



Department of Computing Sciences

**Advanced Programming 3.2**  
**(WRAP302/WRPV302)**

**Assignment List**

**September 2020**

*Please read this document carefully and keep it in a safe place — it contains important information that you will need to refer to throughout the module.*

---

## Table of Contents

About .....	3
Schedule .....	3
<b>ASSIGNMENT 1</b>	
Topics .....	4
Task 1: Android “Hello World” .....	4
Task 2: Android SOS .....	4
<b>ASSIGNMENT 2</b>	
Topics .....	5
Task 1: Pub/Sub Broker .....	5
Task 2: Pub/Sub SOS .....	6
Task 3: Contacts .....	7
<b>ASSIGNMENT 3</b>	
Topics .....	8
Task 1: Sudoku Generator .....	8
Task 2: Threaded Generation of Sudoku Boards .....	9
<b>ASSIGNMENT 4</b>	
Topics .....	11
Task 1: Networked Pub/Sub Broker .....	11
Task 2: Cuppa Java .....	12
<b>CAPSTONE ASSIGNMENT</b>	
<b><i>Simplified</i></b> Segrada Rules .....	16
Minimum Functionality of the Apps .....	17
Bonus Marks .....	20
Preliminary Mark Allocation .....	21
Work to a Schedule .....	22

# WRPV302 Assignment List

## About

This document contains *all* the assignments, except for the 48 hour assignment, that you will be required to complete this year. The 48 hour assignment will be made available to you on the Wednesday of Week 5.

The due dates are provided for each assignment; be sure to submit your assignment in a zipped file on the Learn site before the due date and time. You are also required to *peer assess* a selection of assignments submitted by your peers for Assignments 1-4.

Remember to complete *all* the Activities, as *all* of them count towards the Final Mark for this Module.

**Be sure to read the Capstone Assignment and start working immediately on it!**

## Schedule

The schedule of assignments, peer assessments is given below:

Week	Dates	Activity
1	28 Sep - 2 Oct	Assignment 1
2	5-9 Oct	Assignment 1 submitted by Tuesday, peer assess by Friday
3	12-16 Oct	Assignment 2
4	19-23 Oct	Work on Assignment 2
5	26-30 Oct	Assignment 2 submitted by Tuesday, peer assess by Friday
		<b>48H Assignment (Wednesday &amp; Thursday)</b>
6	2-6 Nov	Assignment 3
7	9-13 Nov	Assignment 3 submitted by Tuesday, peer assess by Friday
8	16-20 Nov	Assignment 4
9	23-27 Nov	Assignment 4 submitted by Tuesday, peer assess by Friday
10	30 Nov - 4 Dec	Work on Capstone Assignment
11	7-11 Dec	Work on Capstone Assignment
12	14-18 Dec	<b>Capstone Assignment (due Tuesday)</b> <b>Reflective Report (due Thursday)</b>

# Assignment 1

## Topics

- Introductory Android
- Activities

## Task 1: Android “Hello World”

Complete and submit the “Build your first app” tutorial on the Android Developer site (<https://developer.android.com/training/basics/firstapp/>). Be sure to deploy it to the emulator and test it.

## Task 2: Android SOS

In WRPV301, you made an SOS game using JavaFX. An excerpt from that assignment is given below:

### Game Rules:

The game of SOS , is played between two players.

The game begins play on a square grid of at least 3×3 squares. For this task, play should be on a 5×5 board.

Each player takes a turn placing an “S” or an “O” in an open square.

If the newly placed letter completes an “SOS” sequence vertically, diagonally or horizontally, that player scores one point and gets another turn (until placing a letter does not complete a sequence, in which case it is the other player’s turn).

Once all the empty squares have a letter in them, the winner is the player with the highest score. If the scores are the same, then the game is a draw.

S	O	S		
O	O			S
S				
		S	O	S
			O	S

You are now required to write an Android app that will allow two players to play the game on the same device, with turns alternating between the two players. Use whatever Views and interaction methods you think most appropriate to implement the game. Be sure to let players know what their scores are at all times and when they need to pass the device to the other player so that they may take their turn. Once the game is over, be sure to clearly indicate who the winner is and what the scores were.

# Assignment 2

## Topics

- Multiple Activities
- Views

## Task 1: Pub/Sub Broker

Last semester, the SOLID<sup>1</sup> principles for OOP were discussed. One of the major issues related to the *de-coupling* of objects/classes so that they did not depend so much on one another. Highly coupled classes often:

- require complex setting up and communication between them;
- require full knowledge of the other class (e.g. its methods being called); and
- have an issue with *scalability*, specifically when wanting addition classes (potentially unknown) to respond to the same messages.

Topic Name	Subscribers
a	S <sub>1</sub> , S <sub>3</sub>
b	S <sub>2</sub>
c	S <sub>2</sub> , S <sub>3</sub>



One way of decoupling classes is to use a messaging system (event notification system) where a *publisher* may *publish* a message about a particular *topic* and if *subscribers* are *subscribed* to receive messages about *that* topic, then they do. The important thing, though, is that neither the publishers, **nor** the subscribers know about one another (i.e. they do not have *references* to the publisher or receiver). Instead, an intermediary, known as the broker<sup>2</sup>, handles message passing.

