COSC1076 | Semester 1 2020
Advanced Programming Techniques

# Assignment 2 (v1.1)
# Programming Project (Azul)

**Weight:** 50% of the final course mark
**Group Registration:** Week 7 Lab
**Progress Updates:** Weekly (with your tutor during labs)
**Group Deliverable Due Date:** 11.59pm Friday 22 May 2020 (Week 11)
**Individual Deliverable Due Date:** 11.59pm Friday 5 June 2020 (Week 13)
**Written Report Due Date:** 11.59pm Friday 5 June 2020 (Week 13)
**Presentation & Marking:** Week 14, by registered time slot
**Learning Outcomes:** This assignment contributes to CLOs: 1, 2, 3, 4, 5, 6

## Change Log

1.2
- Released Enhancements for Milestone 2

1.1
- Removed reference to linked list in Suggestions section.

1.0
- Initial Release
- Group Deliverable
- Written Report and Demonstration
- Enhancements to be released at a later date

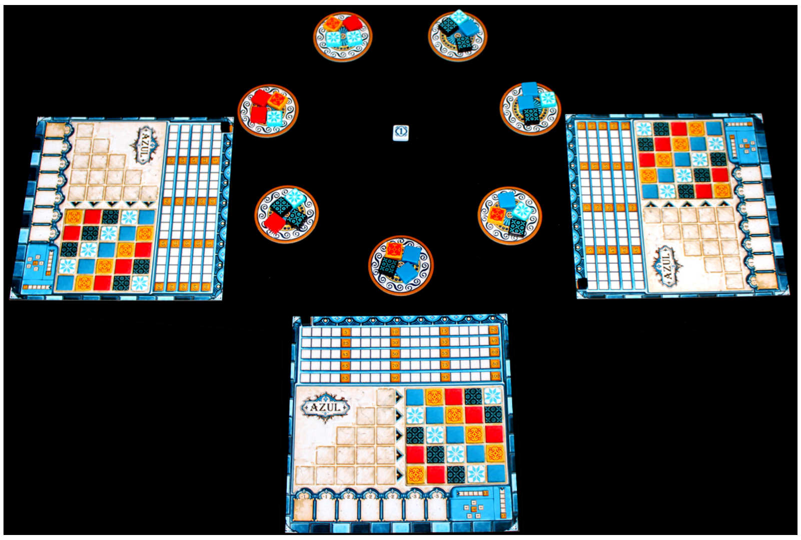
# Contents

# 1 Introduction

## 1.1 Summary

In this assignment you will implement a ***2-player*** text-based version of the Board Game ***Azul***.

(a) Qwirle box and pieces


(b) Example game state

For an explanation of the rules and gameplay:

- Review and rules explanation: by SU&SD (Youtube)
- Rules explanation: by Dice Tower
- Official Rules: Online Website

In this assignment you will:

- Practice the programming skills covered throughout this course, such as:
  - ADTs
  - Data Strcutres (Arrays, Vectors, Liked Lists, Trees, etc.)
  - Dynamic Memory Management
  - File Processing
  - Program State Management
  - Exception Handling
- Practice the use of testing
- Implement a medium size C++ program:
  - Use features of C++14
  - Use elements of the C++ STL
- Work as a team
  - Use group collaboration tools (MS Teams, Git, etc.)

This assignment is divided into three Milestones:

- **Milestone 1, Base Implementation (Group work)**: In your group of 3, you will implement a fully functioning version of the base `Azul` gameplay. This will also include writing tests that show your implementation is fully functional. The group component is due 11.59pm Friday 22 May 2020 (Week 11). The group work is worth 30% of *the course mark*.
- **Milestone 2, Enhancements (Individual work)**. You will *individually* extend upon your groups base implementation with additional functionality (called enhancements). The individual component is due 11.59pm Friday 5 June 2020 (Week 13). The individual work is worth 20% of *the course mark*.
- **Milestone 3, Written report (no more than 4 pages) & Demonstration**. You will write a report that analyses the design and implementation of your software. The report is sue 11.59pm Friday 5 June 2020 (Week 13). As a group you will demonstrate your group's work and each individual's enhancements. This is where your final work will be graded. Demonstrations will be held during Week 14.

The marking rubric shows which elements of the assignment contribute to your group and individual marks.

! To be fair to all groups and avoid giving a head-start, the specification and requirements for the *individual* component will be released in Week 11.

## 1.2 Group Work

This group work component completed groups of 3. All group members **must be from the same Lab**. There will be **no exceptions** to this requirement[1]. Your group must be registered (with your tutor), by your Week 7 Lab. If you are unable to find a group, discuss this with your tutor **as soon as possible**.

If at any point you have problems working with your group, **inform your tutor immediately**, so that issues may be resolved. This is especially important with the online delivery of the course. The weekly progress updates are for communicating the progress of your group work. We will do our best to help manage group issues, so that everybody receives a fair grade for their contributions. To help with managing your group work we will be requiring your group to use particular tools. These are detailed in Section 5.

! There are **very important requirements** about keeping your tutor informed of your individual progress, and if you have been unwell or otherwise unable to contribute to your group (Section5.2).
Remember your actions affect **everybody in your group**. If you fail to keep us informed, you may individually receive a lower grade.

To complete this assignment you will require skills and knowledge from lecture and lab material for Weeks 7 to 10 (inclusive). You may find that you will be unable to complete some of the activities until you have completed the relevant lab work. However, you will be able to commence work on some sections. Note that the mark for your group work requires *consistent work throughout all weeks*. Thus, do the work as you can, building in new features as you learn the relevant skills.

