Programação Orientada aos Objetos

Trabalho Prático

Grupo TP32

12 de Maio 2024

Diogo Costa

A100751

Humberto Gomes

A104348

José Lopes

A104541

Resumo

No âmbito da Unidade Curricular de Programação Orientada aos Objetos, o nosso grupo desenvolveu uma aplicação de gestão de utilizadores, atividades e planos de treino. A aplicação permite a gestão destas entidades: criação tanto de novos utilizadores como de novas atividades, e associação de atividades a utilizadores como execuções individuais ou como parte de um plano de treino. Ademais, a aplicação permite a simulação da passagem do tempo para identificação das atividades como concluídas ou não, e permite a execução de interrogações sobre recordes (por exemplo, do maior número de calorias dispendido por um utilizador). Por último, foi implementado o conceito de atividade *hard*, que de momento apenas é usado para caracterização de atividades na interface de linha de comandos desenvolvida, mas que pode vir a ser utilizado na expansão do programa.

1 Arquitetura de classes

Nesta secção, é descrita e justificada a arquitetura de classes da aplicação desenvolvida. Um diagrama de classes pode encontrar-se no final deste documento.

1.1 Entidades da aplicação

Em primeiro lugar, abordar-se-ão as hierarquias de classes utilizadas para a representação das entidades da aplicação. Todos os utilizadores são compostos pelos mesmos campos, mas podem apresentar diferentes multiplicadores de calorias conforme o seu tipo (principiante, intermédio ou avançado). Logo, a decisão arquitetural lógica foi o uso de uma classe abstrata para a representação de um utilizador qualquer, `User`, que contém todas as variáveis de instância de um utilizador. Não é permitida a instanciação de utilizadores sem conhecimento da sua experiência, pelo que a cada nível de experiência está associada uma classe concreta (`BeginnerUser`, `IntermediateUser` ou `AdvancedUser`), que é subclasse de `User` e que implementa o cálculo do multiplicador de calorias.

Para a implementação das atividades, foi utilizado um padrão arquitetural semelhante ao dos utilizadores. A classe abstrata `Activity`, no topo da hierarquia, é responsável pelas variáveis de instância comuns a todas as atividades (por exemplo, data de execução e duração). Na base da hierarquia, estão as implementações de atividades em concreto, como levamento de pesos (`ActivityWeightLifting`). No entanto, ao contrário do que ocorre com os utilizadores, há certas variáveis de instância comuns a apenas algumas atividades (por exemplo, um número de repetições). Logo, cada tipo de atividade que envolva o conhecimento

de mais dados dá origem a uma classe abstrata que herda de `Activity`, como ocorre com `ActivityRepetition`. Esta classe poderá ter outras subclasses, algumas delas implementações de atividades (como `ActivityPushUp`) e outras também tipos de atividades. Estes subtipos, como `ActivityRepetitionWeighted`, envolvem as mesmas variáveis de instância que o tipo associado à sua classe-pai, mas também outros dados que estarão presentes em todas as atividades do subtipo, as suas subclasses. As atividades *hard* são implementadas com recurso a uma interface vazia, `ActivityHard`. Verificar se uma instância de atividade é difícil (por exemplo, para a construção de um plano de treino) é tão simples como utilizar o operador `instanceof`.

Para compreender como as atividades são associadas a utilizadores, é necessário ter em conta a interpretação do enunciado feita pelo nosso grupo. Considerou-se que uma instância de atividade é executada a uma dada hora por um único utilizador, e que atividades não são identificáveis sem estarem associadas ao seu utilizador. Também os planos de treino foram considerados entidades fracas, associados a apenas um utilizador. Logo, utilizou-se uma estratégia de composição, onde cada utilizador é dono do seu plano de treino e das suas atividades.