When a subscriber is initialised, it registers itself with the broker, indicating which topics it is interested in receiving messages about. For example, when s<sub>2</sub> is initialised, it registers itself to receive messages about topics *b* and *c*.

When a publisher wishes to send a message about a particular topic, it sends the message to the broker, who then forwards the message to *all* subscribers that registered an interest in the topic. For example, when publisher p<sub>1</sub> sends a message about topic *a* to the broker, the broker forwards the

<sup>1</sup> Check out <https://en.wikipedia.org/wiki/SOLID> if you cannot remember.

<sup>2</sup> Note that there is only *one* broker, but there can be many publishers and subscribers. All publishers and subscribers use the *same* broker.

message to subscribers  $s_1$  and  $s_3$ . Multiple publishers can send a message about a particular topic ( $p_1$  and  $p_3$  can both send messages about topic  $a$ ) and multiple subscribers can register to receive messages about a topic (e.g.  $s_2$  and  $s_3$  can receive messages about topic  $c$ ).

Subscribers can register and deregister themselves at *any* time. The pub/sub broker should be easily accessible to *any* object, so it is often a singleton<sup>3</sup> or a static class.

A message will typically contain the publisher (who published it), the topic and optional data containing extra information about the message.

Using SOLID principles design and implement a simple Publish Subscribe architecture in **Java** (i.e. do not use Android-only classes) that can be reused in later assignments (including this one). In other words, be sure to use classes, interfaces, groupings of methods, generics, Lambdas etc. in a manner that is decoupled and cohesive.

Demonstrate that your implementation is working correctly by completing Task 2.

## Task 2: Pub/Sub SOS

In the previous assignment, you implemented the SOS game (using standard Android event handling). Now modify this implementation to make use of the Pub/Sub Broker<sup>4</sup>.

All the cells on the board should now generate *messages* indicating that the user tapped them. Instead of the event being handled directly inside this event, rather have another class (the controller) deal with it.

In addition to this, other types of messages should be generated, including:

- a new game has started or ended;
- a player's turn starts or ends;
- a letter has been placed in a cell (including *which* letter); and
- when an SOS sequence has been completed, etc.

Make use of these events to update the scores, display the "player turn" and "game over" messages.

In addition to this, create player specific stats calculators that monitor the events and calculate:

- how many turns each player had;
- how many O's they placed;
- how many S's they placed;
- how many SOS sequences they completed; and
- the highest number of SOS sequences they completed in a single turn.

You *must not* modify the game logic code to calculate these stats. Simply monitor the messages to do so, i.e. it should be possible to easily add new stats calculators *without* needing to modify existing game logic code (O of SOLID).

---

<sup>3</sup> Read [https://en.wikipedia.org/wiki/Singleton\\_pattern](https://en.wikipedia.org/wiki/Singleton_pattern) for more details.

<sup>4</sup> The changes required for *this* task should not be a lot. If you find yourself writing a *lot* of code, it probably means that you need to rethink what you are doing. This task is about thinking in an event-driven manner, which might take a bit of practise.

The stats calculated for each player should be displayed during game play and updated *as soon as* a stat changes<sup>5</sup>.

### Task 3: Contacts

For this task, you are required to create a simple Android app that will allow a user to keep and manage their contacts and dial them. A contact contains an avatar image, the name and contact number for a person or business. The contacts pictures can be found in the “Assignment 2 Files” folder.



**Figure 1: Contact Card**

The app must have two Activities, namely the Contacts List and Contact Card Activities.

The Contacts List Activity is the main Activity and displays a vertical staggered grid of contact cards, which can be scrolled up and down. A contact card should look similar to that seen in Figure 1. Tapping on a contact card should display an edit, message and dial button on the card.

Tapping the:

- edit button takes the user to the Contact Activity and displays more details about the selected contact;
- message button will take user to the SMS messaging Android screen; and
- dial button will dial the number of the contact.

At the bottom of the Contacts List Activity is a floating action button. Clicking on this button adds a new contact to the list and takes the user to the Contact Activity to enter the details;

The Contact Activity displays the details about a specific contact, including the name, number and an avatar image. It must be possible to change the avatar image used, the name and number of the user. It should be possible to return to the Contacts List Activity once the required changes have been made *and* these changes should be reflected in the Contacts List Activity. There should also be a floating action button at the bottom of the Activity. Tapping it will dial the contact’s number.

All the contacts should be persisted to a file.

Populate the initial contact list at start up with at least 10 contacts.

---

<sup>5</sup> Another message? Hint, hint...

# Assignment 3

## Topics

- Collections
- Threads

## Task 1: Sudoku Generator

For this task, you are required to write a Sudoku generator using the *Wave Collapse Function* algorithm. You *must* make use of the streaming API wherever possible. Hint: consider the data structure(s) you use carefully.

