Universidade de São Paulo

Faculdade de Filosofia Ciências e Letras de Ribeirão Preto

Projeto SO

Alunos: Caio Uehara e Gustavo Bender

Ribeirão Preto, 23 de junho de 2024

Sumário

1 Introdução

1.1 Propósito

Este documento descreve a implementação de um simulador de escalonador de processos utilizando Java.

O projeto é o trabalho final da disciplina **(5954016) Sistemas Operacionais** do Departamento de Computação e Matemática da USP-RP, aplicada pelo professor Clever Ricardo Guareis de Farias.

O projeto está disponível em três formatos.

- Em código fonte no repositório do GitHub
- Em código fonte no pelo SchedulerSimulatorSource.zip, disponível nesse link.
- Compilado no arquivo SchedulerSimulator.java, por esse link.

1.2 Contextualização

Para a implementação do projeto foi seguido a especificação do diagrama de classe dada pela instrução do projeto, mostrada na figura 1.1.

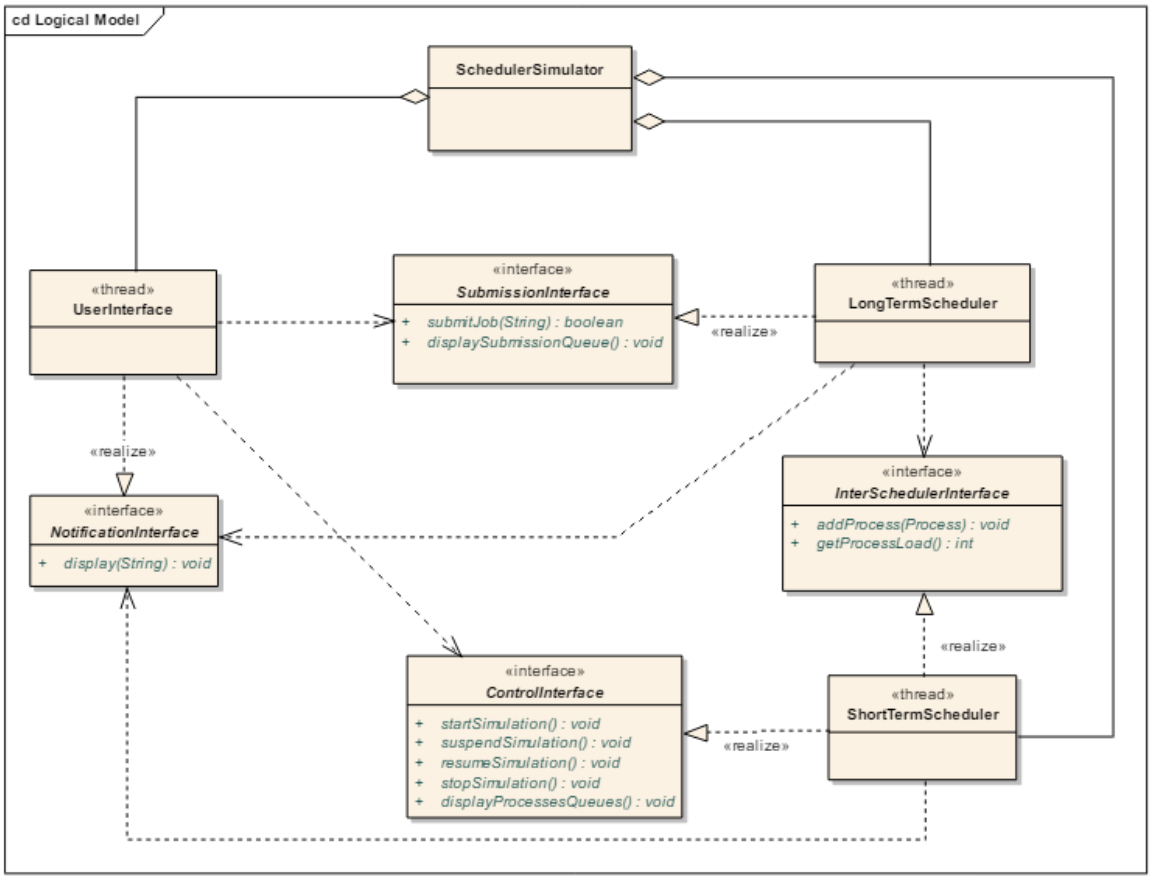

Figura 1.1: Arquitetura do Projeto

O objetivo é simular diferentes estratégias de escalonamento de processos, permitindo a visualização de como cada uma delas se comporta em termos de eficiência e utilização dos recursos do sistema.