Um plano de treino, implementado em `TrainingPlan`, é uma coleção de atividades, que podem ser repetidas várias vezes consecutivamente. Também é constituído pelo conjunto de dias da semana em que é executado. A associação entre atividades e o seu número de execuções é feita através de um mapeamento onde a chave é uma atividade, algo que é possível [1] mas que dificulta a leitura do diagrama de classes UML. Planos de treino garantem a não-sobreposição de atividades e expõem métodos úteis para verificar o mesmo para atividades externas. Ademais, implementam a construção do conjunto de atividades entre duas datas resultantes da sua execução.

A coleção de atividades de um utilizador, `UserActivities`, foi separada da classe `User` que a contém, de modo a respeitar o princípio da responsabilidade única: o utilizador não deve conter a lógica de gestão da sua coleção de atividades. Esta coleção contém um plano de treino e dois conjuntos de atividades, as concluídas e não concluídas. Instâncias desta classe são responsáveis pela movimentação de atividades por concluir para o conjunto de atividades concluídas, quando se lhes passa uma mensagem de avanço de tempo.

Por fim, a classe `FitnessModel` contém todos os dados da aplicação, um mapa entre códigos identificadores de utilizadores e os utilizadores em si, que por sua vez contêm, através de `TrainingPlan` e `UserActivities`, as atividades existentes. Todas as classes mencionadas até este momento implementam a interface `java.io.Serializable`, permitindo o armazenamento persistente do estado da aplicação através da serialização da instância de `FitnessModel`.

[1] `Activity` implementa `hashCode` e `equals`, e `TrainingPlan` garante que atividades não são modificadas sem reinserção no `Map`.

1.2 Interrogações

As interrogações sobre recordes de atividades são todas implementações da interface funcional `java.util.function.Consumer<User>`. `FitnessModel`, durante a execução de uma destas *queries*, itera pelos utilizadores da aplicação, que são consumidos pela instância da interrogação. No final, poder-se-ão enviar mensagens a esta para se conhecer o resultado calculado. Como algumas destas interrogações envolvem restrições entre duas datas, criou-se a classe abstracta `QueryBetweenDates`, que gere este intervalo de datas e fornece um método às suas subclasses, implementações de *queries*, para a determinação de se uma dada atividade cumpre os requisitos do filtro de datas existente. A única interrogação que não se encaixa na descrição providenciada é `QueryDistance`, também um `Consumer<User>` mas que tem em conta um utilizador apenas. Segue-se a associação entre as classes das interrogações e as suas funcionalidades:

- `QueryDistance` – Cálculo da distância percorrida por um utilizador;
- `QueryHardestTrainingPlan` – Determinação do plano de treino que exige o dispêndio de mais calorias (por semana);
- `QueryMostActivities` – Determinação do utilizador que realizou o maior número de atividades;
- `QueryMostCalories` – Determinação do utilizador que dispendeu o maior número de calorias;
- `QueryMostCommonActivity` – Determinação da classe de atividade mais executada.

1.3 Exceções

Foram definidos vários tipos de exceção, para se poder gerir o fluxo de controlo no caso de ocorrerem erros. Por exemplo, quando valores inválidos são providenciados a construtores e a *setters* de utilizadores, estes podem levantar uma `UserException`. O mesmo se tem para atividades e `ActivityException`. As exceções `ActivityDoesntExistException` e `ActivityOverlapException` têm nomes auto-descritivos e são levantadas na inserção e remoção de atividades, respetivamente, ambas em planos de treino e coleções de atividades de um utilizador. Por último, `FitnessModelException` é levantada para os diversos erros de interação com `FitnessModel`, e `FitnessControllerException` é uma exceção utilizada para a vista da interface com o utilizador não aceder aos tipos de exceção do modelo, sendo todas as exceções vindas do modelo recriadas pelo controlador (a arquitetura *Model-View-Controller* será descrita de seguida). Optou-se por não se criar uma classe base para exceções da aplicação *Fitness*, para desencorajar os desenvolvedores a não distinguirem entre os diferentes tipos de erro.

