
# GAC125 - Automação Avançada

## Relatório de Execução da Atividade Avaliativa 1

Pedro Ernesto Machado de Paula Nascimento - 201810663

**Turma 22A**

**2023**

# Índice

# Proposta

Em resumo a proposta se baseia na construção de um sistema que simula uma compania de transporte. Para tanto, espera-se implementar um sistema multithread capaz de realizar os procedimentos e cálculos paralelamente, mantendo a latência em termos aceitáveis. O funcionamento do sistema depende da interação de várias threads, cíclicas ou não. As principais a serem apresentadas são Company, Car, Driver, AlphaBank e FuelStation. Ainda, as comunicações entre elas devem ser feitas utilizando a arquitetura Client-Server, e fazendo uso de pacotes estruturados JSON, com criptografia.

A organização entre as classes vai ser melhor apresentada na seção referente ao Diagrama UML, contudo cada uma delas apresenta uma demanda como explicado a seguir:

**Company:** Trata-se de uma Empresa de mobilidade que detém um conjunto de rotas<Route> que devem ser executadas. Deve funcionar como um servidor<Server> para os carros<Car> e como cliente<Client> para o servidor da AlphaBank. Onde a comunicação com este último deve ser feita a partir da classe<Thread> BotPayment de pagamentos aos motoristas<Driver> por KM rodado. Para isso, durante a execução das Routes, é necessário observar os sinais gerados pelos Cars, o que será detalhado mais a frente. Portanto, a cada KM rodado de uma Route por um Car é preciso gerar um pagamento no valor de R$3,25 para o Driver.

Além disso, as rotas devem obrigatoriamente estar dispostas em um ArrayList<Route> de rotas a serem executadas. Tão logo uma rota seja selecionada para ser executada, a mesma deve migrar para um ArrayList<Route> de rotas em execução. Ao final da execução, a rota deve ser armazenada em um ArrayList<Route> de rotas executadas. A leitura, interpretação e construção das instâncias de Route devem ser feitas a partir do arquivos XML disponibilizados na pasta \data.

**AlphaBank:** Para que as transações financeiras sejam possíveis, será necessário criar um servidor<Server> do AlphaBank, implementando um mecanismo de troca de mensagens, por exemplo: entre a Company e o Alpha Bank; entre o Alpha Bank e o Driver; entre o Driver e a Fuel Station.

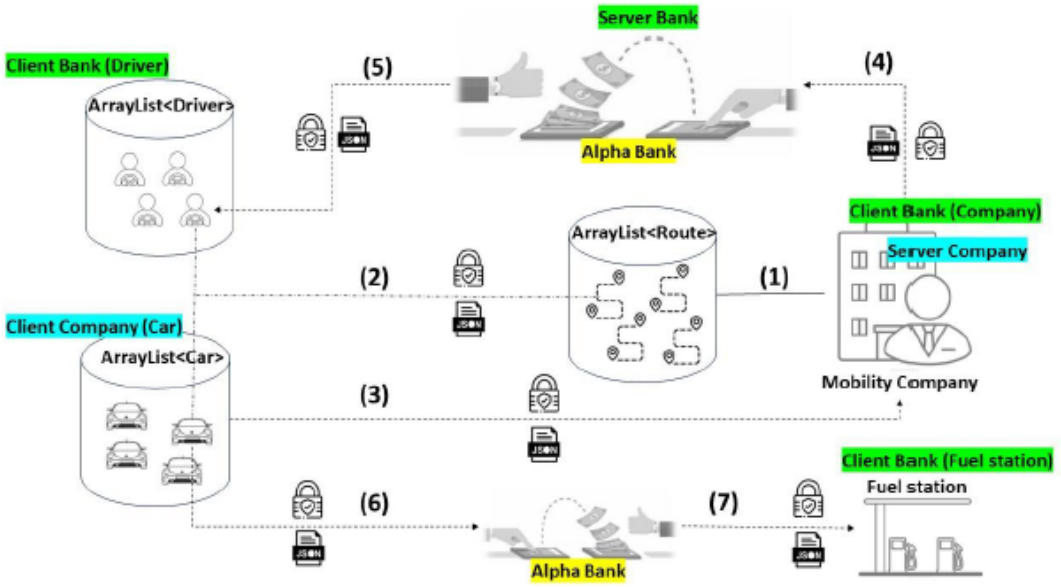
Ainda, é preciso controlar o saldo de cada Account e registrar (extrato) todas as transações realizadas, incluindo o Timestamp em Nanosegundo 8 em que a transação ocorreu. Cada Account deverá ter um login e senha de acesso. Os acessos às Accounts devem ser controlados, especialmente ao atributo saldo, um único acesso por vez é permitido, para que o saldo não sofra acessos simultâneos.