## 1.3 Academic Integrity and Plagiarism

! CLO 6 for this course is: *Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.*

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites. If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own.

RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the RMIT Academic Integrity Website.

The penalty for plagiarised assignments include zero marks for that assignment, or failure for this course. Please keep in mind that RMIT University uses plagiarism detection software.

[1]The reason for this is the group updates will be with your tutor each week during labs. These groups updates *inform your final grade*. Thus, restricting groups to labs allows us to ensure you can attend the weekly updates, practically manage the groups, help ensure everybody is on-track and address group issues.

## 2 Milestone 1: Base Gameplay (Group Component)

In your groups you will produce an implementation of the "base" `Azul` gameplay. This section lists the components that your group must implement. Generally, it is up to your group to decide the best way to implement these components. However, there are some **important requirements** that your group must satisfy.

! For the aspects of this specification that are flexible and open to your interpretation, it is up to *your group* to determine the best course of action. You will need to **analyse** and **justify** your choices in your group's written report.

### 2.1 Base gameplay Requirements

Your group will implement a base `Azul` gameplay which is defined as:

- A **2-player** game, ONLY.
- Both players are "human users" that play the game by interacting the the terminal, that is through standard input/output.
- Using the original/default `Azul` Mosaic, with a fixed colour pattern. This is the colour mosaic as picture in Section 1.1. (Note this pattern is the same for *all* players).
- Use 5 factories (plus the central factory) as specified in the `Azul` rules for a 2-player game.
- Automatic moving of tiles to the mosaic and scoring of points at the end of a round.

In addition to the above gameplay, your base implementation must provide the following functionality:

- A main menu, that allows users to perform actions such as setting up a new game, loading an existing game, showing "credits" (details of the people who wrote the software), and quitting the program.
- Save a game in-progress to a file.
- Load a previously saved game from a file, and resume gameplay from the saved state.
- A way to provide a seed to the random number generator in your program to "turn off" and randomness in the program.
- A way to represent and display the `Azul` mosaics and factories to the user.
- A User prompt for entering all commands from standard input.
- Provide sufficient help to the users about how to use your program (that is help for the commands that can be entered at the user prompt), For this you may assume that the users know the rules of Azul, and just need to know how to use your program.
- Full Error checking of all user input (and save-game files). It a user ever enters an invalid command, your program should notify the user of the reason their input is incorrect and resume operation in a suitable manner. Your program must never crash or exit unless:
  - The user explicit requests for the program to terminate.
  - The EOF character is given on standard input.

! Of course your program must also be error free, should not segfault, crash, or contain logic errors.

How your group implements the above functionality is mostly up to you. This includes the Data Structures, ADTs, Classes and STL libraries you choose to use. However, as part of this course is about analysing data structures, you are required to use a particular set of data structures in your implementation. These **mandatory requirements** are:

- You must use *at least one* linked list to store or represent some aspect of the game.
- You must use *at least one* C++ STL vector to store or represent some aspect of the game.
- You must use *at least one* 1D or 2D array to store or represent some aspect of the game.
- You may only use the C++14 STL. You may not incorporate any additional libraries.

Remember, that in your report your group will be marked on the justifications and analysis of the above choices. You need to be careful which data structures that you choose to represent the aspects of `Azul`, and the reasons behind these choices.

! Your program must stick to this base functionality as **enhancements** is part of Milestone 2.

## 2.2 Save Game Format and Testing

The layout of saving a game to a file to very important. You will need to be able to test if your program is correct. For testing, we will use the load/save game functionality. That is, you can load a game from a file, then execute a given set of commands, before saving the game. A test can then compare the contents of the saved game file against an pre-determined saved game to see if the files match and the program functioned correctly.

Section 6 describes this in more detail. In particular, in labs you will be designing the format of the save-game file. This means that groups in your lab can share tests! This way it will be easier for everyone to check their programs are running correctly.

! Hopefully, you will be sharing tests with other groups in your lab!

## 2.3 Example Base Program

As an example, when you have implemented your base gameplay it might look as follows. Note that the below combines output from the program (to standard output - ie `std::cout`) and input from the user (on standard input - ie `std::cin`). For this example, it is assumed the user prompt it given by `>` and any text after this has been given on standard input.

! Recall that a game of `Azul` takes place across a series of rounds, until one player triggers the end-of-game condition. This is an example game might look it with your `Azul` program, for two players `A` and `B`.

```
$ ./azul -s <seed>
Welcome to Azul!
-------------------

Menu
----
1. New Game
2. Load Game
3. Credits (Show student information)
4. Quit

> 3

----------------------------------
Name: <full name>
Student ID: <student number>
Email: <email address>

<Student 2, 3, etc.>
----------------------------------

Menu
----
1. New Game
2. Load Game
3. Show student information
4. Quit

> 1

Starting a New Game
```

```
Enter a name for player 1
> <user enters name>

Enter a name for player 2
> <user enters name>

Let's Play!

=== Start Round ===
TURN FOR PLAYER: A
Factories:
0: F
1: R Y Y U
2: R B B B
3: B L L L
4: R R U U
5: R Y B L

Mosaic for A:
1:         . || . . . . .
2:       . . || . . . . .
3:     . . . || . . . . .
4:   . . . . || . . . . .
5: . . . . . || . . . . .
broken:
> turn 2 B 3
Turn successful.

TURN FOR PLAYER: B
Factories:
0: F R
1: R Y Y U
2:
3: B L L L
4: R R U U
5: R Y B L

Mosaic for B:
1:         . || . . . . .
2:       . . || . . . . .
3:     . . . || . . . . .
4:   . . . . || . . . . .
5: . . . . . || . . . . .
broken:
> turn 3 L 3
Turn successful.

< the following turns happen >
(A) > turn 2 Y 2
(B) > turn 5 Y 1
(A) > turn 4 U 5
(B) > turn 0 B 2 (gets first player token)
(A) > turn 0 L 1
(B) > turn 0 R 5
(A) > turn 0 U 5

=== END OF ROUND ===
=== Start Round ===

> save savedGame

Game successfully saved to 'saveGame'

< Gameplay continues until end-of-game >

=== GAME OVER ===

Player B wins!
```

