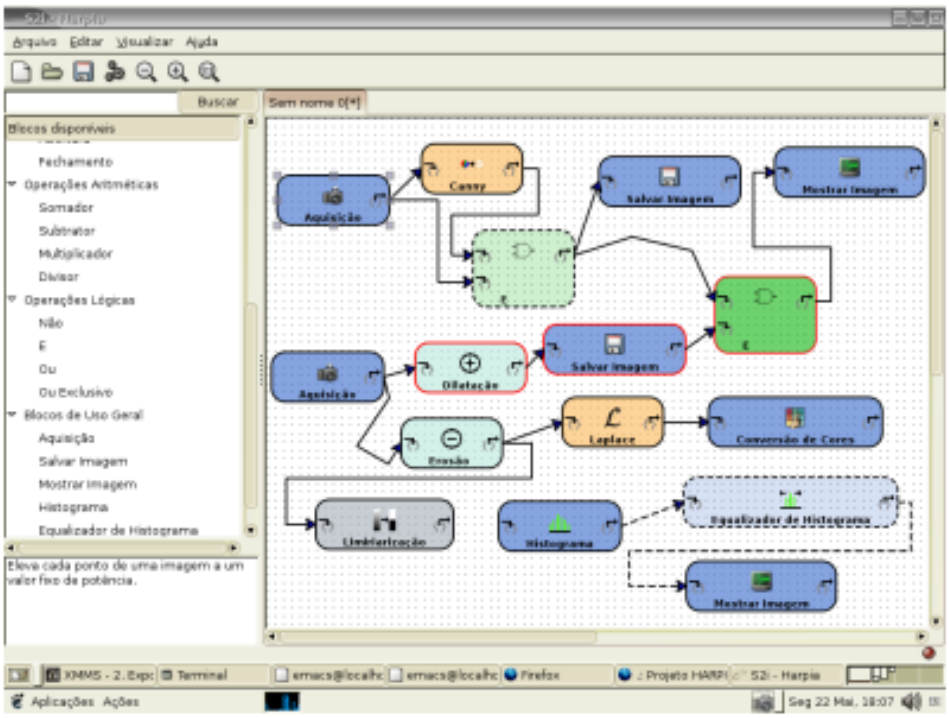

Figura 4 – Diagrama feito no ambiente de programação Harpia

Fonte: (GONÇALVES; SCHIAVONI, 2020)

O Mosaicode é um ambiente de programação visual baseado em blocos, concebido

para permitir a construção de programas por meio de diagramas, sem necessidade de escrever código textual diretamente. Esta ferramenta foi desenvolvida e é mantida pelo grupo de pesquisa ALICE[1] da Universidade Federal de São João del-Rei e pelo professor Flávio Schiavoni, orientador deste trabalho. Seu objetivo principal é auxiliar usuários sem conhecimento profundo de programação (como artistas digitais ou estudantes) a criar aplicações de forma gráfica, oferecendo ao final do processo o código-fonte gerado automaticamente para possíveis customizações.(RESENDE, 2019)

Conforme destacado por (SCHIAVONI, 2018), a ferramenta foi criada para simplificar a criação de programas com propósitos artísticos, preenchendo a lacuna entre a criatividade artística e as habilidades técnicas de codificação. A figura 5 mostra um diagrama feito no ambiente Mosaicode.

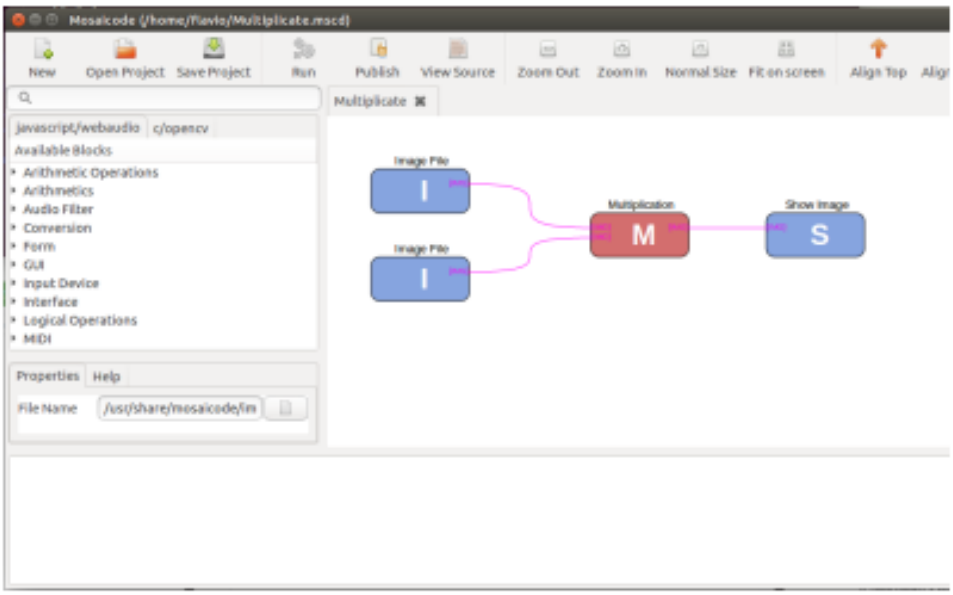

Figura 5 – Diagrama feito no ambiente de programação Mosaicode

Fonte: (GONÇALVES; SCHIAVONI, 2020)

2.1.1 Principais Características

Entre as características do Mosaicode, podemos destacar:

- **Programação por blocos conectáveis:** O software utiliza o paradigma de encapsular trechos de código em segmentos visuais, também chamados de blocos. Cada bloco possui uma funcionalidade específica e pode ser conectado a outros blocos por meio de conexões que formam um diagrama que transmite o fluxo lógico da aplicação. Essas conexões transferem dados entre os blocos, e também definem a ordem de execução. Além disso, os diagramas construídos dentro do ambiente podem até

[1] Sobre o ALICE, consultar <https://www.alice.ufsj.edu.br>.

incluir observações junto aos blocos para que o "código" possa ser documentado visualmente (GONçALVES, 2017).

- **Blocos com portas de entrada e saída:** Cada bloco possui propriedades dinâmicas que são chamadas como portas de entrada (Input) e saída (Output) através das quais dados de controle fluem. As portas de entrada permitem a configuração de um bloco ou fornecem dados de outros blocos. Já as portas de saída, exportam os resultados do processamento realizado pelo bloco. Essas conexões são feitas unindo as portas de saída às de entrada correspondentes, e o sistema garante a ligação apenas entre portas compatíveis, de modo que um bloco possa receber valores de um tipo apropriado às suas funções. Essas conexões de portas servem como caminhos para a troca de informações entre blocos e determinam onde a execução e o fluxo de dados ocorrem no diagrama(SCHIAVONI, 2018).

