Intelligent Game Designer

Specification

Team Juliet

January 26, 2016

Contents

1 Introduction

1.1 Problem Definition

Game designers are often faced with the problem of creating exciting new game levels with the appropriate amount of difficulty. Human error in such a task can lead to users becoming bored (if the level is too easy or isn't aesthetically stimulating), or frustrated (if the level is simply too hard). Furthermore, the time investment of the game designers is a cost to the organisation/company responsible for the game.

In this project, we aim to automate the process of level design in order to save money, and improve the accuracy of difficulty approximation, through the use of simulated players, whilst ensuring that levels remain enjoyable and aesthetically pleasing. The game we will focus on is a match-3 tile based game which includes a subset of the features of Candy Crush Saga.

1.2 Background

We decided on implementing a tile-matching game from an early stage, since levels are easy to format while being non-trivial, and having simulated players of different skill levels is also manageable. Within this genre, we considered a wide range of games. After browsing King's catalogue of puzzle games, we decided that Candy Crush would be the best choice. This is because it has a simple level structure (the level defining parameters like the board shape and special tiles, with the placement of candies being generated by the game at run time) and has a range of features that are largely independent, meaning that we can choose to implement only a subset of the total game. Its popularity is an added bonus, as it has resulted in a dedicated wiki, from which we can easily obtain information on the mechanics.

We considered working from an open source game implementation. While we found some (such as Bejewelled_Java on GitHub) that implemented a rudimentary tile-matching game, we decided against using one of these as a template, choosing instead to create our own, possibly using one of these as a reference point. The issues with basing our level designers and simulated players on such a game are that it is very simplified, having no need for an actual level designer and that it would require us to read and understand the (not-necessarily well documented) code. Building our own interface will allow easier interaction between the game and our simulated/human players. We decided to implement a subset of the features of Candy Crush, instead of creating our own rule set, since coming up with game mechanics and balancing it ourselves would be too distracting from the main focus of the project.

1.3 The Game Rules

1.3.1 Overview

Candy Crush is a match-three game, where the player swaps candy positions to form patterns of matching candies, clearing them from the board. When candies are cleared from the board, candies fill in from above (if there are no candies above, additional unknown candies will be generated and introduced to the board). If a match is formed by the additional candies falling down, these will also be cleared, in what is known as a 'combo', scoring additional points. Although candies normally fall from the column above, if it is blocked for whatever reason, candies can fill in from adjacent columns as they fall. If an adjacent column needs filling, and cannot fill

itself, filling this will take priority.

Particular matches beyond a regular match of three will result in the formation of 'special candies', which have additional effects. When matches are made, the score increases, and the additional effects of special candies can also affect the score. There are six different regular candies, each of a different colour, although not all types have to be in a given level. The goal of a level is to fill the required criteria, within the allowed number of moves. Candy Crush has a wide variety of game modes, which require different criteria, and a subset of these modes will be included in our implementation. If the board is in a state such that no matches are possible, the pieces that are on the board will be shuffled.

The board is a 2D grid of candies. This board does not wrap-around, as in the candies can't be swapped across edges of the board. The board can be of a maximum size of 10 by 10, but importantly the board does not need to be rectangular. A level will have a set layout of places on the board in which candies can't exist (we call these 'layout blocks'). Candies can't be swapped through these places, but if there exist empty spots below a particular candy on the board the candy will drop down to fill that space even if there is a layout block between.

We will keep the basic mechanics of the Candy Crush game, where swapping candies can result in matches that would remove said candies. We plan on implementing subsets of the following features:

- Special candies
- Game modes
- Obstacles

1.3.2 Valid Matches

A move attempts to swap adjacent candies (not diagonally), accepting it only if it results in a valid match. The valid matches are listed below:

- A vertical or horizontal line of three candies of the same colour. This will increase the score by 60.
- A vertical or horizontal line of four candies. This increases the score by 120 and results in a special candy (described below).
- A vertical or horizontal line of five candies. This gives 200 points and results in a special candy.
- Special matches, such as a T, L or + shape (shown below). These will also score you 200, and create a different special candy. These can be in any rotation.

1.3.3 Special Candies

When a match is made that results in a special candy, the special candy will be placed in the location of the candy that was moved in to create the match. The special candies we will consider will be:

- Striped candy: Formed from a horizontal/vertical match of 4, of the colour of the match. When matched, these will clear the entire row/column respectively. The direction of the stripes on the candy formed is dependent on the match direction.
- Wrapped candy: Formed by the 'special' matches, of the colour of the match. Matching these first clears the 3x3 square of candies, centred on the wrapped candy, then activates again after the board has been filled in. This gives 540 points per activation.
- Colour bomb: Formed by the horizontal/vertical matches of 5. This has no associated colour. Whatever candy you swap the colour bomb with, all candies of that colour will be cleared. 3000 points will also be gained.

