INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO

RIO GRANDE DO SUL

CAMPUS CANOAS

CURSO TÉCNICO EM INFORMÁTICA INTEGRADO AO ENSINO MÉDIO

Luiz Henrique Broch Lago

[APÊNDICE III

DOCUMENTAÇÃO DO SISTEMA]

Quartz Wordcode: Plataforma Base de Interpretação com Execução em Baixo Nível

**Orientador**: Rafael Coimbra Pinto

Canoas, 12 de junho de 2019

# Proposta de Trabalho de Conclusão de Curso

## Descrição do Problema

Atualmente o uso de linguagens interpretadas na programação está se tornando cada vez mais frequente, pois com um alto nível de abstração, conseguem facilitar diversas áreas da programação. No desenvolvimento de um sistema, um desenvolvedor se deparará com um barreira de qual linguagem utilizar; cada uma tem sua ‘especialidade’ em algum aspecto, porém surge uma dificuldade quando se necessita que duas ou mais linguagens de programação em um mesmo projeto e que se interajam de forma eficiente. Como Daniel Barrett apresenta em seu artigo “Polylingual Systems: An Approach To Seamless Interoperability” (1998), apesar de existirem algumas abordagens que propõem o uso da *multilinguagem* (duas ou mais linguagens de programação), o principal problema é que apresentam a falta de uniformidade ou a transparência da interoperabilidade, ou seja, a falta de padronização em uma comunicação transparente, o que afetaria sua legibilidade, manutenção e coesão.

## Proposta de Solução

Para resolver o problema abordado, foi proposta a criação de uma plataforma base que comportaria a interpretação de softwares compilados, de linguagens diferentes, em uma linguagem binaria em comum semelhante ao código binário de máquina.

O software proposto, se comportará como uma plataforma virtual, ou também chamada Virtual Machine[[1]](#footnote-1), em que os programas inseridos serão interpretados a partir do código binário originado que permitirá a criação de interfaces de comunicação entre os diferentes programas, por conseguinte, todos os tipos de paradigmas das linguagens de programação serão executados em cima desse mesmo ambiente virtual.

O software interpretará códigos com *opcodes* de tamanhos de 2 bytes, o suficiente para abranger uma grande diversidade de comandos, e que não sature o tamanho do código da aplicação; nesse contexto, o código binário será nomeado de *wordcode*, com a referência ao *word[[2]](#footnote-2)*, que em linguagens de baixo nível simboliza 2 bytes.

### 1.2.1 Objetivo Geral

Desenvolver um software capaz de carregar códigos fontes em uma plataforma virtual e que possibilite compartilhar funções e métodos entre eles a partir de interfaces lógicas.

### 1.2.2 Objetivos Específicos

* Desenvolver o software proposto com um alto nível de desempenho.
* Desenvolver programas para o interpretador proposto para uso de testes e exemplos.
* Implementar instruções gerais em nasm para optimização.
* Executar testes de desempenho entre o interpretador e linguagens mais usadas.
* Implementar JIT no sistema para compilação e execução em tempo real, otimizando o desempenho do software.

# Trabalhos relacionados

## Java

Criado inicialmente em 1991 com o nome de Projeto Green, e algum tempo após nomeada para Java. É uma linguagem hibrida com um alto nível de desempenho se comparando até a linguagens de baixo nível como C++.

Para resolver o problema abordado Java possibilita a interligação de diversos código Java dinamicamente, utilizando sistemas pré-compilados ou pós-compilados durante o tempo de execução. Também permite interligar códigos de outras linguagens interpretadas, não rodáveis a partir da JVM, sendo uma ligação não nativa da linguagem, necessitando bibliotecas externas para a execução de um interpretador da linguagem em cima do processo Java, diminuindo consideravelmente o desempenho pelo alto nível de abstração.

Existem ainda algumas linguagens que são capazes de rodar nativamente na JVM, porém essas linguagens são limitadas a utilização de orientação a objeto como visto no artigo “Conhecendo as Principais Linguagens para JVM” (Felipe Almeida; Flavia Oliveira, 2019), sendo inviável a execução de linguagens com outro tipo de paradigmas ou com níveis de abstrações menores que Orientadas a Objetos.

## WEB SERVICE

Web Service é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Os Web Services são componentes que permitem às aplicações enviar e receber dados, sendo que cada aplicação pode ter a sua própria "linguagem", que é traduzida para uma linguagem universal, um formato intermediário como XML, Json, CSV, etc.