### The Rules of Sudoku

The classic Sudoku game involves a grid of 81 boxes. The grid is divided into nine blocks, each containing nine boxes.

The rules of the game are simple:

- each of the nine blocks has to contain all the numbers 1-9 within its boxes; and
- each number can only appear once in a row, column or block.

One way to randomly generate a valid Sudoku grid is using the *Wave Function Collapse* algorithm<sup>6</sup> from Quantum Mechanics. This algorithm can be used to generate many things, including a Sudoku grid.

The algorithm works by initially having each box contain a set of *all* possible valid values, e.g. 1, 2, 3, 4, 5, 6, 7, 8, 9. So initially the grid contains all possible Sudoku grids that can *ever* be generated.

Then using a process of elimination, the possible Sudoku grids are reduced until finally a *single* particular Sudoku board is generated.

The process of reducing the possibilities works as follows:

- pick a box, *b*, with the least number of possibilities remaining, then randomly pick one of the possibilities, *p*, in that box. This becomes fact for box *b* (i.e. there are no other possibilities for it);
- remove *p* from the possibilities of all boxes in the same row, column and block as *b*.

Repeat this process until all the boxes contain only facts.

The table below shows an example of what a grid could look like after a few reduction steps. Box A4 contains all possibilities (1-9). Boxes B2, B2 and D3 have facts (2, 3 and 6 respectively). Based on these facts, the possibilities of other boxes have been reduced. For example box D2 has possibilities 1, 4, 5,

		3	9			1		
5		1				4		
9		7			5			
6	2	5	3			7		
7			7				8	
8	3		1		9			
	9		2	6			7	
4				3	6	1		

<sup>6</sup> Another example of the Wave Collapse Function being used. Also includes a nice description of the general algorithm used for creating Wedding seating charts: <https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/>



7, 8 and 9. This is because the possibilities 2 and 3 were removed due to being in the same row as boxes B2 and C2 and the possibility of 6 being removed because it is in the same block as D3.

	A	B	C	D	E	F
1	1,4,5,6,7,8,9	1,4,5,6,7,8,9	1,4,5,6,7,8,9	1,2,3,4,5,7,8,9	1,2,3,4,5,7,8,9	1,2,3,4,5,7,8,9
2	1,4,5,6,7,8,9	<b>2</b>	<b>3</b>	1,4,5,7,8,9	1,4,5,7,8,9	1,4,5,7,8,9
3	1,4,5,6,7,8,9	1,4,5,6,7,8,9	1,4,5,6,7,8,9	<b>6</b>	1,2,3,4,5,7,8,9	1,2,3,4,5,7,8,9
4	1,2,3,4,5,6,7,8,9	1,3,4,5,6,7,8,9	1,2,4,5,6,7,8,9	1,2,3,4,5,7,8,9	1,2,3,4,5,6,7,8,9	1,2,3,4,5,6,7,8,9

When picking a new box, one of boxes D2, E2 or F2 will be selected as they all have 6 possibilities left. If box E2 was selected, then one of the possibilities 1, 4, 5, 7, 8 or 9 would be randomly chosen. If, for example, 9 was chosen, then E2 would hold the fact 9 and all the 9s would be removed from the possibilities of the boxes in row 2, column E and the block in which E2 resides namely (D1, E1, F1, D2, F2, E3 and F3).

Should there ever exist a box with 0 possibilities, then the grid is invalid<sup>7</sup>. Display an error message and try again.

Write two classes named `SudokuGenerator` and `SudokuGrid`.

The `SudokuGrid` class represents a board and the values placed in it. It must provide functionality to set, retrieve and clear box values, as well as a method to determine whether placing a value in a given box is valid, invalid due to row conflict, invalid due to column conflict or invalid due to block conflict. Note that it is possible to be invalid for any combination of the three reasons (e.g. invalid due to row conflict AND invalid due to column conflict).

The `SudokuGenerator` class is responsible for generating a valid complete `SudokuGrid` using the Wave Function Collapse algorithm described above.

You must make appropriate use of the Java Collection Framework (collections, algorithms, streaming API, etc.) when completing this task.

Demonstrate that your implementation works correctly by displaying generated boards after the user presses “enter” and display the number of invalid boxes on the board (using the method you wrote previously).

## Task 2: Threaded Generation of Sudoku Boards

In Task 1, you wrote an algorithm to generate Sudoku game boards (full solutions). Now you need to write a program that uses threads to do the following:

- generate 500,000<sup>8</sup> *unique* Sudoku game boards, allocate each a unique number and save these to a file – as quickly as possible.

Game boards are numbered 0, 1, 2, 3... in the order that they are generated. A game board is saved in a text file named 0.txt 1.txt, 2.txt, etc. which simply contains all the numbers in each box of the game board.

<sup>7</sup> Not 100% sure if this case can ever occur though, but just in case.

<sup>8</sup> Or some other large number...