Para a simulação da execução de um programa foi definido nas instruções a seguinte sintaxe em *BackusNaur form (BNF)* [1].

Definição de Programa

```
<program> ::= <program_statement> <program_body>
<program_statement> ::= "program" <whitespace> <file_name> <EndOfLine>
<program_body> ::= <begin_statement> <program_behaviour> <
    end_statement>
<begin_statement> ::= "begin" <EndOfLine>
<end_statement> ::= "end" <EndOfLine>
<program_behaviour> ::= (<behaviour_statement>)+
<behaviour_statement> ::= "execute" <EndOfLine> | "block" <whitespace>
     <block_period> <EndOfLine>
<whitespace> ::= " "
<block_period> ::= "1" | "2" | "3" | "4" | "5"
```

Nosso projeto implementou para as Interfaces a biblioteca Java Swing [2], e a política escolhida para o ShortTermScheduler foi a de **filas múltiplas com alternância ciruclar.**

2 Projeto

O projeto foi desenvolvido por uma equipe de dois membros:

- Membro 1 (Caio Uehara): Implementação do parser e Criação da interface gráfica.
- Membro 2 (Gustavo Bender): Desenvolvimento do escalonador de longo prazo e execução do Processo.
- Membro 1 e 2: Desenvolvimento do escalonador de curto prazo e integração com o Swing.

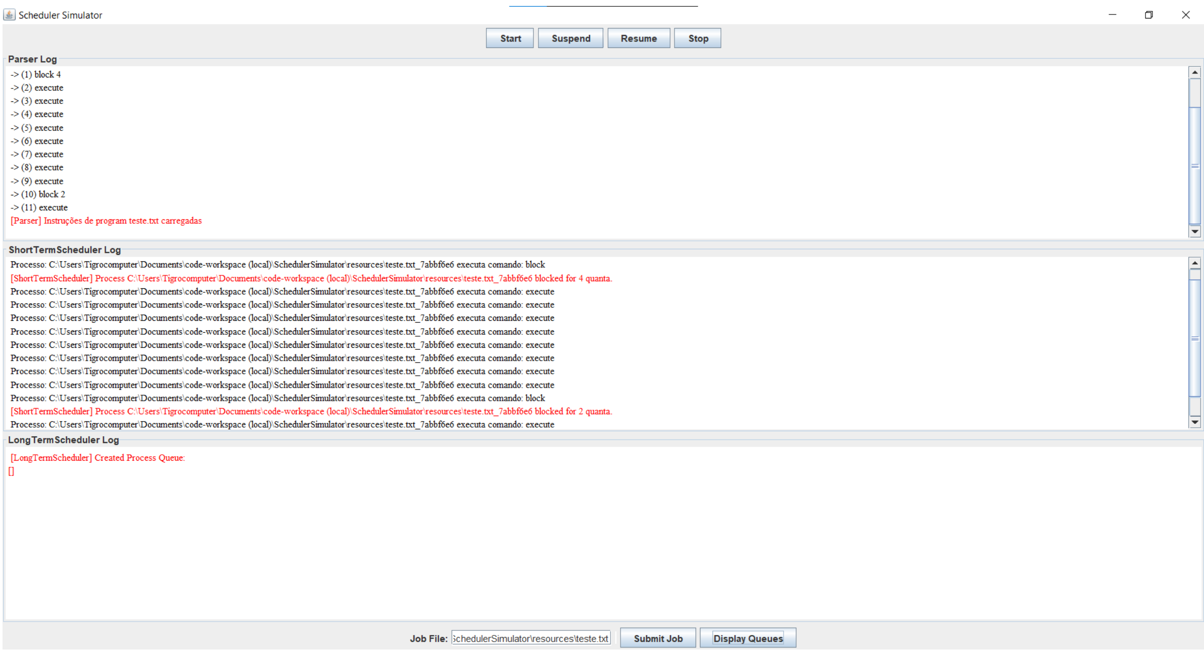

Figura 2.1: Interface do programa

2.1 Implementação do projeto

Para a arquitetura realizada, não houve nenhuma modificação nas classes solicitadas na especificação, mas para uma melhor codificação da solução, houve uma adição de três classes principais, encarregadas de executar ler e validar o programa explicitado pela BNF proposta 1.2:

