CS3217

Relentless.

Members: Yan Chun, Yi Wai, Yi Yin, Zechu

Final Report

Overview

Relentless is a co-op multiplayer game for a maximum of 6 players that runs across multiple iOS devices over the network. In the game, players form a team of delivery workers working for a multinational conglomerate. This company delivers its products very quickly but its workers, namely the players, are overworked and mistreated. Each player has to work with his team to satisfy their customers, which will help them earn commission and feed their family. Their jobs are to pack items into packages based on their customers' orders and deliver the packages to their respective houses.

However, each player will receive different orders in their neighbourhoods on their own devices (Fig. 1 and 2). To complete an order, their own items in the warehouse are not enough (Fig. 3 and 4). Each player will need to ask around to find out who has the extra required items and send a partially completed package to that person. Then, he/she will pack the required items into the package and send it back. After which, the player will send the completed package to the correct house.

There will be different types of items, such as books, magazines, robots and toy cars. Some of these items are designed to be hard to describe. For example, the title of a book or magazine to be delivered can be "the title of a book", words that are difficult to pronounce, or homophones. Describing pictures can also be difficult because different pictures may have similar components. There are also rhythmic items that alternate their states according to a pattern, and items assembled from other items. As players cannot see one another's orders, the game requires players to describe their orders clearly and concisely in order to last longer in the game.

The team's customers' satisfaction level increases according to the accuracy and speed of package deliveries. Hence, it increases if players send a package correctly and quickly but drops if the package is incorrect or they take too long to deliver the package.

At the end of each round, players will be able to see their collective amount of money based on their customers' satisfaction level in the form of commision. At the same time, money will be deducted to pay for daily expenses. Players have to make sure that they do not run out of money to survive by working harder in the next round. This game promotes teamwork and encourages players to cooperate with their friends, whilst allowing them to experience the adrenaline arising from panic and chaos!

Features and Specifications

Feature 1: Multiplayer game connection

- Create a new game
- Join an existing game with a game ID
- Player information and game status are shared and synchronised across the network in real time
- Data such as packages can be sent across the network to another player
- Pause a game for 30 seconds after a round has started when at least one player moves the app to background; resumes only when all players are back.

Feature 2: Generation and Allocation

- Category generation
- Item generation (based on categories generated) and allocation to players
- Order generation and allocation to players
- Order allocation to houses

Feature 3: Maintaining timers

- Timer for each round
- Timer for each order

Feature 4: Marking packages

- Packages are marked with usernames of their creators and unique identification numbers

Feature 5: Packing items into packages

- Tap on "+" button to create a new package
- Tap on existing package in top bar to open package
- View items by categories
- Tap on item to add it to currently open package
- Tap on item in package to remove it from the package

Feature 6: Viewing orders

- Tap on a house to view the order for that house

Feature 7: Delivering packages

- Long press package at top bar to change to map screen
- Tap house icon to deliver package

Feature 8: Fulfilling orders

- Compare orders of house with delivered package
- Fulfilled if items in order are the same as those in the package and order is removed
- If not fulfilled, closest order is removed

Feature 10: Sending and receiving packages between players

- Long press package at top bar to change to map screen
- Tap another player's icon to send package
- Other player receives package

Feature 11: Individual satisfaction level

- Changes when package is delivered
- Change depends on whether package fulfills order and on the amount of time used
- Resets at the start of every rounds
- Order-time-dependence (drops along with time when there are orders)

Feature 12: Team satisfaction and money

- Satisfaction levels of all players are combined at end of the round
- The combined satisfaction level is converted into money, and daily expenses are deducted and is shown after each round

Feature 13: Lose condition

- Players lose if money drops below 0 at the end of the round

Feature 14: Round-end condition

- Round ends if allocated orders run out or timer runs out of time

Feature 15: Different types of items: Titled Items, Stateful Items, and Rhythmic Items

- Titled Items have a title, such as books or magazines, with confusing or hard-to-pronounce names.
- Stateful items have multiple possible states that they can appear in, for example a fruit could be an apple, pear, or orange.
- Rhythmic Items alternate their states according to rhythmic sequences

Feature 16: Assembled items

- Items are made up of other items
- Some items are inventory items only and some are order items only.
- Order assembled items are made up of inventory items
- Assembled items can also be assembled (i.e. toy car can be combined with a siren if such an item exists)

Feature 17: Local scoreboard

- Scoreboard with player's score history and usernames of all players for each game (shown when game ends)

Feature 18: Game parameters and settings view

- Player can adjust difficulty level in settings
- Game parameters (e.g. number of orders, number of categories) vary according to difficulty level and are used during order and item generation
- These game parameters can be modified by the game designer in a plist configuration file (Configuration/DefaultGameParameters.plist), or remotely on Firebase console using Remote Config.

Feature 19: Minimum and maximum number of players

- A minimum number of players (3) is required to start
- A maximum number of players (6) is allowed to join the game

Feature 20: Limit on number of items allowed in a package

- This limit may be imposed in some rounds and is decided based on a probability in the game parameters

Feature 21: Player username and avatar

- Players have to key in their own usernames and choose their avatars before starting a game

Feature 22: Configuration file for designing game items

- Game designers can design items that appear in the warehouse and orders, and link them to corresponding assets, through editing a plist file. Please refer to “Game Designer Manual” below (in Appendix) for more details.

Feature 23: Demo mode

- Only difference is that parameters are obtained from a local plist with a specified set of parameters
- Generates objects from all categories
- Longer round time

User Manual

This application runs on iPhones with iOS version 13.0 and after, with an Internet connection.

Rules and Objectives

The objective of this game is for you and your team to get through as many rounds as possible by ensuring that all of you do not run out of money. Each of you will have some items in their packing areas and will also receive orders. To complete an order, you have to get the items that you do not have from one another by sending packages to others for them to add items in.

You have to describe the item that you require to your teammate without showing them your order list. Each of your customers’ satisfaction levels will change according to the accuracy and speed of your deliveries. At the end of each round, all players’ satisfaction levels are combined and converted to money. Daily expenses will also be deducted each time. If your team runs out of money, you all lose. So remember to work together and have fun!

How to start playing?

Number of players: 3 to 6

There are two ways to start playing:

1. Create a new game

Click on <u>"Create Game"</u> and you will be given a unique game ID. Your friends should join your game using the same game ID.

2. **Join an existing game**
   If your friend has already created a game, click on <u>"Join game"</u> and key in the game ID as displayed on your friend's screen.

Once you have joined a game, set up your profile by choosing a unique username and avatar.

<u>Settings</u>

The player who created the game will be able to see a small gear icon at the top right corner of the screen. Click on it and adjust the slider value if you want to adjust the difficulty level of the game.

The player who created the game should click on <u>"Start"</u> when all players have joined and set up their profiles. Once the game has started, no new players can join.

Click <u>"Proceed"</u> to start the round when all players are ready.

<u>Game Area</u>

1. **Packing area**
   Packages bar
   This is where all your packages are stored.
   - Any packages that are sent to you from the other players will also appear here.
   - Tap on any package to open it.
   - Tap on the <u>"+"</u> button to add a new package

   Currently open package
   This is the package that is currently open. Tap on an item to remove the item from the package.

   Items Corner
   You can view all the items that you have here. You can tap on an item to add them to your package that is currently open.

   Category Button

This button describes the current category of the items in the Items Corner. Click on this to change the category and look at other types of items.

"Assemble" button
This button is used to assemble items that are made up of various parts. When you have consolidated all the parts that are needed to assemble an item, click on the "Assemble" button. Select the parts for that item and click on the "Assemble" button again to assemble the item. The item will only be assembled if the parts you have selected make up a valid item.

To deliver packages
Press and hold on a package in the packages bar to change to the map area if you want to deliver a package to a house or send the package to another player. When you release your finger, you will be redirected to the map area, click on the house or player that you want to deliver the package to.

To change to map view
If you simply want to view the houses and the orders, click on "To Houses".

2. **Map**
   The map shows all the houses present. Houses that have an order have a timer bar underneath representing the time left to complete that order.

   Checking orders
   Tap on a house to see the orders for that house. Different orders are separated by a grey bar.

   Change to packing area
   To go back to the packing area, click on "To Packing"

3. **Satisfaction Bar**
   This shows your customers' satisfaction level. The round ends if the satisfaction level reaches 0.

4. **Money**
   After each round, your team's combined satisfaction is converted into money and daily expenses will be deducted. You can see how much your team has after each round. If your money drops below 0, you all lose.

5. Scoreboard
When the game ends, you will be able to see a scoreboard with your team's score and all your previous scores from past games. See which friends you work better with here!

6. Pausing game
When you or one of your teammates moves the app to the background in the middle of a round, the game will be paused for at most 30 seconds, after which the game will end for all the players. The game will resume after all the team members move their app to the foreground.

Game Designer Manual

Please refer to the Appendix for how the game designer should modify game parameters locally or remotely, and how they can design the configuration of items, using Property List files.

Designs

Overview

Top Level Organisation

The top level design comprises six major components:

- View
- Game Controller
- Network
- Model
- Storage
- Configuration

The View component is in charge of handling user interactions. It calls methods in the Game Controller when the user performs an action. It will also update the view according to notifications from the Game Controller when changes happen within the model or network.

The Game Controller component is in charge of handling the interactions between all other components by translating and passing information amongst them. The host game controller will also generate the items and orders required for the game and will define the game-specific parameters such as the duration of one round and send them to the other devices over the network.

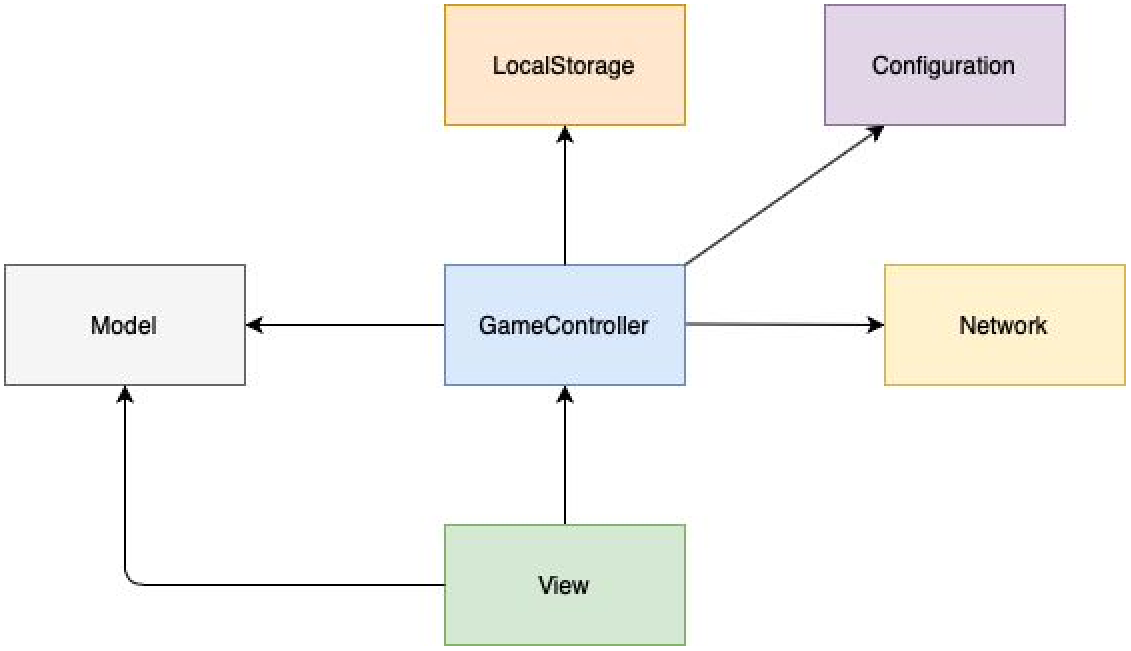
The Model component contains the objects that are involved in the game, such as Packages and Items, amongst others.