A game board is considered the same regardless of how it is rotated, flipped along the rows, flipped along the columns, or any combination therefore<sup>9</sup>. So a game board is considered the *same* as a mirrored version of itself, etc.

You will need to have threads to generate game boards, some to check for uniqueness and another for saving unique game boards to file.

---

<sup>9</sup> Note that the representation of your data can go a *long* way to how efficiently the comparisons, rotations, mirroring, etc. can be done. Think about this, unless you feel like sitting for months waiting to get all 500,000 game boards.

# Assignment 4

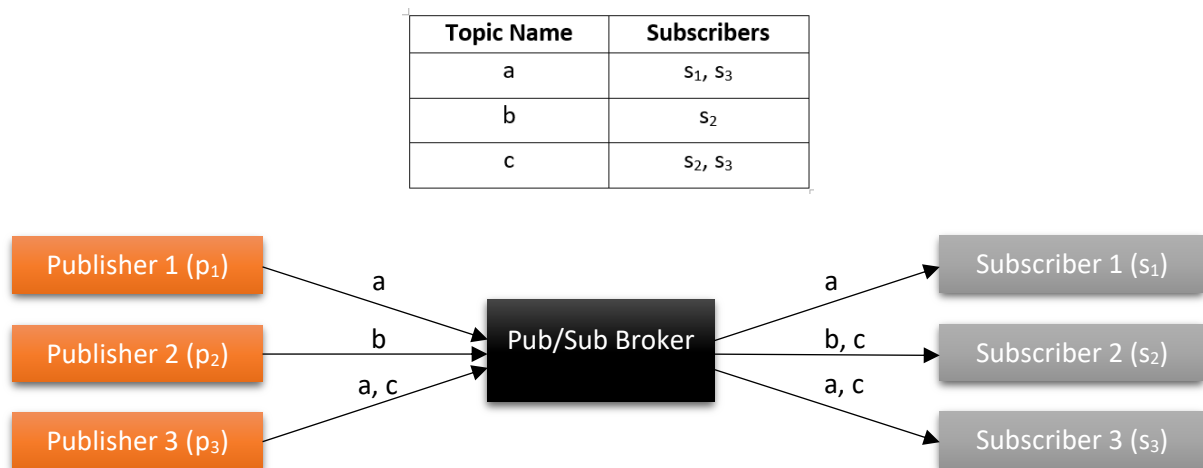
## Topics

- Networking
- Threads

## Task 1: Networked Pub/Sub Broker

In Assignment 2, you were asked to implement a Publish/Subscribe Broker pattern where Subscribers subscribe to receive messages about a specific topic via the Broker. Publishers can publish messages about specific topics, which are then forwarded via the Broker to all subscribers subscribed to those topics. Subscribers can unsubscribe from a topic at any time and will then not receive any further notifications.

A very important aspect of this was that Publishers and Subscribers did not know about each other explicitly. Instead all communication was handled via a single, shared Broker object. In your assignments, the Broker, Publishers and Subscribers were all within the *same* application and on the *same* device.



For this task you are to create a *thread-safe networked* version of the Publish/Subscribe Broker. In this scenario, the Broker is hosted on a Server and the Publishers and Subscribers run in different client applications and possibly different devices. The Publishers and Subscribers communicate with each other indirectly via the Broker. Communication with the Broker is handled using a TCP/IP connection<sup>10</sup>.

You have been provided with an implementation of a Publish/Subscribe Broker class and associated interface where everything was running within the same application in the “Assignment 4 Files” folder, which will become available from week 6. Modify this implementation, or your own from Assignment 2, to allow for a networked version of the pattern.

<sup>10</sup> Networked messages in this task is very similar, in many ways, to the concepts of web services, remote procedure calls, etc. used elsewhere.

Note that your implementation must not interfere with the normal execution of an application, e.g. it must not block an Android app's UI thread. Also a client should not "block" while waiting for publications, it should be possible to publish anything at any time (and receive anything at any time).

For this implementation, when a Client connects to the Broker Server, it is allocated a unique integer ID. This ID is communicated with the Client. The ID is used as the "publisher" in a message.

The Server must be a Java console-based application. The Client side classes must be able to run in either regular Java applications or Android apps, i.e. don't use any Android only or Java only classes.

Make appropriate use of the Java Collection Framework, Lambdas, threads, sockets, etc. in your solution.

## Task 2: Cuppa Java

Cuppa-Java™ is a coffee bar where customers can order specialty coffees and confectionaries. Instead of customers standing in a queue and ordering what they want, the owners of the shop would like to allow customers to be able to place their orders using their mobile device (making it possible to place an order while sitting in the comfy chairs and chatting with their friends or even before the customer enters the coffee bar). In order to test the viability of such a system, the owners have asked you to create a simple prototype Android app to demonstrate the menu, order creation and order collection parts of the system (the final system will handle payments, deliveries, specials, news, etc. as well).

You are to create a simple console-based Java Server, as well as an Android Client, that work together to act as a simple order placing and collection system.