> Process

> ParserBNF

> NamedValidation

E a adicão das classes componentes da interface:

> ControlPanel

> SubmissionPanel

> TextPanel

Assim, o projeto SchedulerSimulator ficou composto pelas seguintes classes:

Classes

Classe	Responsabilidade
SchedulerSimulator	Gerencia a simulação de escalonamento, coordena a interação entre diferentes componentes do sistema. • (Caio e Gustavo) **ShortTermScheduler(int quantum)**: Construtor da classe que inicializa as filas de processos (pronto, bloqueado e terminado) e define o valor do quantum. • (Caio e Gustavo) **void setThreads(NotificationInterface userInterface)**: Método para definir a interface de notificação que será usada para exibir mensagens ao usuário. • (Caio e Gustavo) **synchronized void addProcess(Process process)**: Adiciona um processo à fila de prontos. • (Caio e Gustavo) **synchronized int getProcessLoad()** : Retorna a quantidade de processos na fila de prontos. • (Caio e Gustavo) **void run()**: Método principal de execução do escalonador. Gera a simulação, executa processos da fila de prontos e gerencia processos bloqueados. • (Caio e Gustavo) **private void executeProcess(Process process)**: Executa o processo atual, decrementando o tempo restante e gerenciando comandos de bloqueio e execução. • (Caio e Gustavo) **private void checkBlockedQueue()**: Verifica a fila de bloqueados e move processos para a fila de prontos quando seu tempo de bloqueio expira. • (Caio e Gustavo) **private void processBlock(Process process)**: Bloqueia um processo por um período determinado e o adiciona à fila de bloqueados. • (Caio e Gustavo) **void startSimulation()**: Inicia a simulação, permitindo que os processos sejam executados. • (Caio e Gustavo) **void suspendSimulation()**: Suspende a simulação, pausando a execução dos processos. • (Caio e Gustavo) **void resumeSimulation()**: Retoma a simulação após uma suspensão. • (Caio e Gustavo) **void stopSimulation()**: Para a simulação, limpando todas as filas de processos. • (Caio e Gustavo) **void displayProcessQueues()**: Exibe o estado atual das filas de processos e do processo em execução, se houver.

LongTermScheduler «thread»	Gerencia a fila de processos prontos e decide quais processos serão movidos para a memória principal. • (Gustavo) **Process(String pid, int quantum, Vector<String> instructions, NotificationInterface userInterface)**: Construtor da classe que inicializa os atributos do processo, incluindo o identificador, quantum, tempo restante, tempo de bloqueio, instruções e interface de usuário. • (Gustavo) **String getId()**: Retorna o identificador do processo. • (Gustavo) **int getRemainingTime()**: Retorna o tempo restante para a execução do processo. • (Gustavo) **void setRemainingTime(int remainingTime)**: Define o tempo restante para a execução do processo. • (Gustavo) **int getBlockTime()**: Retorna o tempo de bloqueio do processo. • (Gustavo) **void setBlockTime(int blockTime)**: Define o tempo de bloqueio do processo. • (Gustavo) **void decrementBlockTime()**: Decrementa o tempo de bloqueio do processo. • (Gustavo) **String getCommand(String instruction)**: Extrai e retorna o comando da instrução e, se presente, define o período de bloqueio. • (Gustavo) **int getNextBlockPeriod()**: Retorna o próximo período de bloqueio. • (Gustavo) **String getNextInstruction()**: Retorna a próxima instrução a ser executada e incrementa o número da instrução. • (Gustavo) **String toString()**: Retorna uma string com o identificador do processo, tempo restante e tempo de bloqueio. • (Gustavo) **void resetRemainingTime()**: Redefine o tempo restante do processo com base no número de quantums. • (Gustavo) **void upgradeNumberQuantum()**: Aumenta o número de quantums do processo até um máximo de 6. • (Gustavo) **void downgradeNumberQuantum()**: Diminui o número de quantums do processo até um mínimo de 2.