## 2.4 Suggestions

This section provides suggestions that you might wish to consider when implementing the base gameplay.

! These are suggestions of what to do. You could choose a different approach. However, remember you must **justify** your choices!

### 2.4.1 User Prompt

The user prompt is displayed whenever input is required from the user. In these examples, we use a greater-than symbol (>), followed by a space.

```
> █
```

This assumes that all user inputs are provided as a single line of input, and a return key.

If at any point the user enters input what is invalid then the program should print a useful error message and re-show the prompt so the user can try again.

```
> qwerty
Invalid Input
> █
```

If the user enters the EOF (end-of-file) character[2], then the program should Quit.

```
> ^D
Goodbye
```

### 2.4.2 Tiles

Tiles could be represented by a single-character code, based on their colour as in the table below. Special codes represent the 1st-player maker, and where a tile is not present.

| Colour | Colour Code |
|---|---|
| Red | R |
| Yellow | Y |
| Dark Blue | B |
| Light Blue | L |
| Black | U |
| `first-player` | F |
| `no-tile` | . |

You might also wish to define a "total ordering" over the tiles to help print them out consistently. For example, tiles are always shown in the order: `F`, `R`, `Y`, `B`, `L`, `U`.

[2]Reminder: this is **not** the two characters `^` and `D`, this is the representation of EOF when typing `control-D`

### 2.4.3 Game Board

The board has two elements:

1. The shared central area - "factories"
2. The individual player board - "mosaic".

The factories can be represented by listing all of the tiles on the factory. This labels factories with numbers so the user can refer to them, with factory `0` as the "centre" factory.

```
Factories:
0: F B U
1: R Y Y U
2:
3: B L L L
4: R R U U
5: R Y B L
```

This shows the state of an individual players mosaic, giving:

- Storage rows of unlaid tiles
- Completed grid of tiles.

This is an example of a mosaic. The "broken" tiles (including the 1st player marker) are listed at the bottom. Again, each storage row is numbered so the players can refer to it.

```
Mosaic for <player-name>:
1:         . || . . R . .
2:       Y Y || . . . R .
3:     B B B || . . . . .
4:   . . . . || . . L . .
5: . . U U U || . . . . .
broken: F Y
```

! If you take this approach, you should note that it does not show where tiles are placed in the mosaic. So you will need to come up with a way to show this to the players.

### 2.4.4 Tile Bag & Box Lid

You will need two represent the following, however, you won't show these to the players:

- The Tile bag - used to fill factories at the start of each round.
- The Box Lid - used to store tiles that are removed from the mosaic storage at the end of each round.

The best data structure to store and use the tile bag and box lid is up to your group to determine. However, you might want to think about operations such as:

- Drawing tiles from the bad
- Putting tiles into the box lid
- Refiling the bag

You will also need to work out how to deal with the "randomness' of drawing tiles from the bag. We recommend that you do this *at the start of the program only.* That is you shuffle the tile bag once, and then *always* draw tiles to/from the bad/lid in a sequential manner. This will **be very useful** for testing.

### 2.4.5 Launching the program

Your `Azul` program will be run from the terminal, it might take a command-line arguments such as a seed for generating random numbers.

```
$ ./azul -s <seed>
```

### 2.4.6 Main Menu

The main menu shows the options of your `Azul` program. By default there should be 4 options (new game, load game from a file, credits and quit). The menu is followed by the user prompt.

### 2.4.7 Starting a New Game

The program should start a new game. As part of this you might want to get names for each of the players.

When a new game is started, your program will need to construct the initial state of the game. We recommend that you also construct an initial *random* ordering of the tile bag. Once this random order is determined, the order is **fixed** for the whole duration of the game..

You will need to devise *your own algorithm* to "shuffle" the bag of tiles to create a "random" initial order. This is left up to your own invention. Remember, that randomness makes testing very hard, so it might be a good idea to use a command-line arguement to take a fixed seed for your pseudo-random number generator so that the order of the tile bad will be the same every time!

! The reason for this is that you can consistently test your game. By fixing the tile bag order, everything that happens is completely *deterministic*!

### 2.4.8 Load a Game from a file

The program asks the user for a filename from which to load a game, where the filename is a *relative path* to the saved game file.

```
<main menu>
> 2

Enter the filename from which load a game
> <filename>

Azul game successfully loaded
<game play continues from here>
```

It is highly recommended to conduct validation checks such as:

1. Check that the file exists.
2. Check that the file contains a valid game.

To load a game, your program will need to read the contents of the saved game file, and update all data structures in the program. Specifically, the program should take note of:

- The player's name and scores
- The state of the factories and player mosaic's
- The order of the tiles in *both* bags
- The current player - the next player to take a turn

Once the game has been loaded, gameplay continues resumes with the current player.