1.3.4 Game Modes

The game modes we intend to implement are:

- Score: Each level has a particular score you must achieve within a maximum number of turns. We will implement this feature.
- Jelly: Each level has fixed location on the board containing 'jelly' tiles (which candies can be freely swapped in and out of). The aim of the game is to clear all of these jellies (by making a match including a candy in the jelly location), in the allowed number of turns. An addition 1000 points are awarded for any match containing a jelly. We should implement this feature.
- Ingredients: There will be ingredients that are introduced to the top of the board at points in the game, and these must be moved to the bottom, in the turn limit. Every time an ingredient is brought down, 10,000 points are earned. We could implement this feature.

1.3.5 Obstacles

To add difficulty to levels, a number of obstacle tiles are introduced. These can't be moved, and do not allow candies to move through them. Of these, we will implement:

- Icing: These are cleared by making a match including a candy adjacent to the obstacle.
- Liquorice lock: These each contain a regular candy. They are cleared by making a match involving the candy inside the lock. When the match is made, the lock is removed, but the candy remains in place.

2 System Overview

2.1 Platform

We will implement our project as a desktop application written in Java. The reasoning behind this is that game developers are more likely to be working on laptops / desktop machines rather than mobile devices, and Java is a platform-independent language which we are all very familiar with.

2.2 Major System Components

2.2.1 Level Designer

The level designer will learn to create aesthetically pleasing levels for our specific game, of suitable difficulty, from feedback given by the simulated players and a set of constraints.

2.2.2 Simulated Players

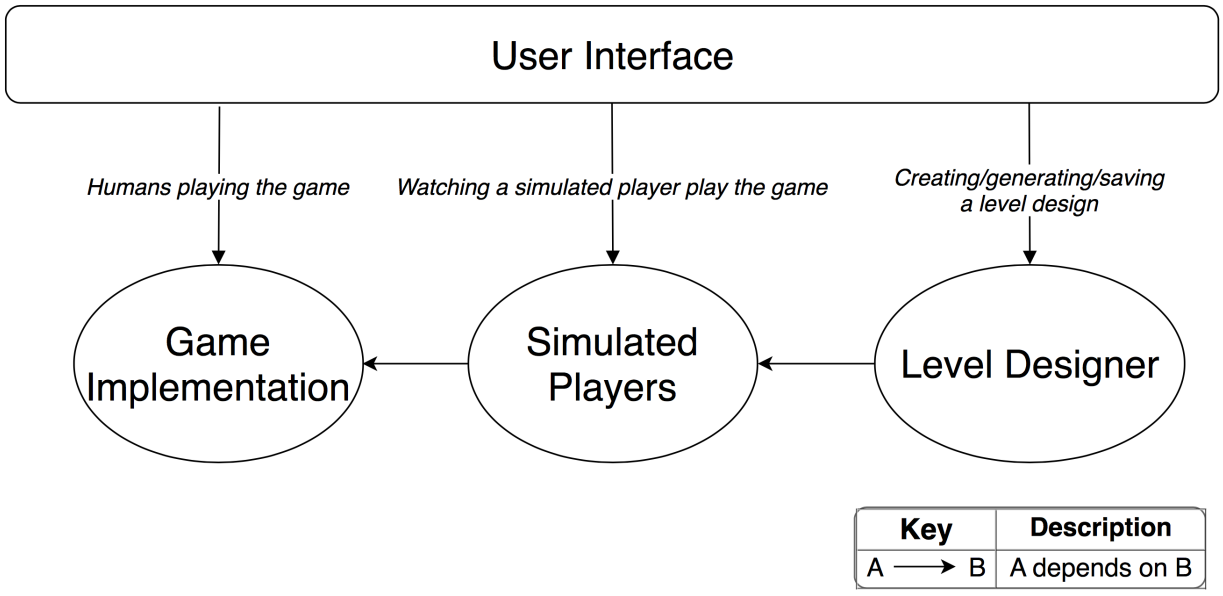
A population of simulated players created for our game of various levels of ability will be used to repeatedly play and attempt to complete levels to assess how difficult they are.

2.2.3 Game Implementation

In order for both humans and simulated players to evaluate the levels which have been generated, we need a module to actually provide the mechanics of the game, such as the state of the board after a given move.