- **Geração automática de código-fonte:** A ferramenta permite gerar automaticamente o código-fonte da aplicação a partir do diagrama construído. É utilizado um template de código (modelo pré-definido) conforme a linguagem de destino, preenchendo-o com os trechos de código correspondentes a cada bloco conectado. Com o diagrama finalizado, o usuário pode obter o código compilável/interpretável em diferentes linguagens, dependendo das extensões instaladas. A geração de código permite que o programa criado graficamente seja exportado como na linguagem selecionada, o código fonte real é produzido e pode ser otimizado ou utilizado fora do ambiente(GONçALVES, 2017).

- **Extensibilidade modular (blocos personalizados):** A estrutura do software foi projetada de forma modular e extensível. Novos conjuntos de blocos podem ser adicionados ao ambiente sob a forma de extensões. Cada extensão define seus blocos, portas e templates de código, podendo abranger um domínio de aplicação (por exemplo, visão computacional, áudio digital, gráficos 3D etc.). Elas são carregadas automaticamente ao inicializar, buscando primeiro por classes Python instaladas, depois por arquivos XML do sistema e, em seguida, por arquivos XML no diretório do usuário (RESENDE, 2019). Essa hierarquia permite que usuários personalizem ou adicionem blocos em sua instância local sem necessitar alterar a instalação global do programa. Desde que, o desenvolvedor tenha conhecimento técnico para definir o bloco em XML ou Python(SCHIAVONI, 2018).

2.1.2 Arquitetura da Ferramenta

De acordo com (SCHIAVONI, 2018), o Mosaicode organiza-se segundo o padrão MVC (Model-View-Controller), o que facilita a manutenção e a compreensão do código-fonte. Nesse modelo, distinguem-se quatro camadas fundamentais:

- **Camada de Modelo:** Responsável por gerenciar os dados essenciais – diagramas, blocos, portas, conexões e templates de código –, representando a estrutura lógica do aplicativo.
- **Camada de GUI (Interface Gráfica):** Constrói a interface com o usuário utilizando a biblioteca GTK+3 e o canvas GooCanvas para renderizar, de forma visual, os elementos que compõem os diagramas.
- **Camada de Controle:** Gerencia a lógica do aplicativo, coordenando as ações do usuário e a interação entre a interface e o modelo, de forma a manter baixo acoplamento entre os componentes.
- **Camada de Persistência:** Responsável por salvar e carregar os artefatos gerados – como diagramas e definições de blocos –, utilizando os modelos e os mecanismos de controle para assegurar a integridade dos dados.

Inicialmente, o sistema fazia uso extensivo de arquivos XML para definir blocos e configurações, o que ainda é mantido para garantir compatibilidade e facilitar a extensão do ambiente. Com o amadurecimento do projeto, passou-se a admitir a definição de blocos via scripts em Python, ampliando as possibilidades de personalização. (GONÇALVES; SCHIAVONI, 2020)

2.2 Python

Python [2] é uma linguagem de programação de fácil aprendizado, caracterizada por sua sintaxe clara, tipagem dinâmica e recursos avançados de estruturas de dados. Reconhecida por sua capacidade de acelerar o desenvolvimento de software, ela é amplamente utilizada para automação de tarefas, desenvolvimento de aplicações, scripts interativos e prototipagem rápida. Por ser interpretada e oferecer suporte ao modo interativo, o Python permite testes e experimentações ágeis. Disponível em múltiplas plataformas, Python é ideal tanto para iniciantes quanto para profissionais, promovendo a criação eficiente de programas compactos e legíveis (HUNT, 2020).

Nas subseções seguintes serão descritos as atualizações relacionada a linguagem Python propostas neste trabalho.

[2] The Python Tutorial. Disponível em: <https://docs.python.org/3.12/tutorial/index.html>. Acesso em: 15 jul. 2025.

2.2.1 Pathlib

De acordo com a documentação do Pathlib [3], o módulo faz parte da biblioteca padrão da linguagem Python, a partir da versão 3.4, e traz uma nova interface para a manipulação de diretórios e arquivos usando abordagem orientada a objetos. Essa nova forma de trabalhar com diretórios e arquivos traz mais clareza e organização se comparado aos métodos anteriores funcionais como os disponibilizados pelo módulo os.path.

A classe Path é a principal do módulo, que representa e encapsula caminhos de sistemas POSIX(Unix, Linux, macOS) e Windows usando as subclasses PosixPath e WindowsPath. Entre as principais funções do Pathlib, se destacam as funções de verificação de existência de arquivos e diretórios, a distinção entre arquivos e diretórios e a leitura e escrita de conteúdo em texto e bytes. O pathlib se destaca como mais seguro e coeso em relação ao mencionado os.path, por isso é mais recomendado para novas aplicações desenvolvidas com versões mais recentes do Python. Seu uso favorece uma melhor adoção de princípios da programação orientada a objetos.

2.2.2 Data Classes

O módulo *Data Classes*[4], que foi adicionado com a PEP 557, fornece uma nova maneira de criar classes destinadas a armazenar dados em Python. Com o decorador `@dataclass`, ele automatiza a geração de métodos especiais como `__init__`, `__repr__`, `__eq__`, entre outros, com base nas anotações de tipo dos atributos da classe, permitindo eliminar grande parte do código repetitivo necessário para definir objetos estruturados e comparáveis. Além disso, as *Data Classes* permitem transformar instâncias em dicionários e tuplas, respectivamente.

No contexto de projetos refatorados, isso se traduz em classes de domínio mais enxutas e manutenção facilitada, uma vez que mudanças nos atributos são refletidas diretamente nos construtores gerados automaticamente.

2.2.3 Type Hints

Type Hints[5] em Python são anotações opcionais que indicam explicitamente os tipos de variáveis, parâmetros e valores de retorno de funções. Foi introduzida oficialmente na PEP 484, tais anotações não afetam a execução do código, mas servem para facilitar a leitura, manutenção e verificação estática do código por ferramentas como o mypy ou

[3] Pathlib — Object-oriented filesystem paths. Disponível em: <https://docs.python.org/3/library/pathlib.html>. Acesso em: 15 jul. 2025.

[4] dataclasses — Data Classes. Disponível em: <https://docs.python.org/3/library/dataclasses.html>. Acesso em: 15 jul. 2025.

[5] PEP 484 – Type Hints. Disponível em: <https://peps.python.org/pep-0484/>. Acesso em: 15 jul. 2025.

editores com suporte à análise de tipos. A figura 6 mostra um exemplo prático do uso de Type Hints em uma função Python.