ShortTermScheduler «thread»	Gerencia a execução dos processos na CPU, decidindo a ordem de execução dos mesmos. • (Caio e Gustavo) **ShortTermScheduler(int quantum)**: Construtor da classe que inicializa as filas de processos (pronto, bloqueado e terminado) e define o valor do quantum. • (Gustavo) **void setThreads(NotificationInterface userInterface)**: Método para definir a interface de notificação que será usada para exibir mensagens ao usuário. • (Gustavo) **synchronized void addProcess(Process process)**: Adiciona um processo à fila de prontos. • (Gustavo) **synchronized int getProcessLoad()**: Retorna a quantidade de processos na fila de prontos. • (Caio e Gustavo) **void run()**: Método principal de execução do escalonador. Gera a simulação, executa processos da fila de prontos e gerencia processos bloqueados. • (Caio e Gustavo) **private void executeProcess(Process process)**: Executa o processo atual, decrementando o tempo restante e gerenciando comandos de bloqueio e execução. • (Gustavo) **private void checkBlockedQueue()**: Verifica a fila de bloqueados e move processos para a fila de prontos quando seu tempo de bloqueio expira. • (Gustavo) **private void processBlock(Process process)**: Bloqueia um processo por um período determinado e o adiciona à fila de bloqueados. • (Gustavo) **void startSimulation()**: Inicia a simulação, permitindo que os processos sejam executados. • (Gustavo) **void suspendSimulation()**: Suspende a simulação, pausando a execução dos processos. • (Gustavo) **void resumeSimulation()**: Retoma a simulação após uma suspensão. • (Gustavo) **void stopSimulation()**: Para a simulação, limpando todas as filas de processos. • (Caio e Gustavo) **void displayProcessQueues()**: Exibe o estado atual das filas de processos e do processo em execução, se houver.

NamedValidation	Responsável por validar os programas de entrada definidas na gramática BNF. Validações: Validation Name File, Validation Begin/End Command e Validation Inner Program Structure.
	
	• (Caio) **NamedValidation(String name, Supplier<Boolean> validationFunction)**: Construtor da classe que inicializa os atributos 'name' e 'validationFunction'.
	  – **name**: O nome da validação.
	  – **validationFunction**: Uma função fornecedora que retorna um valor booleano, representando o resultado da validação.
	
	• (Caio) **String getName()**: Retorna o nome da validação.
	
	• (Caio) **Supplier<Boolean> getValidationFunction()**: Retorna a função de validação.

ParserBNF	Implementa o parser da gramática BNF para processar e validar as entradas do sistema. • (Caio) **ParserBNF(NotificationInterface userInterface)**: Construtor da classe que inicializa os atributos da classe, incluindo a interface de usuário e o vetor de instruções. • (Caio) **Vector<String> parse(String fileName) throws FileNotFoundException, ParseException**: Método principal de análise que lê um arquivo, valida seu conteúdo de acordo com as regras da BNF e retorna as instruções. • (Caio) **private void print()**: Imprime as instruções lidas no formato esperado. • (Caio) **private void detectOS(String fileName)**: Detecta o sistema operacional e ajusta o nome real do arquivo conforme necessário. • (Caio e Gustavo) **private boolean validateName(String fileName, String name)**: Valida o nome do programa no arquivo. • (Caio e Gustavo) **private boolean validateBegin(String begin)**: Valida a linha de início do programa. • (Caio e Gustavo) **private boolean validateEnd(String end)**: Valida a linha de término do programa. • (Caio e Gustavo) **private boolean validateProgram()**: Valida o conteúdo das instruções do programa.

Process	Classe que representa um processo no sistema, contendo informações como ID, tempo de execução, prioridade, etc. • (Gustavo) **Process(String pid, int quantum, Vector<String> instructions, NotificationInterface userInterface)**: Construtor da classe que inicializa os atributos do processo, incluindo o identificador, quantum, tempo restante, tempo de bloqueio, instruções e interface de usuário. • (Gustavo) **String getId()**: Retorna o identificador do processo. • **int getRemainingTime()**: Retorna o tempo restante para a execução do processo. • (Gustavo) **void setRemainingTime(int remainingTime)**: Define o tempo restante para a execução do processo. • (Gustavo) **int getBlockTime()**: Retorna o tempo de bloqueio do processo. • (Gustavo) **void setBlockTime(int blockTime)**: Define o tempo de bloqueio do processo. • **void decrementBlockTime()**: Decrementa o tempo de bloqueio do processo. • (Gustavo) **String getCommand(String instruction)**: Extrai e retorna o comando da instrução e, se presente, define o período de bloqueio.