### 2.4.9 Credits (Show student information)

The program should print the name, student number, and email address of each student in the group.

### 2.4.10 Quit

The program should safely terminate ***without crashing***.

### 2.4.11 Typical Gameplay

In `Azul`, 2 players take turns drawing tiles from factories and placing them in storage on their individual mosaic, starting with the first player. Once all tiles a drawn, the round ends and scoring automatically happens. Scoring should happen **automatically**. Then either the next round commences, or the game ends (including the end-of-game scoring).

### 2.4.12 Starting a Round of `Azul`

At the beginning of a round, the factories need to be filled by drawing tiles from the Tile Bag. To ensure consistency (and help with testing), your might want to fill factories in **order**, starting with factory `1`. Remember, if the tile bag runs of out tiles, then all of the tiles from the Box Lid are placed back into the Tile bag. Don't forget to add the 1st-player marker should be added to the "centre" factory (number `0`).

### 2.4.13 A Player's Turn

A player might take their turn using the command such as

```
turn <factory> <colour> <storage row>
```

The command contains three elements:

1. A number of the factory to draw from
2. The colour to take from the factory
3. The row in the mosaic storage to place the tiles

After the player enters the turn command, the game should:

- Validate that the turn is *legal*, checking the player's action against all of the rules of `Azul`
- Update the game-state based on the player's turn, then continue with the next player's turn.

### 2.4.14 End-of-round

Once all tiles have been drawn from the factories, the program should:

- Move tiles from each player's storage to their completed mosaic grid, as per the rules of Azul. (Don't forget to move excess tiles to the Box Lid)
- Update the players score as the tiles are moved.
- Subtract points for broken tiles (Don't forget to move these tiles to back to the Box Lid).

You might want to then show:

- The mosaic for each player
- How many points each player scored on that round
- How many total points each player has

The game play then either:

- Proceeds to the next round. Remember to re-fill the factories from the tile bag!
- If the end-of-game condition is met (that is, one player has finished a *full row* of their mosaic), terminates gameplay, showing the winner.

### 2.4.15 End-of-game

If the end-of-game condition is reached (at least one player has completed a whole row of their mosaic), then the game ends. Don't forget to complete the **end of game scoring**, in particular, for completed rows, columns, and colour sets. You should then show the winner.

### 2.4.16 Saving the Game to a file

At any point during gameplay, the current player may save the game to a file using a command such as:

```
save <filename>
```

Your program should save the current state of the game to the provided filename (overwriting the file if it already exists).

The format for the saved-game is up to you to decide in your labs.

! Remember that you will decide the format of the file in your labs to that you can share tests!

# 3 Milestone 2: Enhancements (Individual Component)

In Milestone 2, as an **individual** you will make signification expansions(s) to the functionality of your group's `Azul` program. You will select your enhancements from a suggested list, to be provided around Week 11. You may also **negotiate** potential enhancements with your tutor.

! This milestone is your chance to showcase to us your skills, capabilities and knowledge!

Milestone 2 is a very open-ended milestone with lots of room for you to showcase your abilities. You are given some directives, however, there is a lot of room for you to make considered and well-reasoned choices. Take careful note that this showcase of your skills is not just about "making the code work". A major focus is on *how* you choose to the implement an enhancement, and the *justifications of the reasons why* you chose a given data structure, class hierarchy, language feature, or algorithm to name a few examples.

Enhancements are classified as **minor** or **major**. Enhancements only count towards this Milestone if they are *fully functional*, *error-free*, and have *suitable justifications*. If you break your Milestone 1 implementation, this will count against your Milestone 1 grade. So, be careful to make sure everything is working.

! Enhancement will only be considered in marking if they make a significant expansion, change in functionality, or modification to the implementation of your group's base `Azul` program. If your group's base `Azul` program already *substantially satisfies* an existing enhancement **you must select and implement an alternative enhancement**.

The following enhancements are based on the suggested approach to Milestone 1. If you wish to use any C++ STL libraries in implementing your enhancements, consult with your tutor or ask on the forum.

If you group's Milestone 1 solution has *significant errors* or is *significantly incomplete*, please read Section 3.3.

## 3.1 Minor enhancements

### 3.1.1 User Interface Improvements

The basic user interface in the suggested implementation of Milestone 1 has many deficiencies which may make is hard for a user to understand and play a a game of `Azul`. For example:

- The information to standard output has limited formatting
- It may be difficult to distinguish or interpret the ASCII character codes
- Identifying the location where a tile will go on the mosaic may be difficult
- It may be important to see the state of the opponents board, requiring scrolling through the text
- The user may not know the commands they can enter to play the game

This enhancement requires you to *significantly improve* the terminal-based user interface. You may wish to consider implementing the following:

- Provide a way for the user to query what commands they may use, such as typing "`help`". This help should be aware of the program/game state and only given relevant help.
- Use colour to enhance the display of game information (such as the tiles and mosaic). Linux, Mac (and most similar) terminals support the use of colour through the use of *escape codes*.
- Use Unicode to enhance the display of game information. You can embed Unicode symbols symbols directly into C++ files which can then be written to output streams. Most modern terminals also support the display of unicode symbols.
- The format of the saved-game file does not need to change and can remain plain-text.

You may need to investigate how to correctly use some of the above features in C++14.

Simple or cursory cosmetic changes will not be suitable for satisfying this enhancement. You will need to take a wholistic and comprehensive approach to have the best terminal-based user interface you are able to provide.