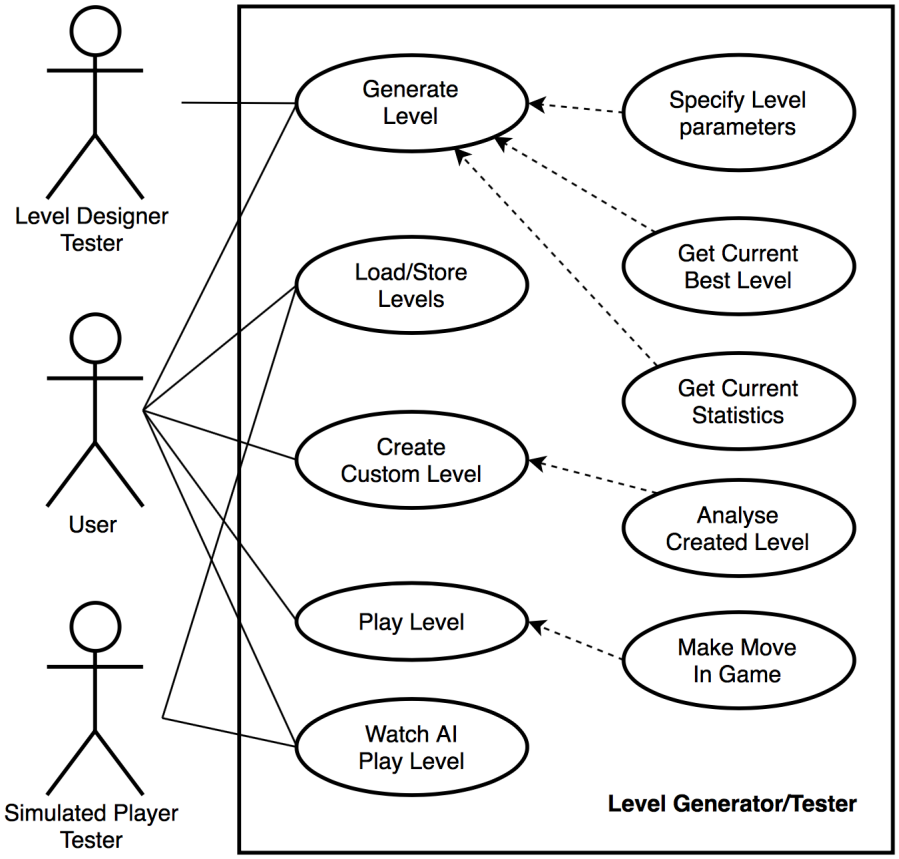
2.2.4 User Interface

The user interface will provide a variety of ways to interact with our system. Users will be able to play the game, watch simulated players play and most importantly, generate new level designs.



2.3 Target Users

We consider the target users of our product to be game designers/developers, who would use the product to create additional levels with reduced manpower, and game players who could use this as a level creating application to increase the longevity of their game and so their overall enjoyment.



2.4 Project Acceptance Criteria

The project will be deemed successful if the following system is thoroughly implemented, tested and documented:

- Users can, within the level designer interface, specify the type of level they would like generated and its target difficulty.
- With this information, the level designer should produce a finite series of suggested level designs, that are readable by our implementation of the game.
- The simulated players should be able to play the game, and capture a range of abilities similar to that of human users.
- Levels made by the level designer should be close to the difficulty that it is requested by the user.

3 Functional Requirements

3.1 Level Designer

The level designer's role is to generate a level that fits within a given set of parameters, for example it could generate a level close to a given difficulty. It would be best to create a level designer that internally has no understanding of how a specific game such as Candy Crush works, and instead generates levels randomly, then learning which features make a good level based on feedback from the simulated players and some sort of function to score the level's aesthetic. To do this, we will use a genetic algorithm, such as the Feasible-Infeasible Two-Population algorithm[1], that will maintain a population of level designs. The idea is that, with the use of mutation and crossover between generations, the population of solutions will converge to a set of designs that meet the criteria, but don't appear too idiosyncratic. Keeping the core level designing algorithm general (not game specific) means the level designer could in theory be used to generate levels for similar games by creating simulated players for that game instead.

We are aware that a generalised evolutionary approach may be ambitious, and may not give strong results. For this reason, we will also implement a more hard-coded backup solution which would work by adjusting the number of moves available (based on the performance of the simulated players) for a pseudo-arbitrary level design.

3.1.1 Aesthetic Fitness

Levels should not only conform to a given difficulty, but should also be aesthetically pleasing. This would mean they had some level of symmetry and each element of the level would be sensible (for example, there should not be single squares of blocks dotted around the level). Checking the aesthetics of a level is likely to be much quicker than simulating play, so levels should reach a certain aesthetic fitness before they are scored for difficulty fitness by the simulated players, which will probably be much more computationally expensive.