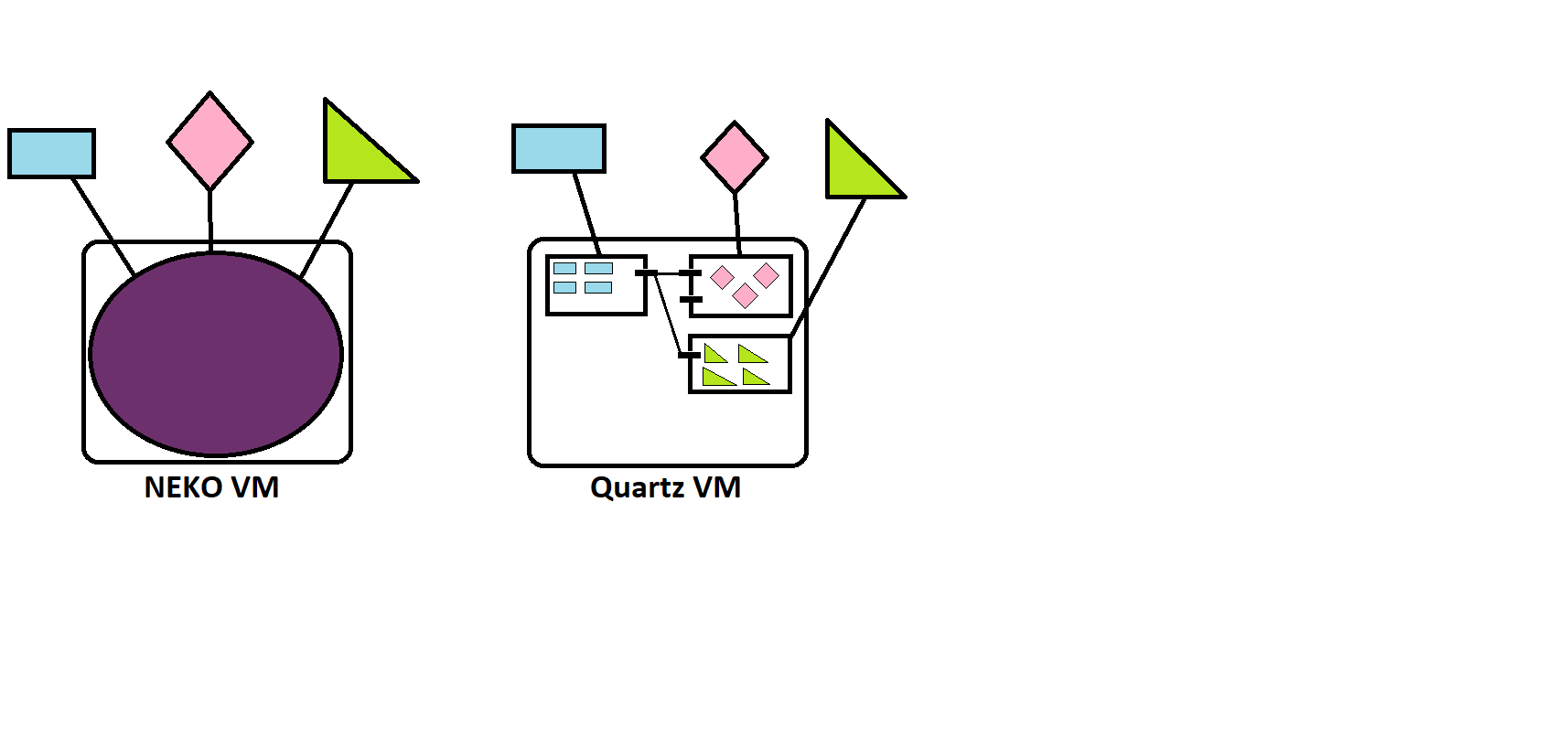
Usar Webservices pode ser uma boa ideia, porém a cada chamada de "método" teria um custo alto em performance, pois se utilizaria de sockets, o que tornaria o programa lento demais para softwares que necessitam de altos níveis de desempenho.

## NEKO

A linguagem Neko de acordo com a documentação do site oficial (Nicolas Cannasse, 2019), apresenta uma proposta de linguagem de alto nível dinamicamente tipada, projetado para fornecer uma linha de execução para várias linguagens diferentes. O desenvolvedor passa o código fonte da linguagem desejada para um gerador Neko, que o transforma para um código fonte da linguagem Neko, em que após pode ser compilado e executado. A grande vantagem da linguagem Neko, é a sua capacidade de multilinguagens, e por ser dinamicamente tipada, consegue se adequar em diversos tipos de cenários com linguagem diferentes.