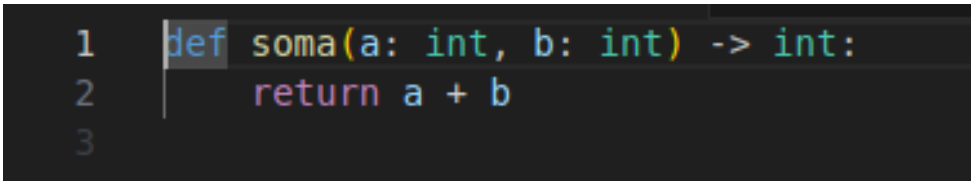

Figura 6 – Função em Python que utiliza Type Hints

2.2.4 Logging

O módulo de *Logging* [6] do Python padroniza a forma de rastrear o comportamento do sistema, ajudando o desenvolvedor a detectar e corrigir possíveis erros. Ele faz parte da biblioteca padrão do Python e possui diferentes tipos de níveis de mensagem padrão. Abaixo estão listados os diferentes tipos de mensagem padrão:

- **DEBUG**: informações detalhadas para diagnóstico durante o desenvolvimento.
- **INFO**: mensagens que indicam o funcionamento normal do sistema.
- **WARNING**: alerta sobre algo inesperado, mas que não impede a execução.
- **ERROR**: erro que afeta parte do sistema, mas não o interrompe completamente.
- **CRITICAL**: erro grave que compromete ou encerra a aplicação.

2.2.5 Técnicas de Caching

O módulo *functools* [7] da biblioteca padrão do Python oferece mecanismos eficientes de cache de resultados. Entre eles estão os decoradores `@lru_cache` e `@cached_property`, utilizados para melhorar o desempenho e evitar recomputações desnecessárias.

- **`@lru_cache` (Least Recently Used Cache):** é um método que armazena em cache os resultados de chamadas anteriores. Quando a função novamente chamda utilizando os mesmos argumentos, o resultado é retornado diretamente do cache, evitando reprocessamento.
- **`@cached_property`:** transforma um método de instância em uma propriedade de instância com cache, ou seja, o método é executado apenas uma vez, e seu valor

[6] Logging. Disponível em: <https://docs.python.org/pt-br/3/howto/logging.html>. Acesso em: 15 jul. 2025.

[7] functools — Higher-order functions and operations on callable objects. Disponível em: <https://docs.python.org/3/library/functools.html>. Acesso em: 15 jul. 2025.

é armazenado. Nas chamadas subsequentes, o valor armazenado é retornado, sem nova execução do método.

2.3 Boas Práticas Durante o Ciclo de Vida do Software

A seguir, listaremos algumas práticas a serem adotadas durante o ciclo de vida do software. Estas práticas foram estudadas para auxiliar na refatoração do código do Mosaicode no presente trabalho.

2.3.1 Qualidade do código

A qualidade estrutural do código-fonte exerce influência direta na manutenibilidade de um sistema. Código claro, bem organizado e consistente é mais fácil de compreender, depurar e estender. Boas práticas de programação, como padronização de nomenclaturas, organização modular do software, definição clara de responsabilidade para cada função ou classe, e separação de preocupações (por exemplo, entre lógica de negócio e lógica de interface), contribuem para um código mais limpo e coerente (BARICHELLO, 2022). A Teoria Fundamental da Legibilidade, citada por (GOMES, 2013), aponta que o código deve ser redigido de modo a reduzir o tempo necessário para sua compreensão, adotando boas práticas de escrita resultando na melhoria da compreensibilidade e facilita a manutenção.

Essa legibilidade superior implica que, diante de uma base de código legível e padronizada, erros e incoerências são mais facilmente percebidos e corrigidos, e novos membros da equipe podem se ambientar ao projeto com agilidade. Quanto mais legível o código, maiores as chances de ele ser mais fácil de modificar, menos propenso a erros, mais manutenível, mais reutilizável e mais confiável (LEE, 2014).

2.3.2 Manutenção de Código

Como mostra (BORQUE; FAIRLEY, 2014), a manutenção de software é um processo do ciclo de vida dos softwares, responsável por modificar e aprimorar aplicativos existentes a fim de preservar sua utilidade e qualidade ao longo do tempo. Ela envolve atividades pós-entrega que visam corrigir defeitos, adaptar o software a mudanças no ambiente operacional, adicionar novas funcionalidades e melhorar atributos internos do código, como desempenho ou manutenibilidade .

Essas ações de manutenção podem ser categorizadas em manutenção corretiva (para corrigir falhas), adaptativa (para ajustar o sistema a novas plataformas ou requisitos de ambiente), aperfeiçoativa ou perfectiva (para implementar melhorias e novas funcionalidades) e preventiva (para refatorar e otimizar o código visando evitar proble-

mas futuros). De fato, a manutenção deve ser realizada regularmente para corrigir falhas, melhorar o design interno, implementar melhorias e manter o software utilizável diante de novas condições de hardware ou software (LUCAS, 2016).

Essa predominância reflete a importância da manutenção contínua para manter o software alinhado aos requisitos dos usuários e às mudanças tecnológicas. A falta de práticas consolidadas de manutenção pode levar ao acúmulo de código obsoleto e crescente complexidade, tornando o sistema difícil de compreender e evoluir. Como observado por (PADUELLI, 2007) que um sistema de software de larga escala tende a aumentar em complexidade à medida que evolui, a menos que sejam tomadas ações explícitas para reduzir essa complexidade.

2.3.3 Testes

Os testes são uma etapa essencial dentro do ciclo de vida do desenvolvimento de software e consistem em um processo sistemático e contínuo de avaliação que verifica se cada componente do sistema funciona corretamente conforme os requisitos inicialmente especificados. Além disso, os testes atuam como uma rede de segurança contra a introdução de falhas durante modificações ou evoluções futuras do software (JAMIL, 2016).

Com base nisso, os testes sistemáticos auxiliam na verificação de que os requisitos foram implementados conforme especificado, ao executar os testes após cada mudança, é possível detectar imediatamente se algo que antes funcionava passou a falhar, garantindo que as alterações no código funcionem conforme o esperado.

De acordo com (SOMMERVILLE, 2011) para cobrir diferentes aspectos e granularidades do comportamento do sistema, as técnicas de teste são organizadas em vários níveis complementares.