3.1.2 Design Constraints

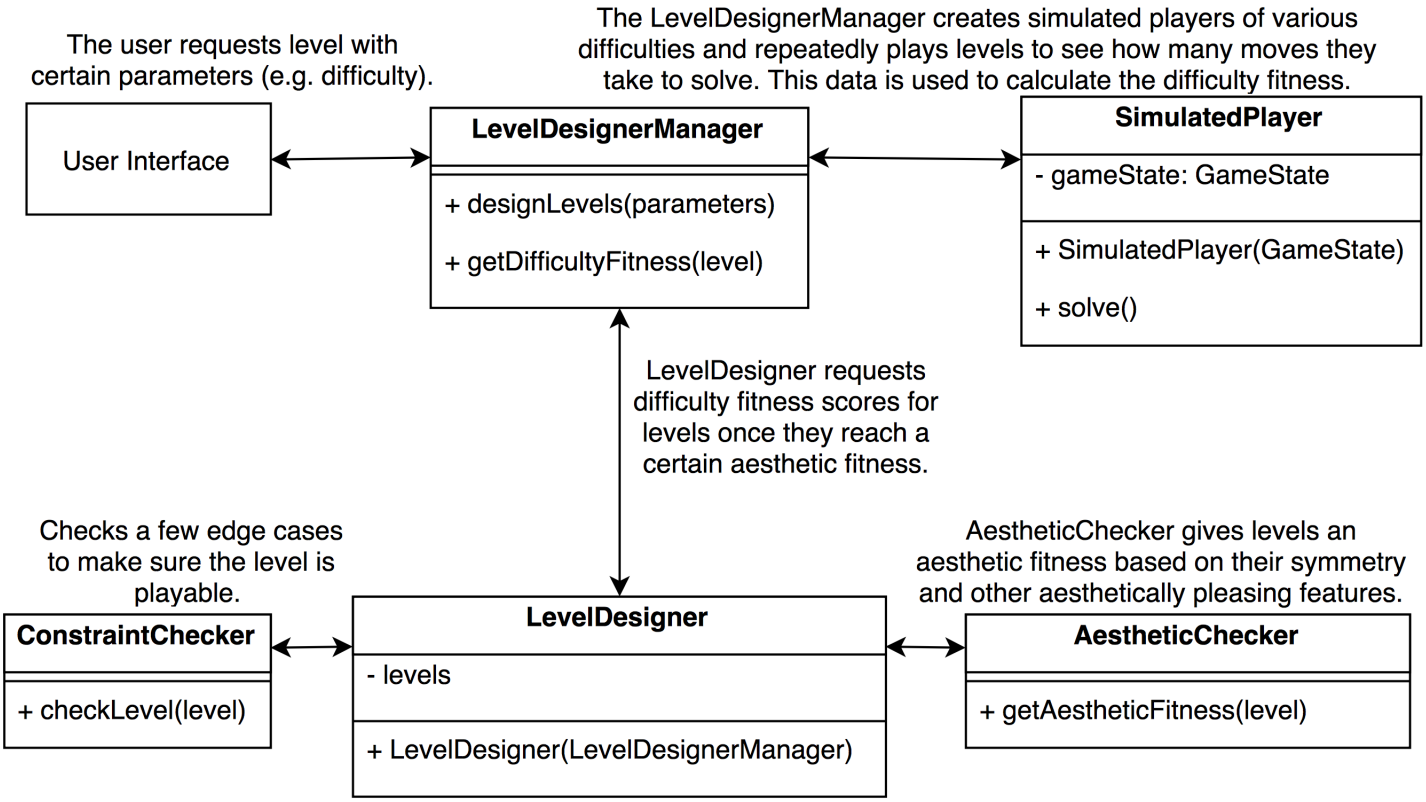
As well as checking the aesthetic of the board, we will need to ensure that the level is actually playable. This will involve checking for edge cases such as a blocked entrance across the top of the board. The level designer will check that these constraints are satisfied before requesting that the simulated players play the level.

3.1.3 Managing Simulated Players

Once a threshold aesthetic fitness is reached, the level designer will request a difficulty fitness from a manager class, which will in turn create a set of simulated players of various abilities. Each ability will need to play a level multiple times, since progress can vary a lot due to randomness. The manager class should run simulated players in multiple threads in order to compute the difficulty quicker. Once the simulated players are done, the difficulty fitness can be calculated using things like the number of moves it took to complete a level, and how the different ability levels did relative to each other. This score will be returned to the level designer and used when creating the next generation of levels.

[1] Sorenson, N & Pasquier, P - Towards a Generic Framework for Automated Video Game Level Creation

The structure of the level design component of the system is summarised below:



3.2 Simulated Players

3.2.1 Types of Player

As mentioned in section 1.3.4, we aim to deliver three kinds of level. Each level requires a slightly different approach to play, and thus it is likely that three different forms of simulated player will have to be implemented (though, much of the functionality may be shared between them). These players each have 3 distinct objectives:

Maximise the score (and minimise the number of moves taken to do so)
Players whose aim is to maximise their score within some number of moves will need to be created of varying ability.

Minimise the moves taken to remove all jelly blocks
Players whose aim is to minimise the number of moves taken to remove all jelly blocks will need to be created of varying ability.

Minimise the moves taken to sink ingredients to the bottom
Players whose aim is to minimise the number of moves taken to get ingredients to the bottom of the board will need to be created of varying ability.