1.4 Interação com o utilizador

A interação com o utilizador foi implementada de acordo com o padrão de conceção MVC *Model-View-Controller*. `FitnessModel`, classe já apresentada, contém os dados da aplicação, enquanto que `FitnessController` constitui um adaptador entre `FitnessModel` e `FitnessView`, a interface com o utilizador, desacoplando assim o modelo da interface com o utilizador. Algumas classes auxiliares foram criadas para suportar a vista: `Menu` e `MenuEntry`, um menu formado por entradas compostas por uma `String` e um *handler* que é chamado quando essa opção é escolhida, e `UserInput`, um *wrapper* de um `java.util.Scanner` que permite uma leitura da entrada do utilizador resiliente a erros.

Foram utilizadas capacidades de reflexão da linguagem Java para simplificar a implementação do controlador e da vista, o que também facilita a adição de novos utilizadores, atividades e interrogações. Por exemplo, o controlador expõe à vista o nome das classes dos utilizadores disponíveis. Após ler todos os campos necessários, a vista passará, juntamente com o nome da classe escolhida, a informação lida ao controlador, que criará o utilizador do tipo correto e o adicionará ao modelo. A implementação da entrada de atividades é um pouco mais complexa, pois exige dar a conhecer à vista que outros campos têm de ser pedidos ao utilizador. Mesmo assim, julgamos que esta solução é menos extensa do que a enumeração de tipos de atividade na vista. Com o uso de reflexão, a expansão do programa torna-se muito simples: para a criação de um novo tipo de utilizador ou de uma nova atividade de um tipo existente, além da criação da nova classe, é apenas necessária a inserção de uma linha de código em `FitnessController`, a adição da classe criada à lista de classes conhecidas.

2 Descrição da aplicação

Após transferir o repositório `git`, a aplicação pode ser executada com o comando `gradle run`. Em alternativa, pode correr-se a aplicação externamente após a sua compilação, evitando a elevada latência adicionada pelo redirecionador de I/O do `gradle`:

```
gradle build && unzip build/distributions/poo.zip && ./poo/bin/poo
```

Após correr o comando anterior, o utilizador deparar-se-á com o seguinte menu:

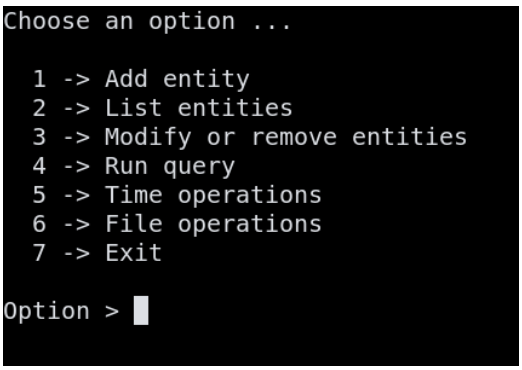


Figura 1: Menu principal da aplicação.

Para adicionar uma nova entidade, a opção 1 é escolhida, e o utilizador pode escolher entre adicionar um novo utilizador ou uma nova atividade, como se vê na figura abaixo. Enquanto não forem adicionados utilizadores, não é possível adicionar atividades, como informaria uma mensagem de erro caso essa opção fosse escolhida. Segue-se um exemplo do que ocorre quando se opta por inserir um utilizador.

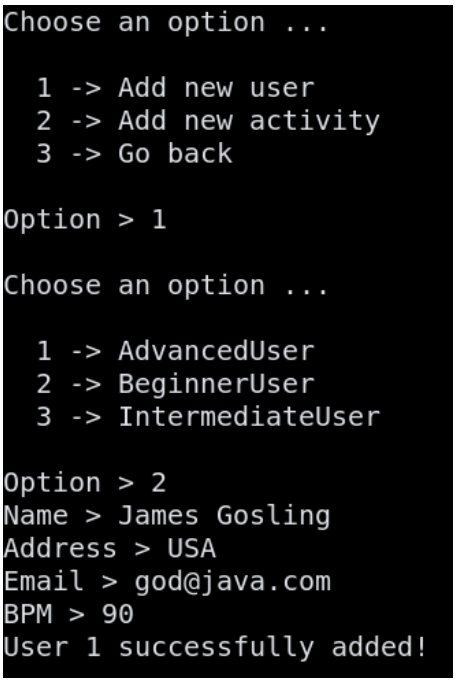


Figura 2: Inserção de um novo utilizador.