**Driver:** Drivers representam os motoristas que prestarão serviços à empresa<Company> a qual detém o conjunto de rotas<Route> que devem ser executadas. A classe Driver deverá conter uma instância Car como atributo. Nesta implementação a classe Driver é responsável por criar, iniciar e finalizar as Threads TransportService que executam os veículos no SUMO. Enquanto monitorando seu atributo Car, se este levantar a flag indicando que precisa abastecer, o motorista então deve iniciar uma Thread de abastecimento e uma BotPayment<Thread> para pagamento da FuelStation. A BotPayment do motorista é reconhecida pelo AlphaBank que transfere todo o seu saldo para a conta<Account> da FuelStation e a notifica. Quem controla os litros e a execução do abastecimento é o posto de gasolina.

**Car:** é uma thread client do servidor de Company (Servidor Company), responsável por enviar os dados do veículo<Vehicle> para que a empresa possa realizar seu monitoramento e relatório, sendo, portanto, também responsável por calcular a distância percorrida atráves da posição XY em latitude e longitude. A Classe Car tem um atributo privado Fuel Tank que se inicia com um valor igual a 10 litros. A cada iteração do simulador, o valor do atributo fuelTank é decrementado do equivalente em litros consumidos, considerando-se para isso a informação obtida no SUMO (FuelConsumption).
Quando o valor do atribuito fuelTank estiver com 3 litros restantes, uma flag é lançada e o Driver deve providenciar o abastecimento, chamando um serviço específico na Fuel Station. O valor do abastecimento dependerá do saldo disponível na Account do Driver. Quem deve realizar o abastecimento é a Fuel Station, isto é, apenas ela pode acrescentar o equivalente em livros ao atributo fuelTank do Car. O processo de abastecimento leva 2 minutos. Durante esse intervalo de espera, o Car fica parado, isto é, com velocidade igual a zero.

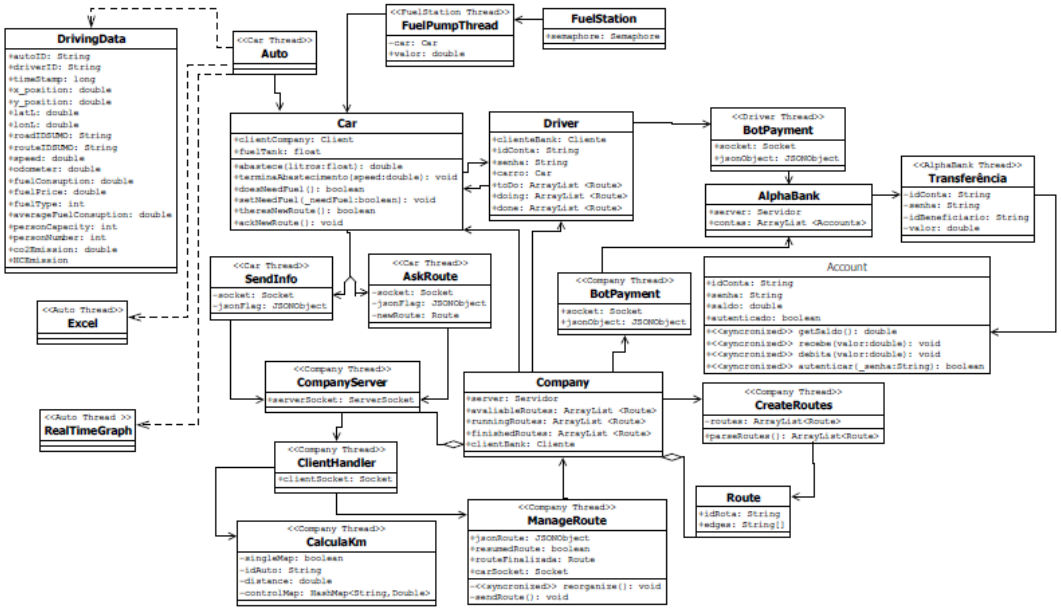
**FuelStation:** posto de gasolina pelo qual Cars podem ser abastecidos. Implementada como um Service<Thread>, gerencia um semáforo que controla o acesso aos recursos, que são duas bombas representadas por sub-threads independentes. A classe ainda possui um Client para o Server de AlphaBank, por onde recebe as confirmações de pagamento de cada Driver, permitindo o cálculo do valor em litros a ser abastecido. Quem executa o abastecimento é de fato a Fuel Station, isto é, apenas ela acessa os métodos que acrescem o atribuito fuelTank do Car. O processo de abastecimento deve levar 2 minutos. Durante esse intervalo de espera, o Car permanecerá parado, e deverá retornar à velocidade anterior quando o abastecimento terminar.