The Network component is in charge of synchronising all devices throughout the game. It updates the game state and handles the transfer of information (e.g. packages) between devices. The database in the cloud maintains shared data for all the games that are currently active.

The Storage component is in charge of local storage. It currently contains a scoreboard that keeps track of the player's score history.

The Configuration component is in charge of retrieving the parameter configuration values for the game, either from RemoteConfig or from local plists.

The diagram below shows the overall architecture diagram comprising the four components involved in this application.



<u>**Interesting Design Issues**</u>

Issue 1: Whether to implement `Item` as a class or a protocol

Initially, we attempted to use protocol-oriented programming and implemented `Item` as a protocol. However, our implementation also required `Item` objects to conform to `Hashable` and `Comparable`, which therefore restricted the collection of `Item` objects to a homogenous collection. In other words, different types of objects that conformed to `Item` could not be stored together. Therefore, `Item` was changed to become a parent class of different types of objects where some dummy methods are written to be overridden by its child classes instead.

Issue 2: How to identify which image was selected/dragged and linking it to the correct object in the Model
We started with subclassing UIView with an attribute that holds the object with the model. However, we found that Apple discourages subclassing UIViews.

So, we considered storing an identifier under the “Tag” attribute of UIView, then searching for an object that matches that tag in the Model (e.g. storing package number in Tag, then searching all packages for that tag). The benefit was that ViewController would have no knowledge of the model. However, there was not always a suitable identifier, e.g. for Houses that have no identifiers. Furthermore, a lot of information from each model could be required, which could be hard to pass.

Finally, we modelled each display as a UICollectionView, with Cells holding references to the objects from the model (i.e. PackageCell has a Package). UICollectionView has a lot of benefits, such as scrolling, multiple sections, section headers, etc. Although the ViewController now holds objects from the model, it does not modify the model without passing through GameController, hence separation of concerns still exists.

Issue 3: How to retrieve data from the online database
Initially, we considered calling a function that is supposed to return a specific data value from the database. However, we realised that reading data from the network is asynchronous and we have to wait for the results to come when we need certain data. As a result, all the methods that read data in Network protocol are designed to take in a listener (a closure), which will be triggered whenever there is a change in the relevant node of the database. The game controller initialises listeners to receive data from the network. Afterwards, when a listener is called, the game controller updates the model and notifies View to update its UI through NotificationCenter.

Issue 4: How to structure the various subtypes of Items
Initially, Item had four subclasses, namely, TitledItem, RhythmicItem, AssembledItem and Part (which represents a constituent item of an assembled item). Whenever we needed to add a new category of items, we would have to

create a new subclass of TitledItem, RhythmicItem or AssembledItem. This results in a lot of duplicated code every time a new subclass is created, and reduces flexibility for the game designer to design new items. In addition, an assembled item could only contain parts specially designed for this item (i.e. it was not possible for a “car” and a “lorry” to both have the same “wheel” as a component). It was also not possible for an assembled item to contain another assembled item.

We decided to separate the design and configuration of items from the game model. Currently, our Item has four subclasses: StatefulItem, TitledItem, RhythmicItem, AssembledItem. Items are now generated from a Property List (Plist) file, where the game designer can design any number of items for each subtype by following the specified format. An assembled item can contain any type of items, including StatefulItem, TitledItem, RhythmicItem, and even another AssembledItem. Assembled items can be nested to an arbitrary depth as designed by the game designer. This design significantly reduces code duplication and increases flexibility as the game designer can now design the game without editing any code.

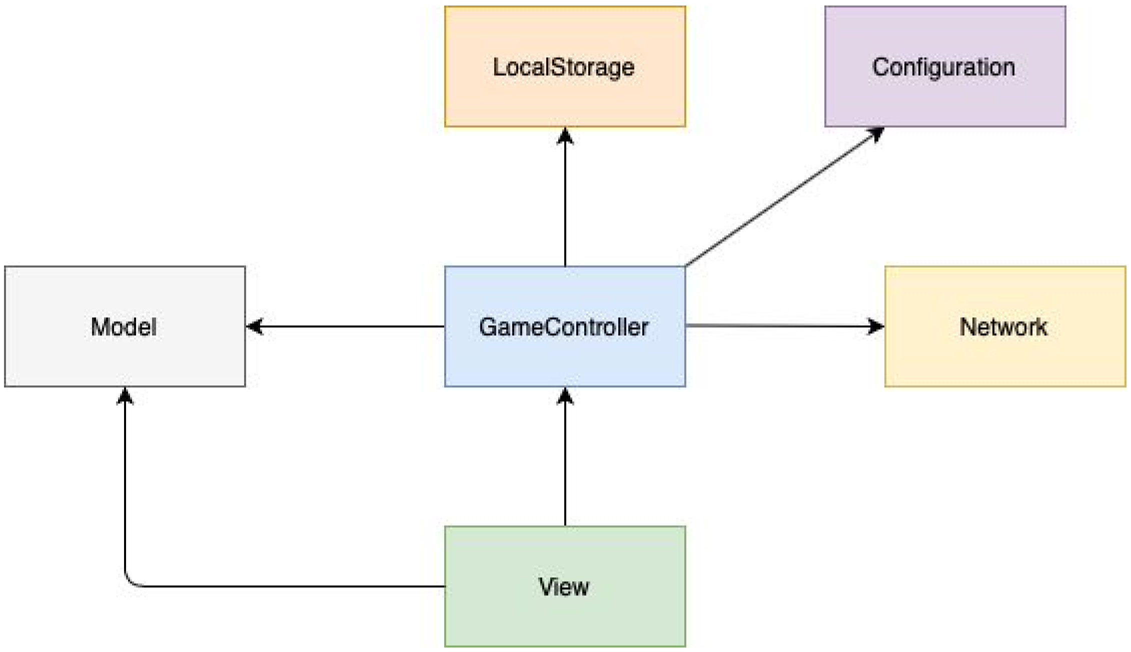
Issue 5: How to change the game parameters.
Initially, all the game parameters, such as difficulty level, round time etc were represented as static values in a class. Then, we decided to put the game parameters in a plist file, and their values can be modified in the config file or remotely on Firebase console. This allows the game designer to make small changes to the parameters without modifying the code. Furthermore, users do not have to update the application in order to receive new updates to the parameters, as it is accessed remotely. This design makes deployment of new parameters easier.

Runtime Structure

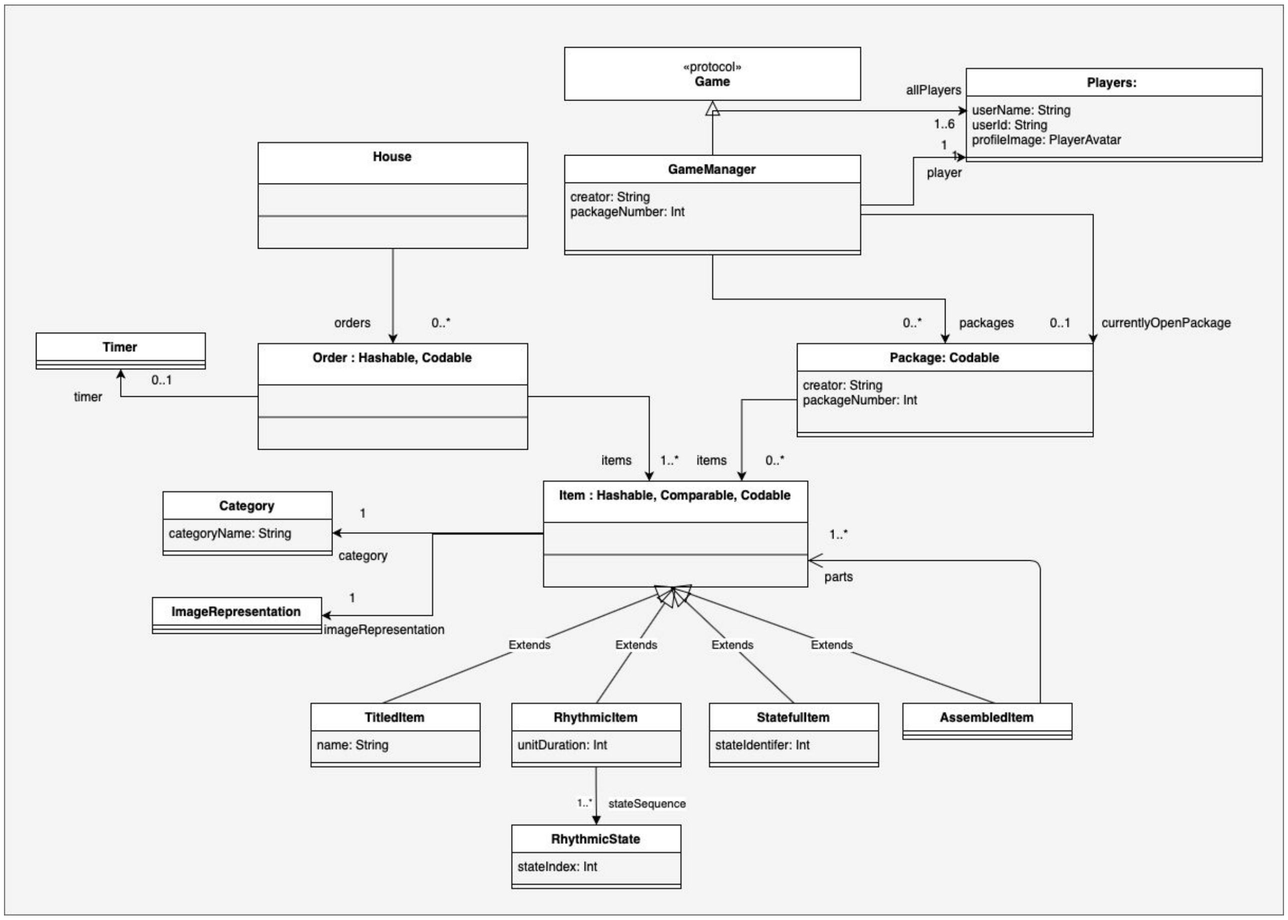
When the application is running on each player’s device, the game only stores the information (packages, orders etc) of one player (the player him/herself) instead of all the players currently in the game. The player only knows other players’ profiles such as their game IDs and nothing else. The online database does not store all the game state information either. It only stores information that needs to be communicated from one device to another, such as game status and user IDs of all players. When a package is sent from one player to another, it is placed under the recipient’s node in the database, and is deleted from the database as soon as it is downloaded onto the recipient’s device. Please refer to the Model Class Diagram under Module Structure below for how data is represented during runtime.

Module Structure

We used the NotificationCenter, which is based on the *Observer Pattern*, to handle the interactions between the components. In general, the GameController observes the Model and the View observes the GameController. The *Facade Pattern* was also used to separate the view from the internal components of the application. This decouples the view from the internal structure and implementation.



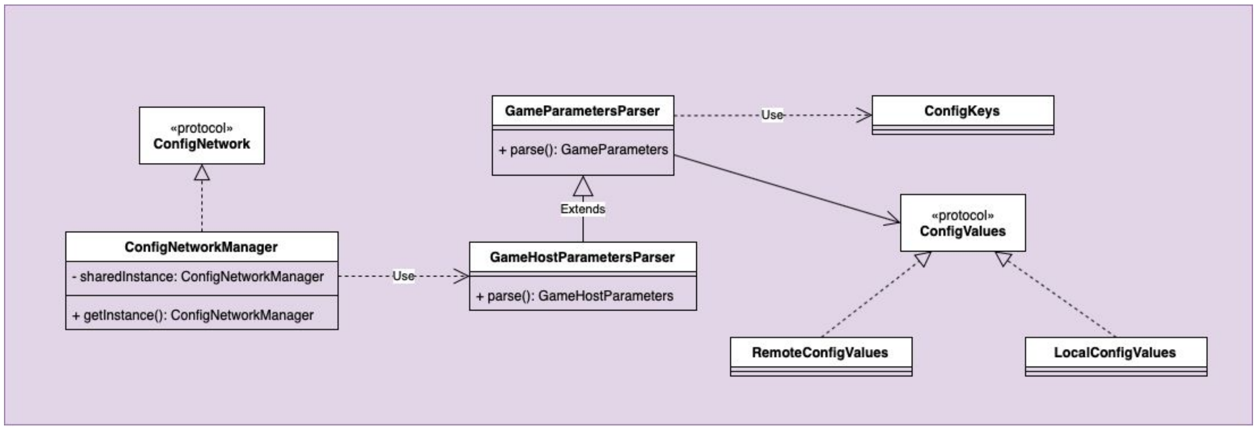
Model Class Diagram:



The Game Controller accesses the model via the Game protocol. This component comprises all the objects needed for the game, including Package, House, Order, etc. Orders are stored in a Set to prevent duplications, hence requiring the Order class to conform to the Hashable protocol. As a result, Items also have to be made Hashable. Order, Package and Item conform to the Codable protocol to allow them to be transferred over the network to other players.

