NOME: João Paul Fernandes Carvalho

N.º DE ESTUDANTE: 2304353

CURSO: Licenciatura de Engenharia Informática

Relatório:

Comecei o **e-Fólio B** pegando no código do **e-Fólio A** e incorporando todas as sugestões do professor: reorganizei toda a parte Prolog para que, ao adicionar, editar ou remover factos, nunca mais fosse apagado inadvertidamente o resto do banco de dados; e, ao mesmo tempo, isolei cada grupo de funcionalidades numa camada própria, facilitando a manutenção e a extensibilidade.

No coração da aplicação Java está a função **main**, que recebe os argumentos da linha de comando e imediatamente despacha o pedido para o método mais indicado, através de um **switch(cmd)**. Essa estrutura mantém a **main** limpa e focada apenas em roteamento: primeiro vêm os cinco comandos OCaml, depois os comandos Prolog agrupados por domínio (orçamentos, inventário, categorias/descontos e, por fim, clientes). Essa escolha modular esclarece de relance quais são as responsabilidades de cada parte do código e simplifica o fluxo de execução.

Para os comandos OCaml, criei o método **invokeOcaml(cmd, ids)**. Nele, a função **buildCommand(cmd, ids)** gera dinamicamente a lista de strings necessárias para o **ProcessBuilder** — em geral ["dune", "exec", "mainA", "--", cmd] mais, quando aplicável, o argumento ids. Em seguida, um **switch(cmd)** escolhe um “mapper” funcional: **Part::fromLine**, **LaborCost::fromLine**, **PartDiscount::fromLine** ou a função utilitária **fixedPriceFromLine**, que converte uma linha do tipo "ID;valor" num array **double[]**. Por fim, invocamos ou **runAndPrint**, que simplesmente imprime cada linha da saída, ou **runAndParse**, que aplica o mapper a cada linha e retorna uma lista de objetos Java. Essa separação evita duplicar lógica de criação de processos ou parsing em cada comando OCaml.

Todas as chamadas Prolog foram organizadas em dois padrões distintos. Para inventário (Questão 2) e orçamentos/clientes (Questão 1) — onde já havia handlers — mantive dois métodos dedicados: **handleInventory(cmd, idArg)** e **handleLogic(cmd)**. Cada um inicializa a máquina Prolog com **JPL.init()**, faz **consult** do ficheiro de regras correspondente (inventory.pl ou Logic.pl) e do **database.pl**, e em seguida executa a query certa. Essa abordagem resolve um bug original em que Prolog não consultava o **database.pl** na ordem correta quando se invocava OCaml logo antes.

Para categorias/descontos (Questão 3) e gestão de clientes (Questão 4), introduzi o helper genérico **withProlog(sourceFile, action)**. Ele faz **consult** primeiro do ficheiro de regras específico (categ_disc.pl ou cliente.pl), depois do **database.pl**, e finalmente executa a lambda **action.get()**, normalmente contendo um **new Query(...).hasSolution()** e uma mensagem de confirmação. Esse helper uniformiza o carregamento de factos e elimina repetições de código, garantindo sempre que o Prolog carregue primeiro as definições dos predicados antes de ler os dados.

No lado Prolog, cada módulo possui um predicado de persistência cuidadosamente desenhado para preservar apenas o seu próprio conjunto de factos. Em **inventory.pl**, o predicado **read_all_facts/3** percorre **database.pl** inteiro, filtra todos os factos que não sejam **item/7** e retorna-os numa lista. Depois, **persist_db** reabre o ficheiro em modo escrita: primeiro reescreve esses factos “outsider” intactos, depois grava todos os factos atuais **item/7**. Isso assegura que qualquer operação sobre inventário não elimine factos de orçamentos, categorias, descontos ou clientes. A mesma lógica se aplica em **categ_disc.pl** (filtrando **servico/6**, **item/7**, **desconto_marca/2** e **desconto_mao_obra/2**) e em **cliente.pl** (filtrando apenas **cliente/3**).

Em cada módulo Prolog, também incluí verificações de existência e mensagens de erro claras antes de inserir ou remover factos. Por exemplo, em **adicionar_item/7** verifico se já existe um **item(ID,…)** e, em caso afirmativo, escrevo no **user_error** e retorno **fail**. Nos predicados de edição e remoção, uso **retract** ou **retractall** seguido de **assertz** para atualizar

apenas o campo desejado (nome, marca, tipo, custo, preço ou quantidade, no caso do inventário; nome ou distrito, no caso de clientes).

Dois desafios técnicos merecem destaque. Primeiro, garantir a ordem correta de **consult** em Prolog: sem os helpers e sem a sequência regras-de-domínio → dados, várias queries falhavam por não encontrarem factos ainda não carregados. Segundo, compatibilizar o OCaml e o Java/JPL7 no mesmo projeto: houve conflitos de ambiente e de classpath, resolvidos apenas ao especificar manualmente **-cp "/usr/lib/swi-prolog/lib/jpl.jar"** tanto no **javac** quanto no **java**, já que nem o IDE nem o classpath automático reconheciam a dependência JPL7.

Finalmente, para evitar números com dígitos de ponto flutuante de mais (por exemplo **0.20000000298023224**), faço a formatação em Prolog sempre que reescrevo descontos, usando **format(..., "~2f", ...)** para fixar duas casas decimais. Em Java, limito-me a **Double.parseDouble** sem reformatar de volta no ficheiro, delegando a apresentação numérica ao Prolog.

Em resumo, a aplicação resultante é altamente modular: o Java orquestra OCaml e Prolog através de três tipos de métodos (**invokeOcaml**, handlers especializados e **withProlog**), e cada módulo Prolog persiste seletivamente apenas os seus factos, conservando a integridade de todo o **database.pl**. Essa arquitetura DRY (Don’t Repeat Yourself) torna fácil adicionar novas funcionalidades — basta criar mais **cases** em Java e novos predicados Prolog com o mesmo padrão de persistência — e garante que inventário, orçamentos, categorias/descontos e clientes coexistam sem interferências.