- **Testes de unidade:** verificam partes específicas do código-fonte, isolando cada unidade para confirmar que sua lógica interna está correta.
- **Testes de integração:** avaliam a interação entre módulos ou serviços, garantindo que componentes que funcionam separadamente também operem corretamente em conjunto.
- **Testes de sistema:** validam o comportamento do software simulando o uso real, é verificado se todos os componentes e camadas da aplicação interagem de forma adequada para atender aos requisitos globais.
- **Testes de regressão:** são repetidos após atualizações no código fonte para confirmar que o sistema continua se comportando corretamente, sem que nenhuma funcionalidade pré-existente seja afetada.

A adoção de uma estratégia de testes bem definida assegura a qualidade, a confiabilidade e a manutenibilidade de sistemas de software. Os diferentes níveis de teste se complementam na etapa de detecção de erros em diversas fases do desenvolvimento, contribuindo para a conformidade com os requisitos especificados e para a prevenção de falhas introduzidas por modificações futuras (JAMIL, 2016).

3 Proposta

A contextualização do problema refere-se à necessidade de reengenharia do sistema Mosaicode. A proposta parte da premissa de que, para manter a relevância da ferramenta em termos de programação visual e arte digital, torna-se imprescindível atualizar sua base tecnológica, reorganizar sua arquitetura interna e revisar seus mecanismos de configuração e extensão. Para isso, este projeto propõe inicialmente realizar um estudo do sistema para que seja possível entender seu funcionamento, a arquitetura de seus arquivos e diretórios. Com isso, é possível mapear templates e configurações, a distribuição de blocos e portas, e ainda confrontar isto com a utilização de arquivos XML e práticas de codificação que estão obsoletas. Esta análise apoia a identificação dos pontos mais críticos do código, como redundância e acoplamento excessivo entre módulos.

Em função desse exame preliminar, um plano de **reescrita incremental e controlada** será seguido, priorizando a lógica do sistema, mas permitindo a substituição gradual de partes obsoletas por outras mais sustentáveis. Esse plano terá, em um primeiro momento, a definição de uma base de código mais uniforme e utilização de recursos atuais e que ainda possuem suporte do Python. Essa nova base viabiliza, em etapas subsequentes, a reorganização da arquitetura interna da aplicação, promovendo separações mais nítidas entre diversas camadas do software, incluindo a camada de controle, modelo, interface gráfica (GUI)e persistência de dados.

Neste sentido, o projeto propõe o **uso de JSON**, como novo formato de configuração que ofereça um melhor armazenamento e maior legibilidade do código. A atualização para uso de JSON também pretende ser utilizada para as extensões do mosaicode que utiliza arquivos Python para definição de blocos e portas e XML para armazenar os diagramas.

Essa mudança requer, também, o desenvolvimento de um novo sistema de **carregamento e validação de configurações** que carregue e valide as configurações, garantindo a consistência dos dados utilizados pelo Mosaicode.

O projeto também propõe a revisão nos processos que estão relacionados à **instalação e ao gerenciamento das extensões**. Atualmente, o Mosaicode lida com extensões de uma forma que contraria a praticidade e automação que o sistema tem como objetivo oferecer. Para contornar esse desafio, será criado um fluxo que torne o processo de importação, validação e ativação de extensões mais direto, seguro e amigável com a interface da aplicação.

A proposta acrescenta a incorporação de **validação e controle da qualidade do código fonte** em toda a etapa de reescrita. Isso englobará a definição de limites para

checagem da estrutura em camadas da aplicação, além da utilização de um padrão de testes automáticos que auxiliem na validação do comportamento de todos os componentes do Mosaicode , reduzindo os riscos de regressão e garantindo que a nova versão da ferramenta cumpra com todas as funcionalidades esperadas.

Em suma, este trabalho propõe uma reengenharia da ferramenta com ações planejadas que envolvem análise, refatoração, reorganização arquitetônica, melhorias na experiência do usuário e garantia da qualidade da ferramenta. Ao final, isso deverá resultar em uma versão do Mosaicode que seja mais estável, compreensível e alinhada aos paradigmas contemporâneos de desenvolvimento em Python, permitindo uma evolução ininterrupta.

Como demonstrado na Figura 7, ela representa, de forma estruturada, todas as etapas metodológicas que compõem a ideia do projeto. Cada fase corresponde a um ciclo do processo, e seu fechamento determina a progressão para a próxima etapa. A sequência inicia com uma análise da versão original da aplicação, atualização da base de código, migração de formatos de dados até a implementação de novos recursos e a validação do sistema. Ao final desse processo, o resultado esperado é uma nova versão da ferramenta que foi reavaliada, modernizada e esteja validada.

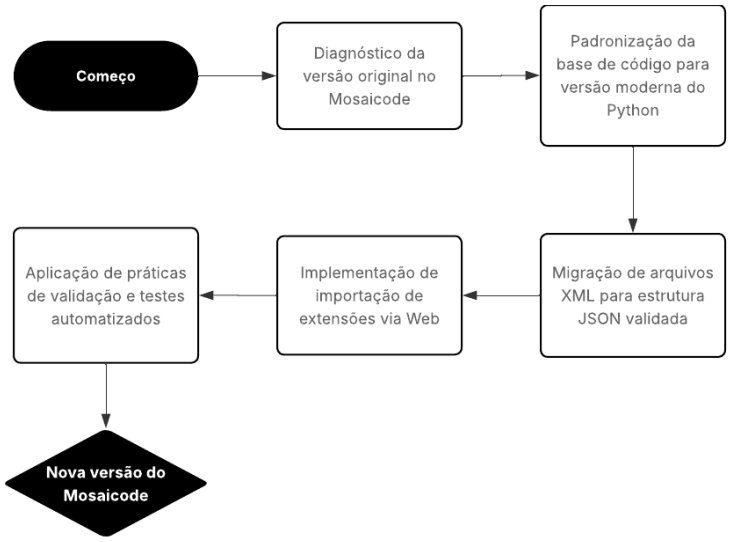

Figura 7 – Fluxograma para reescritura do Mosaicode

4 Desenvolvimento

Definida a proposta de modificação e refatoração do código, neste capítulo trataremos do desenvolvimento da ação proposta.

4.1 Diagnóstico do Código Original

A reescritura do Mosaicode iniciou com a análise detalhada de sua versão original. O diagnóstico teve como objetivo identificar os principais aspectos técnicos que dificultavam a manutenção do software, aspectos que possivelmente poderiam comprometer sua extensibilidade e limitações de compatibilidade com versões atuais da linguagem Python.

Com base nos pontos apresentados na seção 2.2, observou-se durante o levantamento, uma predominância de práticas consideradas ultrapassadas da linguagem Python. Entre os principais problemas identificados, destacam-se o uso explícito da herança da classe *object*, comum em versões antigas do Python, e o uso de *print()* para geração de mensagens de depuração. Outro ponto notado foi a ausência completa de tipagem estática. As funções e classes não utilizavam anotações de tipo *(type hints)*, o que dificulta o entendimento do código fonte. Por fim, havia arquivos utilizando *os.path* ao invés do uso de *Pathlib*, módulo recomendado para manipulação de diretórios.