This architecture can support various types of items. Currently, we have four different types of items, mainly TitledItem, RhythmicItem, StatefulItem and AssembledItems, all of which extend Item. In particular, AssembledItem contains a collection of Item objects that make up the assembled item.

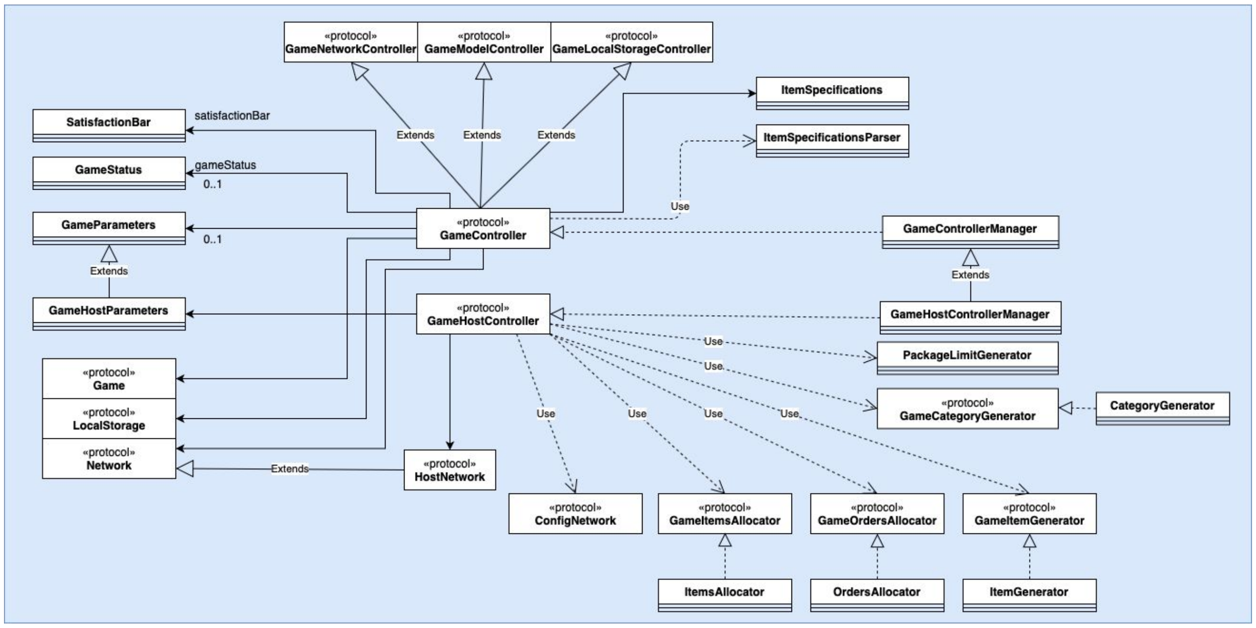
Configuration Class Diagram



The Singleton Pattern is used for ConfigNetworkManager, which fetches and activates RemoteConfig values. It creates a RemoteConfigValues object and passes it into the GameHostParametersParser, which parses the values into a GameHostParameters object.

There is another type of ConfigValues object -- LocalConfigValues. This can be used to get values from local plists. The ConfigNetworkManager has a method to store the values from RemoteConfigValues in a LocalConfigValues object that can be sent over the network to the other devices (refer to game logic section below)

Game Logic Class Diagram:



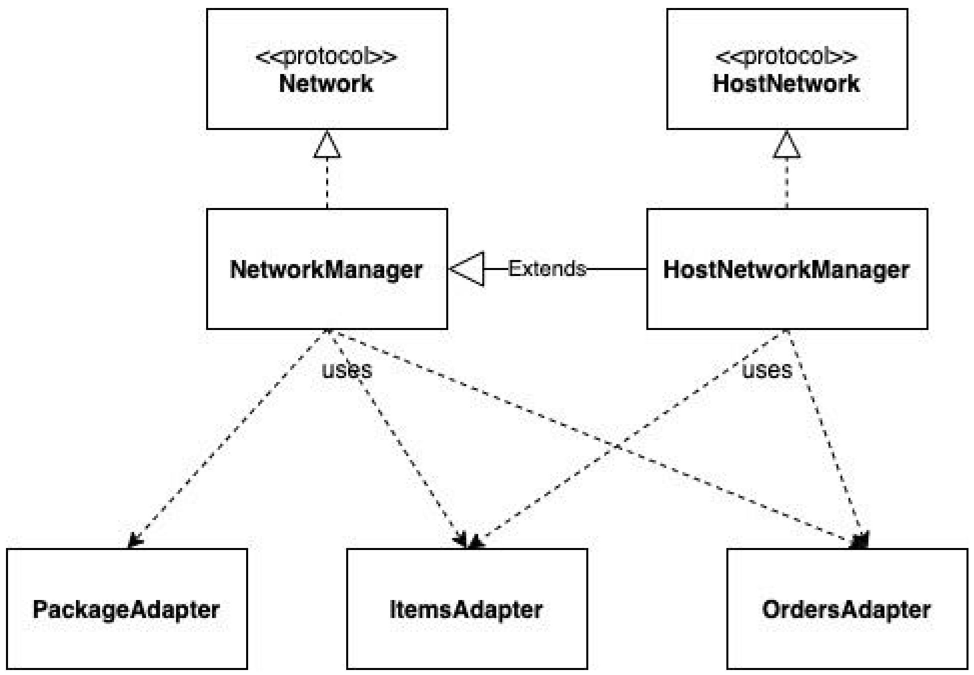
The game controller acts as the bridge between the Model, View and Network. It also handles the game logic such as the game status, round timers and customer satisfaction. It also contains a GameParameters object, which represents certain properties of the game.

Hosts have additional methods and parameters that are specified in the GameHostController protocol and represented as GameHostParameters respectively. The GameHostController either gets the GameHostParameters object from ConfigNetwork or from a local plist, which is parsed using a parser (not shown in diagram). The GameHostController will send a LocalConfigValues object that contains the values corresponding to those that it uses to the other devices, and the other game controllers will parse them into GameParameters objects on their own. This is to ensure consistency of the game parameters across all devices as fetching and activating of Remote Config values can sometimes fail.

The design for game generators and allocators is based on the Command Pattern. It uses the protocols GameCategoryGenerator, GameOrdersAllocator, GameItemsAllocator and GameItemGenerator. This gives the classes that implement the protocols the freedom to decide which parameters to use in generating and allocating the objects. To allocate items or orders in a different way, the developer can simply implement another kind of allocator without having to change other parts of the code.

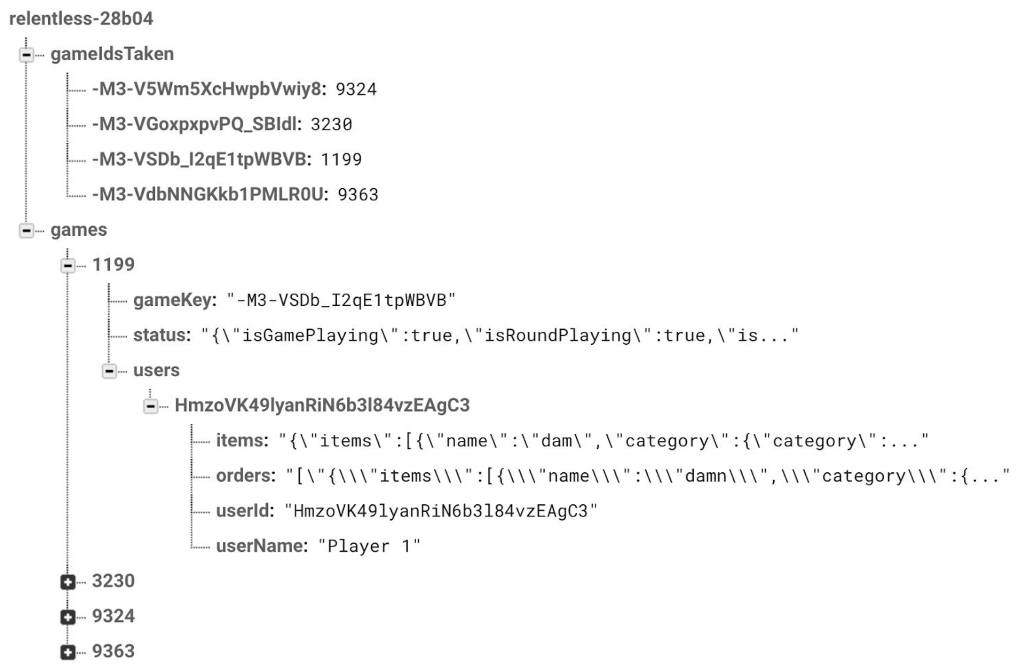
The game controller has an ItemSpecifications object, which contains information about available items for warehouse and orders. An ItemSpecifications object is created by the ItemSpecificationsParser upon the creation of the game controller. The parser gets all the available items by parsing the GameConfig.plist file and creating new categories and items as specified by the game designer.

Network Class Diagram:



The Network component exposes its functionalities to its client through the Network protocol (for non-host players) or HostNetwork protocol (for host player). The methods in the Network protocol enable the user to join and exchange data with other players. The methods in the HostNetwork protocol allow the host to create, start and terminate a game or a round, and share with other players pre-generated items and orders. These methods can be separated into two categories: methods that send information to other players, and those that attach listeners to receive changes in the online database.