Interfaces

Interface	**Responsabilidade**
SubmissionInterface	Define métodos para submissão de trabalhos e exibição da fila de submissão. Métodos: - `submitJob(String): boolean` - `displaySubmissionQueue(): void`

NotificationInterface	Define método para exibir mensagens ao usuário. Métodos: - `display(String): void`
InterScheduleInterface	Define métodos para adicionar processos e obter a carga de processos. Métodos: - `addProcess(Process): void` - `getProcessLoad(): int`
ControlInterface	Define métodos para controle da simulação, como iniciar, suspender, retomar e parar a simulação, além de exibir as filas de processos. Métodos: - `startSimulation(): void` - `suspendSimulation(): void` - `resumeSimulation(): void` - `stopSimulation(): void` - `displayProcessesQueues(): void`

Componentes Visuais

Classes (UI)	Responsabilidade
UserInterface «thread»	Responsável pela interação com o usuário, permitindo a submissão de trabalhos e exibindo informações ao usuário. • (Caio) **UserInterface()**: Construtor da classe que inicializa os painéis de log e o mapa de cores. • (Caio) **void setThreads(ControlInterface shortTermScheduler, SubmissionInterface longTermScheduler)**: Método para definir as interfaces de controle do escalonador de curto prazo e de submissão do escalonador de longo prazo. • (Caio) **void createAndShowGUI()**: Método que cria e exibe a interface gráfica do usuário (GUI), incluindo os painéis de log, painel de controle e painel de submissão. • (Caio) **void run()**: Método que inicia a GUI utilizando o SwingUtilities para garantir que a criação da GUI ocorra na thread de despacho de eventos do Swing. • (Caio) **private void createColorHashMap()**: Método que inicializa o mapa de cores associando níveis de importância (1, 2, 3) a cores específicas (preto, vermelho, azul). • (Caio) **void display(String info)**: Método que exibe informações nos painéis de log apropriados com base no prefixo da mensagem (`<ss>`, `<ls>`, `<is>`), ajustando a cor conforme o nível de importância.

SubmissionPanel	Painel de submissão utilizado na interface gráfica para permitir que o usuário submeta novos trabalhos ao sistema. • (Caio) **SubmissionPanel(SubmissionInterface actor)**: Construtor da classe que inicializa o painel de submissão com um layout de fluxo, campos de entrada e botões. • (Caio) **JLabel jobLabel**: Rótulo para o campo de entrada do arquivo de trabalho. • (Caio) **JTextField jobField**: Campo de texto para entrada do nome do arquivo de trabalho. • (Caio) **JTextArea textArea**: Área de texto para exibição adicional. • (Caio) **JButton submitButton**: Botão para submeter o trabalho, que aciona o método `submitJob` do objeto `actor` com o nome do arquivo fornecido. • (Caio) **JButton displayButton**: Botão para exibir as filas de submissão de longo prazo, que aciona o método `displaySubmissionQueue` do objeto `actor`.
TextPanel	Painel de texto utilizado para exibir informações e mensagens ao usuário na interface gráfica. • (Caio) **TextPanel(String title)**: Construtor da classe que inicializa o painel de texto com um título, configurações de estilo e comportamento do caret. • (Caio) **void append(String text, Color color)**: Método para anexar texto ao painel com a cor especificada e garantir que o comportamento de rolagem automática funcione corretamente. • (Caio) **private boolean isAtBottom(JScrollBar verticalScrollBar)**: Método auxiliar que verifica se a barra de rolagem está na parte inferior.

ControlPanel	Painel de controle utilizado na interface gráfica para controlar a simulação (iniciar, pausar, retomar, parar). • (Caio) **ControlPanel(ControlInterface actor)**: Construtor da classe que inicializa o painel de controle com botões de start, suspend, resume, stop e display, associando-os às ações correspondentes no objeto `actor`. • (Caio) **startButton.addActionListener(e -> actor.startSimulation())**: Adiciona um ouvinte de ação ao botão start para iniciar a simulação. • (Caio) **suspendButton.addActionListener(e -> actor.suspendSimulation())**: Adiciona um ouvinte de ação ao botão suspend para suspender a simulação. • (Caio) **resumeButton.addActionListener(e -> actor.resumeSimulation())**: Adiciona um ouvinte de ação ao botão resume para retomar a simulação. • (Caio) **stopButton.addActionListener(e -> actor.stopSimulation())**: Adiciona um ouvinte de ação ao botão stop para parar a simulação. • (Caio) **displayButton.addActionListener(e -> actor.displayProcessQueues())**: Adiciona um ouvinte de ação ao botão display para exibir as filas de processos de curto prazo.