Essas características prejudicam softwares com grande quantidade de parâmetros e objetos compartilhados entre módulos distintos, pois dificultam o entendimento correto do software, como é o caso do Mosaicode, que utiliza estruturas complexas de blocos, portas e templates.

Em relação à estrutura de dados, os arquivos de configuração estavam armazenados em duas formas: diretamente em arquivos Python, ou em arquivos XML. A Figura 8 mostra como eram carregadas as preferências antes da reescrita, em um arquivo Python.

```python
import os


class Preferences(object):

    def __init__(self):
        self.conf_file_path = "configuration"

        self.author = ""
        self.license = "GPL 3.0"
        from mosaicode.system import System as System
        self.version = System.VERSION

        self.recent_files = []
        from mosaicode.system import System
        self.default_directory = os.path.join(System.get_user_dir(), "code-gen")
        self.default_filename = "%n"
        self.grid = 10


        self.width = 900
        self.height = 500
        self.hpaned_work_area = 150
        self.vpaned_bottom = 450
        self.vpaned_left = 300

        self.connection = "Curve"

```

Figura 8 – Arquivo de Preferências utilizando Python

A codificação de dados diretamente em arquivos Python compromete a modularidade do sistema e impede atualizações ou substituições dinâmicas em tempo de execução. Já o uso de XML para representar blocos e portas apresentava limitações de legibilidade, ausência de validação automática e necessidade de *parsers* específicos para leitura, o que tornava a manutenção e a extensão desses componentes mais complexas e propensas a erros. A figura 9 mostra um arquivo de um diagrama de exemplo da extensão *Javascript / Web Audio API* [1] que usa a estrutura XML.

```xml
<?xml version="1.0" encoding="utf-8"?>
<mosaicode>
 <version value="0.0.1"/>
 <zoom value="1.0"/>
 <language value="javascript"/>
 <blocks>
  <block collapsed="False" id="2" type="mosaicode_lib_javascript_webaudio.extensior
   <position x="250.0" y="260.0"/>
  </block>
  <block collapsed="False" id="3" type="mosaicode_lib_javascript_webaudio.extensior
   <position x="270.0" y="70.0"/>
   <property key="freq" value="440"/>
   <property key="type" value="sine"/>
  </block>
  <block collapsed="False" id="4" type="mosaicode_lib_javascript_webaudio.extensior
   <position x="440.0" y="160.0"/>
  </block>
  <block collapsed="False" id="6" type="mosaicode_lib_javascript_webaudio.extensior
   <position x="690.0" y="120.0"/>
  </block>
  <block collapsed="False" id="7" type="mosaicode_lib_javascript_webaudio.extensior
   <position x="680.0" y="230.0"/>
   <property key="width" value="640"/>
   <property key="height" value="240"/>
  </block>
  <block collapsed="False" id="8" type="mosaicode_lib_javascript_webaudio.extensior
   <position x="520.0" y="310.0"/>
  </block>
 </blocks>
 <connections>
  <connection from_block="2" from_out="0" to_block="4" to_in="1"/>
  <connection from_block="3" from_out="3" to_block="4" to_in="0"/>
  <connection from_block="4" from_out="2" to_block="8" to_in="0"/>
  <connection from_block="8" from_out="4" to_block="7" to_in="0"/>
  <connection from_block="8" from_out="4" to_block="6" to_in="0"/>
 </connections>
 <comments/>
</mosaicode>
```

Figura 9 – Arquivo *filter.mscd* da extensão Javascript / Web Audio API

[1] Javascript / Web Audio API. Disponível em: <https://github.com/Alice-ArtsLab/mosaicode-javascript-webaudio>. Acesso em: 15 jul. 2025.

Além dos fatores estruturais, foram observados níveis elevados de acoplamento entre os módulos: por exemplo, um ciclo de importação entre *system.py* e *port.py* fazia com que cada arquivo dependesse do outro, bloqueando mudanças que pudessem ser feitas de forma isolada. A arquitetura, por sua vez, carecia de uma divisão clara de responsabilidades e, dentro desse quadro, arquivos como o *blockmodel.py* reunia lógica de controle (validação de dados), visualização (definição de cores e posicionamento) e persistência (serialização XML) numa única classe.

Por fim, o software utilizava uma abordagem de carregamento completo das informações durante a inicialização, o que resultava em maior consumo de memória e maior tempo para abertura da interface gráfica. Não havia qualquer mecanismo de carregamento sob demanda (*lazy loading*), o que fazia com que todos os blocos, portas e templates fossem carregados integralmente.

4.2 Reescritura do Código Fonte

A reescritura do código fonte do Mosaicode iniciou com a atualização do código-fonte com o objetivo de alinhar a aplicação às melhores práticas do Python (versões 3.9+). Foram feitas as seguintes alterações:

- **Modernização da Sintaxe:** todas as classes deixaram de herdar de *object*, instruções *print()* foram substituídas por chamadas ao módulo *logging* e *os.path* foi trocado por *pathlib*. Essas alterações eliminam redundâncias do código e estão alinhadas com as boas práticas do Python contemporâneo.

- **Uso de Type Hints:** foram adicionadas anotações de tipo em todos os modelos de dados, controladores, campos de entrada, componentes de interface e métodos principais. Essa prática favorece a segurança estática e facilita o uso de ferramentas como mypy [2] - ferramenta para verificação estática de tipos em Python, para checar automaticamente se os tipos usados no código estão corretos, baseado nas type hints.

- **Conversão de Modelos para Dataclasses:** as classes do diretório *Model* foram refatoradas como *dataclasses*, eliminado código repetitivo e facilitando operações dinâmicas através da reflexão.

- **Criação de Enums e Tipagem Segura:** constantes dispersas foram substituídas por enumeradores (Enum), para maior legibilidade do código. A Figura 10 mostra um exemplo de Enum aplicado no arquivo de modelo de portas para constantes de entrada e saída.