Como se observa, primeiro é pedido o tipo do utilizador e depois o valor dos seus campos. Segue-se uma mensagem com o identificador do novo utilizador criado.

Podem agora inserir-se novas atividades. Estas podem ser eventos isolados ou parte do plano de treino do utilizador providenciado, informação que a aplicação pede, como se observa na figura abaixo. Depois, serão pedidos os valores dos atributos necessários conforme o tipo de

atividade escolhido.

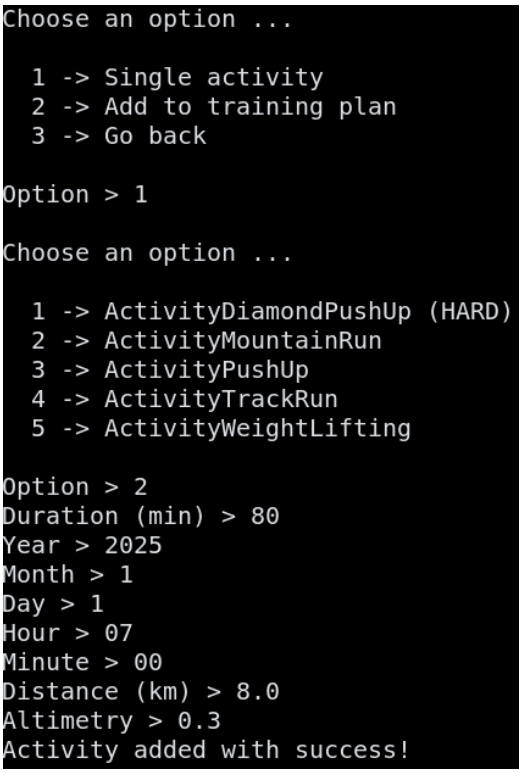


Figura 3: Inserção de uma nova atividade.

Os vários utilizadores e as várias atividades podem ser consultados escolhendo a opção 2 no menu principal. Note-se que as atividades têm de ser filtradas conforme o seu estado de conclusão ou a sua pertença a um plano de treino.

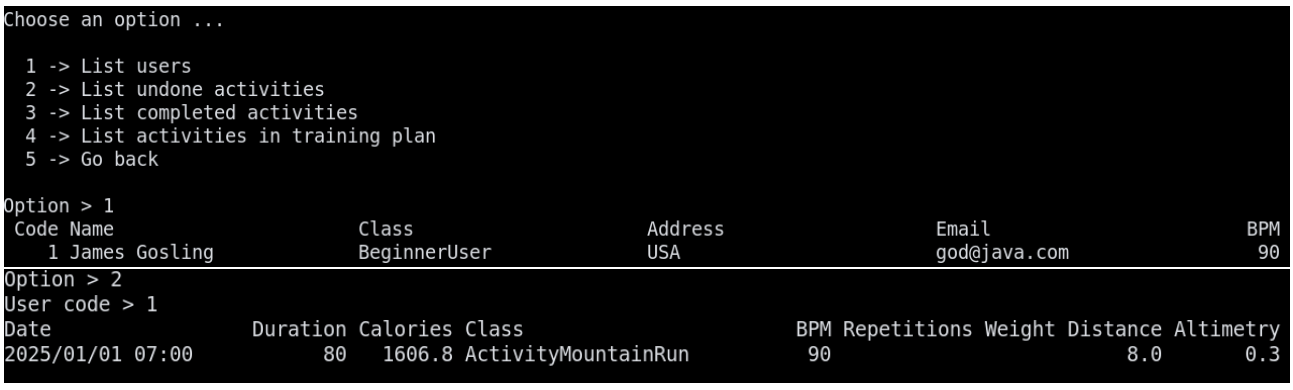


Figura 4: Enumeração de utilizadores e de atividades.