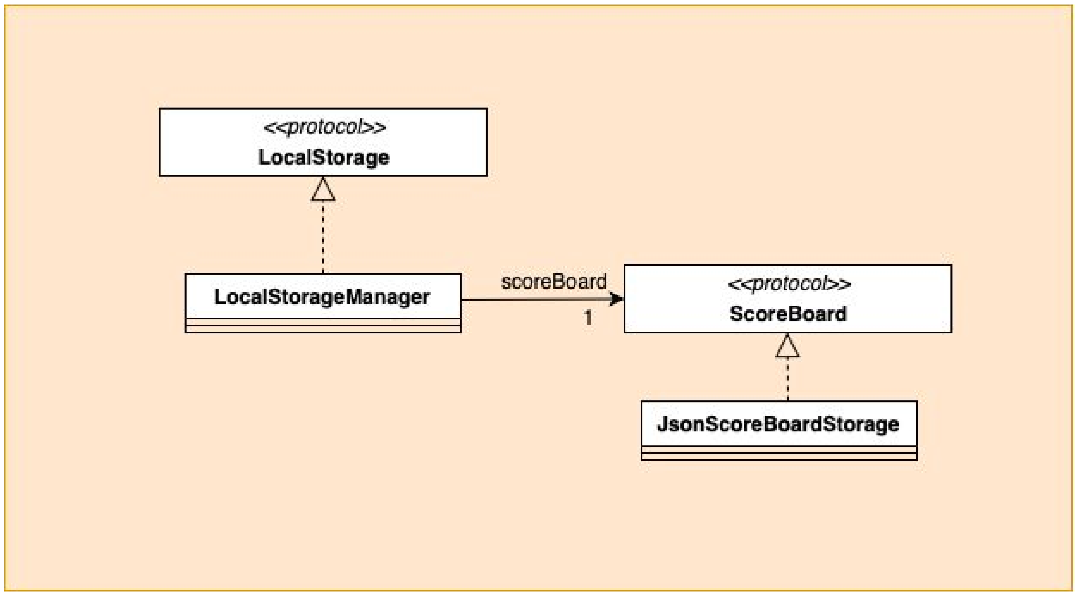
The Observer Pattern is used here for Game Controller to observe changes in relevant values in the network. HostNetworkManager extends NetworkManager, as the host player should have access to all the functionalities available to a normal player. Classes such as PackageAdapter, ItemsAdapter and Orders Adapter serialise Packages, Items and Orders to be sent over the network. The implementation classes NetworkManager and HostNetworkManager use Firebase Realtime Database. A snapshot demonstrating how this online database is structured is attached below:



```
relentless-28b04
  gameIdsTaken
    -M3-V5Wm5XcHwpbVwiy8: 9324
    -M3-VGoxpxpvPQ_SBIdl: 3230
    -M3-VSDb_I2qE1tpWBVB: 1199
    -M3-VdbNNGKkb1PMLR0U: 9363
  games
    1199
      gameKey: "-M3-VSDb_I2qE1tpWBVB"
      status: "{\"isGamePlaying\":true,\"isRoundPlaying\":true,\"is..."
      users
        HmzoVK49lyanRiN6b3l84vzEAgC3
          items: "{\"items\":[{\"name\":\"dam\",\"category\":{\"category\":..."
          orders: "[\"{\\\"items\\\":[{\\\"name\\\":\\\"damn\\\",\\\"category\\\":{..."
          userId: "HmzoVK49lyanRiN6b3l84vzEAgC3"
          userName: "Player 1"
    3230
    9324
    9363
```

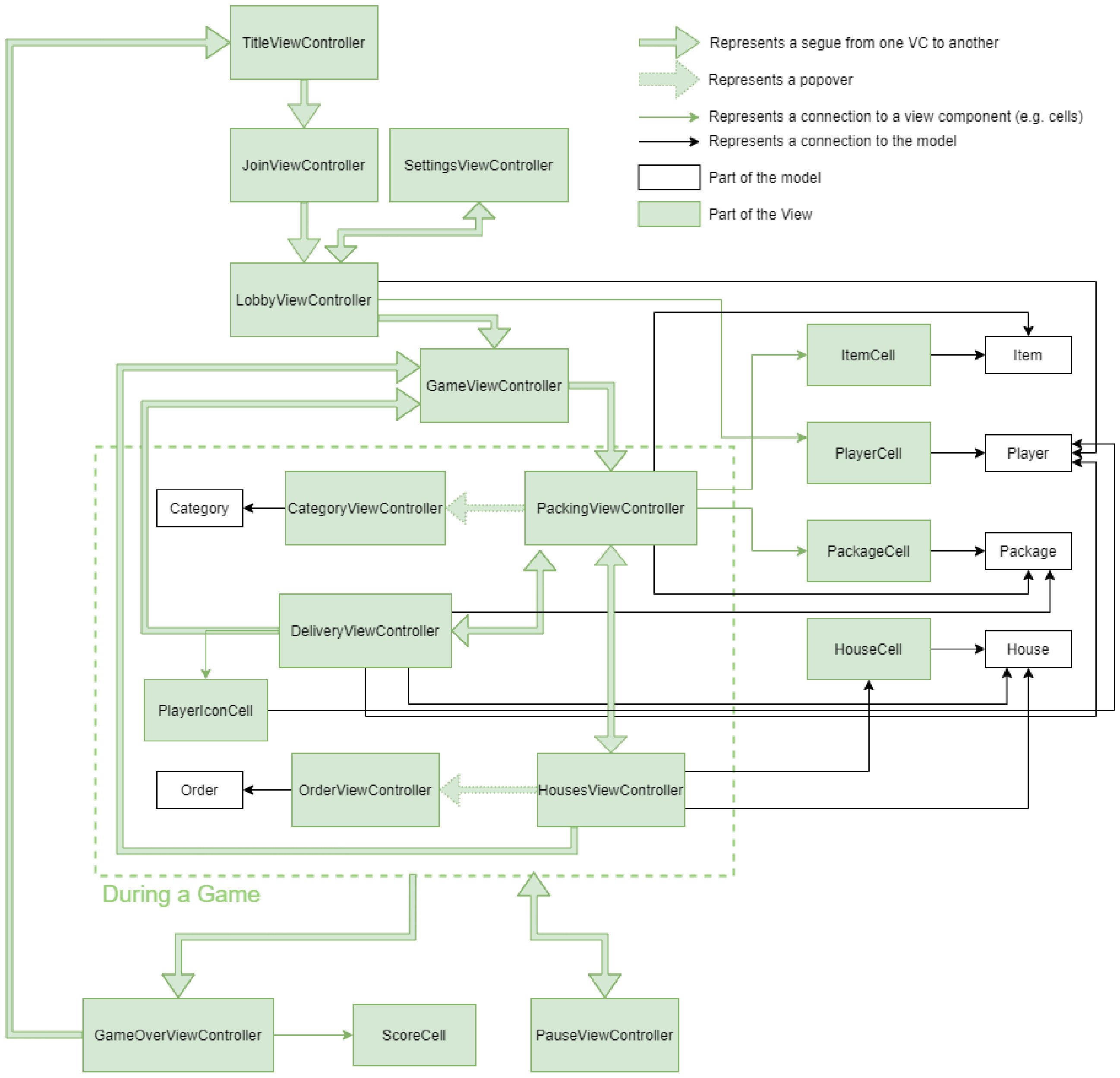
`gameIdsTaken` is stored in a separate tree to flatten the database structure so as to reduce the amount of data downloaded when the creator of a game checks which game IDs are unavailable. `games` store all the currently active games keyed by their game IDs. Each game consists of shared game status and individual player information.

Local Storage Class Diagram:



The local storage manager contains a scoreboard. The scores are currently encoded using JSON. The ScoreBoard protocol allows for the abstraction of the encoding method.

View Class Diagram:



The View Component consists of a Main.storyboard file and these view controllers as depicted above. The View Component firstly presents information from the model, and secondly calls methods from the GameController interface based on user input. The View uses the Observer pattern to observe changes in the GameController, before knowing to update its views. This Observer pattern is implemented through NotificationCenter.

Below is a brief explanation of what each View Controller does:
TitleVC - Starting view where players can choose to join a game or create a game
JoinVC - For joining games with a game ID
LobbyVC - To see other players and wait for the game to start. Only the host can start the game.
SettingsVC - To adjust the difficulty level for the game
GameVC - Appears in between each round
PackingVC - Displays the player's packages and items. Allows creation and modification of packages.
CategoryVC - Allows changing of categories to be displayed in the PackingVC
DeliveryVC - To send a specific package to a house or another player
HousesVC - Allows viewing of the orders from each house
OrderVC - Shows the orders from a specific house
GameOverVC - Shows the local scoreboard when the game ends
PauseVC - Countdown from 30 when a player has left; ends game upon 30 seconds up

All presentations of the model are displayed via UICollectionViews. The Item, Player, Package, House, and PlayerIcon cells extend from UICollectionViewCells. Each cell holds a reference to a model object of their type. They convert the model object into a NIB/visual representation for display in the View.

Testing

Test Strategy

Glass box testing was done with unit tests. They are done for classes in the Model component e.g. Package, Order etc. as well as item generators, network adapters and parsers. These classes were chosen to ensure that the logic of the game is deterministic and expected. For example, unit tests were done on `ItemGenerator` to ensure that the assembled items that were generated for orders can be assembled by inventory items. UI tests are also performed for integration testing in a single-player context. Our team did blackbox and greybox manual integration testing for multiplayer mode by following our test plan using a Demo Mode. In Demo Mode, all categories of items will appear in our inventory as well as orders, and the round time is extended. This allows us to repeatedly test parts where errors are likely to occur. Please refer to the Appendix for our test plan.

Reflection

Evaluation

1. Performance issues: Lags in responsiveness of application (e.g. when round ends, the screen takes a while to change on other players' devices)
    a. Delays are most likely due to Firebase and notification lags
2. Important features
    a. ItemSpecifications and GameParameters makes the architecture more extensible as the game designer can change and add new items/parameters with limited changes to the code
    b. Separating host and non-host game controllers and network enhances Separation of Concern
3. We did a lot of work in previous sprints in terms of implementing the necessary features for a user to play the game. Hence, in this final sprint we did not have to worry about creating necessary game features, but could focus on improving the abstraction and architecture of the software, as well as on fixing the bugs that had cropped up.
4. We managed to meet our goals for our software that we set out at the beginning of the project in terms of the playability and extendability of the game. After this module, we can now easily proceed to work on game design and playtesting as the software is largely complete.

Lessons

1. We learnt that there can always be more abstraction. For example, previously, we had parameters as constants and classes of items extending from certain types of items, like TitledItem and RhythmicItem. We previously considered this abstracted already, as parameters can just be numerically edited and new items just require extending from the needed classes. However, in this Sprint, all of the items and parameters were changed to be decided by config files that can be edited by a game designer with no knowledge of code. We realised that this is a much better form of abstraction, as there can easily be many different config files to generate entirely different sets of items. There could also be different modes with different parameters, such as a demo mode which is now easily implemented.
2. Communication is very important when working as a team on software. Because we were all working on different components of the software, we had to communicate regarding the links between the components. We found that frequent integration was crucial. Furthermore, certain aspects required

another component's implementation. In such cases, it was important to communicate on how the interfaces would look like, such that stubs could be created, or on how long it would take to develop, such that it is more efficient to wait for that change to be merged into the master branch.

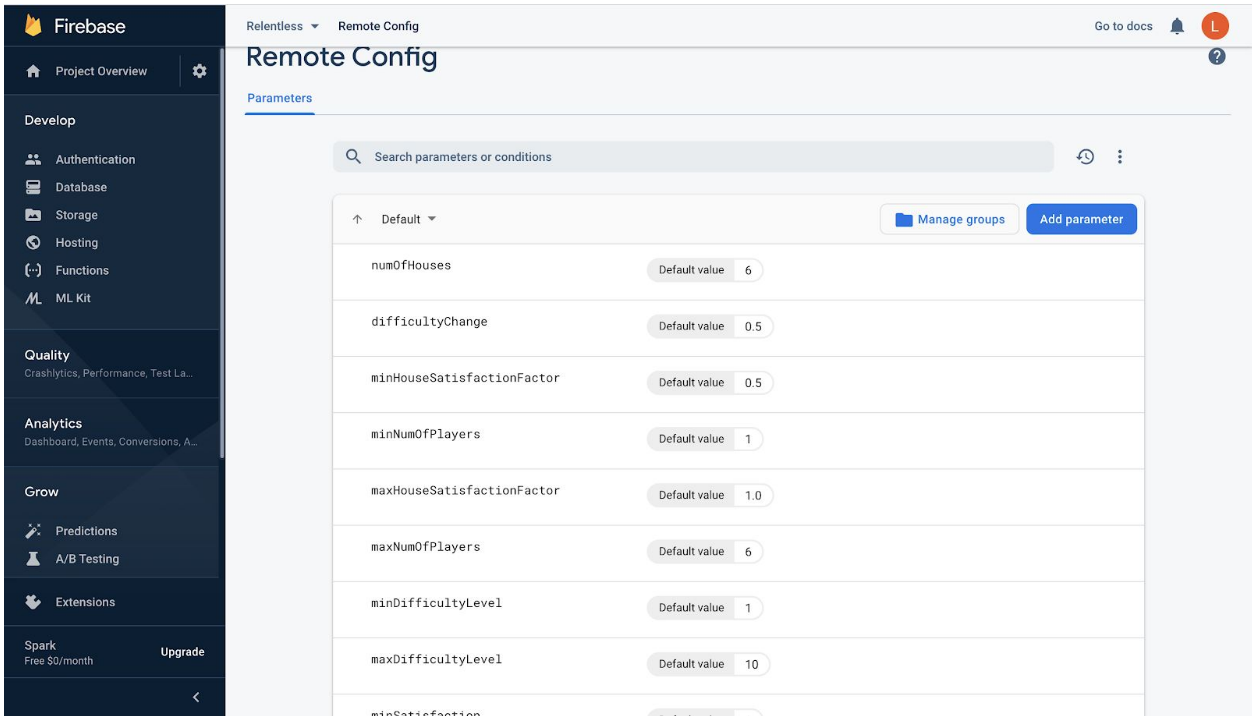
3. Known bugs and Limitations
    a. If the host presses back in the lobby screen while other players are in the game, the other players will remain in the lobby screen and see no players - they should instead be forced back to the title screen. This issue can be solved by having a listener in the lobby screen to detect if the game is deleted.
    b. If the player opens the orders on a house and closes the order at the exact same time as the round ends, this could cause that player to be stuck on the same screen while all other players move on to the next round. This problem arises due to the attempt to present a view controller on the OrdersViewController just as it is dismissed. The dismissal of the OrdersViewController has to be prevented when the round is ended to solve this problem.