[2] mypy. Disponível em: <https://mypy-lang.org/>. Acesso em: 15 jul. 2025.

```
10
11  class ConnectionType(Enum):
12      INPUT = "input"
13      OUTPUT = "output"
14
```

Figura 10 – Arquivo de modelo de portas utilizando Enum

- **Separação de Dados e Código:** informações anteriormente em código Python e XML, foram migradas para arquivos JSON. A Figura 11 mostra o código da Figura 8 de preferências após a reescrita.

```
{
    "conf_file_path": "configuration",
    "author": "",
    "license": "GPL 3.0",
    "version": "${System.VERSION}",
    "recent_files": [],
    "default_directory": "${System.get_user_dir()}/code-gen",
    "default_filename": "%n",
    "grid": 10,
    "width": 900,
    "height": 500,
    "hpaned_work_area": 150,
    "vpaned_bottom": 450,
    "vpaned_left": 300,
    "connection": "Curve"
}
```

Figura 11 – Arquivo de Preferências utilizando JSON

  Os arquivos da extensão *Javascript / Web Audio API* também foram migrados para a estrutura JSON, para poderem ser utlizados usando a nova estrutura de código do Mosaicode. Para utilizar a extensão nesta nova versão o usuário precisa somente adicionar o diretório na pasta raiz do projeto.

  Além disso, um carregador de configurações (ConfigLoader), ele é responsável por gerenciar todo o carregamento e pesistência de configurações JSON, desde preferências de usuário, caminhos, valores padrão de blocos e portas, configurações de sistema, persistência e extensões.

- **Reestruturação Arquitetural:** houve uma separação mais clara entre as responsabilidades das camadas do sistema, reduzindo a dependência entre elas e aumentando a testabilidade da aplicação:
  - **model/:** representação dos dados.
  - **control/:** lógica de controle e orquestração.
  - **persistence/:** leitura e escrita de arquivos.
  - **GUI/:** interface gráfica e interação com o usuário.

- **Técnicas de Caching:** foram aplicadas técnicas como `@cached_property` e `@lru_cache`, de modo que resultados sejam guardados em memória e reutilizados em chamadas subsequentes. Essa técnica foi utilizada nos recursos que são mais usados do Mosaicode. A Figura 12 e 13 mostram exemplos da aplicação `@cached_property` e `@lru_cache`, respectivamente.

```
13  class BlockModel:
62      @cached_property
63      def properties_dict(self) -> Dict[str, Any]:
64          return {prop.get("name", f"prop_{i}"): prop.get("value", "")
65                  for i, prop in enumerate(self.properties)}
66
67      @cached_property
68      def input_ports(self) -> List[Any]:
69          return [port for port in self.ports if hasattr(port, 'conn_type') and port.
70
71      @cached_property
72      def output_ports(self) -> List[Any]:
73          return [port for port in self.ports if hasattr(port, 'conn_type') and port.
74
75      @cached_property
76      def port_count(self) -> int:
77          return len(self.ports)
78
79      @cached_property
80      def has_properties(self) -> bool:
81          return len(self.properties) > 0
82
83      @cached_property
84      def display_label(self) -> str:
85          if self.label and self.label not in ["A", ""]:
86              return self.label
87          return self.type.replace("_", " ").title()
88
```

Figura 12 – Exemplo de métodos da classe *BlockModel* que utiliza `@cached_property`

```
33  class System:
47      @classmethod
48      @lru_cache(maxsize=1)
49      def load_system_config(cls) -> Dict[str, Any]:
50          from mosaicode.utils.config_loader import ConfigLoader
51          try:
52              return ConfigLoader.load_config("system")
53          except (ConfigurationError, FileOperationError) as e:
54              logger.warning(f"Failed to load system config: {e}")
55              return {}
56
57      @classmethod
58      @lru_cache(maxsize=32)
59      def get_system_value(cls, key: str, default: Any = None) -> Any:
60          config = cls.load_system_config()
61          return config.get(key, default)
62
63      @classmethod
64      @lru_cache(maxsize=1)
65      def get_user_dir(cls) -> Path:
66          return Path.home() / cls.APP
```

Figura 13 – Exemplo de métodos da classe *System* que utiliza `@lru_cache`

5 Resultados

Este capítulo apresenta os resultados obtidos com este trabalho.

- **Código Compatível com Python 3.9:** A base de código passou a adotar recursos modernos da linguagem Python (3.9+), eliminando práticas obsoletas. A Tabela 1 mostra um resultado comparativo entre o número de arquivos e total de linhas.

Versão	**Total de Arquivos**	**Total de Linhas**
Mosaicode	185	≈ 9.000
Mosaicode Reescrito	275	≈ 14.000

Tabela 1 – Comparativo entre versões do Mosaicode (arquivos e total de linhas)

Mosaicode novo possui significativamente mais linhas de código que o antigo devido a uma transformação na arquitetura. No sistema antigo, as classes eram preenchidas em tempo de execução através de reflexão, importação dinâmica de módulos e cópia automática de atributos entre objetos. Por exemplo, enquanto o sistema carregava automaticamente classes Python, criando instâncias dinamicamente sem validação ou tratamento de erros.

O sistema novo, implementa uma abordagem diferente utilizando *dataclasses* com tipos explícitos, *type hints* em todos os métodos, propriedades computadas com cache automático, validação de dados, tratamento específico de exceções, *logging* detalhado e carregamento estruturado de configurações JSON. Essas alterações são descritas abaixo:

- **Modelos de dados mais simples e coesos:** conversão das classes do módulo *model* em *dataclasses* contribuiu para a redução de código repetitivo, aumentando a coesão e a clareza dos modelos de dados utilizados na aplicação.

  Outras classes não foram reescritas utilizando *dataclasses*, pois classes que herdam de widgets gráficos do GTK, como *MainWindow* ou *Block*, já possuem métodos `__init__` específicos. Usar *dataclasses* nesse caso causa conflito na ordem de inicialização, resultando em erros. Além disso, padrões como *Singleton*, usado na classe *System*, exigem controle sobre a criação de instâncias — algo que as *dataclasses* não suportam nativamente.

  Por fim, classes que gerenciam arquivos, como *Persistence*, precisam lidar com falhas, exceções e estados variáveis — o oposto da proposta de simplicidade e imutabilidade das *dataclasses*. Por isso, apenas classes como *BlockModel*, que armazenam dados e não contêm lógica complexa, utilizaram o recurso.

- **Maior clareza e segurança com type hints:** a introdução das anotações de tipo possibilitou validações automáticas com ferramentas como o *mypy*, facilitando a identificação de erros em tempo de desenvolvimento, além de tornar o código mais legível e acessível para novos desenvolvedores.

- **Separação de responsabilidades e baixo acoplamento:** a nova organização em camadas promoveu uma divisão clara entre controle, modelo, persistência e interface gráfica. Essa abordagem melhora o isolamento de funcionalidades, reduz o acoplamento e favorece testes automatizados e manutenção evolutiva.

- **Maior flexibilidade com arquivos JSON:** a substituição dos arquivos XML e Python por arquivos JSON para armazenar preferências, blocos e templates tornou os dados mais fáceis de editar, validar e integrar com outras ferramentas. Isso também permite a edição externa das configurações e facilita a extensibilidade da plataforma.