Explicadas as principais demandas de cada classe base, uma representação simplificada do funcionamento é apresentada abaixo.



# Diagrama UML

A fim de melhor visualizar o relacionamento entre as classes e suas sub-threads, foi construído o diagrama a ser explicado abaixo. Vale ressaltar que um diagrama de classes UML é uma ótima ferramenta ao se desenvolver aplicações com princípios de orientação a objetos, no entanto, não se alcança o mesmo efeito quando se refere a um sistema Multithread, pois este envolve conceitos como escalonamento e dependências de threads. Dessa forma, o diagrama mostrado abaixo não se caracteriza fiel às normativas de um diagrama UML.




*Algumas classes de execução como App.java e Simulation.java são omitidas
**Classes que implementam serviço client-server também foram omitidas para melhor visualização

A primeira thread a ser executada após a criação da conexão sumo é a thread **Company**, que por sua vez constrói as classes Route, Car e Driver que irão compor a simulação. Portanto, sua execução segue a linha:

1. **CreateRoutes**: Thread(tarefa) lançada para a interpretação dos arquivos XML e instância dos objetos Route que serão armazenados e gerenciados pela Company. Após a obtenção dos objetos Route, eles são alocados ao ArrayList<Route> avaliableRoutes e o arquivo XML esvaziado.
2. **CompanyServer:** Thread(cíclica) que herda de Service e implementa um servidor que permanece buscando novas conexões e as transfere para novas sub-threads independentes quando acontecem.
3. **Instancia Car e Driver**: Acontece logo no construtor da Company, onde são instanciados os objetos-threads que implementam as classes Car e Driver. A fim de controle e gestão, a company mantém uma cópia referente à lista de cada carro e motorista cadastrado em seu domínio.
4. **Start**: Por fim, antes de encerrar seu método run(), a Company coloca seu servidor e todos os seus motoristas e carros na fila de pronto para que iniciem a sua execução.

A thread Company então finaliza a sua execução e o sistema se mantém organicamente a partir do relacionamento das outras Threads.

**CompanyServer**: Thread que espera por novas conexões e as atribui a novas sub-threads ClientHandler independentes quando acontecem. As derivações a partir dos ClientHandler são tais:

- **ManageRoute**: quando a conexão envia pacote solicitando rota
    - Tenta obter a rota finalizada pelo veículo, se não houver significa que é a primeira solicitação de rota.
    - Reorganiza os ArrayLists que contemplam o banco de rotas da Company, atualiza as rotas executadas, coloca nova rota em execução e envia uma cópia ao Car.

- **CalculaKm**: quando a conexão envia um relatório com os dados atualizados do Auto
    - Obtém a última distância percorrida, calculada a partir da posição geográfica, e a armazena num HashMap com chave igual ao ID do carro.
    - Verifica se a distância completa um quilômetro. Se sim, emite uma nova thread de pagamento ao driver com aquele ID e zera a distância percorrida. Se não, apenas guarda o novo valor acrescido.
        - **BotPayment**: thread que envia solicitando ao Alpha Bank uma transferência de R$3,25 ao driver em questão.