3.2.2 Techniques & Approach

Several techniques will be adopted for creating these simulated players. Some will follow naively certain greedy approaches such as the finding the longest match or choosing any match with a jelly. Others will employ searching the state space using AI Techniques including the A* search algorithm and variants, with numerous different metric functions as well as learning approaches. A variety to the levels of ability will essentially be achieved by giving better players better search algorithms.

3.2.3 Assessment of Players

As part of the development of the simulated players, we will need some method of assessing their performance. There are basically two approaches for this:

- Find statistics for already existing Candy Crush Levels and based on the number of attempts to solve the game assess the simulated player.
- Create boards by hand and do comparative analysis. (When we have the Level Designer ready we can use levels produced by that).

If a Neural Network is employed for making the next move decision, then the feedback from assessment will be used to improve learning.

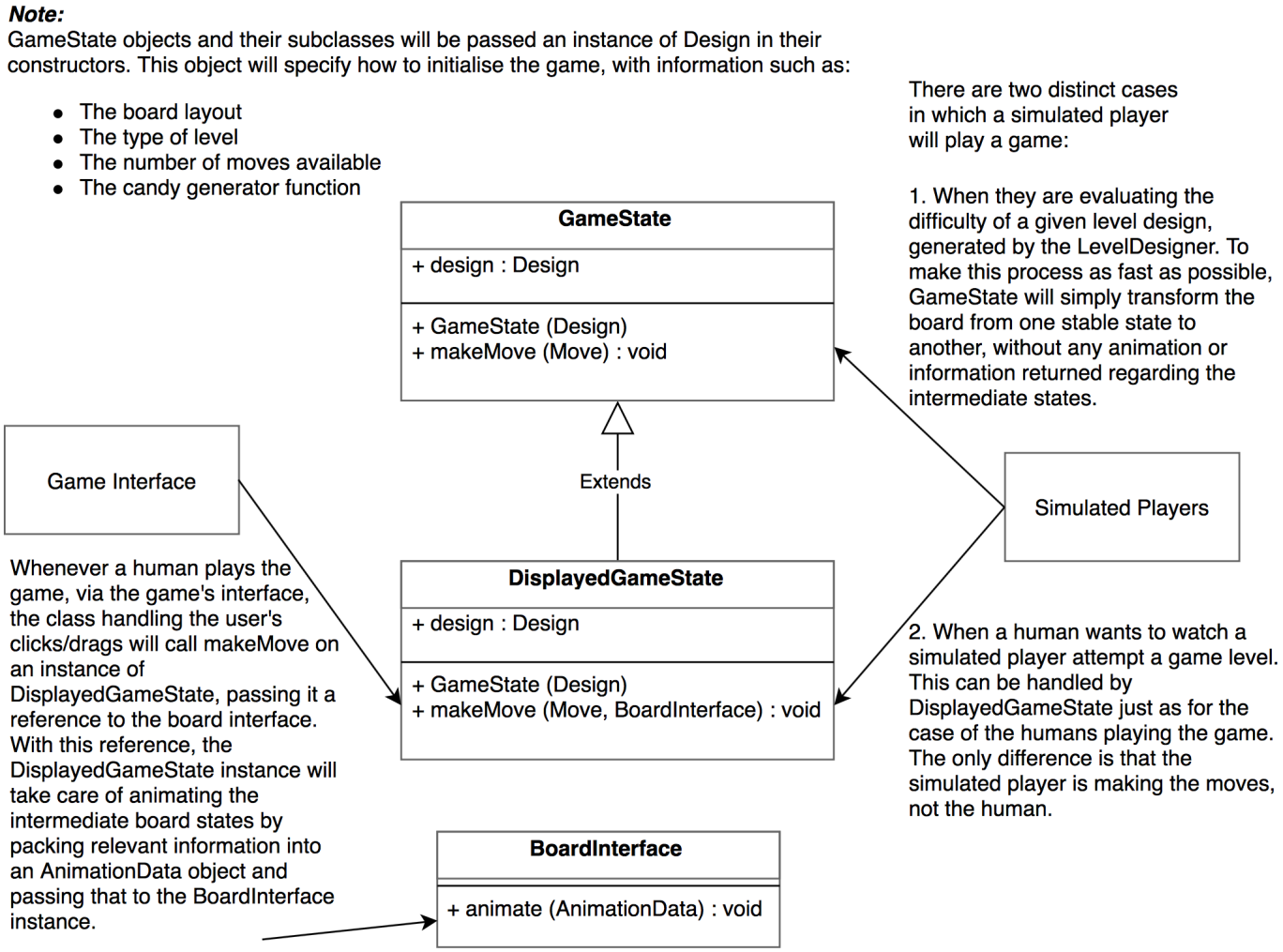
3.3 Game Implementation

The key idea behind separating the implementation of the game from the other system components is to reduce code duplication and increase modularisation. Both the human users and simulated players need to be able to play the game, so it makes sense to define a common interface through which this can be achieved.

