Textual explanation for the different class diagrams.

1. Domain “entity” classes diagram

Note: This diagram was presented in the first delivery but it was lacking some explanations that now we hope to provide.

The diagram that this explanation references can be found in the directory docs/domain-entity-classes-diagram.pdf

Detailed explanation:

In our system we identify a **User** with a unique name and we don’t require a password for signing in.

Note: The reasoning behind this is that we want it to feel like a classic arcade program that one can have in a computer. If, however, a password was found necessary by our stakeholder we could possibly implement it for the last version of the project although we weren’t planning on it.

Our system keeps track of a series of **Kakuro** puzzles identified by a name that is defined at the time of creation by the user that creates it or by the system itself if it is “automatically” generated. These kakuro puzzles have a **Difficulty** property that can be one of four: Easy, Medium, Hard or Extreme (if it was generated by the system) or it can have a value of null if it was created “by hand” by a user. This means that if there is a User associated as creator, we can’t ensure the difficulty of that kakuro because it wasn’t generated by the system’s standards.
A kakuro puzzle also holds a reference to an instance of a **Board** which is identified by a unique id and has a certain height (number of rows) and a certain width (number of columns), and is made of exactly |rows|*|columns| number of **Cell**s which store their coordinates in that board, that is: coordRow ∈ [0..height-1] and coordCol ∈ [0..width-1]. So a cell is identified by its coordinates and the board it is associated to. This board associated to a kakuro is “empty” or in “initial state” (and will remain that way) in the sense that it is in the same state that when it was created/generated, and it will serve as a blueprint or template for the board to be created when a User wants to play said kakuro puzzle.

In a certain board there are two, and only two, types of cells, **WhiteCell**s are the ones that can hold a value (which contributes to the sum of its row and column as described in the rules of the game), and some notations. We will store these notations as bits in an integer variable, where bit 0 represents the state of notation number 1, bit 1 → notation 2, up to, bit 8 → notation number 9, for efficiency reasons.

These notations represent values that a user marks on a certain white cell for self-help purposes, indicating which values he/she/it thinks could possibly go in that cell when solving a kakuro puzzle.
For reference: it is equivalent to the well known “Pencil mode” option that a lot of sudoku solving games implement, and a quite useful/helpful one that we wanted to include.

Note: We also use this variable when generating a new kakuro puzzle or solving one in our algorithms because it results useful to reuse methods and have it stored in the WhiteCell instance. Moreover, in the manual creation of kakuros we’re able to give feedback to the user as we assist him/her with our algorithms by using this same variable. In these algorithms we use it to indicate which values are still possible for that cell after a certain point in the process of generation/solving. There is parallelism with both uses as it is always an indication of “possible values” but we understand that giving it these two different uses can make it a bit difficult to comprehend and we wanted to make it clear.

The second type of cells are **BlackCell**s which hold the vertical sum (if they have a white cell immediately under them) of the column, and the horizontal sum (if they have a whihte cell immediately to the right of them) of the row.

At a given time, a User can start a new **Game** by selecting a certain Kakuro, when this happens a new instance of **GameInProgress** is created with timeSpent = 0 and its board will be a new instance of Board copied from the board of the game’s associated kakuro (using it as a blueprint as we discussed earlier). We also store the last time the game in progress was played to be able to order them, and thus, if the user leaves the game unfinished, later he/she is able to find it quickly and continue it from the state it was left.

The list of **Movement**s of a game in progress is initially empty, but it will procedurally increase in size as the game progresses when the user assigns values to white cells. We consider a movement a change of the value of a white cell (not it’s notations) and we store the index of the movement, the previous value, the value that was assigned to it, and the coordinates of the white cell it changed (they must coincide). A Movement is identified by its index (number of move from 1 to n) and the game in progress that it was played. Keeping track of the movements in a game lets us offer the user a tool to undo, redo, visit old positions, mark key moves and tell him/her what move was incorrect if a hint is asked for, which we believe is a big improvement in user experience and usability of the program.

When a user either completes a kakuro game or gives up and asks the system to solve the kakuro instead, an instance of **GameFinished** is created from the one in progress the user was playing, the timeSpent solving the puzzle is stored, also the finnish DateTime and a certain score that the user achieved.
The User’s score is the sum of all the scores of the games it has played and a score of a game is calculated by the following formula:

$$max(0,\ \frac{1}{time\ in\ sec} * d - num\ of\ hints * k)$$

d = 1000*difficulty
k = $num\ of\ black\ cells\ /\ num\ of\ white\ cells$

Note: This formula is provisional but we wanted to show a little the intention behind it: First, the faster you solve the kakuro the more points you get. The parameter d is based on the difficulty so the more difficult the kakuro is the more points you obtain. Lastly the player screen offers a help option that is VERY helpful, so we subtract points from the user if he/she uses it too much specially if the percentage of white cells is low. As there is a subtracting component, we take the maximum between zero and this calculation.