A opção 3 do menu principal permite a remoção de utilizadores e a modificação dos dias em que planos de treino são executados. Exemplifica-se esta segunda operação na figura abaixo,

onde a aplicação pede o código do utilizador e os dias de execução um a um:

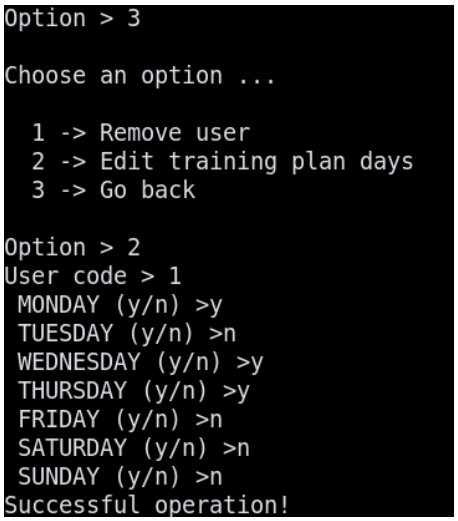


Figura 5: Edição dos dias de execução de um plano de treino.

A opção 4 do menu principal permite a execução de uma *query*. É pedida ao utilizador a interrogação que deve ser executada, seguida dos argumentos necessários para a sua parametrização. Segue-se o exemplo da consulta da atividade mais comum, que não requer qualquer argumento. Note-se que nenhuma atividade será encontrada, pois esta interrogação apenas contabiliza atividades já executadas.

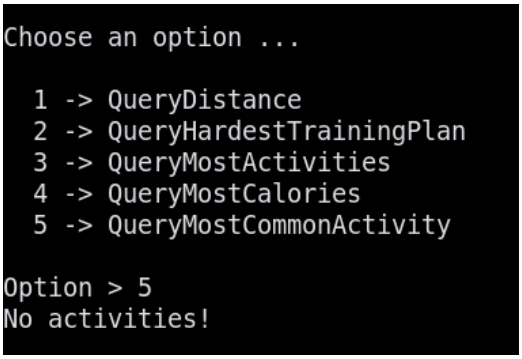


Figura 6: Execução de uma interrogação.

Se seguida, a opção 5 do menu principal permite consultar o tempo atual na aplicação e avançar o tempo, operação que é vista na figura abaixo.

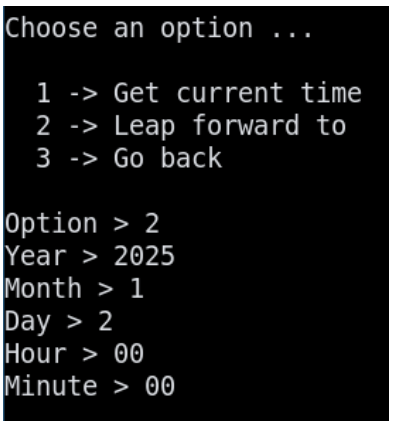


Figura 7: Avanço do tempo para uma data posterior.

Agora, caso o utilizador repita a *query* executada anteriormente, poderá verificar que, como a única atividade criada já foi executada, a atividade mais frequente será, com uma única execução, `ActivityMountainRun`.

Por último, podem realizar-se operações como leituras e escritas para ficheiro. Após se escolher um caminho de ficheiro, o programa a serializa o estado da aplicação e a armazena-o com sucesso. Este ficheiro pode ser depois lido e o estado da aplicação restaurado.

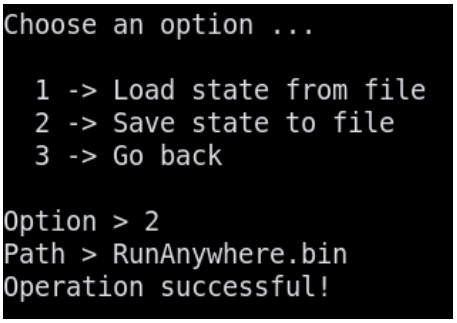


Figura 8: Serialização do estado da aplicação para ficheiro.