A key issue we have to consider is the difference between playing the game where we animate the output in a GUI (such as when the human plays the game, or when they choose to watch a simulated player play it), and playing the game internally (such as when a simulated player tests the difficulty of a game for the level designer). In the former case, we have to handle the intermediate states of the game board, and must focus on the aesthetics of the animation. In the latter case, the priority is speed of interaction, and so any information about intermediate board states can be abstracted away from the player.

The game should generate candies at constant time per candy, so that it is playable, and does not lag.

The key components of the game implementation and an illustration of the above issues are highlighted in the diagram below:



3.4 User Interface

3.4.1 Main menu

Our product must have some sort of overarching display from which users can select and view particular features. All the following subcomponents that are implemented must be accessible through this menu.

3.4.2 Request level

To present our product to others, it is important to have a mechanic where we can request levels. This will be implemented with the use of a 'level requester' interface. From this, we will alter particular parameters of the desired level, such as its game mode, desired difficulty, and possibly the presence of particular blocks/candies.

Once a level is created, the interface gives feedback on the progress of the level designer, giving the estimated difficulty levels of the levels being designed. This will notify us once it has created

some levels of roughly the desired difficulty. It will then give us the opportunity to view these levels and test them out. Whilst generating the levels, there could be the option to poll for the current 'best' design. From there, there would be the option to either test it out, or to continue generating. When displaying the level, it will show a map of the level.

3.4.3 Displaying the game

There should be an interface from which you can view the game state (showing the candies and score etc.). The interface should allow for a human to make moves (most likely by clicking and dragging the tiles) and the game should respond accordingly. Time allowing, we could improve the interface with the inclusion of animations for swapping, falling, etc.

3.4.4 Simulated player viewer

Time allowing, we should implement the option to visualise how the simulated players are behaving. The user should be able to select a simulated player and a level for it to attempt. The game window will appear the same as when a human plays, but instead of waiting for a human to input a move, the simulated player will choose the moves. There could be an option to select the play style, between prompting you to click 'next' before continuing to the next move, or to automatically play the moves (possibly at a selectable speed).