A principal diferença entre a linguagem Neko e o software proposto nesse artigo, é a forma que as maquinas virtuais atuam para fazer a interação entre linguagem de níveis de abstrações diferentes. Enquanto a linguagem Neko soluciona esse problema elevando o nível de abstração para que todas as linguagem consigam interagir umas com as outras, a Quartz VM apresenta um conceito diferente, que separa cada tipo de linguagem em contextos distintos, que preserva as características da nível de abstração, e não as mistura entre si, como pode ser visto representado na figura 1. Para estabelecer a comunicação entre os programas, é criado uma abordagem de interfaces, que se assemelham a funções, a quais são pré definidas, e disponibilizadas para que qualquer outro contexto às acesse.

Figura 1. Imagem ilustrativa entre os níveis de abstração da Neko VM e da Quartz VM.



Fonte: Figura autoral.

Para exemplificar, cada tipo de paradigma trabalha com *array[[3]](#footnote-3)*’s de forma diferentes, então para que todos tenham acesso aos dados do *array* a Neko VM aumenta o nível de abstração das linguagens para que nenhuma acabe tendo incompatibilidade com a outra. Já na Quartz VM, cada contexto possui uma memória distinta, que não pode ser acessado por qualquer contexto, então para acessar a um *array* que esteja em outro contexto, o programa deve acessar a interface pré-programada daquele contexto, no qual irá ser executada uma de suas funções internas, que irá manusear o vetor de forma que ele não seja danificado ou manuseado de forma indevida.

## LLVM

A LLVM é uma coleção de tecnologias modulares de compiladores (LLVM Project Documentação, 2019), que fornece uma estratégia moderna de compilação capaz de suportar a união estática e dinâmica de linguagens de programação diferentes. Atualmente possui capacidade de executar diversos tipos de linguagens de programação, como C, C++, Fortran, entre outros.

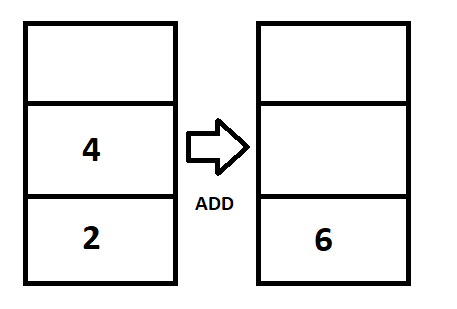
O funcionamento da LLVM se baseia na compilação dos códigos fontes para código objetos, onde eles são ligados, e no final é gerado um executável. Foi escrito originalmente com o objetivo de otimizar o tempo da compilação, ligação, e execução de programas em linguagens diferentes. A principal vantagem da LLVM é a facilidade do uso para fazer projetos de linguagens diferentes, porém de mesma forma que linguagens compiladas para código de máquina, o executável não pode ser utilizado em outro sistema operacional sem compatibilidade.

## LUA

Uma linguagem de alta simplicidade, portabilidade, eficiência e embutilidade, de acordo com o artigo “The Implementation of Lua 5.0” (Roberto I., Luiz F., Waldemar C., 2019), projetada para expandir aplicações em geral, por ser um linguagem extensível, ou seja, que possa unir programas feitos em linguagens diferentes, e também para ser embarcada em softwares mais complexos. O linguagem lua é executada via interpretação de *bytecodes*, utilizando uma VM[[4]](#footnote-4) que possui diversas implementações para diferentes tipos de linguagens, como Java ou C.

Até a versão 4 do lua, sua arquitetura era baseada em *stack* (do inglês, pilha), onde todos os cálculos eram feitos usando a pilha, como exemplo, para efetuar a soma de 2 com 4, era colocado esses dois números na pilha, e efetuado o comando de soma, e o resultado era colocado de volta na pilha, como ilustrado na figura 2.

Figura 2. Representação de uma soma em máquinas virtual do tipo *Stack-Based* (do inglês, baseado em pilha).

  
Fonte: Figura autoral.