E para a estrutura de arquivos do projeto fica:

Estrutura de Arquivos

```
SchedulerSimulator/
│
├── .idea/
├── docs/
├── libs/
├── out/
├── resources/
│   ├── arquivos_para_teste/
│   │   └── (...)
│   ├── generator.python
│   ├── Simple-Math.bnf
│   └── teste.txt
├── scripts/
├── src/
│   ├── Parser/
│   │   ├── NamedValidation.java
│   │   └── ParserBNF.java
│   ├── UserInterface/
│   │   ├── UIComponents/
│   │   │   ├── ControlPanel.java
│   │   │   ├── SubmissionPanel.java
│   │   │   └── TextPanel.java
│   │   ├── ControlInterface.java
│   │   ├── NotificationInterface.java
│   │   ├── SubmissionInterface.java
│   │   └── UserInterface.java
│   ├── InterSchedulerInterface.java
│   ├── LongTermScheduler.java
│   ├── Process.java
│   ├── SchedulerSimulator.java
│   └── ShortTermScheduler.java
├── .gitignore
├── README.md
├── SchedulerSimulator.java
└── SchedulerSimulator.iml
```

- /.idea: Diretório utilizado pela IDE IntelliJ IDEA para armazenar configurações específicas do projeto.
- /docs: Contém a documentação do projeto.
- /libs: Diretório para bibliotecas externas utilizadas pelo projeto

- /out: Diretório para arquivos compilados.
- /resources: Contém recursos adicionais necessários para o projeto, como os exemplos de entrada.
- /scripts: Diretório para scripts auxiliares utilizados no projeto.
- /src: Diretório do código fonte.

2.2 Interface: Java Swing

As interfaces foram desenvolvidas usando o Java Swing.

A interface contém uma classe principal "UserInterface"que é a responsável por encapsular toda a interface. Isso é realizado a partir da criação de um *JFrame* (Janela), que contém um corpo diagramado em *BorderLayout* (Esquema de posicionamento), podendo anexar em cada orientação (Cima, Centro, Baixo) os *JPanels* (Painéis) que irão conter os componentes.

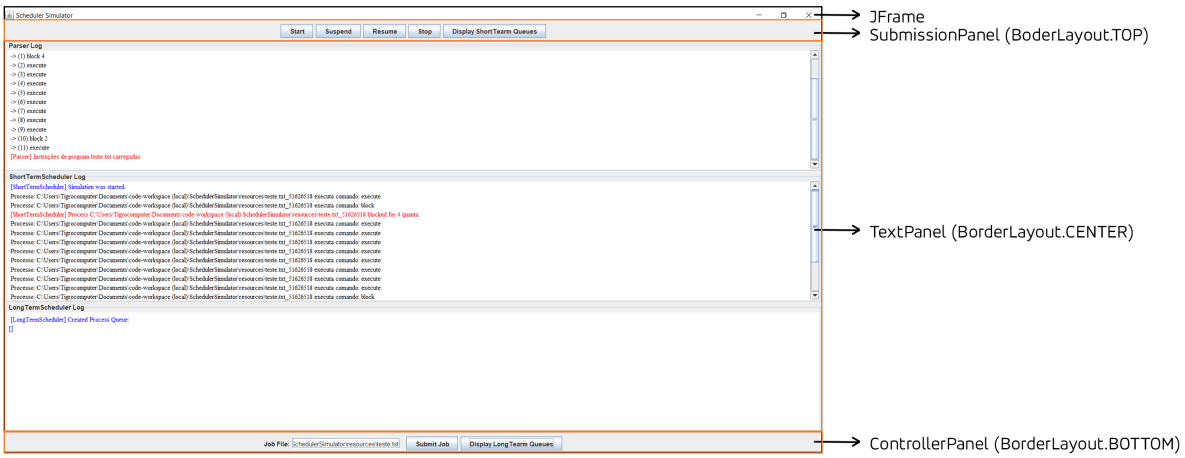

Figura 2.2: UserInterface por Painés

A interface contém três painéis: SubmissionPanel, TextPanel, ControlPanel, como mostra a figura 2.2.