Some small corrections of the first version of the diagram:

We've noticed a couple of missing associations (thanks for mentioning them in the first correction as well), we've added the "creator" association between Kakuro and User, the multiplicity of the user end is 0..1 because a kakuro can either be created manually by a user or generated by the system. We've also added the creation DateTime to kakuro as it is a piece of data that we have decided to display in the final version.
We've removed the association between Movement and WhiteCell because we have decided it is easier to store the coordinates and always make sure that those belong to a white cell (as the coordinates aren't specific to this class but to the Cell class).
We have removed the "inferred" variables that seemed to result confusing and we have left those that are actually implemented as part of a class. The inferred values will be calculated at runtime to display several things like rankings, statistics, information on certain kakuros, etc. Kakuros now have names as identifiers instead of IDs. Games now have IDs for easier persistence layer implementation (doing "joins" between user, kakuro and dateTime seemed unnecessarily complex without the ability to use a database).

2. Presentation class diagram

The diagram that this explanation references can be found in the directory docs/presentation-class-diagram.pdf

Detailed explanation:

The **PresentationCtrl** is the general controller for the presentation layer, which we have organized in a specific way:

There are dedicated classes that are in charge of displaying the necessary information to the screen and capturing user triggered events (like button clicks or keyboard typing). We've called these classes "Screens" and there is one for each different view that we have on the program. These Screen classes inherit from **AbstractScreen** so they all implement necessary functions that are called from the controller (e.g. Screen resizes, window gets closed, minimized or loses/gains focus, etc.). Each screen is controlled by a dedicated screen controller that inherits from **AbstractScreenCtrl** (this is an abstract class for basically the same reasons as the one discussed before) and that is in charge of telling its specific screen what to display and communicate with the domain layer to ask for the necessary

information (PresentationCtrl passes them an instance of DomainCtrl -explained in the next section- so they can do so).

PresentationCtrl holds one, and only one, instance of each of the screen controller classes except for the two special cases of the GameScreenCtrl and CreatorScreenCtrl. These two classes are special as the interaction of the user with these screens is usually maintained over a certain period of time, and, unlike the others, do not just make one call to the domain controller to ask for information but are continuously in communication with dedicated domain controllers that need to take care of updating everything as the user plays or creates a kakuro to be able to save the user's progress at any point and give the user advice, help, feedback or whatever is needed.
Also, as a User can play more than one game or create more than one kakuro puzzle in a single "session" it wouldn't make much sense to just have one instance of these controllers.

Lastly, PresentationCtrl holds a "currentScreenCtrl" variable that is a reference to the screen that is visible at the time, and it is to which it will pass the window events that we discussed earlier.

Some clarifications:

We also have some classes in a directory called src/presentation/views and src/presentation/utils that any screen or screen controller that needs them can access but we didn't find them relevant to include in the diagram, and if we had included them it would've made the diagram more dense and possibly difficult to understand at first glance.

3. Domain class diagram

The diagram that this explanation references can be found in the directory docs/domain-class-diagram.pdf
There is a big section in the diagram that is called "Domain entity classes" that, for simplicity and ease of reading, replaces the diagram that you can find in the document docs/domain-entity-classes-diagram.pdf, which has already been explained in section 1 of this document.

Detailed explanation:

The **DomainCtrl** class is the general controller for the domain layer, it is instanciated by a PresentationCtrl instance and it holds a reference to five other domain controllers, each of which does specific, related, tasks.
UserCtrl takes care of "list all users", "user login" and "user creation" use cases and needs access to the UserRepository to perform these operations.
RankingCtrl is in charge of retrieving the necessary data and building the different ranking listings that the program offers (for example, ordering users by our point system, or by average time spent solving kakuros of a certain difficulty, etc.). It needs access to the UserRepository and the GameRepository to perform these operations.

StatisticsCtrl is the controller that can prepare information about a given user and its various statistics (like games played, games that have given him/her the most points, average play time, etc.) in order to respond to a Presentation layer request that wants to display such information. It needs access to the UserRepository and the GameRepository to perform these operations.
KakuroCtrl is able to list all the kakuros in the database ordered by a certain difficulty or filtered by users (if they are handmade in the creator or imported from an external file by a user). Moreover, it takes care of saving new kakuros to the database (imported, generated or created). It needs access to the UserRepository and the KakuroRepository to perform these operations.
GameCtrl is responsible for saving finished or in progress games to the database and also assembles the list of the "history" section of the dashboard by ordering all the user's games by last played date. It needs access to the UserRepository and the GameRepository to perform these operations.