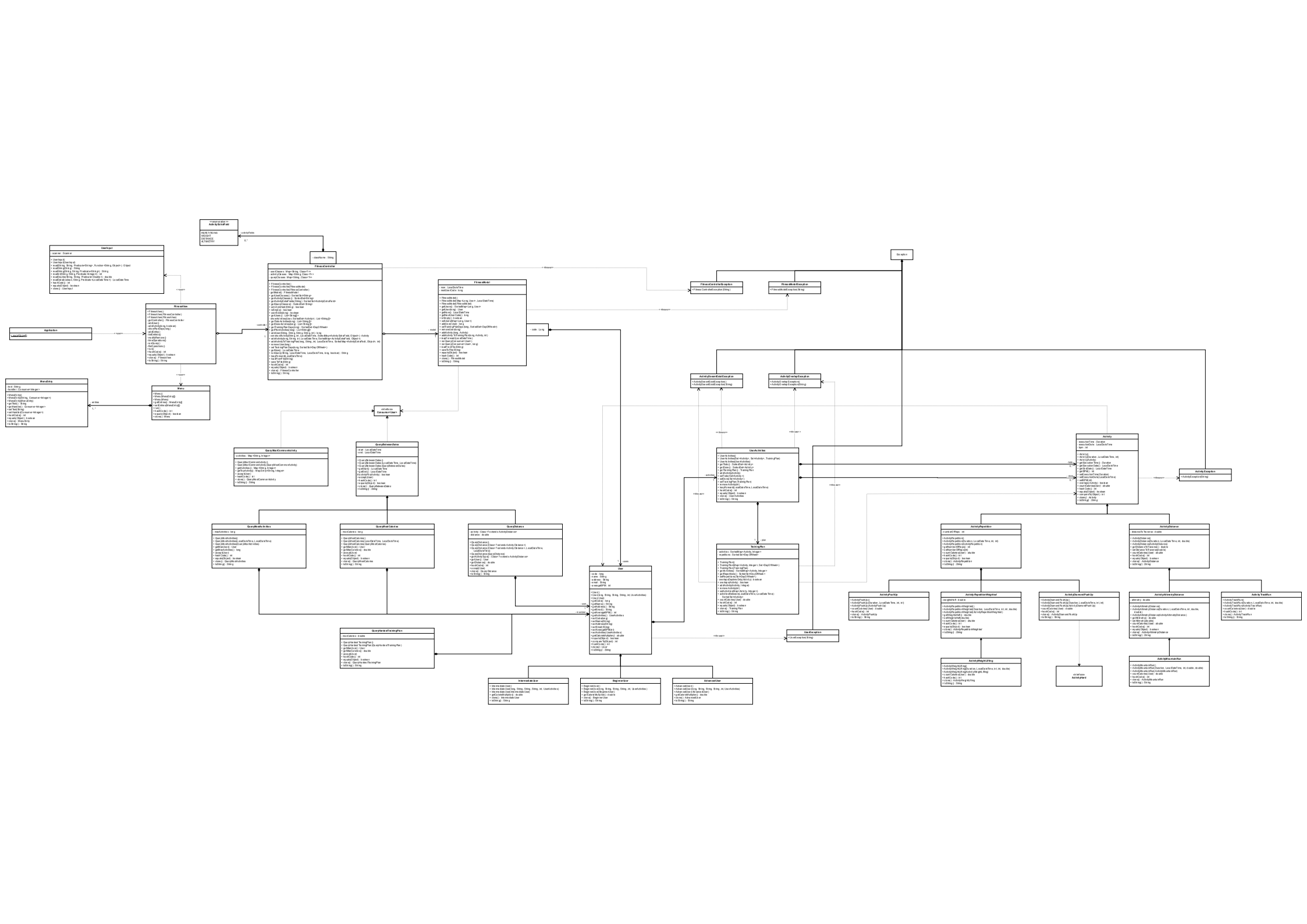
3 Conclusão

Em suma, o nosso grupo foi capaz de desenvolver a aplicação proposta, cumprindo a maioria dos requisitos, para ser específico, os requisitos para a nota máxima de 18 valores. O leitor pode questionar-se por que motivo o nosso grupo não conseguiu cumprir a totalidade dos requisitos.