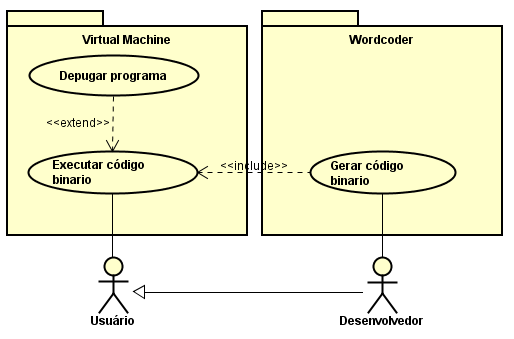
A partir da versão 5.0, a máquina virtual da linguagem Lua começou a se basear no tipo *register* (do inglês, registro), no qual os cálculos eram feitos a partir de registradores, que podiam reter a memória para poderem reutilizar os números mais futuramente na aplicação. A Quartz VM também se fundamenta em máquinas virtuais baseadas em *registers*. A principal diferença entre a Quartz VM e a VM utilizada na linguagem Lua, é o seu método de execução, pois na Linguagem Lua as variáveis não utilizadas são liberadas automaticamente e de forma incremental, e o gerenciamento da memória é definido totalmente por parte da VM. Na Quartz VM, todo o gerenciamento de memória é feito pelo próprio software.

# Modelo de Requisitos

## Diagrama de Casos de Uso

É possível observar na figura 3 que para o usuário, o único caso de uso possível para esse software, será a execução do código binário, que foi gerado anteriormente. Também para a execução da máquina virtual, é possível utiliza-lo com a opção de depuração, que é disponibilizada ao passar o parâmetro de depuração (-d). Essa opção irá dar uma saída detalhada com informações da execução do programa a cada passo de forma que um desenvolvedor ou um analista possa trabalhar no código em que ele está desenvolvendo.

Figura 3. Figura referente ao caso de uso geral do sistema



Fonte: Figura autoral.

Para o desenvolvedor, ele herdará os casos de uso do usuário, ele terá acesso a biblioteca *Wordcoder*, que permite ao desenvolvedor programar e gerar códigos fontes binários compatíveis com a Quartz VM. Para todo o código gerado, ele deve ser executado em algum momento, então ele será passado para a VM que executaria o código.