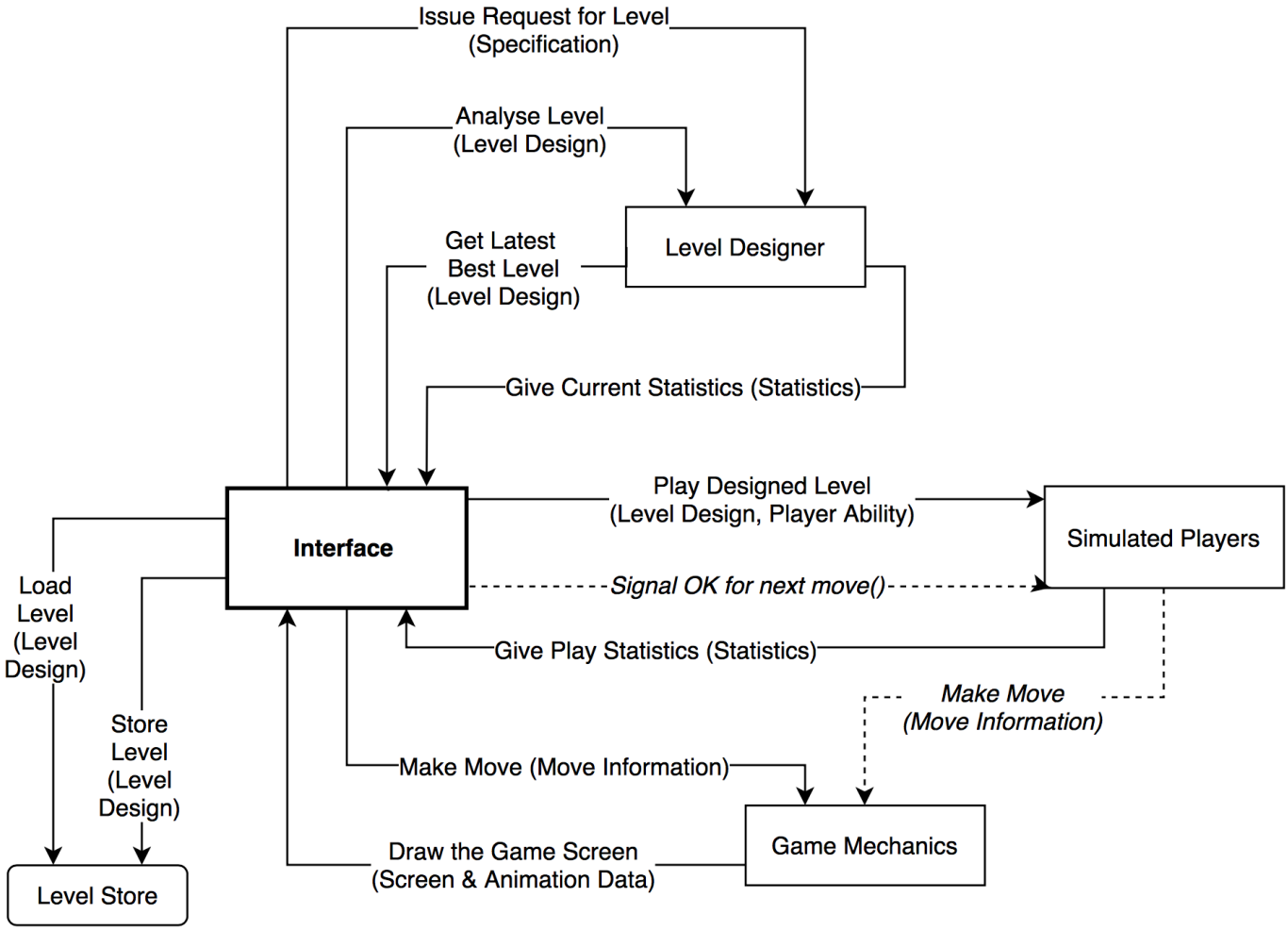
3.4.5 Level creator

If there is additional time, we could implement a simple level creator, where a human can use the designed interface to create levels (of the same format as the level designer), for instance by clicking (and dragging) on tiles to change their type. There should be an option to change the game board's size (within parameters), the game mode, the shape of the board and the presence/location of special tiles. These levels could be sent to the simulated players, and so analysed to determine their difficulty.