- SubmissionPanel: É o responsável pelos botões da "SubmissionInterface", criando os eventos do escalonador de longo prazo, também como a criação de um "Input Text Field"para o caminho do arquivo de execução.
- ControlPanel: É o responsável pelos botões da interface da "ControlInterface", criando os eventos para o escalonador de curto prazo.

- TextPanel: É aquele que tem os componentes de "Text Log", na qual renderiza os textos que são enviados pela "NotificationInterface".

Note-que, para conseguirmos fazer uma interface com diferentes cores e diferentes painés de display, foi aplicado um parseamento da String enviada pelo *"NotificationInterface.display(String info)"*, na qual via *hashMap* para as cores e *Switch-Case* para os dpainés, se é parseado a *String* da forma:

Parseamento de NotificationInterface.display(String info)

<command> : «painel» «color» <String>

- <painel> : "ls"(Long Tearm) | "ss"(Short Tearm) | "is"(Instructions).
- <cor> : "1"(Color Black) | "2"(Color Red) | "3"(Color Blue)
- <string>: Texto para ser mostrado

2.3 ParserBNF

Essa classe foi construída, pois um requisito do projeto era ler um arquivo texto e interpretar seu conteúdo como um conjunto de instruções.

Dessa forma, uma decisão do time foi introduzir uma classe ParserBNF que é responsável pela tradução desse arquivo texto em instruções que seriam executadas por uma classe Process, conforme as regras propostas pela Definição de Programa da BNF.

A classe ParserBNF possui as seguintes funcionalidades principais:

- Construtor da Classe: O construtor ParserBNF(NotificationInterface userInterface) inicializa a classe com uma interface de notificação para comunicação com o usuário e um vetor de strings para armazenar as instruções.
- Método parse: O método parse(String fileName) é responsável por ler o arquivo de programa especificado pelo nome do arquivo (fileName). Este método detecta o sistema operacional para ajustar o caminho do arquivo conforme necessário. Utiliza um objeto Scanner para ler o arquivo linha por linha e processar o conteúdo conforme definido pelas regras da BNF.

A classe ParserBNF é projetada para garantir que os arquivos de programa que seguem a notação BNF sejam corretamente interpretados e convertidos em um formato que possa ser executado pelo sistema de processamento, garantindo a correta interação com os Schedulers, para essa validação é aplicado a classe NamedValidation, que garante as seguintes validações:

- Name File: Garante que o programa começa definindo seu nome, igualmente está em seu nome de arquivo.

- Program Begin/End: Garante que o programa começa e termina com um begin e um end
- Inner Program Strucutre: Garante que as instruções seguem as especificações da BNF

2.4 Schedulers

A abordagem que utilizamos no projeto para implementação do política de escalonamento é semelhante a um sistema de fila circular, porém, dependendo do número de execuções de operação de block por parte do processo, ele perde prioridade, até que, os processo que tiverem uma prioridade muito reduzida, terão seu tempo de CPU reduzido com base nesse valor. Esse valor está presente na class Process com o nome de number_quantum.

2.4.1 LongTermScheduler

Esta classe foi implementada com base no diagrama definido na descrição do trabalho, implementando a interface SubmissionInterface.

- Construtor da Classe: O construtor da classe LongTerScheduler cria a fila de processos ainda não preemptados, define a carga do escalonador de curto prazo e o quantum.
- setThreads: Pega a referencia das intefaces Notification inteface e InterSchedulerInterface.
- submitJob: Adiciona um novo programa a fila de criados, e garente que estão seguindo os padrões de escrita do programa usando a classe ParserBNF.
- run: Aqui irá checar a carga do ShortTermScheduler, e se ele tiver uma carga abaixo da configurada ele passa o processo para a fila de prontos no ShorTermScheduler.

2.4.2 ShortTermScheduler

Esta classe foi implementada com base no diagrama definido na descrição do trabalho, implementando as interfaces ControlInterface e InterSchedulerInterface.