## Especificação de Casos de Uso

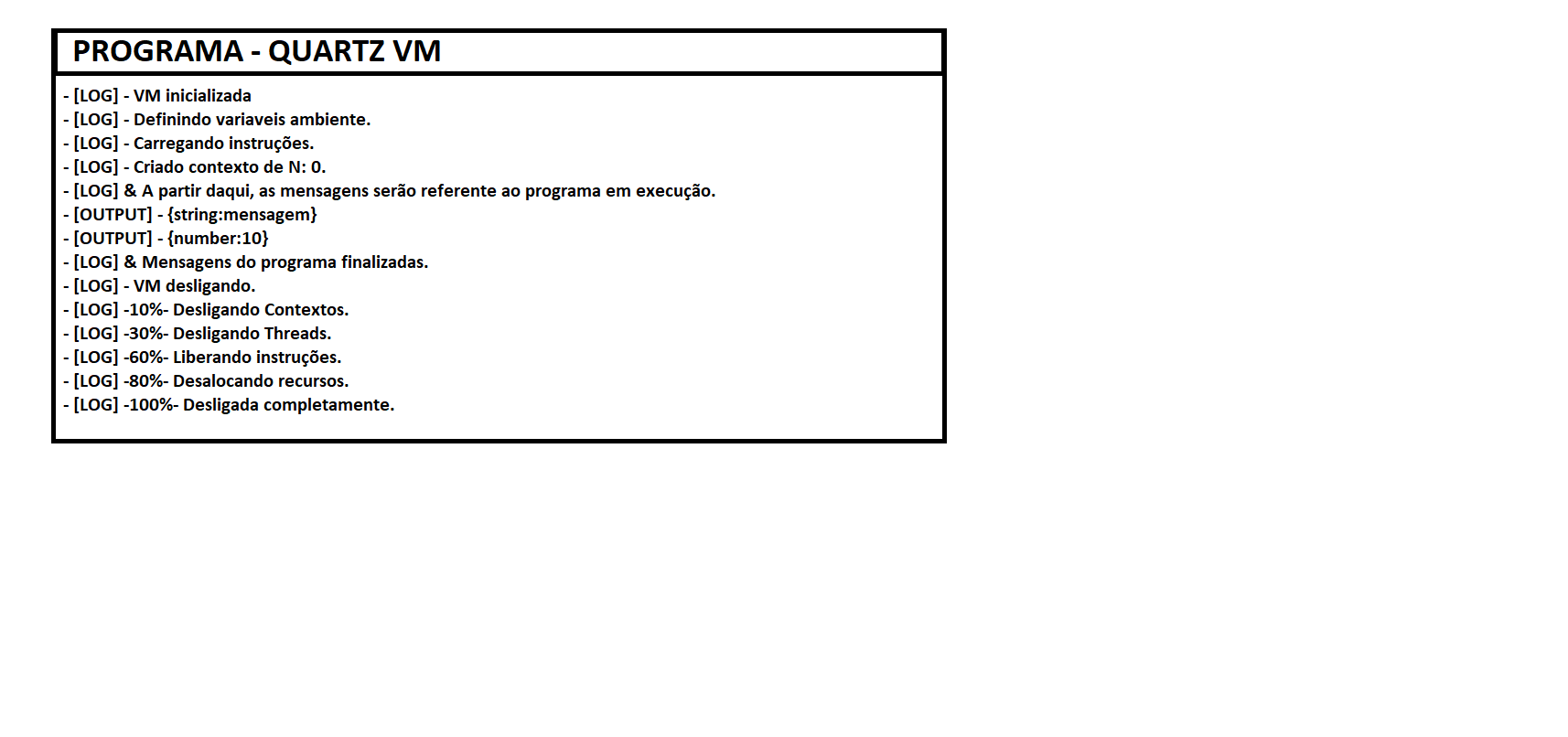
|  |  |
| --- | --- |
| Código e Nome do Caso de Uso | Executar código binário |
| Ator Primário:  Ator Secundário: | Usuário |
| Fluxo Principal de Eventos | P1. O usuário chama o software via linha de comando ou pelo executável passando como parâmetro o programa a ser executado.  P2. O software retorna ao usuário sucessivamente a saída do programa, se houver, até o fim do mesmo. (A1) (E1)  P3. O usuário pressiona uma tecla qualquer.  P4. O software finaliza.  P5. O caso de uso se encerra. |
| Fluxos Alternativos | A1. Caso passado o parâmetro de debug (-d).  A1.1. O software da a saída na tela das memórias depois da execução do comando.  A1.2. Retorna para P2. |
| Fluxos de exceção | E1. Caso se ative a bandeira de erros da execução  E1.1 O software encontra o erro correspondente  E1.2 O software da a saída na tela do erro.  E1.3 O usuário pressiona uma tecla qualquer.  E1.4 O software finaliza.  E1.5 O caso de uso se encerra. |

|  |  |
| --- | --- |
| Código e Nome do Caso de Uso | Gerar código binário |
| Ator Primário:  Ator Secundário: | Desenvolvedor |
| Fluxo Principal de Eventos | P1. O desenvolvedor cria um objeto da classe Wordcoder.  P2. O desenvolvedor prepara o programa que deseja utilizando as funções da classe.  P3. O desenvolvedor solicita para a classe o binário.  P4. A classe retorna o binário para o desenvolvedor.  P5. O desenvolvedor salva esse binário em um arquivo.  P6. O desenvolvedor inicia a máquina virtual com o arquivo contendo o binário.  P7. Inicia o caso de uso Executar o código binário.  P8. O caso de uso se encerra. |

## Protótipos DE TELA e/OU Storyboards

O protótipo apresentado é uma tela de um console, pois o programa não irá possuir interface gráfica interativa com o usuário. A saída de informações no console será informações da execução da VM, que serão diferenciadas entre cada tipo de informação, que pode ser notado entre colchetes na figura 4.

Figura 4. Protótipo de tela da execução do software  
  
Fonte: Figura autoral.



Todos os tipos de informações possíveis são:

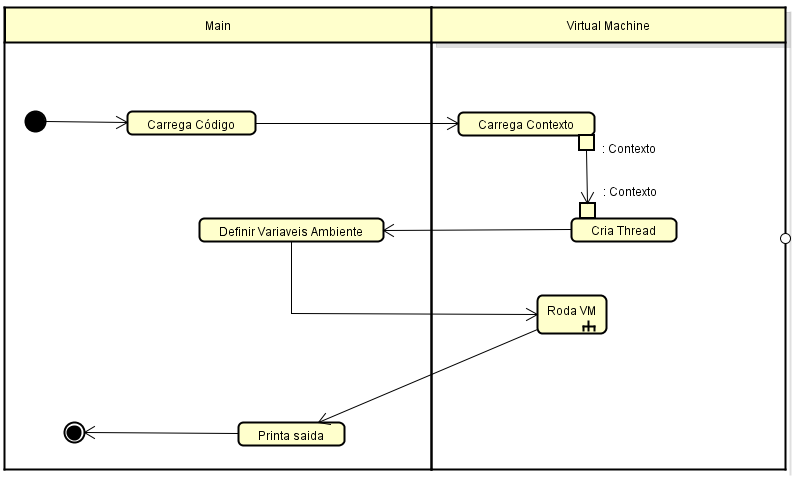
* [LOG]: Exibe informações decorrentes de processos chaves para o funcionamento da VM.
* [INFO]: Exibe informações adicionais que não interferem no funcionamento da VM.
* [DEBUG]: Exibe informações da aplicação em execução para visualização do funcionamento do código.
* [OUTPUT]: Exibe informações que foram pré programadas no Wordcode inserido na execução da VM.
* [SUCESS]/[ERROR]: Mostra a saída no console de procedimentos feitos que estavam sucessíveis a erros. Caso o programa passou sem nenhuma interferência é exibido o tipo de informação ‘*[SUCESS]*’, caso contrário
* [LOG] -[*CONTEXTO*]-: Semelhante ao tipo ‘*[LOG]*’, porém apresenta apenas processos chaves pra a execução do de um contexto específico. Dentro dos colchetes é mostrado qual contexto pertence aquela informação.

# Modelagem do Sistema

## DIAGRAMAS DE ATIVIDADE

O diagrama de atividades da figura 5 representa quando o programa é inicializado para a execução. Ao inicializar, o software busca o código binário para carrega-lo em um contexto, logo após é criado uma thread inicial para a execução. Antes da inicialização da VM são definidas algumas variáveis ambiente, para fins de medições do software, e após inicializa a execução da Máquina Virtual. No final é apenas mostra ao usuário a saída final do programa com detalhes de erro e logs de execução.

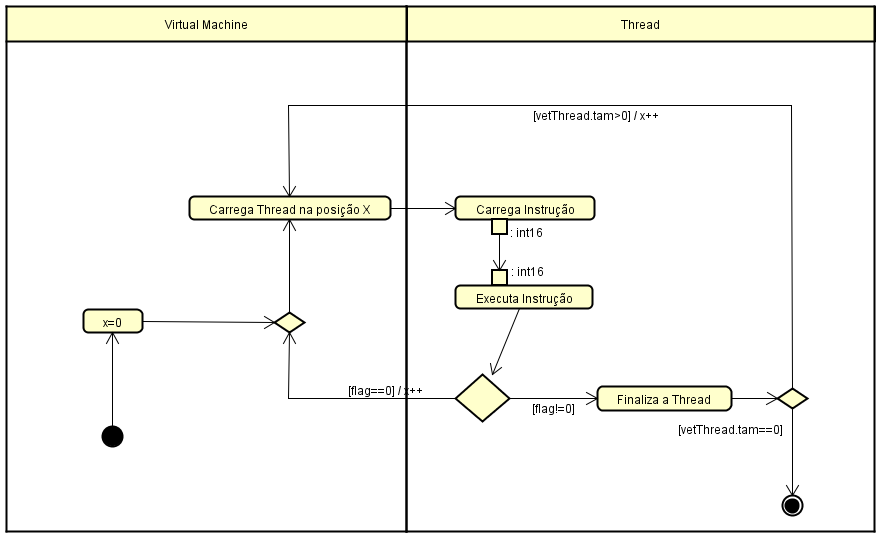
Figura 5. Diagrama de atividades geral, executado ao iniciar o software



Fonte: Figura autoral.

No diagrama da figura 6, é descrito a atividade realizada internamente na VM para a execução do programa. Primeiramente se é definido o vetor de Threads que serão gerenciadas, e após e inserido um laço que irá repetir alternando as Threads de dentro do vetor, onde todas irão executar sua instrução. Quando uma Thread causa uma exceção, ou finaliza sua execução, sua variável *flag[[5]](#footnote-5)* é modificada, finalizando-a logo após. Quando a última Thread finalizar é encerrado o processo da Máquina Virtual.

Figura 6. Diagrama de atividades referente a execução da Máquina Virtual.



Fonte: Figura autoral.