In your *individual* report you should justify why the user interface is *significantly improved*.

### 3.1.2 Data Structure Improvements, Generics & Code Re-use

Additional data structures can be incorporated into your group's `Azul` program. This enhancement requires you to *significantly modify* and reconsider the data structures used in your `Azul` program. You must:

- Add a *Binary Search Tree*, which uses C++ Generics.
- Modify your implementation of a *linked list* to be a *double-ended linked list* which uses C++ Generics. You must implement this generic double-ended link, and may not rely on the C++ STL containers.
- Ensure your modified program still contains:
  - You must use *at least one* binary search tree to store or represent some aspect of the game.
  - You must use *at least one* double-ended linked list to store or represent some aspect of the game.
  - You must use *at least one* C++ STL vector to store or represent some aspect of the game.
  - You must use *at least one* 1D or 2D array to store or represent some aspect of the game.

You should also consider if your group's `Azul` program uses the most suitable data structures to represent the elements of the `Azul` game. If the use of data structures could be improved, then you must modify the program to use the most appropriate data structures. In particular you may wish to consider using a stack or queue.

Finally, you need to ensure your program follows the principles of code re-use that have been discussed in this course. You may wish to consider:

- Are constructors (including copy constructors) and methods, calling other constructors and methods to avoid duplicating code and logic within a class?
- Could class inheritance and abstract classes be used to reduce code duplication?
- Where class inheritance is used, should code be relocated to common parent classes to re-use code?

In your *individual* report you should justify why the data structures and code re-use has *significantly improved.*

### 3.1.3 Replay Mode

This enhancement requires that you *significantly modify* the functionality to your `Azul` program to allow a user to load a saved-game and then replay all of the turns of the game from the beginning of the game. The replay mode should be interactive, allowing the user to decide when to observe each move. That is, the replay mode should not just dump the entire replay of the game to the terminal, instead wait for input from the user before moving onto the next turn.

This enhancement will require you to modify your saved-game format. However, you must be *efficient* in your representation. You should *not* just store the state of the game after each turn. Instead you should store minimal information, such as a description of each turn.

For this enhancement, you may wish to consider:

- You will need to track each turn that is played, and use a suitable data structure to store this information. You will also need to load into this data structure when reading a saved-game.
- The user may wish to resume a game after viewing the replay. So you may need to modify the behaviour of your main menu.
- The replay functionality should work with a game that is fully completed. If your `Azul` program does not yet allow a user to save a completed game, you will need to add this functionality.
- You should provide sufficient feedback to the user about each turn that is executed. For example, in the normal gameplay the actual command the user entered may not be printed to the terminal.
- The user may wish to exit the replay early.

In your *individual* report you should justify your choice of data structure, why the representation of information in the saved-game is *efficient* and justify any other necessary changes to your group's `Azul` program.

## 3.2 Major enhancements

### 3.2.1 3-4 Player Mode & 2 Centre Factories

The Milestone 1 version of `Azul` is for 2 players only. The enhancement requires you to allow `Azul` to be played with either 2, 3, or 4 players. Ensure that you read the rules of `Azul`, as the number of factories changes depending on the number of players. This will also have significant impacts on your text-based UI for playing `Azul`. This is because it's common for players to look at opponents mosaic(s) when considering their turn. In the 2-player game this is easy as the opponents board may still be on the screen. However, for more players, it becomes too difficult to keep scrolling up in the terminal. Thus you should consider ways to improve the UI to make it easier to see an opponents mosaic.

The enhancement *also* requires you to allow the user to specify whether 1 or 2 centre factories are used for the game. If 2 centre factories are used, the rules of `Azul` change as follows:

- When playing a turn from a 'normal' factory, the user must choose which of the 2 centre factories in which to place the excess tiles.
- A user may choose to select tiles (following standard rules) from either one of the 2 centre factories
- The first player to draw from any centre factory receives the first player marker.

For this enhancement, you may wish to consider:

- This gives up to 6 different combinations of `Azul` games, with varying numbers of players and factories.
- You may need to *redesign and change* the data structures that are used to represent `Azul`. However, you must *still* meet the mandatory minimal use of each data structure as given in Milestone 1.
- You will need to modify the saved-game format to support the varying number of players and centre factories.
- You may need additional commands to support your UI changes.

In your *individual* report you should justify any *significant* changes to the choice of data structure(s), why the representation of information in the saved-game is *suitable*, describe any *significant* changes to your text-based UI, and justify any other necessary changes to your group's `Azul` program.

### 3.2.2 Advanced Azul (grey-board and 6-tile modes)

`Azul` can be played in more advanced modes that can bring additional strategy and challenge for players. The enhancement requires you implement two "advanced modes" for `Azul`, the "grey-board" and "6-tile" modes.

The "grey-board" is an advanced mode of `Azul` where the location that tiles are placed into a mosaic is not fixed. You should consult the `Azul` rules for the "grey-board" mode. Importantly, this mode requires players to manually place tiles onto their mosaic after all the factories are empty for end-of-round scoring, rather than the process being automatic. You will need to devise a way for players to place tiles onto their mosaic.

The "6-tile" mode introduces a 6th tile, which for the purposes of this enhancement will be denoted by the colour "orange". To permit play with the 6th tile, the following changes to the `Azul` board are made:

- The mosaic is expanded to a 6x6 board
- The 6th row has space to store 6 tiles
- An additional broken tile slot is added, worth `-4` points

For this enhancement, you may wish to consider:

- You may need to *redesign and change* the data structures that are used to represent `Azul`. However, you must *still* meet the mandatory minimal use of each data structure as given in Milestone 1.
- You will need to modify the saved-game format to support the advanced options.
- You may need additional commands to support the advanced modes.