You may use your implement in Task 1 to assist you if you wish.

The Server *accepts orders* from a Client app (via a network connection), will log the orders to a local text file<sup>11</sup> (including date, time, telephone number, order details, total cost), and after a random period of time (between 2-5 minutes<sup>12</sup>), will send an "*order complete*" message back to the Client, indicating that the order is ready to be collected. The Server will resend the "*order complete*" notification to the Client every 2 minutes, until the Client acknowledges that the *order has been collected*. The Server will also, on request from a Client, *send the latest menu details* to it (to be displayed to a user). As network requests are processed on the Server, information should be displayed on the console to see what is happening.

Below are mock screen shots of the activities that need to be created Client Android app. The dark brown colour used is #CC9933 and the light brown/yellow colour is #FFCC33. You may use the View, found in the Advanced section, to make the yellow bar behind the Java logo of the Main Activity. The icons and images may be found in the "Assignment 4 Files" folder.

---

<sup>11</sup> Using a "nice" format, such as XML or json is a good idea.

<sup>12</sup> For testing purposes, you may shorten the times mentioned here.



When the app starts, the Main activity is displayed. When no order has been created, the activity only displays the Cuppa-Java logo and the Create/Modify Order icon centred horizontally. The Order Cost and Send Order views are not displayed.



Clicking on the Create/Modify Order icon will take the user to the Menu activity<sup>13</sup>. If an order has been created, then the Order Cost is displayed, as well as the Send Order icon. Clicking on the Send Order icon will “send” the order to the Server, display a toast stating that the order has been sent and then returns to the Main activity. When back at the Main activity, it must be obvious to the user that the order has been sent to the Server and is being processed. When the Server notifies the Client that the order is ready to be picked up, this must be displayed to the user. When the user clicks on a “Collect Order” button, the Server is notified of this and the order details are cleared on the device.

The Menu activity will display a menu of items on offer, their cost and a short description of the item. The quantity of each item ordered may be adjusted using the plus and minus buttons. Clicking the Return icon will take the user back to the Main activity. The order must be persisted, i.e. closing the app and opening it again should “remember” the order that has been created so far. Coming back from the Main activity should allow the user to resume editing the order. Only once the Send icon is clicked will an order become uneditable and after the collect order notification has been received and the user collects the order will the order be cleared.

<sup>13</sup> The latest menu needs to be obtained from the Server before displaying it to the user.

The menu items must be recycled.

Have at least ten items on the menu (it must be able to scroll). It is not shown in the screen designs above, but a menu item<sup>14</sup> must include all of the following:

- the item's name, e.g. cappachino;
- a short description;
- an optional picture of the item (at least 2-3 items on your menu must have an image, the others not. Download appropriate images from the internet for this);
- cost per item; and
- quantity of the item wanted (the  2  bit).

Ensure that the app can work in English and one other language you know. Ensure that all messages and text work in both languages. Note that the company name, Cuppa-Java, remains the same regardless of the language.

---

<sup>14</sup> You *may* tweak the design of the menu if you wish, but it *must* be in a recyclable View and have the information specified.

# Capstone Assignment:

## Mobile Multiplayer Segrada

Segrada<sup>15</sup> is a die-placing board game<sup>16</sup> for up to four people. The game is inspired by the beautiful stained glass windows (see Figure 1) of the Segrada Familia Cathedral<sup>17</sup> in Barcelona, built by Gaudi.