Appendix

Game Designer Manual

Modifying Game Parameters

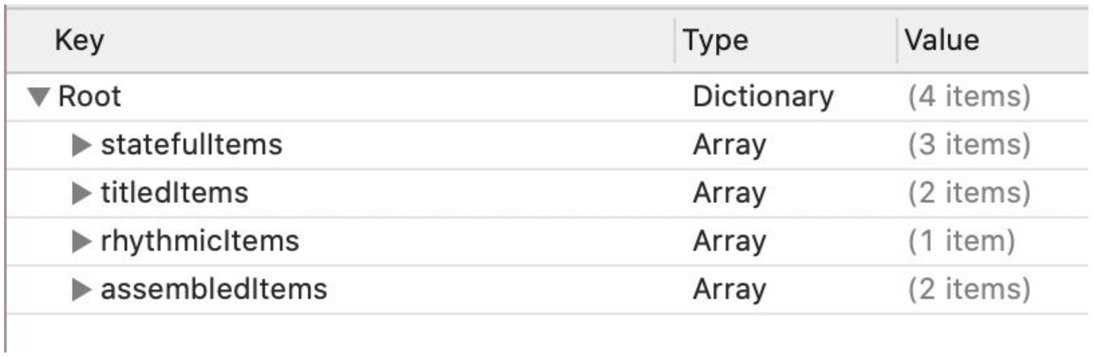
The game designer can change the values of game parameters on the Remote Config console (as shown below) on Firebase, and then publish the changes. When a device launches the application, it will fetch the updated values from Remote Config, but will only activate them on the next launch. This is because fetching values may take some time. Hence, the previously fetched values are used to reduce user wait time and also to avoid the need to use a loading screen. Default local values can be changed in Configuration/DefaultGameParameters.plist.



numOfHouses	Default value 6
difficultyChange	Default value 0.5
minHouseSatisfactionFactor	Default value 0.5
minNumOfPlayers	Default value 1
maxHouseSatisfactionFactor	Default value 1.0
maxNumOfPlayers	Default value 6
minDifficultyLevel	Default value 1
maxDifficultyLevel	Default value 10

Modifying Configuration of Items

The game designer can design items that appear in the game by editing the file Configuration/GameConfig.plist.



Key	Type	Value
▼ Root	Dictionary	(4 items)
▶ statefulItems	Array	(3 items)
▶ titledItems	Array	(2 items)
▶ rhythmicItems	Array	(1 item)
▶ assembledItems	Array	(2 items)

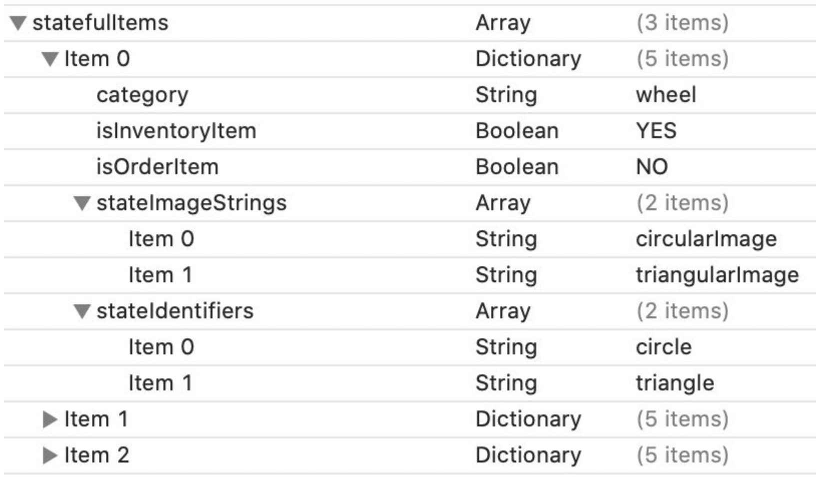
General rules:

- There are four subtypes of items: stateful, titled, rhythmic and assembled. These four subtypes are fixed.
- Under each subtype, there can be an unlimited number of categories of items. For example, titledItems can have “book”, “magazine”, “newspaper” etc.
- To add a new item under each subtype, just add a new entry (dictionary) in the corresponding array, and follow the rules for each subtype as described below.

- For each category of items, there are a few properties that are common to all four subtypes:
    - category: a string of the category name
    - isInventoryItem: a boolean indicating whether this item will appear in a player's warehouse
    - isOrderItem: a boolean indicating whether this item will appear in a player's order
    - Image strings that represent the corresponding assets will also need to be specified for various items.
- Important
    - *Every field* needs to be filled when creating a new category.
    - *Every* category has to have at least N number of inventory items, where N is the maximum number of players for the game
        - Note: number of permutations of an assembled item has to be at least N
        - Exception: category does not contain any isOrder items

We will now go through how to design items for each subtype by editing this plist document.

1. StatefulItem



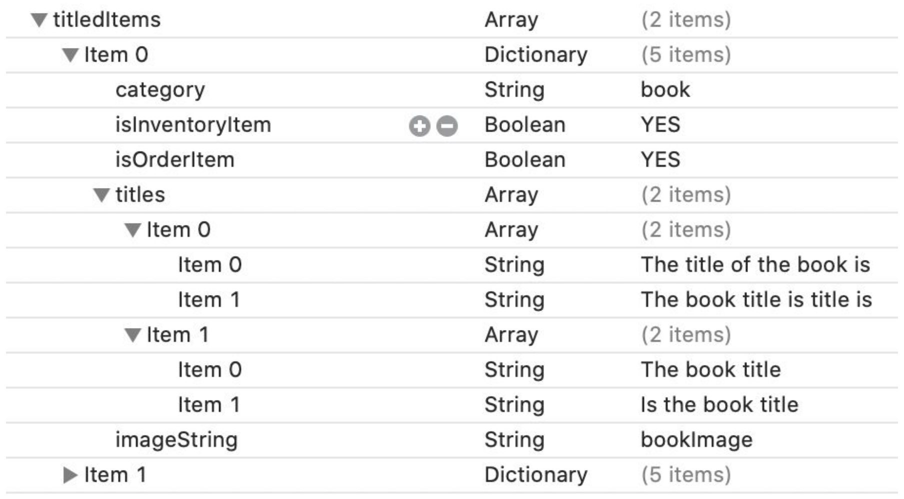
▼ statefulItems	Array	(3 items)
▼ Item 0	Dictionary	(5 items)
category	String	wheel
isInventoryItem	Boolean	YES
isOrderItem	Boolean	NO
▼ stateImageStrings	Array	(2 items)
Item 0	String	circularImage
Item 1	String	triangularImage
▼ stateIdentifiers	Array	(2 items)
Item 0	String	circle
Item 1	String	triangle
▶ Item 1	Dictionary	(5 items)
▶ Item 2	Dictionary	(5 items)

A stateful item is one with several distinct states. For example, a wheel can be circular, triangular or rectangular. A fruit can be an apple, an orange or a pear. Each state is indexed from 0 in the order they are specified under a category.

Properties:
- stateImageStrings: an array of image strings for all the states, in the order of their state indices

- stateIdentifiers: an array of strings that specify the name for each state, in the order of their state indices as well.

2. TitledItem



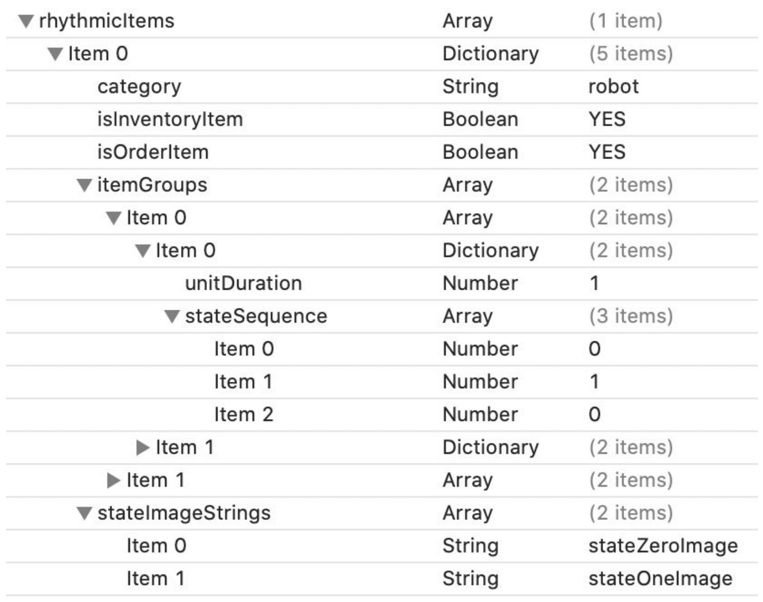
▼ titledItems	Array	(2 items)
▼ Item 0	Dictionary	(5 items)
category	String	book
isInventoryItem	Boolean	YES
isOrderItem	Boolean	YES
▼ titles	Array	(2 items)
▼ Item 0	Array	(2 items)
Item 0	String	The title of the book is
Item 1	String	The book title is title is
▼ Item 1	Array	(2 items)
Item 0	String	The book title
Item 1	String	Is the book title
imageString	String	bookImage
▶ Item 1	Dictionary	(5 items)

A titled item is an item with a title. Examples include “book” (confusing phrases) and “magazine” (homophones), but the game designer can add more categories if needed, just like other subtypes.