Besides these specific controllers there are also two special controllers that do not have access to the persistence. They are dedicated to making sure that the user can play games and create kakuros and be offered help and/or advice throughout the process, as well as keep track of the generated data that must be persisted (in the case of the kakuro player) if the user suddenly exits the game or the program shuts down.
These two controllers are the **GameplayCtrl** and the **KakuroCreatorCtrl** classes, the gameplay controller uses the **Solver** algorithm to offer the requested hints, check if a user's solution is correct or even solve the kakuro for the user if this one asks for it. The creator controller uses both main algorithms, **Solver** and **Generator**, to also offer help, and it is able to generate a kakuro from the state that the user has left it. Both controllers use the **KakuroFunctions, SwappingCellQueue and KakuroConstants** helper functions to validate the user's actions and offer help.

4. Persistence class diagram

The diagram that this explanation references can be found in the directory docs/persistence-class-diagram.pdf

Detailed explanation:

At its core, the persistence system in our project uses a very simplistic approach: all data which we wish to persist (which coincides with the main domain entities shown in domain-entity-classes-diagram.pdf) will be stored in the local file system, inside the /data/DB directory, which will emulate some sort of non-relational database. The instances of the domain entities will be serialized to JSON objects and stored in JSON files. This might not be the most efficient solution, but storing objects in JSON files makes serialization very straightforward, and it allows us to easily read and interpret the contents of the files which can be very helpful when debugging.
For serialization and deserialization, we took the approach of using a well-known third-party library: Gson by Google. This enabled us to focus our efforts into other parts of the project instead of re-writing our own serialization library, which would have been very time-consuming and error-prone.

The structure of the persistence layer is very intuitive; there is one class which directly reads from and writes to the JSON files in the local machine. This class, named **DB** (since it tries to emulate a very basic DBMS) has two methods, one for reading the contents of a file and returning them as an array of Java Objects, and another for writing a Collection of Java Objects to its corresponding JSON file. This class is not attached to any other class in the project, since its only job is to persist any entity that is given to it.

As a layer between the database class and the use cases are the **Repository** classes. There is a single repository interface for each domain entity that we want to persist: **UserRepository**, **GameRepository**, **KakuroRepository** and **BoardRepository**. Each of these interfaces define the methods that the use cases need to perform their functions. These methods are in charge of saving an entity, retrieving it from the database, updating it, deleting it, etc.
For example, in order to update a User, we only need to instantiate a new UserRepository, call its getUser(String userName) method which returns a User instance, modify that user instance however we want and finally call the saveUser(User user) method of the repository. This allows the upper layers to use persistence in a very abstract and straightforward way.

5. All layers class diagram

The diagram that this explanation references can be found in the directory docs/all-layer-class-diagram.pdf

Detailed explanation:

This is a diagram that includes all the diagrams discussed above, and it shows the connections between the three different layers.

1. Connections between presentation layer and domain layer:

PresentationCtrl initializes and holds a reference of a **DomainCtrl** instance, which passes to the different screen controllers so they are able to access it and ask for the information they need.
As we explained before, there are the two special cases with playing a kakuro game and creating a kakuro that have a direct 2-way connection to their dedicated domain controllers. These two domain controllers (**GameplayCtrl** and **KakuroCreationCtrl**) are instantiated by the DomainCtrl instance and passed up to the PresentationCtrl so it can pass them to the corresponding screen controllers when the user decides to start playing or creating a kakuro. After the creation of the instance, neither of the general controllers holds an instance to the dedicated domain controllers, only the specific screen controller communicates back and forth with it every time the user makes a relevant action to offer continuous monitoring, help, and to keep track of the time the user is spending, as it is crucial for calculating the score at the end of a game.

1.1. A possible improvement

For modularity and reusability purposes maybe it would be a good idea that the communication from the special dedicated domain controllers to the GameScreenCtrl and CreatorScreenCtrl was implemented via an interface instead of the domain controllers holding an instance of those presentation controllers. In our case it didn't seem very necessary at the moment and the change would be more or less easy to make if needed in the future.

2. Connections between domain layer and persistence layer:

DomainCtrl initializes the **DB** instance that will be in charge of storing and retrieving data to/from the file system, and passes it to the different **Repository** objects that it also instanciates. The DomainCtrl holds a reference to the User, Game and Kakuro repositories as they are the ones needed by the specific domain controllers, so they can access these repositories to store or ask for data, but it doesn't hold the association to either the DB or the BoardRepository as it won't access them directly and they were only needed by the other repository instances.
As it is shown in the diagram, the specific domain controllers can access the repositories they need for their well-functioning.