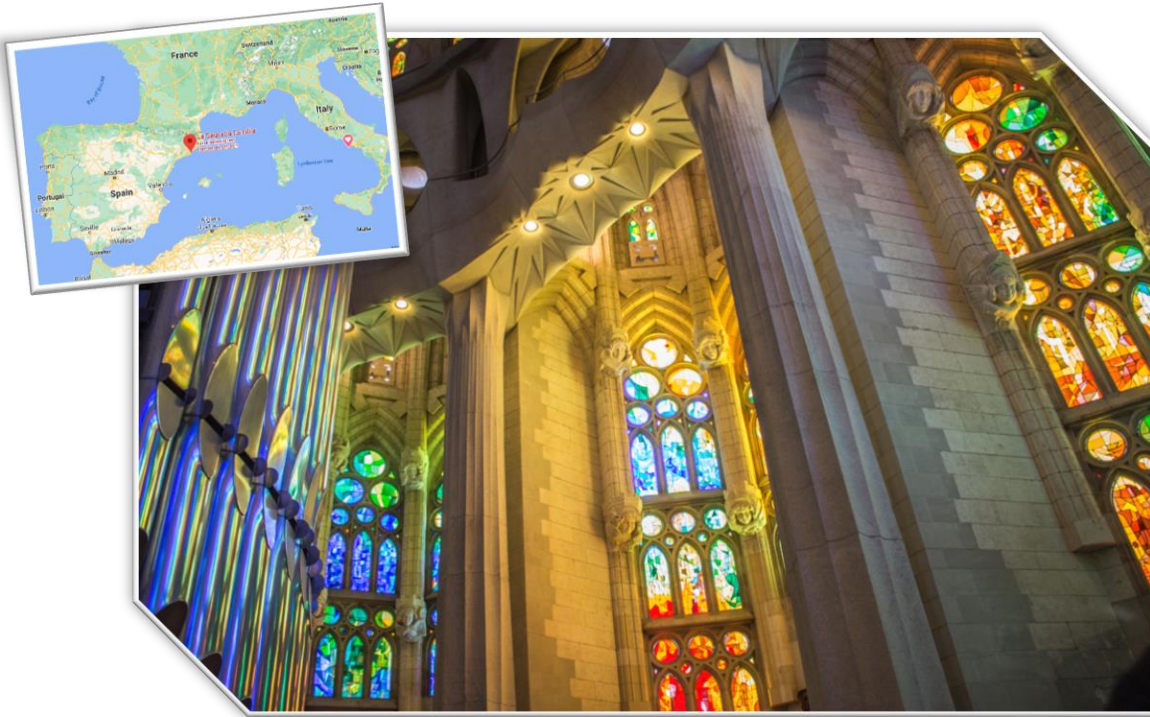


Figure 1: Some stained glass windows in the Cathedral (only natural light from outside)

For the Capstone Assignment for this module, you must:

- create a Client-Server version of the Segrada board game that uses a **simplified sub-set**<sup>18</sup> of the game rules and allows four people to play the game (each using their own device). This will require both a console-based Java Server and an Android Client;
- create a polished Android Client app that allows for a networked, multiplayer version of the board game to be played;
- create a console-based Java Server app that allows multiple Android clients to play against one another, managing their interaction;
- compile a development journal of the process you followed throughout the semester, showing how you went from the brief to the final product. The journal must include UML class diagrams of both the Client and Server apps that you implemented; and

<sup>15</sup> <http://floodgategames.com/Sagrada/>

<sup>16</sup> It's *really* fun to play!

<sup>17</sup> [https://en.wikipedia.org/wiki/Sagrada\\_Fam%C3%ADlia](https://en.wikipedia.org/wiki/Sagrada_Fam%C3%ADlia)

<sup>18</sup> The full rule set can be downloaded from (1), but you will only be required to implement *some* of the rules. You also watch some videos on how it is played on YouTube, such as this one <https://youtu.be/kl-seAITBW8>



- **submit the journal as a single PDF, the projects and source code for both the Server and Client, an installable APK for the Client, and a runnable JAR for the Server.**

The more polished and professional the final product is, the better marks you will obtain. Pay attention to the UX, as well as the functionality of the system. In other words, make sure that the app plays smoothly, intuitively, the UI looks good and makes sense, etc.

### *Simplified Segrada Rules*

Use this version of the rules in your solution<sup>19</sup>!

The aim of the game is for each player to create the “most beautiful” stained glass window by placing coloured die into a grid representing a stained glass window (see Figure 2). The “most beautiful” window is determined by a scoring system discussed later.

Every new game, each player’s window has a different window pattern that places some restrictions on how dice may be placed in that window.

On a player’s turn, the player may place a single die while obeying some simple placement rules:

- the new die must be placed next to a die that has already been placed in the window (above, below, to the left, to the right, or any of the four diagonals). The exception to this is on the first round (where no dice have been placed), in which case the new die must be placed along one of the four edges of the grid (i.e. in column 0 or 4, or in row 0 or 3);
- the die above, below, to the left and to the right of a die may not be the same colour, or have the same value;
- if a window’s grid cell has a grey value, then the die placed there must also have the same value;
- if a window’s grid cell is a specific colour, then the die placed there must also have the same colour.



<sup>19</sup> For *bonus marks*, you may implement the full set of rules, including tool and scoring cards. These rules are a lot more complex though.



Figure 2: A player's partially completed window

The game is played with 90 die of five different colours (purple, red, yellow, green and blue). There are 18 die of each colour. At the beginning of the game, all 90 dice are put into a bag. Each player is secretly<sup>20</sup> allocated one of the five colours (purple, red, yellow, green or blue). No two players have the same colour.

The game is played over 10 rounds and at the end of the 10<sup>th</sup> round, the windows are rated by how "beautiful" they are. The player with the most "beautiful" window wins.

At the start of the game, a random player is chosen as the one to start the round. Each round thereafter, the player who starts the round is the player to the left of the player who started the previous round.

A *round* consists of the following steps:

1. The player starting the round takes 9 dice from the bag of dice;
2. These 9 dice are rolled and form the *draft pool*;
3. On a player's turn, they pick 1 die from the draft pool and place it in their window grid making sure not to violate the placement rules mentioned above. If they *cannot* place a die, their turn is skipped.
4. After a player has had a turn (or skipped their turn), then the next player has a turn. The next player is chosen clockwise until it reaches the player to the *right* of the starting player. This player then has *two* turns directly after each other, and the next player is chosen *anti*-clockwise (returning to the starting player). For example, if there are four players A, B, C and D. If A is the starting player, then the order of play is  $A \Rightarrow B \Rightarrow C \Rightarrow D$  (clockwise), then  $D \Rightarrow C \Rightarrow B \Rightarrow A$ . In other words, each player has two turns in the round.
5. Once the player who started the round has their last turn, the remaining dice (normally 1) in the draft pool *are removed from the game*<sup>21</sup>.