- **eliminação de ciclos de dependência:** ao remover interdependências problemáticas entre módulos, como entre *system.py* e *port.py*, o código tornou-se mais modular, permitindo alterações localizadas sem impacto em outras partes do sistema.

- **Melhora de desempenho com cache:** a aplicação de decoradores contribuiu para otimizações de desempenho. O uso desses decoradores possibilitaram o a reutilização de processamentos custosos no software.

- **Carregamento de dados mais eficiente:** foi introduzido a centralização do carregamento de configurações via a classe *ConfigLoader*, que além de unificar o acesso aos dados, eliminou a necessidade de carregamento completo na inicialização.

- **Instalação de extensões por meio da interface gráfica**: com o desenvolvimento desta funcionalidade tornou-se viável instalar novos blocos, sem a necessidade de intervenção direta no código-fonte ou nos diretórios do sistema. Embora a nova funcionalidade esteja funcional, atualmente somente uma extensão para o Mosaicode foi reescrita no novo formato JSON, sendo ela *Javascript / Web Audio API*. As demais extensões ainda estão no formato antigo e devem ser reescritas para funcionar normalmente com a nova versão do sistema.

  A Figura 14 mostra um diagrama desenvolvido no ambiente reescrito.

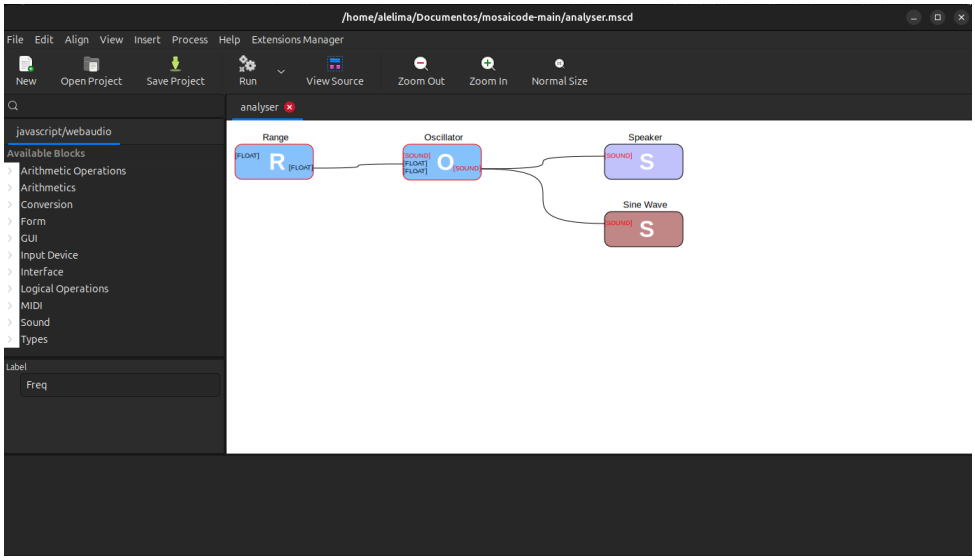

Figura 14 – Diagrama feito no ambiente de programação Mosaicode reescrito

Foram realizados testes automatizados visando avaliar o comportamento das principais camadas da aplicação. Os testes foram conduzidos utilizando a ferramenta pytest [1], com os testes cobrindo os diretórios que possuem os componentes de lógica de negócio e persistência de dados.

A estrutura de testes abrange 29 arquivos de teste, organizados em 180 casos de teste individuais que cobrem as principais funcionalidades do sistema. A configuração de testes utiliza marcadores específicos para categorização e filtros para suprimir avisos de depreciação, garantindo execução limpa e organizada.

De modo geral, os testes associados à camada de modelo apresentaram resultados satisfatórios, com ampla cobertura e execução bem-sucedida. Os módulos responsáveis pela definição de entidades fundamentais — como diagramas, blocos, portas e preferências — também foram validados com sucesso. Também foram validados os testes básicos relacionados a funcionalidades utilitárias e estrutura do sistema.

Por fim, testes mais dependentes de interface gráfica e integração com suas bibliotecas externas, apresentaram falhas em sua execução devido à ausência de ambiente gráfico configurado para testes automatizados.

Os códigos reescritos do Mosaicode e da extensão *Javascript / Web Audio API* estão publicados no site GitHub para seguirem sendo aprimorados dentro de sua proposta.

[1] pytest. Disponível em: <https://docs.pytest.org/>. Acesso em: 15 jul. 2025.

6 Conclusão

Este projeto foi criado com a finalidade de atualizar a base de código do ambiente Mosaicode, tornando-a mais alinhada com as versões mais recentes da linguagem Python e apta para expansões futuras. Ademais, buscou-se proporcionar maior manutenibilidade, clareza estrutural e facilidade de uso para a comunidade de desenvolvedores e artistas que empregam a ferramenta em projetos de programação visual.

A aplicação foi reformulada com ênfase na modularidade, segurança de tipos, verificação de dados e performance. A implementação de práticas contemporâneas, como tipagem estática, uso de `dataclasses`, reorganização de arquivos de configuração no formato JSON e realização de testes automatizados, auxiliou na criação de uma base robusta e expansível.

Durante este trabalho, mesmo sem conseguir implementar todas as melhorias que imaginei inicialmente, o processo trouxe aprendizados durante o caminho. Analisar um sistema legado me forçou a entender profundamente suas decisões de projeto, suas limitações e, ao mesmo tempo, reconhecer seus méritos. Em vários momentos me deparei com trechos de código difíceis de interpretar ou manter. Essas dificuldades mostraram a importância de boas práticas de desenvolvimento.

Também ficou claro para mim como a arquitetura influencia diretamente na possibilidade de colaboração e evolução do software. Ao reestruturar o código, priorizei uma organização que permitisse maior testabilidade e reutilização de componentes, abrindo caminho para que outros desenvolvedores possam contribuir mais facilmente com o projeto. Foi um exercício contínuo de análise crítica, muitas vezes optando por soluções mais simples, em vez de tentar manter estruturas complexas apenas por compatibilidade.

6.1 Trabalhos Futuros

Nesta seção são apresentadas algumas sugestões, como trabalhos futuros.

- Reescritura das demais extensões do Mosaicode;
- Realizar a implementação e ampliação de testes para todas as camadas de arquitetura;
- Criação de um gerenciamento sistêmico com atualização automática de extensões, verificando versão, segurança e compatibilidade entre os pacotes;

- Exportação e integração com outros sistemas de modelagem visual, através da conversão de arquivos **.mscd** em **YAML** ou **SQLite**.