**Car**: sua árvore de execução se divide em duas principais vertentes, AskRoute e SendInfo, a execução da classe segue o seguinte fluxo:

1. **Instancia Auto (construtor)**: A classe Car como previamente citado, serve de meio para gerenciar um objeto da classe Auto, sendo este a nossa representação direta do veículo inserido na simulação.
2. **AskRoute**: Thread responsável por solicitar nova rota<Route> à Company. Ao iniciar a simulação, a rota atual aponta pra null e portanto é necessário solicitar uma rota à Company. Recebido o JSON que contém o objeto Route enviado pela Company, esta thread o aloca no atributo currentRoute do objeto Car e levanta uma flag dizendo que uma nova rota foi definida.
3. **auto.esperaSensores()**: São chamados métodos criados para sincronização da atualização das informações. O objeto Auto atualiza seus atributos e repport a uma taxa fixa de aquisição, dessa forma a Car deve esperar até que os sensores sejam atualizados para que ela possa se atualizar deduzindo o combustível gasto e verificando a posição do carro na rota.
4. **SendInfo**: Thread que é executada toda vez que Car atualiza, enviando os dados para o servidor da Company, que atribuirá o tratamento dos dados a um novo procedimento.

**Driver:** funciona em um loop de verificações:

1. Se a flag de nota rota do carro está levantada. Se não, continua as verificações. Se sim, ele atualiza a sua rota atual e confirma a nota rota, derrubando a flag. Em

seguida encerra a TransportService atual, se houver, e inicializa uma nova thread TransportService com a rota atualizada.
2. Verifica se o carro precisa abastecer ou iniciar nova rota. Se não, ele entra em um laço onde deve permanecer atualizando até que uma dessas condições aconteça. Se sim, ele segue as verificações.
3. Se o carro então quebra o laço, verifica-se se ele precisa de combustível e se for o caso são geradas a thread de abastecimento da FuelStation e o BotPayment para o pagamento do combustível. Além disso é levantada a flag que indica que o carro está esperando a FuelStation concluir o abastecimento.

**FuelStation:** detém um semáforo que gerencia o acesso aos recursos (FuelPumpThread). Salva os valores enviados pelos motoristas para calcular a quantidade de combustível a ser adicionada.

1. A classe FuelStation implementa o padrão de projeto **SingleTon** para garantir que apenas uma instância desta classe existirá;
2. É instanciado um objeto **Semaphore com permissão igual a 2** para que apenas dois carros possam abastecer ao mesmo tempo.
3. A FuelStation fica escutando em loop para receber as notificações de pagamento enviadas pelo AlphaBank. Recebidas, os valores são armazenados no controle de fluxo **HashMap<String, Double> flowControl.**
4. Quando é criada uma nova Thread FuelPumpThread, esta espera até que possa ser executada. Ao ser executada, deverá obter o valor pago pelo driver a partir do HashMap flowControl, que representa o fluxo de caixa do posto.
5. Ao final da thread, são chamados os **métodos da classe Car** para que este veículo retorne à velocidade anterior e finalize seu abastecimento.

**AlphaBank**: implementa Service como um gerenciador de conexões Socket. Controla uma classe privada Account e gerencia as suas instâncias.

1. Estende e constrói um Service: a classe Service foi criada para se implementar um gerenciador de conexões independentes, dessa forma ao ser executado, este espera por novas conexões Socket e as atribui a novas threads “handlers” quando acontecem. No contexto do AlphaBank, nossas handlers são threads nomeadas **Transferência.**
2. Cria-se um HashMap para armazenar os objetos Account referentes a cada usuário do banco. O uso de HashMap é pautado na **eficiência de busca** e na existência de identificadores facilitados no contexto do trabalho.
3. A cada conexão realizada, uma mensagem é lida e então tratada no método ProcessMessage, onde será lançada uma nova Thread Transferência, que antes de seguir **verifica o login** do usuário e **autentica seu acesso.**
4. Será verificada em seguida a existência de um valor específico a ser transferido. Caso não haja, isso indica que a solicitação de pagamento foi gerada por de Driver para a FuelStation e então **todo o saldo** daquele Driver é transferido à conta da FuelStation e uma notificação é enviada.