Once the 10 rounds are completed, each player's window is allocated a score<sup>22</sup>:

1. add all the dice in the player's colour;
2. subtract 1 for each empty cell (hole in the window);
3. for every column that does not have repeated colours, add 5;
4. for every unique pair of 5's and 6's, add 2;
5. for every unique set of purple, red, yellow, green and blue dice, add 4.

The winner is the player whose window has the highest score.

## Minimum Functionality of the Apps

The apps need to have the following minimum functionality<sup>23</sup>:

1. There will need to a Main (landing) screen, on which the app starts. There should be an option that allows the Client to enter the Server's IP address and connect to it. The Server will match the first *four* unmatched players connecting to it and start a game between these four players.

<sup>20</sup> The player knows their colour, but no-one else does.

<sup>21</sup> This is different from the full rules.

<sup>22</sup> The full game has many other scoring rules as well, but these will not be used for this version.

<sup>23</sup> This is mostly specified from the Client app point of view.

The Server must be able to handle *multiple* games at the same time, i.e. more than one game and four players at a time.

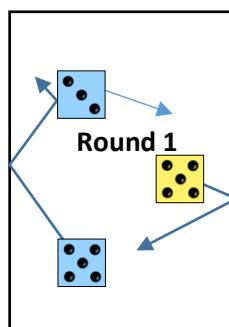
2. When the game starts up, each of the four players is randomly allocated a unique colour and window pattern (see Figure 3 and the rules PDF for some example patterns on the different pages). These patterns are predefined and each player has a different pattern.



Figure 3: Some example window patterns

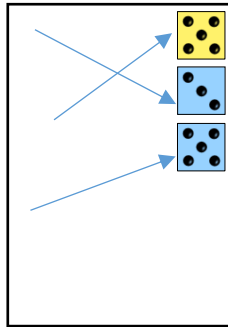
3. At the start of each round, a screen displaying the “Round 1”, “Round 2”, etc. is displayed. After 2 seconds, the 9 dice for that round will be “drawn” from the bag and will be “rolled” on the screen. “Drawing” and “Rolling” the dice involves shooting the 9 die from a specific location on the screen (the bag) and moving them off with random speed and direction. The dice then bounce around the screen. A die changes direction when it collides with the sides of the screen or another die. Whenever a die collides with something, its value randomly changes and it slows down a bit. As the die moves around the screen, its speed slowly decreases<sup>24</sup>. When all the dice have stopped moving, their values are finalised.

All players (of the same game) will see the *same* movement, bounces and value changes as the dice bounce around on the screen. In other words, if you placed the four phones next to each other, the movements should look the same.

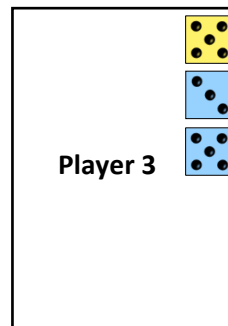


<sup>24</sup> Accurate physics do *not* need to be simulated and a die does not need to spin, i.e. it can remain a square aligned with the sides of the screen at all times.

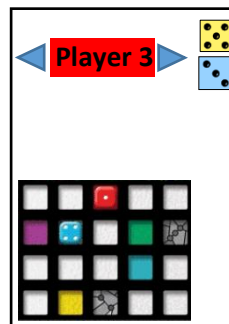
4. Once the dice values have been finalised, the dice need to be animated moving to the right-hand<sup>25</sup> side of the screen (i.e. the draft pool), where the dice are sorted first by colour, then by value from top to bottom.



5. Once the dice have been placed in the draft pool, the player whose turn it is must be displayed, e.g. "Player 1", "Player 2", etc.



6. For this app, you only need to display the window pattern and the dice already placed. You do *not* need to display the actual image of the window (unless you *want* too... because it looks pretty). This screen may have a "next" and "prev" button that will allow the player to view other player's windows, but not their colour and not allow their window to be changed<sup>26</sup>.



7. On the player's turn, the player can select one of the dice in the draft pool to place it. When selected, the *valid* locations where it can be placed must be highlighted in the window. Tapping one of the valid places will animate the die moving from the draft pool into the

<sup>25</sup> If your screen design might look better with the dice being at the bottom, left or top of the screen, this is fine. It is up to you to choose the best UX for your app. But the die must move from being scattered all over the screen to being placed in a sorted order somewhere.

<sup>26</sup> If you decide you want to rather swipe left or right to move between the different windows, that's fine too. But you need to be able to see all the windows in turn.

selected place. If unable to place a die, a “skip turn” button should be displayed. Clicking the skip button or placing a die will move to the next player (and return to step 4).