In your *individual* report you should justify any *significant* changes to the choice of data structure(s), why the representation of information in the saved-game is *suitable*, describe any *significant* changes to your text-based UI, and justify any other necessary changes to your group's `Azul` program.

### 3.2.3 Write an AI

As with many computer-base board-games, it would be nice to be able to player in a single-player mode against the computer. This enhancement requires you to develop an AI (Artificial Intelligence) so that your program can be used in single-player mode, and a person can play against the computer AI. When it's the AI's turn, it should make its move automatically without the user having to input a command.

Take careful note that an AI implies **intelligence**. Thus, an AI doesn't make random moves. It uses logic and **heuristics** to determine a "good" move, and hopefully the "best" move to make. That is, your AI needs to have "smarts" or "intelligence" to figure out a good move to make. You may not be able to decide the most "optimal" move, however, the choice of move must be better than random.

For this enhancement, you may wish to consider:

- You may need to *redesign and change* the data structures that are used to represent `Azul`. However, you must *still* meet the mandatory minimal use of each data structure as given in Milestone 1.

- The AI should not take a long time to calculate its move.
- You will need to modify the saved-game format to record if an AI is being used.

In your *individual* report you should justify why your AI has a *"good" heuristic* and why the moves that are made by the AI are intelligent. You should also describe any *significant* changes to the choice of data structure(s) describe and justify any other necessary changes to your group's `Azul` program.

## 3.3 Milestone 1 Code with Significant Errors

! This section is a **general** statement and overview. This is provided as a **starting point** from which to approach your tutor. You may **ONLY** consider things here as counting toward enhancements if you have discussed the matter with your tutor.

Given the scope of the group work, it is possible that your group's `Azul` program may contain errors. These may range from small logic/gameplay errors on minor edge cases within the code, to significant errors where the program may not even compile.

If the errors in the your group's `Azul` program are small, while you should fix these in your individual work, they are too minor to be considered a *significant* change to the `Azul` functionality. Fixing minor errors is encompassed within the scope of the above enhancements.

However, if for whatever reason, your group's `Azul` program has significant errors or is missing significant functionality, then it **may be able to negotiate** with your tutor for fixing these errors to count as an enhancement. As each group's program is different, this possibility will be determined on a case-by-case basis.

Please note that this does not excuse you from (as an individual) failing to make sufficient contributions to your group's `Azul` program. While you may be able to "make-up" some functionality in this milestone, you should not use this as an excuse to make an insufficient contribution to your group. You may still receive a grade penalty as per the Milestone 1 rubric.

## 3.4 HD+ submissions

To receive a grade of `HD+` your submission need to be **outstanding**. This means the submission need to stand apart from other submissions. If you would like to receive top marks, you will need to go well above-and-beyond the minimum implementation of each enhancement. Of course, a `HD+` submission will be error free and be well-justified.

## 4 Milestone 3: Written report & Demonstration

Your group must write a report, that *analyses* what your group has done in this assignment. Additionally, each **individual** must write a short report that *analyses* their individual enhancement(s). The report is due at the same time as the individual submission.

- The report should be A4, 11pt font, 2cm margins and single-space.
- The section of the report describing the group's work must be no more than **4 pages**.
- Each individual must add **1 additional page** about their individual work (ie, their enhancements).
- Thus the final report **must not exceed 7 pages** (for a group of 3).
- Only the first 4 pages (group), and 1 page (individually) will be marked. Any contents in the report that is over-length will not be marked. Modifying fonts and spacing will count as over length.
- Figures, Tables and References count towards these page limits.

In this assignment, you are marked on the analysis and justification of the choices you have made for different aspects of your assignment. Your report will be used (in conjunction with the demonstration) to determine the quality of your decisions and analysis.

Good analysis provides factual statements with *evidence* and *explanations*.
Statements such as:

> *"We did <xyz> because we felt that it was good"* or *"Feature <xyz> is more efficient"*

do not have any analysis. These are unjustified opinions. Instead, you should aim for:

> *"We did <xyz> because it is more efficient. It is more efficient because ..."*

> **!** We are asking for a combined report as it keeps the context of each individual's enhancements with the whole group's original implementation.

### 4.1 Group Component of the Report

In the **group** section of your report, you should discuss aspects such as:

- Your group's use of *at least one* linked list, and the reasons for where the linked list is used.
- Your group's use of *at least one* C++ STL vector, and the reasons for its use.
- Your group's use of *at least one* 1D or 2D array, and the reasons for its use.
- Your group's choices of ADTs, and how these leads to a "well designed" program.
- The efficiency of your implementation, such as the efficiency of your use of data structures.
- The reason for the tests that your group contributed to the lab.
- Your group co-ordination and management.

### 4.2 Individual Component of the Report

In the **individual** section of your report, you should discuss aspects such as:

- The design of your enhancements, including any changes (and additions) to the data structures and ADTs you had to make when enhancing your group's base implementation.
- The efficiency of your enhancements, such as the efficiency of your use of data structures.
- Limitations and issues that you encountered with your group's base implementation.
- The overall quality of your individual work.