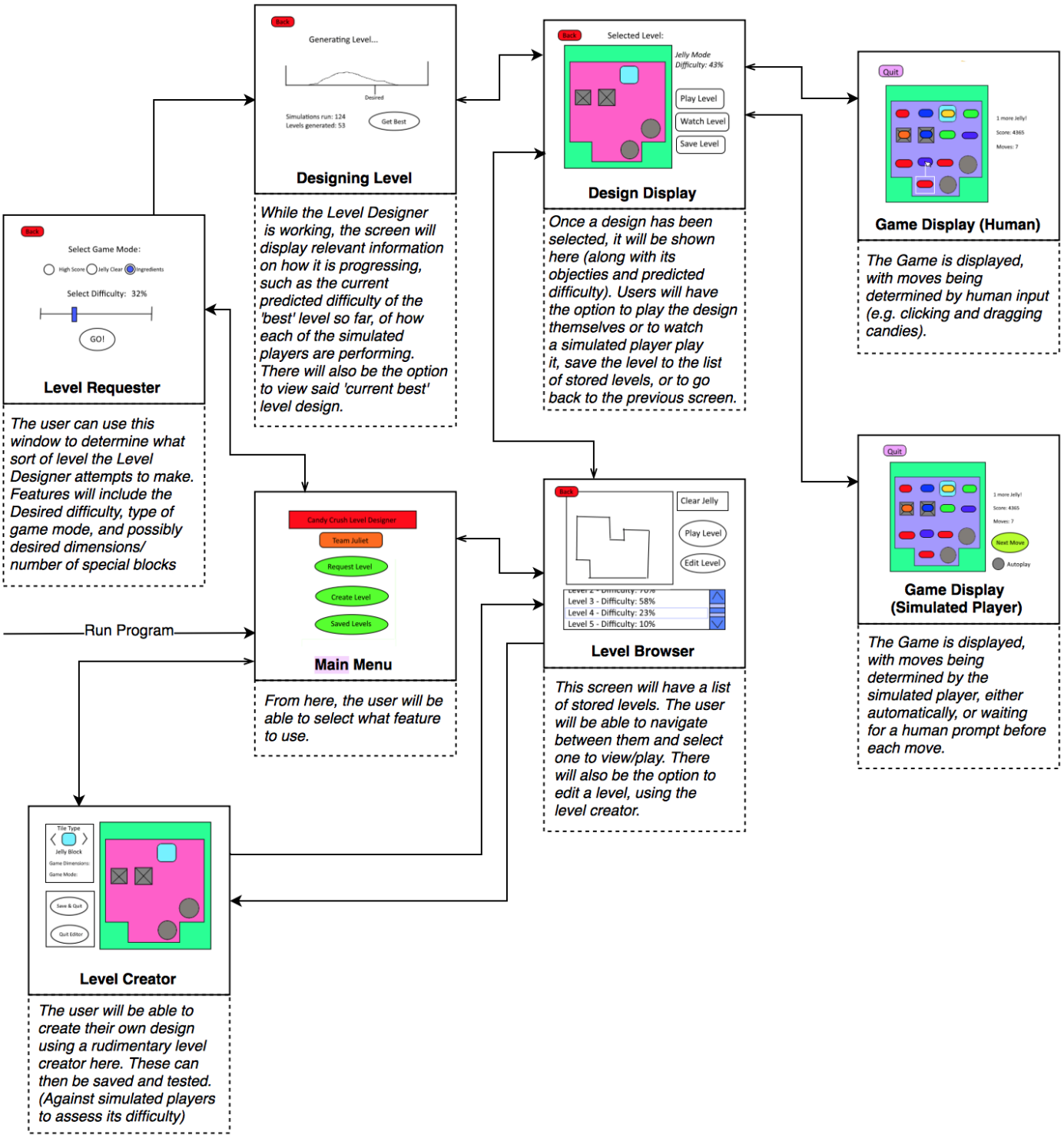
3.4.6 Example levels

As well as generating levels 'on the fly', we should have a list of ready-made levels, that users can view and test out, or watch a simulated player test out. Along with the level, there should be adjacent information on how difficult the level is (for instance saying that 40% of simulated players managed to complete it). There could be an additional folder to which levels created by the player in the level creator, or levels made by the 'on the fly' level generator can be saved and so viewed again.

Below is a summary of the interaction between the user interface and the rest of the system:



Here is an overview of the navigation of the interface:



3.5 Constraints

3.5.1 Time constraints

Due to the nature of this group project, we will have limited time available to produce a non-trivial amount of work. The time available, and how effectively we manage our time will determine how many of the additional features we are able to implement.

3.5.2 Space constraints

We need to consider space constraints, since the level designer will be running many simulated players, each of which will have their own copy of the board state. As an upper bound, we will ensure that our system doesn't occupy more than 2GB of RAM.

3.5.3 Performance constraints

Due to the nature of this project, some of the areas of interest may be very computationally expensive, and for speed reasons, we aim to have many perform in parallel. For example, we wish to minimise the time taken for each simulated player to complete an entire game (to return its results to the level generator), and we want as many of these players testing, as frequently as possible (potentially concurrently). The learning/selection involved in creating improved levels in the 'Level Designer' may equally be computationally expensive. Depending on whether we wish the designer to generate levels in somewhat real-time, we may set a very high requirement on the amount necessary to do in a short amount of time while the product is running. Therefore, the performance of our computers, and our ability to write efficient code will affect how fast our product functions.

4 Management Strategy

4.1 Group Organisation

4.1.1 Group Meetings

The group will meet on Tuesdays and Thursdays from 11:00 to 13:00, as allocated by the group project organisers. As and when we feel it is required, we will organise additional meetings either in person or via Facebook/Slack.

4.1.2 Group Structure

As a group, we have decided to stick with a flat management structure, and to work without appointing a project manager. Devan will serve as the point of contact, and will provide all of the project deliverables.

4.1.3 Communication

During the implementation phase, team communication will be achieved using Slack. This software allows for multiple channels of conversation, each corresponding to different components of the system. It is also less distracting than Facebook.

4.2 Technical Management

4.2.1 Version Control

We will use Git for version control, and GitHub to host our project. Once implementation is fully underway, the master branch will be reserved for production-ready code. Since the project consists of 4 distinguishable components (the game implementation, the simulated players, the user interface and the level designer), we will also have a branch corresponding to each one. Code on feature branches will be peer-reviewed before merging to master.

4.2.2 Arrangement of Workload

We intend to use Trello, a kanban-board like interface, to organise what features are needed by other members of the team, in order to effectively prioritise work and ensure everyone has something they are capable of doing, and are aware of what needs doing. This will allow for clearer communication of objectives and requirements within the group.

4.2.3 Coding Standards

In order to enforce consistency across the codebase, we will follow the coding standards set out for the Java ticks[2], since these are standards everyone is already familiar with.

4.2.4 Testing

All code should be tested frequently as it is developed. A series of unit tests will be constructed in a separate directory using the annotations and assertions provided by Java's JUnit. We may also consider using a continuous integration platform such as TravisCI.

[2] Java Tick coding standards

Whenever one part of the team depends on a module written by another part of the team, they will write black box tests to comply to the interface agreed between the two teams.

5 Documentation

5.1 Technical Documentation

For the implementation phase to be as efficient as possible, we will need to ensure that all code is readable/maintainable by every member of the team. Therefore, each team member will comment their code as they write it using JavaDoc-style comments, allowing for class documentation to be automatically generated without any exhaustive effort.

5.2 User Documentation

Our user interface will be designed to be as intuitive as possible, therefore a user manual should be unnecessary. The only component of the system for which users may need an explanation is when they want to play the game. For this, we will provide a concise textual description of how to play the game in an unobtrusive pop-up that the user can choose to display (e.g. a 'How to Play' button).