Referências

AHMED, S. U.; JACCHERI, L.; SINDRE, G.; TRIFONOVA, A. Conceptual framework for the intersection of software and art. In: *Handbook of Research on Computational Arts and Creative Informatics*. [S.l.]: IGI Global, 2009. p. 26–44. Citado na página 10.

TRIBE, M.; JANA, R.; GROSENICK, U. *New Media Art (Taschen Basic Art)*. [S.l.]: Taschen America, LLC, 2006. Citado na página 10.

HILS, D. D. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, Elsevier, v. 3, n. 1, p. 69–101, 1992. Citado na página 10.

RIBAS, E.; BIANCO, G. D.; LAHM, R. A. Programação visual para introdução ao ensino de programação na educação superior: uma análise prática. *RENOTE*, v. 14, n. 2, 2016. Citado na página 10.

SCHIAVONI, F. L.; SANDY, J. M. d. S.; CARDOSO, T. T. S.; GOMES, A. L. N.; RESENDE, F. R. O ambiente de programação visual mosaicode. In: *Anais da 9ª Sessão de Ferramentas do CBSoft*. [S.l.]: Sociedade Brasileira de Computação, 2018. v. 1, p. 25–35. Citado 3 vezes nas páginas 10, 18 e 19.

RESENDE, F. R. *Processamento Digital de Imagens e Visão Computacional no ambiente Mosaicode*. Tese (Monografia (Bacharelado em Ciência da Computação)) — Universidade Federal de São João del-Rei, 2019. Citado 3 vezes nas páginas 11, 18 e 19.

SCHIAVONI, F. L.; CARDOSO, T. T. S.; GOMES, A. L. N.; RESENDE, F. R.; SANDY, J. M. S. Utilização do ambiente mosaicode como ferramenta de apoio para o ensino de computação musical. In: *Proceedings of the VIII Workshop on Ubiquitous Music (UBIMUS)*. São João del-Rei - MG - Brazil: [s.n.], 2018. v. 8, p. 33–44. Citado 2 vezes nas páginas 11 e 19.

COSTA, L. O.; SCHIAVONI, F. L. Colaboração em criação artística com o mosaicode. p. 1–8, 2019. Trabalho apresentado no Departamento de Ciência da Computação da UFSJ. Citado na página 11.

BREJE, A.-R.; GYORÖDI, R.; GYORÖDI, C.; ZMARANDA, D.; PECHERLE, G. Comparative study of data sending methods for xml and json models. *International Journal of Advanced Computer Science and Applications*, Science and Information (SAI) Organization Limited, v. 9, n. 12, p. 198–204, 2018. Citado na página 11.

GATTO, R. A. *Estratégias para reestruturação de código legado visando à utilização de aspectos*. Dissertação (Mestrado) — Universidade Estadual de Maringá, 2007. Citado na página 13.

BARICHELLO, L. G. E. *Identificação de oportunidades de refatoração e análise da qualidade de código por meio de métricas de código-fonte*. Dissertação (B.S. thesis) — Universidade Tecnológica Federal do Paraná, 2022. Citado 2 vezes nas páginas 13 e 23.

ACITO, F. Predictive analytics with knime. *Analytics for citizen data scientists. Switzerland: Springer*, Springer, 2023. Citado na página 13.

BERTHOLD, M. R.; CEBRON, N.; DILL, F.; GABRIEL, T. R.; KÖTTER, T.; MEINL, T.; OHL, P.; THIEL, K.; WISWEDEL, B. Knime-the konstanz information miner: version 2.0 and beyond. *AcM SIGKDD explorations Newsletter*, ACM New York, NY, USA, v. 11, n. 1, p. 26–31, 2009. Citado na página 14.

FERENCZ, K.; DOMOKOS, J. Using node-red platform in an industrial environment. *XXXV. Jubileumi Kandó Konferencia, Budapest*, p. 52–63, 2019. Citado na página 14.

GARBEV, A. Possibilities for visual programming in robotics via node-red. In: IEEE. *2022 International Conference Automatics and Informatics (ICAI)*. [S.l.], 2022. p. 367–371. Citado na página 15.

PUCKETTE, M. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, Tachikawa, p. 37–41, 1996. Citado na página 15.

MOORE, R. K. On the use of the'pure data'programming language for teaching and public outreach in speech processing. In: *INTERSPEECH*. [S.l.: s.n.], 2014. p. 1498–1499. Citado na página 15.

HOWARD, D. M.; DAFFERN, H.; BRERETON, J. Four-part choral synthesis system for investigating intonation in a cappella choral singing. *Logopedics Phoniatrics Vocology*, Taylor & Francis, v. 38, n. 3, p. 135–142, 2013. Citado na página 16.

GONÇALVES, L. L.; SCHIAVONI, F. L. Do harpia ao mosaicode a evolução de um ambiente de programação visual. In: SBC. *Escola Regional de Engenharia de Software (ERES)*. [S.l.], 2020. p. 316–324. Citado 3 vezes nas páginas 17, 18 e 20.

GONçALVES, L. L. *Sound Design com o Mosaicode*. Tese (Monografia (Bacharelado em Ciência da Computação)) — Universidade Federal de São João del-Rei, 2017. Citado na página 19.

HUNT, J. *Advanced Guide to Python 3 Programming*. Cham, Switzerland: Springer Nature Switzerland AG, 2020. (Undergraduate Topics in Computer Science). Citado na página 20.

GOMES, L. T. P. S. Criação de um documento de padronização de codificação java no lcc. Universidade Federal de Minas Gerais, 2013. Citado na página 23.

LEE, M.-C. Software quality factors and software quality metrics to enhance software quality assurance. *British Journal of Applied Science & Technology*, v. 4, n. 21, p. 3069–3095, 2014. Citado na página 23.

BORQUE, P.; FAIRLEY, R. Guide to the software engineering body of knowledge version 3.0. *IEEE Computer Society Staff*, 2014. Citado na página 23.

LUCAS, A. P. L. Gestão da manutenção de software: Um estudo de caso. 2016. Citado na página 24.

PADUELLI, M. M. *Manutenção de Software: problemas típicos e diretrizes para uma disciplina específica*. Tese (Doutorado) — Universidade de São Paulo, 2007. Citado na página 24.

JAMIL, M. A.; ARIF, M.; ABUBAKAR, N. S. A.; AHMAD, A. Software testing techniques: A literature review. In: IEEE. *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*. [S.l.], 2016. p. 177–182. Citado 2 vezes nas páginas 24 e 25.

SOMMERVILLE, I. Software engineering (ed.). *America: Pearson Education Inc*, 2011. Citado na página 24.