Properties:

- titles: an array of arrays of strings. Within the titles array, each array is a group of titles that should appear in a round together as these titles in the same group resemble each other. The game designer may add any number of strings in a group, and any number groups in the titles array.
- imageString: the image asset string for this category

3. RhythmicItem



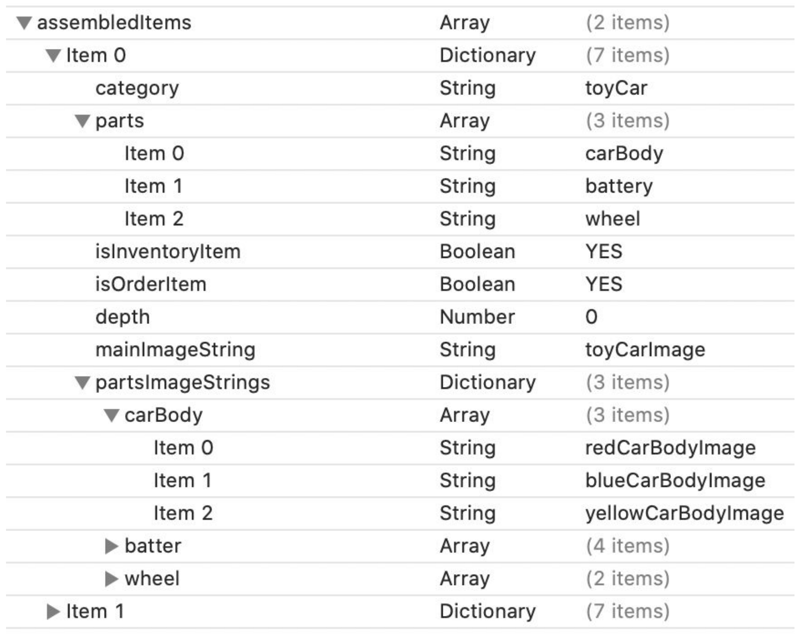
▼ rhythmicItems	Array	(1 item)
▼ Item 0	Dictionary	(5 items)
category	String	robot
isInventoryItem	Boolean	YES
isOrderItem	Boolean	YES
▼ itemGroups	Array	(2 items)
▼ Item 0	Array	(2 items)
▼ Item 0	Dictionary	(2 items)
unitDuration	Number	1
▼ stateSequence	Array	(3 items)
Item 0	Number	0
Item 1	Number	1
Item 2	Number	0
▶ Item 1	Dictionary	(2 items)
▶ Item 1	Array	(2 items)
▼ stateImageStrings	Array	(2 items)
Item 0	String	stateZeroImage
Item 1	String	stateOneImage

A rhythmic item has a repeating pattern of various states, such as a siren, or a lightbulb that lights up according to some rhythm. Each rhythmic item is defined by (1) unit duration and (2) state sequence.

Properties:

- itemGroups: similar to “titles” in titled items, itemGroups is an array of arrays. Each inner array is a group of similar-looking rhythmic items that are confusing for the players to distinguish, and therefore grouped together. There can be any number of groups, and any number of elements within a group. Each element in a group defines a unique item:
  - unitDuration: the number of seconds in the interval between two rhythm states
  - stateSequence: an array of integers, where each integer is the index of a rhythm state. This sequence will be looped repeatedly in the UI.
- stateImageStrings: an array of strings of image assets, where the image string at index n of the array corresponds to rhythm state n.

4. AssembledItem



▼ assembledItems	Array	(2 items)
▼ Item 0	Dictionary	(7 items)
category	String	toyCar
▼ parts	Array	(3 items)
Item 0	String	carBody
Item 1	String	battery
Item 2	String	wheel
isInventoryItem	Boolean	YES
isOrderItem	Boolean	YES
depth	Number	0
mainImageString	String	toyCarImage
▼ partsImageStrings	Dictionary	(3 items)
▼ carBody	Array	(3 items)
Item 0	String	redCarBodyImage
Item 1	String	blueCarBodyImage
Item 2	String	yellowCarBodyImage
▶ batter	Array	(4 items)
▶ wheel	Array	(2 items)
▶ Item 1	Dictionary	(7 items)

An assembled item consists of other items. Each constituent item can be *any* type of items, including stateful, titled, rhythmic and other assembled items (it's possible to have nested assembled items!).

Properties:

- parts: an array of strings, where each string is the category name of a constituent item. If there are more than one item of the same category, then repeat the category name for the corresponding number of times. (e.g. if there are 2 wheels, 1 car body and 1 battery, then the array will be ["carBody", "battery", "wheel", "wheel"].
- depth: this integer indicates how nested an assembled item is. When an assembled item does *not* have another assembled item as a part, it has a depth of 0. An assembled item with a depth-0 item has a depth of 1, and so on. Constraint: constituent assembled items should always have a *strictly smaller* depth (and therefore an item cannot have itself as a part).
- mainImageString: the main image frame of this assembled item, which serves as an additional backdrop to the constituent images. Can be useful
- partsImageStrings: this is a dictionary that maps each constituent category name to an array of image strings. For a titled or assembled item, only one string is needed. For a stateful or rhythmic item, the image string at array index n corresponds to state n.

Test cases

Unit tests:

- `TitledItem`
    - Testing < comparison
        - `TitledItem` with lexicographically smaller category than another `TitledItem` with lexicographically larger category should return true
        - `TitledItem` with lexicographically smaller category than another `TitledItem` with lexicographically smaller name but larger category should return true
        - `TitledItem` with lexicographically smaller name than another `TitledItem` with larger name in the same category should return true
    - Testing `equals` method
        - Different isInventory status, isOrder status and image representation should still return true (all others constant)
        - Different name, category should return false
        - `TitledItem` with same name and category should return true
- `AssembledItem`
    - Testing < comparison
        - `AssembledItem` with lexicographically smaller category than another `AssembledItem` with lexicographically larger category should return true
        - `AssembledItem` with lexicographically smaller category than another `AssembledItem` with fewer parts but larger category should return true
        - `AssembledItem` with fewer parts than another `AssembledItem` with more parts in the same category should return true
    - Testing `equals` method
        - Different isInventory status, isOrder status and image representation should still return true (all others constant)
        - Different parts, category should return false
        - `AssembledItem` with same parts and category should return true

- `RhythmicItem`
    - Testing < comparison

        - `RhythmicItem` with lexicographically smaller category than another `RhythmicItem` with lexicographically larger category should return true
        - `RhythmicItem` with lexicographically smaller category than another `RhythmicItem` with smaller duration but larger category should return true
        - `RhythmicItem` with smaller duration but same sequence as another `RhythmicItem` with longer duration in the same category should return true
        - `RhythmicItem` with same duration but smaller sequence magnitude as another `RhythmicItem` with bigger sequence magnitude in the same category should return true
    - Testing `equals` method
        - Different isInventory status, isOrder status and image representation should still return true (all others constant)
        - Different unit duration, state sequence, category should return false
        - `RhythmicItem` with same unit duration, state sequence and category should return true
- `Order` tests
    - Testing checking for correct package
        - If pass in correct package that matches order, should return true
        - If pass in correct package with a different sequence than the array of items stored in the order, should return true
        - If pass in package with at least one different/wrong item, should return false
    - Testing getting number of differences between package and order
        - Correct package should return 0
        - Incorrect package should return the number of items that are different from the order
    - Testing ==
        - Order with the same items should return true
        - Order with the same items in different sequence should return true
        - Order with different number of items/different items should return false
- `Package` tests
    - Testing add item

        - Package.items should contain the the newly added item
        - package.items’s count should increase by 1
    - Testing delete item
        - Package.items should no longer contain the deleted item
        - package.item’s count should decrease by 1
    - Testing ==
        - Package with same items in same sequence should return true
        - Package with same items in different sequence should return true
        - Package with different items/different number of items should return false
- `House` tests
    - Testing checking for correct package
        - If pass in correct package that matches at least one order in the house, should return true
        - If pass in correct package with different sequence that matches at least one order in the house, should return true
        - If pass in package that does not match any order in the house, should return false
    - Testing get closest order
        - If pass in a correct package that matches an order in the house, should return that order
        - If pass in an incorrect package that has one difference from one order and two differences from another order, should return the order that the package differs from by one item
        - If pass in an incorrect package that has one difference from both orders, should return the order with the earlier timeout i.e. lesser time left
- `GameManager` tests
    - Testing add package
        - Packages should have contain the package
        - Count should increase by 1
    - Testing add new package to empty array of packages
        - Packages count should increase by 1
        - Packages should contain a new empty package
        - Currently open package should be the new empty package
        - Adding another package: the second package added should have a bigger package number than the first one
    - Testing remove package that exists

        - Packages count should decrease by 1
        - Packages should not longer contain the deleted package
        - If package was open,
            - Currently open package should now be nil
    - Testing remove package that does not exist
        - Packages count should remain the same
- `ItemGenerator` tests
    - Sufficient items
        - Inventory items >= number of players
        - Order items and inventory items split correctly
    - Insufficient items
        - Should not have infinite loop
    - Assembled items
        - Inventory items can assemble all assembled items generated
- `ItemAllocator` tests
    - Number items allocated should be equal to total number of items that were supposed to be allocated
- `OrderAllocator` tests
    - Non-assembled items
        - Max number of items in orders and number of orders per player should be less than the specified number
    - Assembled items
        - Same as above
        - All assembled items that are created can be assembled
- `ItemAssembler` tests
    - Correct parts
        - Should be able to return the assembled item that is made up of those parts
    - Incorrect parts
        - Too few, too many or wrong parts should throw exception
- `SatisfactionBar` tests
- `GameHostParametersParser` and `GameParametersParser` tests

Integration Testing

- Starting a game
    - Host(first) device: Click “Create Game” button
        - Should display a four digit number
        - Should display one player on the screen
    - Other devices: Click “Join Game” button

        - All devices should see the new player on the screen
    - Host(first) device: Click “Start” button
        - All screens should change to the “Are you ready for the new day” screen
    - Other devices: Should not be able to click “Start”
    - Host(first) device: Click “Proceed” button
        - All screens should change to the packing area screen
    - Other devices: Should not be able to click “Proceed”
- Adding a package
    - Click “Add” button on the top bar
        - Should have a new box added to the bar
        - Box should be opened (name of package will displayed on top left corner)
- Adding an item to the open package
    - Click on item on the right panel
    - Item should appear on the left panel
- Deleting an item from the open package
    - Click on item in the left panel
    - Items should disappear from left panel
- Delivering a package
    - Click and hold the package on the top bar
    - Screen should change to map screen upon releasing hold
    - Click on a house to deliver selected package to that house
- Changing from packing screen to map screen
    - Click on “To Houses”
- Changing from map screen to packing screen
    - Click on “To Packing”
- Viewing an order
    - Click on house view in map view
    - Pop-up showing orders for that house should appear
- Sending a package to another player
    - Click and hold the package on the top bar
    - Screen should change to map screen upon releasing hold
    - Click on a player icon to send the selected package
    - Package should appear in the top bar in the other player’s screen
- Changing satisfaction level
    - Should change only when a package is delivered
    - For a given amount of time taken, if items in the package match an order for that house, satisfaction level should increase

    - If time taken is more, satisfaction level should increase by a smaller amount
    - For a given amount of time taken, if items in the package match do not an order for that house, satisfaction level should decrease
    - If time taken is less, satisfaction level should decrease by a smaller amount
- Pausing the game
    - Host device: create a game
    - Other devices: join the game
    - Host device: start the game
    - During a round, any one of the players can move the app to background
    - All other devices should see a white screen pop up, showing count down from 30 seconds to 1 second.
    - Another device can move the app to and from background
    - As long as at least one device is not in foreground, the pause screen should continue counting down.
    - When all the devices are back to foreground, the pause screen should disappear and the game should resume.