A complexidade do projeto proposto não é elevada, e o nosso grupo não sentiu dificuldades na implementação da funcionalidade pedida. No entanto, a maioria das dificuldades encontradas têm origem numa única característica da linguagem Java: a sua verbosidade. A implementação

de cada funcionalidade tomou muito mais do nosso tempo do que o esperado, devido à quantidade de código que era necessário escrever. A título de exemplo, o comportamento de cada interrogação é definido por apenas um método, mas a criação de cada interrogação exige também a definição de vários construtores, *getters*, e métodos como `equals` e `clone`. Este maior esforço necessário não era esperado, conduzindo a um atraso no desenvolvimento que impediu a realização do último dos requisitos. Por outro lado, esta verbosidade é o que torna o código Java extremamente legível, pelo que esta característica da linguagem é definitivamente uma espada de dois gumes.

No entanto, considerando apenas os requisitos implementados, o nosso grupo encontra-se satisfeito com o resultado produzido: uma aplicação modular, que respeita os princípios do encapsulamento, facilmente extensível, que faz uso de mecanismos de reutilização de código, fartamente documentada e extensivamente testada.


Application
+ app.setUp():void()
«square»
FitnessView
+F fitnessViews()
+F fitnessViews(FitnessController)
+F fitnessViews(FitnessModel)
-getEventsView( ) : FitnessEventsView
-setEvent()
-setActivityLog(log, b:voidw ev)
-setActivityLog(String log)
-setAllUIIds()
-findEUIIds obj
-setEventsListeners()
-removeEventsListeners()
+ clo obj()
+ hasMEvent obj : int
+ setEventsListeners() : boolwe e
+ close obj : FitnessUser
+ toStr log() : String
«code use»
1
FitnessController
- user fClasses = Map< Str ing, Class<?>>
- activit yClas sses : Map <Entity, Class <?> >
- qu eryCla sses : Map< String , Class<> ?>>
+ F fitnessCo ntro ller( )
+ F fitnessCo ntro ller(FitnessModel model)
- getEventsFromDatabase( ) : boolwe eventsFromDatabase
- getAllActivityLogs( ) : boolwe allActivityLogs
- getActivityLogFromDatabase( ) : boolwe activityLogFromDatabase
- getAllActivityLogFromDatabase(String) : boolwe allActivityLogFromDatabase
- getUserId(Context ct) : Stable dUserID map
- addUserIdToPref(SStrin g) : boolwe
- isEmpty( ) : boolwe
- getUserId( ) : boolwe
- getUserId( ) : Lo ngLong
- storeUserId(long) : boolwe
- getUserId( ) : Lo ngLong
- getActivityLogs( ) : boolwe activityLogs
- getEvents( ) : Lo ngLong
- addEvent(String , String , String , String , int) : boolwe
- getEvents( ) : boolwe events
- getEventsFromDatabase( ) : boolwe eventsFromDatabase
- getEventsFromDatabase(String , String , int , Local Date toFilter , Local Date to) : boolwe eventsFromDatabase
- getEventsFromDatabase(String , String , int , Local Date toFilter , Local Date to , String , int , Local Date toFilter , Local Date to) : boolwe eventsFromDatabase
- getEventsFromDatabase(String , String , int , Local Date toFilter , Local Date to , String , int , Local Date toFilter , Local Date to , String , int , Local Date toFilter , Local Date to) : boolwe eventsFromDatabase
- getEventsFromDatabase(String , String , int , Local Date toFilter , Local Date to , String , int , Local Date toFilter , Local Date to , String , int , Local Date toFilter , Local Date to , String , int , Local Date toFilter , Local Date to) : boolwe eventsFromDatabase
- getEventsFromDatabase(String , String , int , Local Date toFilter , Local Date to