> **!** *Analysis* is about breaking down your work into its parts and then determining how those parts:
> - are necessary and serve a purpose
> - interrelate with each other
> - affect the structure and performance of your program

## 4.3 Demonstration

During Week 14, your group will demonstrate and discuss your `Azul` program to your tutor and/or lecturer. In your demonstration you should:

- Demonstrate your base `Azul` gameplay implementation by running the program.
- Demonstrate how your test cases prove your implementation is correct
- Discuss the design and efficiency of your software

Additionally, each individual will demonstrate their **individual** enhancements by running their program.

- Each individual student will be required to make a short demonstration.
- They should not be interrupted or assisted by other students during this time.

Each presentation will be 20 minutes long. Make sure you prepare for the demonstration and have a plan of what you want to show. It is up to your group to decide how to best conduct this presentation. The purpose of the presentation is to demonstrate and convince the assessor of the quality of your group's software, each individual's enhancements and the quality of your overall work. In particular for your group work, you are marked on how well you demonstrate what your group did, how easy it is to understand your group's work, and how honest you are about limitations and issues your group encountered.

# 5 Managing Group Work

! Having effective group work will be **critical** to the success of your group and reducing your stress levels.

This group assignment will (most likely) be conducted entirely online, without you ever meeting your group members face-to-face. This isn't a problem. What changes is the way you (and your group) must manage yourselves and work together. The challenge for you is not using online apps, it is using those apps *effectively*.

You will need to **make extra efforts** and be **very dedicated and diligent** in working with your team members. This will include setting up dedicated times for meetings, group programming session, and even just hanging out.

! This 5 Minute Video from the Minute Physics YouTube channel contains a number of really good suggestions for how to work effectively as a team from home.

## 5.1 Group Work Tools

To help manage your group work, and demonstrate that you are consistently contributing to your group, we are going to require you to use a set of tools[3]. Your first group update will including setting-up these tools.

### 5.1.1 MS Teams

Each group will be required to create a team on the RMIT MS Teams platform. Your group must **add your tutor** to your MS team. You may also set up various channels to discuss aspects of the assignment.

Your MS team will be the *only official* communication platform for the assignment. This means you must:

- Keep the *weekly tutor update spreadsheet* in the Files section of the "General" channel.
- Only us the MS Team channels for group chats
- Hold all team meeting through MS Teams
- Record all team meetings

If there are *disputes* over group work, we will use the record on MS Teams as the source of evidence.

### 5.1.2 Git Repository

Your group must have a central *private Git repository* that hosts your group's code for the assignment. This may be on BitBucket or Github. Your group must **add your tutor** to this repository. This git repository will be used as the *evidence of your individual contribution* to your group. Therefore your commits will be used as the evidence of your work. Git has officially been taught as a tool in this course (see the Week 5 Echo360 videos), therefore there is no excuse for having insufficient individual contributions recorded in Git[4].

For the individual component you may make either a new branch or fork of this repository before commencing your individual work.

### 5.1.3 Optional Tools

These tools may optionally be used by your group. However, they must be linked to your group's MS Team.

- Trello, for task allocation and management
- MS Planner, to layout weekly work

## 5.2 Weekly Progress Updates

Every week during your lab, you will update your tutor on your group's progress. You can discuss:

[3]These requirements are different from previous year's. However, we know working fully online will be a new experience, so we are putting some very clear rules in place.

[4]"A sufficient contribution" does not necessarily mean a large number of commits. It means you have contributed a sufficient amount of work to the group, and this work is evidenced by commits **you** have made.

1. The stage of implementation that your group has achieved
2. Your individual progress and contributions
3. Your software design and implementation
4. Any issues that have arisen

The final grade for your group-work will be *informed* by these notes, however, the grade will not be decided until the demonstration in Week 14.

## 5.3 Weekly Update Record

Part of this update will the weekly progress record. This is a spreadsheet that will officially record you groups weekly progress. It will be stored in your group's MS team. A template for this record is provided on Canvas.

**Before** your update each week, you (and each member of your group) will need to fill-in this record with:

- Your contributions for the week.
- Any issues that your have encountered through the week.

During the update with your tutor they may add additional notes to the record.

The record also tracks if your group is on-track with the assignment. Each week there are a series of tasks which your group should complete **by the Friday of that week**[5]. Your tutor note if your group is:

- Ahead of schedule
- On-track
- Falling behind schedule
- Well-behind schedule

Additionally, as we are using Git, in the update for week `N`, your group can show what you achieved by Friday of week `(N-1)`, which is especially useful for labs that are earlier in the week.

If you are unable to attend an update, then you must still fill in the record, and this will be used as your update for that week. You should review any comments from your tutor.

The tasks for each week are flexible if your group does run into issues, such as a student being sick. If you **do tell us of issues** you can negotiate with your tutor for revising the weekly tasks so that you still finish on-time. Note your tutor **won't grant an extension**, instead, the tasks will be **re-arranged**.

## 5.4 Notifying of Issues

If there are **any issues** that affect your work, such as if you are unable to contribute for some weeks (eg. being sick), you **must keep your group informed** in a timely manner. Your final grade is determined by you (and your group's) work over the entire period of the assignment. We will do our best to treat everybody fairly, and account for times when issues arise when marking your group work and contributions during the demonstrations in Week 14. However, you must be upfront and communicate with us.

! You **must** keep both your group and your tutor informed of any reasons that you are unable to contribute to your group, in a **timely fashion**, that is, **as soon as you are able**. If you fail to inform us of issues and we deem that your actions significantly harm your group's performance, we may award you a reduced grade. It is academic misconduct if you are *deliberately dishonest*, or otherwise *lie to and mislead* your group or teaching staff in a way that significantly harms your group.