GUI Screenshots

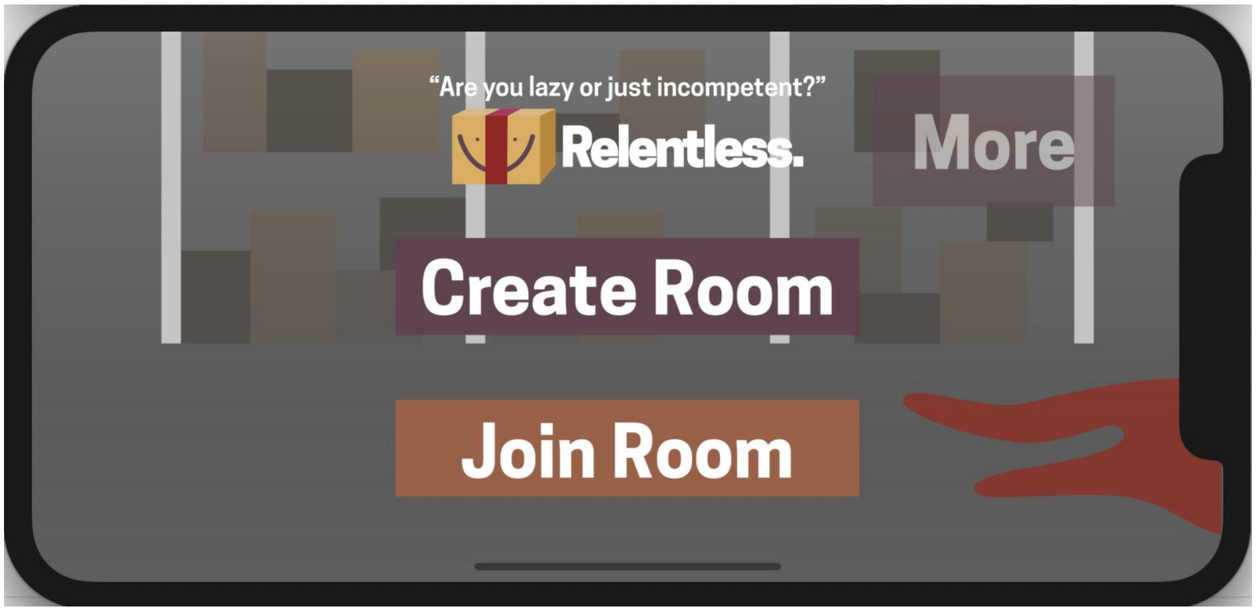


Figure 1: Starting screen

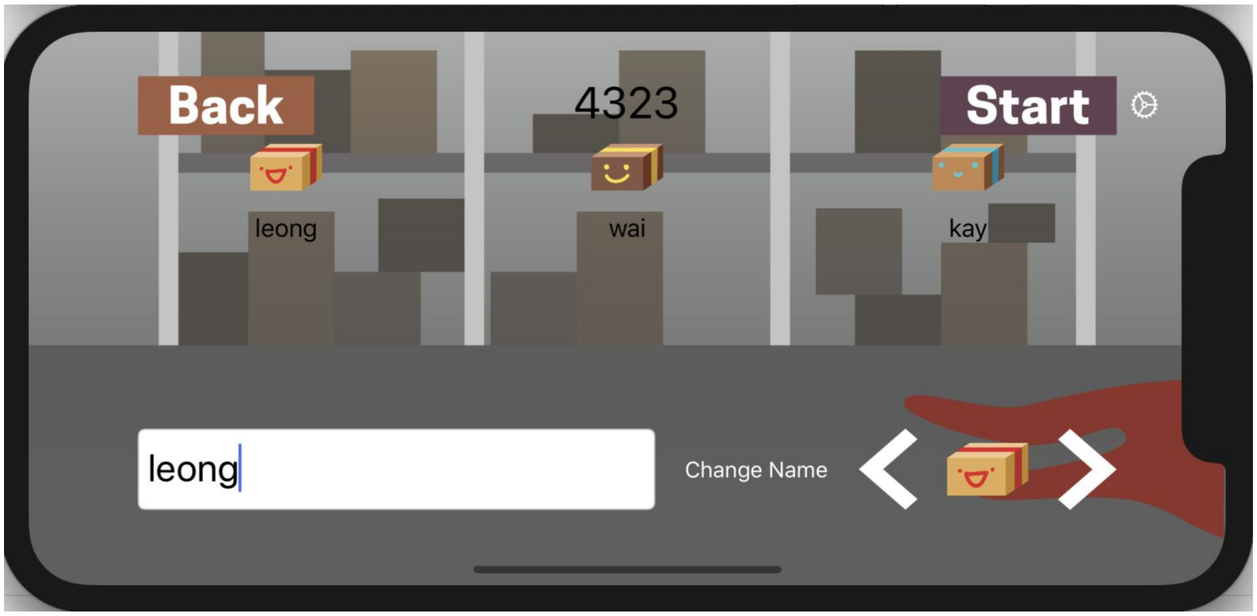


Figure 2: Lobby screen

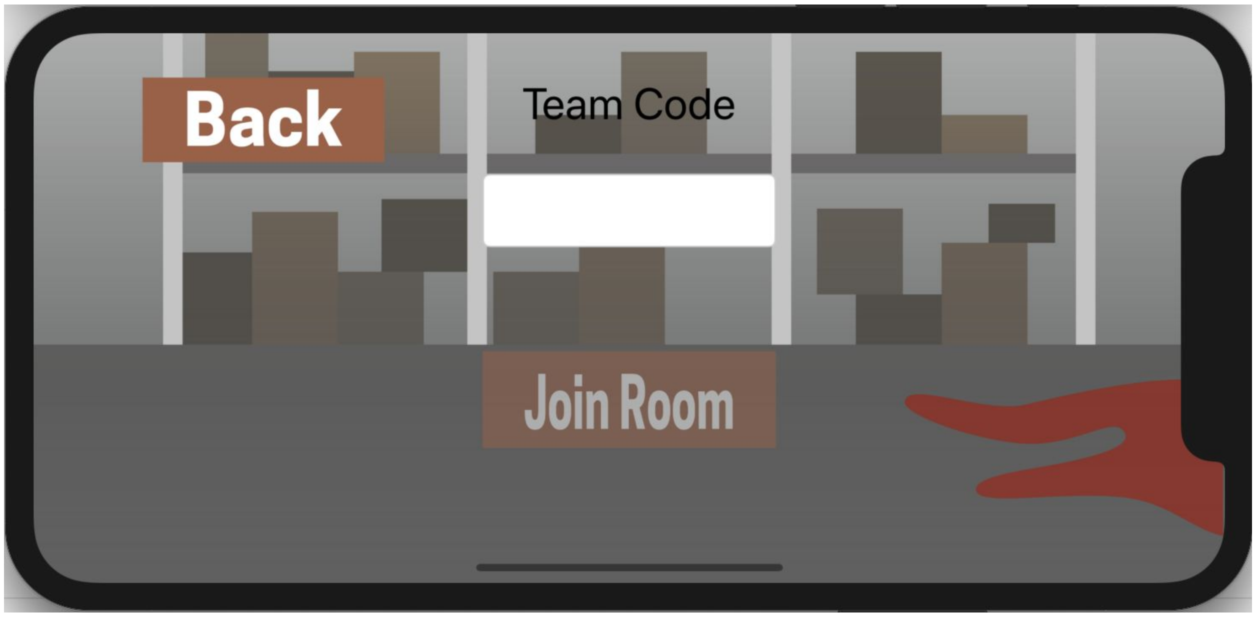
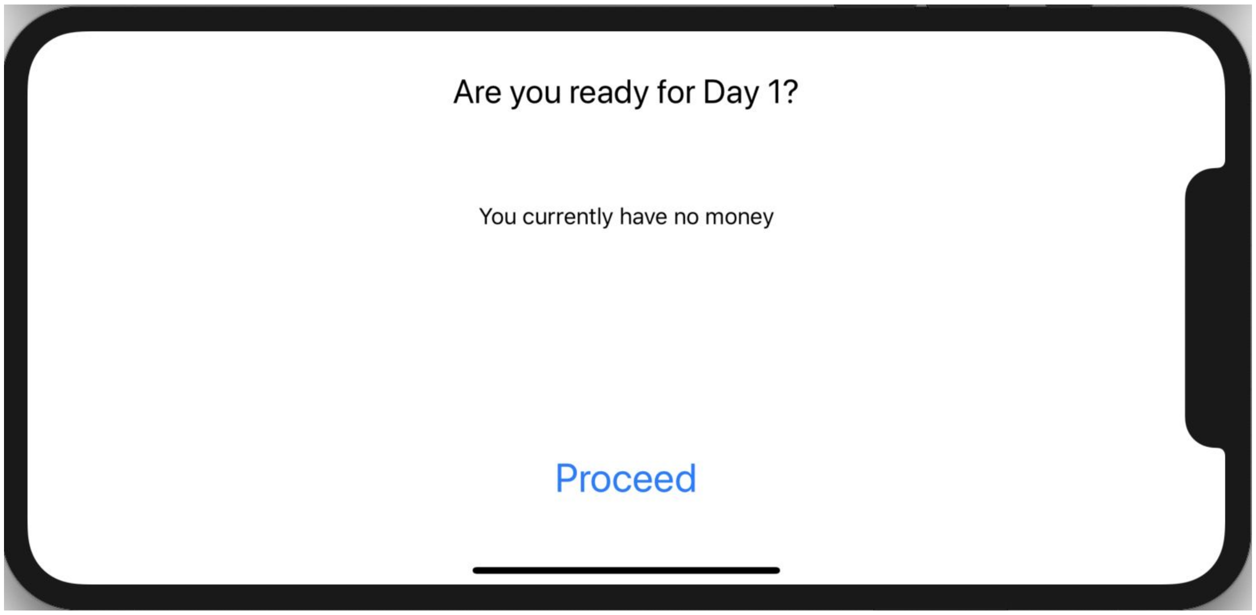


Figure 3: Join game screen



Are you ready for Day 1?

You currently have no money

Figure 4: Start round screen

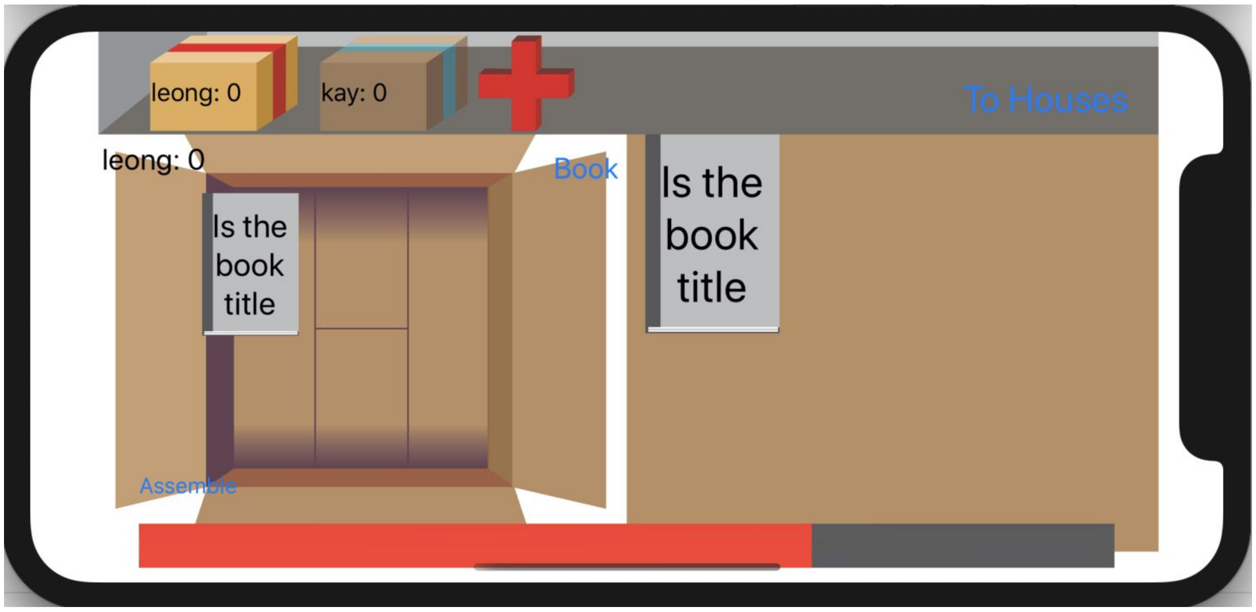


Figure 5: Packing screen

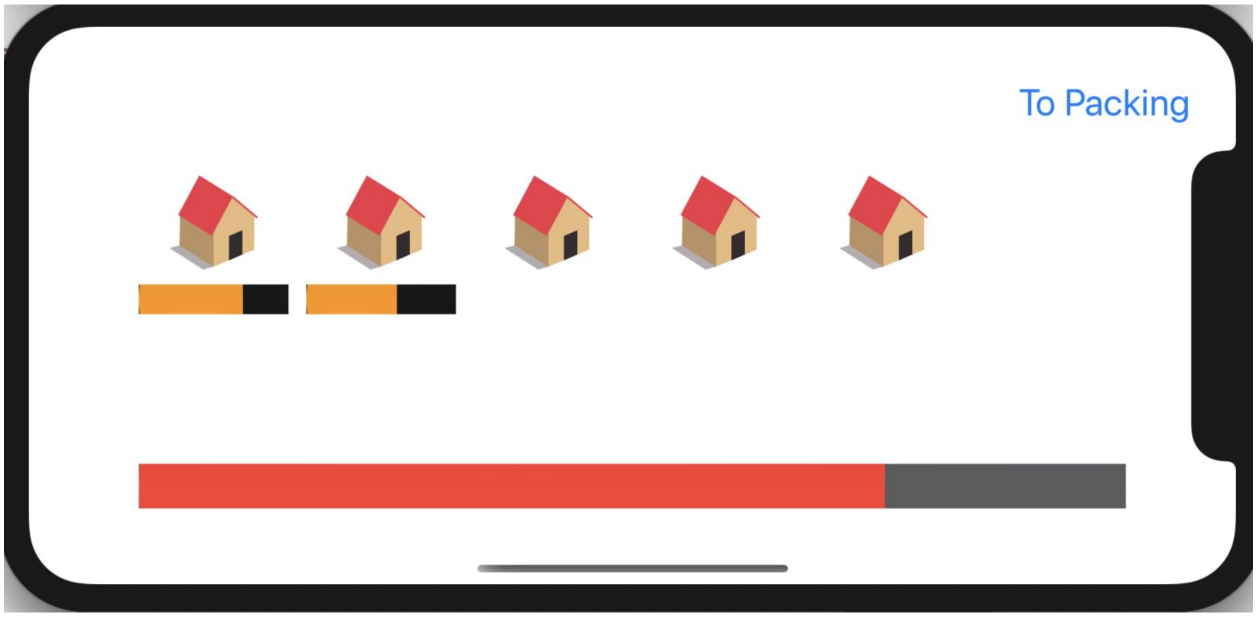


Figure 6: Map screen (houses)