- Construtor da Classe: O construtor da classe ShortTermScheduler cria a fila de processos prontos, bloqueados, terminados e define o quantum. Além disso seta os valores inicias para algumas variaveis de controle como: suspended, executionProcess e started_simulation.
- setThreads: Pega a referencia da inteface NotificationInterface.
- executeProcess: simula a execução do processo escolhido pela política e define para qual fila o processo terá que ir e sua prioridade com base em sua execução.

- run: Aqui irá controlar a execução com base nas opções selecionadas no menu.
- checkBlockedQueue: Analisa a fila de bloqueados e move os processo que já terminaram sua operação de IO para a fila de prontos.

2.5 Ferramentas auxiliares

- Para versionamento dos arquivos foi utilizado a ferramenta Git e plataforma Github.
- A IDEs de ambos os membros foi a IntelliJ.
- Este documento foi escrito e organizado no Overleaf com template da FAPESP.

2.6 Observações

Como observação final, pode-se observar na estrutura de arquivos uma biblioteca chamada *Bullwinkle*, disponibilizada no repositório do GitHub. Esta foi uma tentativa de utilizar um *ParserBNF* de forma completa, ou seja, um *parser* que lê uma definição BNF e consegue aplicar essa definição em qualquer arquivo de texto, o que seria uma generalização da definição de programa apresentada pelo professor. Como era uma especificação não solicitada, não seguimos com esse módulo, mas colocamos aqui como próximos passos de uma possível extensão desse projeto.

Além disso, foi disponibilizado um arquivo `generator.py` escrito pelo Gustavo Bender, no qual são gerados exemplos aleatórios de arquivos de texto de programa, para o time garantir a execução do simulador com diferentes arquivos. Um conjunto de exemplos gerados com esse *script* também foi disponibilizado.

3 README (Anexo)

```
# Projeto Sistemas Operacionais

Esse projeto trata-se de um **simulador de escalonamento
de processos**, proposto como trabalho final da disciplina **(5954016)
Sistemas Operacionais** do Departamento de Computação e Matemática
da USP-RP, aplicada pelo professor Clever Ricardo Guareis de Farias.

## Instruções básicas

1) Para inicilizar o programa basta abrir um projeto java e executar a classe principal
SchedulerSimulator ou apenas executar o arquivo SchedulerSimulator.jar e a carga suportada pelo ShortTermScheduler.
```bash
java -jar SchedulerSimulator.jar n_process
```
2) Com o programa aberto, você precisa indicar em "Job file" o caminho absoluto
completo do "arquivo programa" definido pela BNF apresentada pelo professor.

```
<program> ::= <program_statement> <program_body>
<program_statement> ::= "program" <whitespace> <file_name> <EndOfLine>
<program_body> ::= <begin_statement> <program_behaviour> <end_statement>
<begin_statement> ::= "begin" <EndOfLine>
<end_statement> ::= "end" <EndOfLine>
<program_behaviour> ::= (<behaviour_statement>)+
<behaviour_statement> ::= "execute" <EndOfLine> | "block" <whitespace> <block_period> <EndOfLine>
<whitespace> ::= " "
<block_period> ::= "1" | "2" | "3" | "4" | "5"
```

3) Após indicar o caminho em "Job file", deve-se usar os botões para manipular
a execução do programa.

Instruções para gerador

Para executar nosso gerador de programas aleatórios,
é necessário ter o Python 3 instalado.
A execução é simples: é necessário passar três parâmetros.
Primeiro, o número de instruções por programa;
segundo, o número de programas gerados;
e, por último, o nome da pasta que irá guardar os arquivos.
```

```
41 A pasta será criada na mesma pasta em que o script for executado.
42
43 ```bash
44     python3 generator.py n_instrucoes n_arquivos nome_pasta
45 ```
46
47 Já adicionamos 15 arquivos testes na pasta `resources/arquivos_para_teste`.
48 Lembrando que o nome do arquivo não deve ser modificado, já que há um teste
       para verificar
49 se o nome real do arquivo é o mesmo especificado no programa.
50
51 ### Autores
52 - Caio Uehara Martins (13672022)
53 - Gustavo Bender (13725695)
```

Referências Bibliográficas

[1] WIKIPEDIA. *Backus-Naur form*. Form (2024, May 22). In Wikipedia.

[2] ORACLE Java Docs, The Swing Tutorial. [S.l.].