## Modelagem do Banco de Dados

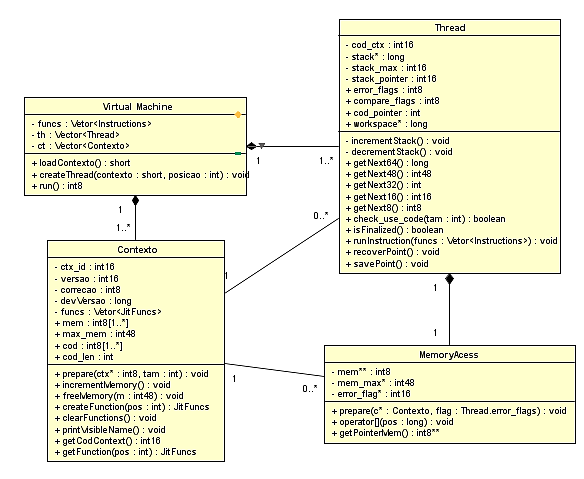
O software proposto não irá fazer uso de banco de dados para o seu funcionamento, pois não haverá necessidade de armazenar nenhuma informação. Na execução independente do software o desenvolvedor será capaz de utilizar acesso a arquivos ou a sockets para acessar a bancos de dados e salvar informações da execução.

## DIAGRAMA DE CLASSES

No diagrama de classes conforme a figura 7 apresenta apenas 4 classes que compões o software. A *Virtual Machine* é a principal classe, e dentro dela que são encontradas as classes de *Thread*’s e *Contexto*’s. Cada *Contexto* contém um código que vai ser executado, e uma memória de tamanho dinâmico para trabalho uso geral, além de possuir algumas informações de qual versão foi compilado aquele código, e a versão do código. Possui também funções para manipular seus dados e mudar dinamicamente a memória de uso geral.

As *Thread*’s estão sempre vinculadas a algum *Contexto*, pois são elas que executam os códigos e fazem o programa funcionar. A *Thread* possui tanto acesso a memória de uso geral do *Contexto* quanto também possui uma memória de trabalho de tamanho estático de 256 espaços de 64 bits. Essa memória não possui verificação de acesso, que diferente da memória do *Contexto* possui, o que a torna mais rápida para uso em tarefas que necessitam constantemente variáveis especificas. Como a memória de trabalho é vinculado a *Thread*, ao mudar de contexto, a memória de uso geral é alterada, porém a memória de trabalho se mantem os valores.

Figura 7. Diagrama de classes

  
Fonte: Figura autoral.

# Proposta de Solução Tecnológica Escolhida

## bREVE DESCRIÇÃO DA METODOLOGIA

A pesquisa proposta nesse artigo classifica-se como exploratória, pois de acordo com Antônio Gil em seu livro “Como elaborar projetos de pesquisa” (2002):

*“Estas pesquisas têm como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses. (...)”*

Essa pesquisa tem como objetivo construir soluções a cerca de um problema ou o aprimoramento de ideias. Já a classificação dos procedimentos técnicos utilizados se encaixaria no tipo experimental, pois serão utilizados diversos experimentos para verificar a capacidade de desempenho do software.

Pra a análise de requisitos foram utilizadas as técnicas de *brainstorm* e o desenvolvimento de *User Stories* para o desenvolvimento do escopo do projeto. A *User Storie* desenvolvida para a análise dos requisitos do projeto:

*“Como um Desenvolvedor, eu quero produzir um software utilizando diversas linguagens de modo que possa usufruir das vantagens de cada uma”*

Na modelagem dos requisitos usando o UML, foram usados os diagramas de Caso de Uso, diagrama de Classe e diagrama de Atividades, pois melhor se adequam ao sistema, e por não utilizar banco de dados, não foi usado o diagrama ER.

O programa será feito utilizando as linguagens C++ e *assembly[[6]](#footnote-6)* em algumas partes, que irá proporcionar o máximo de desempenho do software. A Quartz VM irá implementar uma tecnologia conhecida como JIT (*Just In Time[[7]](#footnote-7)*) para otimizar códigos e processes do software do usuário durante a execução da VM, e para isso será usado a biblioteca AsmJIT feita por Petr Kobalicek. Será disponibilizado uma biblioteca *Wordcoder.h*feita em C++ que possibilitará a criação de wordcodes compatíveis com a Quartz VM.