[5]As labs are held on different days, there will be different expectations for each day. Friday labs will need to be mostly finished. Wednesday classes should be well on track to finishing. Monday classes will need to show they have started the week's task, and have a suitable plan to finish.

## 6 Writing and Sharing Tests within your Lab

A big challenge in this project is making sure your software is *fully functional and correct.* To do this you will need an ability to test your program. There are many ways of going about this. For the purposes of this assignment we will use a similar black-box testing to assignment 1. However, instead of using standard I/O, we will use file I/O through the save/load functionality.

Still, writing a lot of tests will be hard work. Thus to help, you are going to be writing tests **with your labs**. By the end of this sharing process, your lab should have a pretty comprehensive set of tests. Your group will be marked on how well they contribute to the sharing of tests. To do this, however, your lab will need to decide on a format for the save-game file. Your group's `Azul` program should then both *load* a game from this format, and *save* a game using this format. Note that because of this, the tests for every lab will be different.

! During **Weeks 7 & 8**, your lab will decide on a format for the save-game file. From Week 8 onwards, your group will be sharing your test cases with other groups in your tutorial!

Once the saved-game format is determined, a single test is then 3 files:

- An *input* file to load a game.
- A file with a command (or series of commands) representing the user(s) making moves in the game.
- An *expected output* file which is the game state once the commands are run and the game state saved.

The file of commands to run, can be given through standard input. it might look something like below:

```
2
test.input
turn 2 B 3
save test.output
```

Of course, the commands might be slightly different depending on what your group does. This means as you share tests with your lab, you might need to adjust the commands file so it works with your group.

To help your lab make the saved-game file format, you might want to think about representing things such as:

- Factories, including the centre factory. Note factories might be empty.
- Mosaics, including the storage grid, partially-filled grid, and broken tiles.
- Tile Bag contents.
- Box Lid contents.
- Player details, including name, and score.
- Current player
- Ability to provide a comment about the file (such as at the very top of the file)

## 7 Getting Started

This assignment requires you and your group to make a number of design choices including:

- Where to use linked lists, vectors and arrays
- How to create ADTs
- The format for getting input from the user

It is up to your group to determine the "best" way to implementing the program. There isn't necessarily a single "right" way to go about the implementation. The challenge in this assignment is mostly about software design, not necessarily the actual gameplay.

It is **very important** that you think about these ADTs early in your work. In an early group update you will need to show your tutor the classes and ADT interface methods that your group would plans to use for the assignment. These ADT methods should be detailed enough so that your tutor can see how your group can develop a base implementation of `Azul`.

Since you may not have designed many pieces of software before, to help you get started, we recommend that you think about ADTs for the following aspects:

- `Game` - This store all of the state of a game of Azul. You will need to think about where you load and save a game state. This could be inside the Game ADT, or another part of the code that uses methods of the Game ADT.
- `Factories` - An ADT within `Game` that stores all information about the factories. You should think about if you need a further AFT for an individual factory.
- `Mosaic` - An ADT within `Game` that stores *one* mosaic for a player, including the storage rows, mosaic grid and broken tiles.

You can get help about your ideas and progress through the lab updates. This is where you can bring your ideas to your tutor and ask them what they think. They will give some and ideas for your progress.

# 8 Submission & Marking

To submit, follow the instructions on Canvas for Assignment 2.

## 8.1 Notes on Marking

The marking rubric is available on Canvas. You notice there are not many marks for "trying" or just "getting started". This is because this is an *advanced* course. You need to make *significant* progress on solving the task in this assignment before you get any marks. Additionally, the HD+ category is reserved for the absolute best programs and groups. If you are aiming for full marks, your assignment needs to be impressive, and one of the best in the course.

The purpose of this is for you to focus on successfully completing each Milestone *one-at-a-time*.

## 8.2 Late Submissions & Extensions

For Milestone 1, the late submission penalty is built into the marking rubric.

For Milestone 2, late submission accrue a penalty of 10% per day up to 3 days, after which you will lose ALL the assignment marks. This penalty is applied to the full mark of the assignment, NOT the individual component.

Extensions will not be given for this assignment.

## 8.3 Special Consideration

! In addition to special consideration processes, you are ***required*** to keep your group and tutor informed of when you are unable to contribute, ***in a timely manner***. If you apply for special consideration, but do not notify of issues before doing so, then you have *failed* to keep everybody informed.

Where special consideration is granted, generally it is awarded on an individual basis, not for the entire group.

Extensions of time will not be granted, instead an *equivalent assessment* will be conducted which may take the form of a written or practical test. Further, to ensure equivalence of assessment, this equivalent assessment will only count towards the non-group components of the Assignment 2 rubric. The group-work component will be assessed based on your group participation for the duration of Assignment 2 for which you were unaffected and able to contribute. This will *take into consideration* how well you *kept your group informed* of your ability to work and contribute to the group.

## 8.4 Group Work Penalties

In severe cases of poor group work, we may undertake the following actions:

- We may apply an individual mark to any or all categories of the rubric, rather than using one mark for the whole group. This may include a reduced individual grade.
- If we determine that a student has been *deceitful* or *untruthful* in a manner than has a severe adverse academic impact on the other students in the group, either wilfully or by ignorance, we may file a charge of academic misconduct.

! Be aware that:

- All activity on MS Teams, Github and group collaboration tools
- Weekly Group updates

are **official records** and will be used as evidence of your group participation for the purposes of academic judgement and marking.