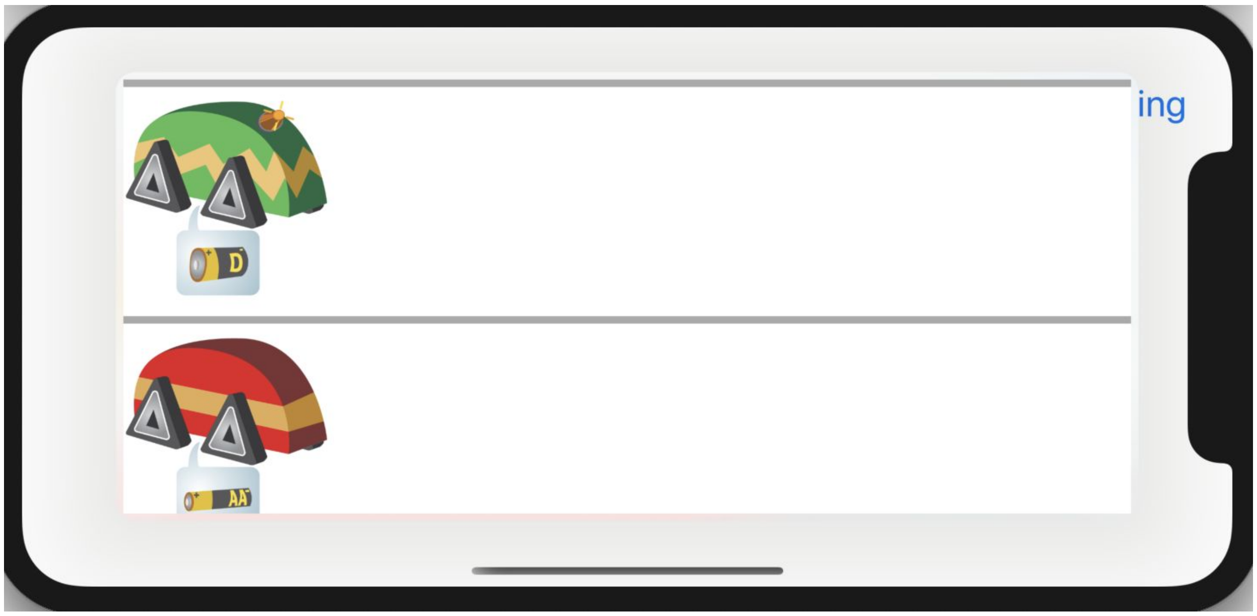


Figure 7: Map screen (orders)

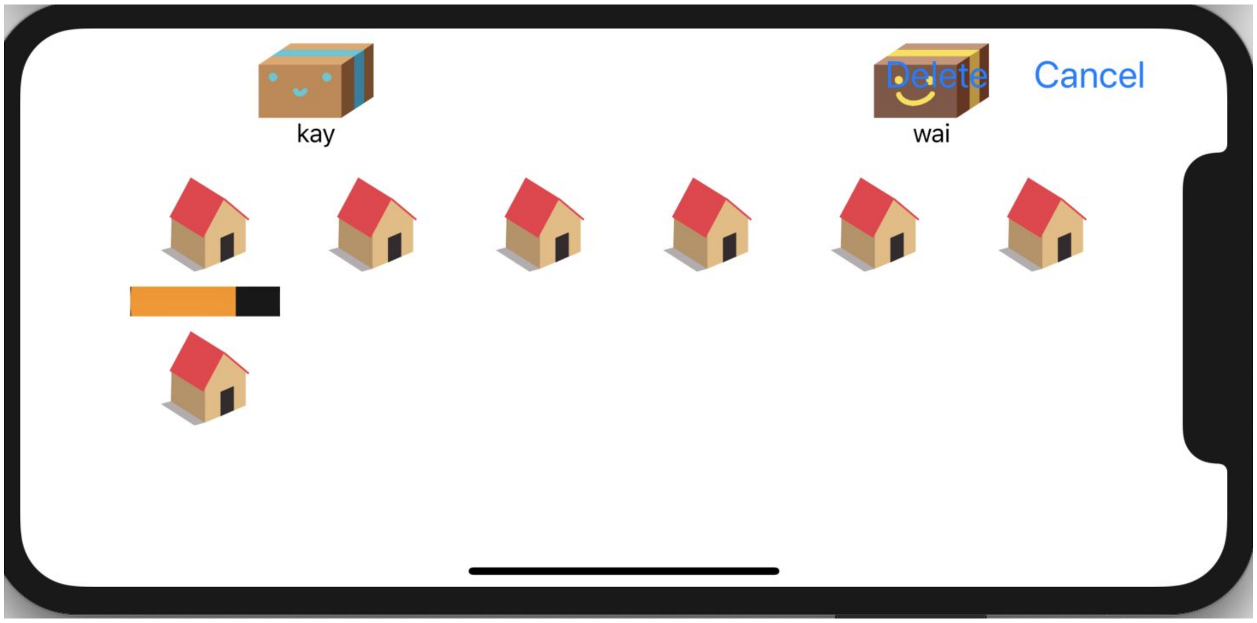


Figure 8: Delivery Screen

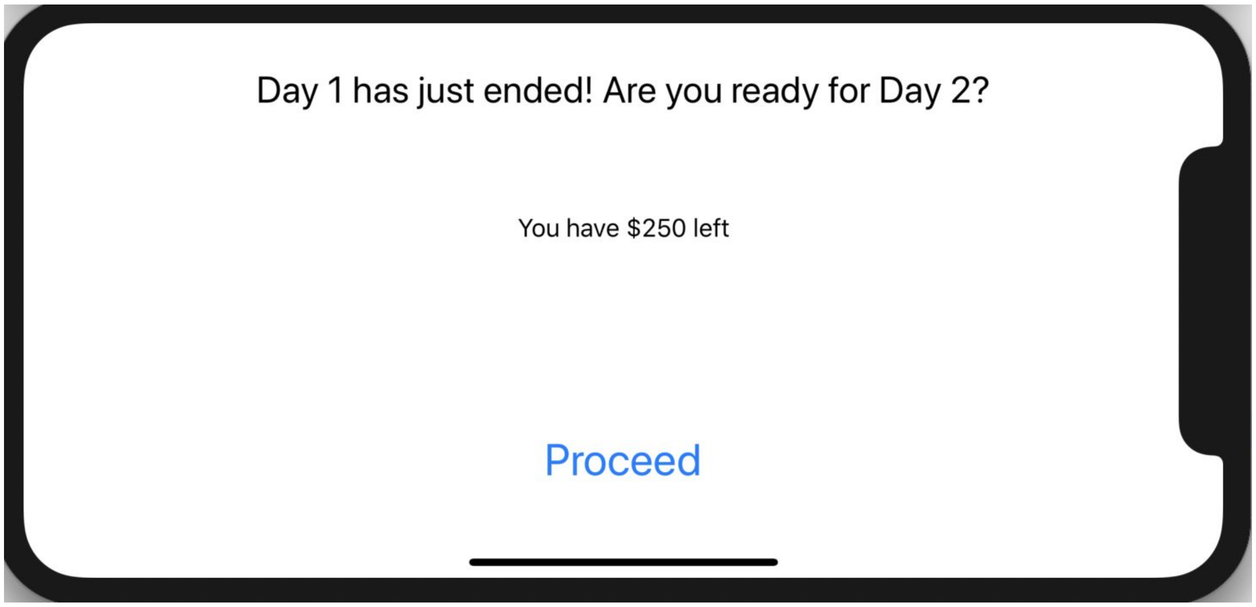
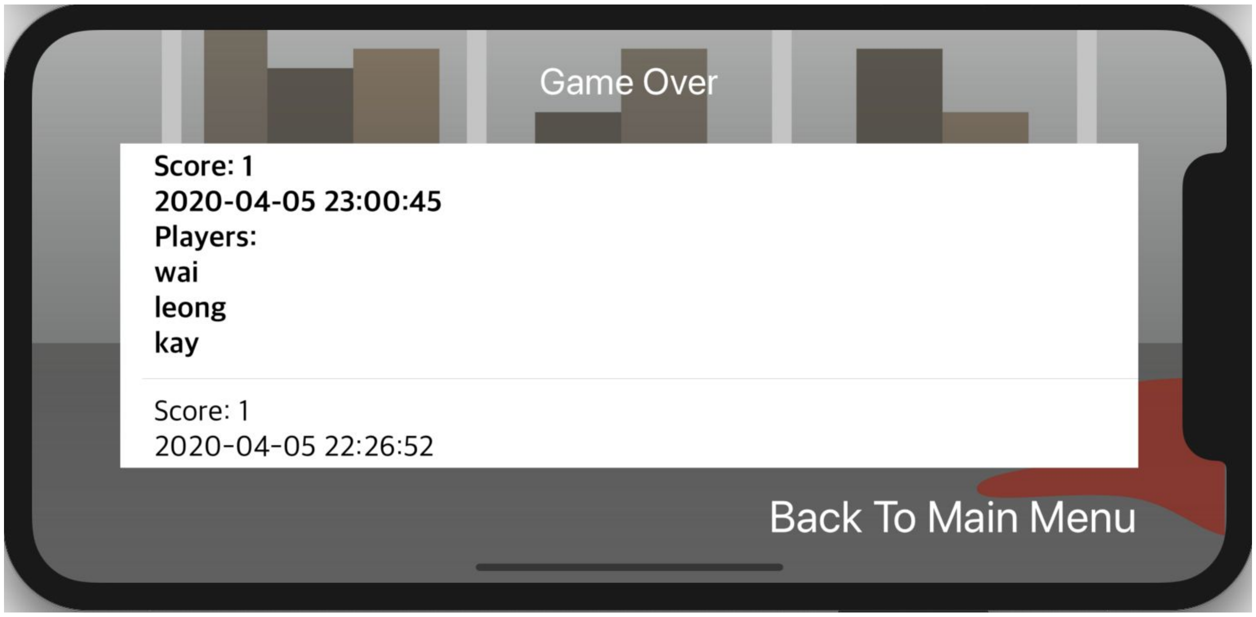


Figure 9: End round screen



Score: 1
2020-04-05 23:00:45
Players:
wai
leong
kay

Score: 1
2020-04-05 22:26:52

Figure 10: Game over screen