8. When other players are taking their turn, a player will see the die being removed from the draft pool. If the player is viewing another player’s window when they place a die, the placement will also be animated. i.e. if *you* are viewing a window that has a die being placed in it, you’ll see the die moving into place. If *you* are not viewing the window, you’ll only see the die being removed from the draft pool.
9. Once each player has had 2 turns, the round is completed. Return to step 2 if there are still more rounds to be played.
10. Once all 10 rounds have been completed, a results screen must be displayed. The scores of the players must be displayed in descending order. For each player, a breakdown of the score must be given, e.g. “player colour = 20, holes = 0, columns = 10, 5 + 6 pairs = 4, colour sets = 4, total = 38”

The above is some of the *minimum* functionality required. It is up to you to decide on the different screens and how to move between them.

## Bonus Marks

An additional 10% is available as *bonus marks*. This means that it is *potentially* possible to score a mark of 110% for the Capstone Assignment.

It is up to you to decide if you *want* to implement additional functionality to be considered for *bonus marks*. This can include implementing the full rule set, in-game communication (e.g. being able to talk to one another during a game), integration (e.g. using WhatsApp to launch a game between friends), rendering (e.g. using actual 3D die), etc.

Discuss the functionality you have in mind with your lecturer first to get an idea of the feasibility and value (in terms of bonus marks) of what you have in mind.

## Preliminary Mark Allocation

To give you an idea of how marks might be allocated, a preliminary mark allocation is provided. Please note that the distribution may change (to become more specific if needed). You will be notified should such changes be made.

Criterion	Description	Marks
<b>Documentation</b>		<b>10%</b>
1	Journal	7%
2	UML Class Diagrams.	3%
<b>Game Play Logic (Non-Visual)</b>		<b>20%</b>
3	Game & Turn Management	5%
4	Dice Drawing & Draft Pool logic	3%
5	Dice Placement logic	6%
6	Scoring logic	6%
<b>UX/UI (Visual)</b>		<b>30%</b>
7	Clean, understandable UI. Visually appealing.	3%
8	Interaction techniques chosen are intuitive and appropriate to mobile devices.	2%
9	Screen Navigation	2%
10	Network Connection screen	2%
11	Window screen (dice placement functionality). Showing valid positions to place selected die, turn skipping, player information.	5%
12	Window screen (viewing functionality). Viewing another player's window, without being able to alter it.	3%
13	Draft Pool functionality (select die, animate die moving into place or being removed from pool, sorting by colour and value)	5%
14	Results screen (player points and order)	3%
15	Animated views, including rolling dice, moving into sorted positions in die pool after rolled.	5%
<b>Multiplayer Functionality</b>		<b>40%</b>
16	Four players play a game. Waiting screen while waiting for other players. Once enough players, game starts.	5%
17	Multiple game management on Server (i.e. many simultaneous games should be possible)	10%
18	Communication system between Server & Clients (i.e. Commands or Networked Pub/Sub Messages, etc.)	7%
19	Turn management & communication	5%
20	Die placement & communication	5%
21	Results management & communication	5%
22	Ending a game gracefully	3%
<b>Total:</b>		<b>100%</b>

## Work to a Schedule

It is recommended that you set up and follow a schedule in order to complete the Capstone Assignment on time. An example schedule is given below.

Week	Dates	Regular Activity	Capstone Assignment Activity
1	28 Sep - 2 Oct	Assignment 1	Plan (on paper) the model of game, screens that will be created and how you will move between them, draw initial UML class diagram(s). It is recommended that you plan for using <i>Command Pattern</i> <sup>27</sup> or the <i>Networked Pub/Sub Broker</i> (from Assignment 4).
2	5-9 Oct		
3	12-16 Oct	Assignment 2	Implement and test game model, start creating Activities, navigation, etc.
4	19-23 Oct		
5	26-30 Oct	<b>48H Assignment (Wednesday &amp; Thursday)</b>	Connect activities together, implement functionality for single device, pass-and-play.
6	2-6 Nov	Assignment 3	Fully functional single device pass-and-play version.
7	9-13 Nov		
8	16-20 Nov	Assignment 4	Implement dice bouncing animations, basic connectivity between devices.
9	23-27 Nov		
10	30 Nov - 4 Dec		Finalise network communication. Polishing, winning screens, extras (e.g. full game rules, chat, etc.)
11	7-11 Dec		
12	14-18 Dec	<b>Capstone Assignment (due Tuesday)</b> <b>Reflective Report (due Thursday)</b>	Compile into presentable form to submit on Tuesday. Do <b>not</b> be late submitting!

<sup>27</sup> Read [https://en.wikipedia.org/wiki/Command\\_pattern](https://en.wikipedia.org/wiki/Command_pattern) and [https://www.tutorialspoint.com/design\\_pattern/command\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/command_pattern.htm). The library also has the book mentioned in the references if you want to read it: *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.*