Os testes serão realizados para fins de comparação no desempenho do software. Serão feitos testes utilizando os algoritmos de ordenação *BubbleSort* e *QuickSort*¸ além da função de Ackermann. Foram selecionados esses algoritmos por suas características marcantes; a característica do *BubbleSort* por ser de complexidade quadrática, enquanto o *QuickSort* foi escolhido por ser recursivo. Já a função Arckermann foi escolhida por possuir um alto nível de complexidade e recursividade.

## TECNOLOGIAS ADOTADAS

* C++: www.cplusplus.com/
* Biblioteca AsmJIT: github.com/asmjit/asmjit

## FERRAMENTAS ADOTADAS

* Eclipse IDE C++: www.eclipse.org/
* NASM:www.nasm.us
* Mingw-w64: mingw-w64.org
* BinaryViewer: www.proxoft.com/BinaryViewer.aspx
* DontPad++: notepad-plus-plus.org
* IDA Freeware:www.hex-rays.com

# Cronograma

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Atividade** | **Jan** | **Fev** | **Mar** | **Abr** | **Mai** | **Jun** | **Jul** | **Ago** | **Set** | **Out** | **Nov** | **Dez** |
| Escolha do Orientador | X | X |  |  |  |  |  |  |  |  |  |  |
| Escolha do Tema | X | X |  |  |  |  |  |  |  |  |  |  |
| Apêndice II |  | X | X |  |  |  |  |  |  |  |  |  |
| Comandos Básicos |  | X | X | X | X | X |  |  |  |  |  |  |
| Apêndice III |  |  | X | X | X |  |  |  |  |  |  |  |
| Programas Testes |  |  | X | X |  |  |  | X | X | X |  |  |
| Diagramas |  |  | X | X | X | X |  |  |  |  |  |  |
| Comandos Avançados |  |  |  | X | X | X | X | X |  |  |  |  |
| JIT avançado |  |  |  |  | X | X | X |  |  |  |  |  |
| IFCITEC |  |  |  |  |  |  | X | X |  |  |  |  |
| Versão Final |  |  |  |  |  |  | X | X | X | X |  |  |
| Documento Final |  |  |  |  |  |  |  | X | X | X | X |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

# referências

ALMEIDA, Felipe Rodrigues; OLIVEIRA, Flavia Castro. “Conhecendo as Principais Linguagens para JVM. Conhecendo as Principais Linguagens para JVM”. Disponível em: http://www.univale.com.br/unisite/mundo-j/artigos/42jvm.pdf. Acesso em: 26 maio 2019.

IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique; CELES, Waldemar. “The Implementation of Lua 5.0”. Disponível em: https://www.lua.org/doc/jucs05.pdf. Acesso em: 30 maio 2019.

JUNGTHON, Gustavo; GOULART, Cristian Machado.” Paradigmas de Programação”. Disponível em: https://fit.faccat.br/~guto/artigos/Artigo\_Paradigmas\_de\_Programacao.pdf. Acesso em: 26 maio 2019.

Linguagem Neko. Disponível em: https://nekovm.org/doc/multilang/. Acesso em: 30 maio 2019.

LLVM Project. Disponível em: https://llvm.org/. Acesso em: 04 junho 2019.

MOSS, Eliot B.; BARRETT, Daniel J.; ELIOT, J. “Polylingual Systems: An Approach To Seamless Interoperability. Polylingual Systems: An Approach To Seamless Interoperability”. Disponível em: https://www.researchgate.net/publication/2354798\_Polylingual\_Systems\_An\_Approach\_To\_Seamless\_Interoperability. Acesso em: 28 maio 2019. Acesso em: 28 maio 2019.

GIL, Antonio Carlos. “Como elaborar projetos de pesquisa”. Disponível em: http://www.ufrgs.br/cursopgdr/downloadsSerie/derad005.pdf. Acesso em: 6 jun. 2019.

1. Do inglês, máquina virtual. [↑](#footnote-ref-1)
2. Do inglês, palavra. Na programação em baixo nível é atribuído a 2 bytes o nome *word*. [↑](#footnote-ref-2)
3. Do inglês, Vetor. Geralmente mantém uma cadeia de dados em sequência. [↑](#footnote-ref-3)
4. Virtual Machine, do inglês, máquina virtual. [↑](#footnote-ref-4)
5. Do inglês, bandeira. Usado para simbolizar se há a presença de um erro. [↑](#footnote-ref-5)
6. Linguagem de mais baixo nível legível que é transformada em código binário que o computador interpreta. [↑](#footnote-ref-6)
7. Tecnologia que compila o programa durante a execução do mesmo. [↑](#footnote-ref-7)