# Build Repport

Realizar este trabalho foi uma experiência verdadeiramente enriquecedora. Durante todo o processo, foi possível mergulhar no mundo das threads e demais facetas da tecnologia atual. Esta tarefa não apenas desafiou minha compreensão existente, mas também me proporcionou uma grande oportunidade de aprendizado. Abaixo é apresentado um relatório resumido com os problemas encontrados e aprendizados absorvidos.

Primeiramente a ambientação, configuração e utilização do gestor **Maven** se mostraram valiosas no que se trata aos processos de concepção de aplicações, e portanto devem ser destacadas.

Ao explorar o **uso de threads**, fui capaz de compreender a importância de dividir tarefas complexas em unidades menores e executá-las simultaneamente, otimizando o desempenho e também permitindo que sistemas lidem de maneira eficiente com tarefas simultâneas, aumentando a eficiência e responsividade. A coordenação e sincronização adequada entre threads se mostraram aspectos cruciais para evitar condições de falha e garantir a integridade dos dados. Destaca-se, que a maior dificuldade encontrada neste tópico se deu na concepção de uma modelagem ao problema, que se mostrou ineficiente em métodos intuitivos e, portanto, faz-se importante o estudo de técnicas de modelagem concorrente para a construção de um código profissional.

A **manipulação de JSON** desempenhou um papel crucial, fornecendo uma maneira eficaz de armazenar e transmitir dados de forma estruturada. Extrair informações precisas e relevantes de conjuntos de dados complexos tornou-se uma habilidade valiosa que pude desenvolver. Isso não apenas otimizou o processo de análise, mas também abriu portas para uma compreensão mais profunda da integração de sistemas. Em sua totalidade, poucos problemas foram encontrados neste tópico, sendo eles restritos a conflitos entre diferentes bibliotecas JSON que são encontradas na internet.

A **criptografia**, por sua vez, entende-se por conferir uma camada adicional de segurança ao projeto. O estudo das técnicas de criptografia permitiu visualizar a implementação da proteção de dados sensíveis e garantia que a informação confidencial fosse acessível somente para as partes autorizadas. Esta prática de segurança se faz essencial na criação de um ambiente robusto e confiável. No entanto, as criptografias mais simples já alcançam certo nível de complexidade e se mostraram sensíveis ao uso em comunição cliente-servidor, dessa forma corroborando com grande parte dos problemas encontrados na execução.

A **geração de gráficos e planilhas em tempo real** foi um ponto culminante do projeto. A capacidade de visualizar dados em tempo real trouxe uma nova dimensão ao entendimento dos padrões e tendências. A criação de gráficos e tabelas dinâmicas não apenas facilita a interpretação dos dados, mas também proporciona uma representação visual impactante. A implementação não se mostrou complexa, sendo baseadas no uso de bibliotecas e APIs disponíveis, no entanto as bibliotecas eleitas apresentaram muitos conflitos com dependências pré-existentes no arquivo pom.xml.

Concluo que, ao lidar com multithreads e demais temas, deparei-me com desafios e nuances que me incentivaram a aprimorar minhas habilidades de resolução de problemas. Esta experiência não apenas expandiu meu repertório técnico, mas também me proporcionou uma apreciação mais profunda pela complexidade e a beleza da automação moderna

# Referências


1. Urban Traffic Simulation with SUMO - A Roadmap for the Beginners - https://cst.fee.unicamp.br/sites/default/files/sumo/sumo-roadmap.pdf. Acesso em 18/09/2023

2. Java Sockets: Criando comunicações em Java - https://www.devmedia.com.br/java-sockets-criando-comunicacoes-em-java/9465. Acesso em 22/09/2023

3. Utilizando Criptografia Simétrica em Java - https://www.devmedia.com.br/utilizando-criptografia-simetrica-em-java/31170 Acesso em 30/9/2023

4. Trabalhando com JSON em Java: o pacote org.json - https://www.devmedia.com.br/trabalhando-com-json-em-java-o-pacote-org-json/25480. Acesso em 30/9/2023

5. Como criar um Chat Multithread com Socket em Java - https://www.devmedia.com.br/como-criar-um-chat-multithread-com-socket-em-java/33639. Acesso em 09/10/2023

6. Apache POI: Manipulando Documentos em Java - https://www.devmedia.com.br/apache-poi-manipulando-documentos-em-java/31778 Acesso em 12/09/2023