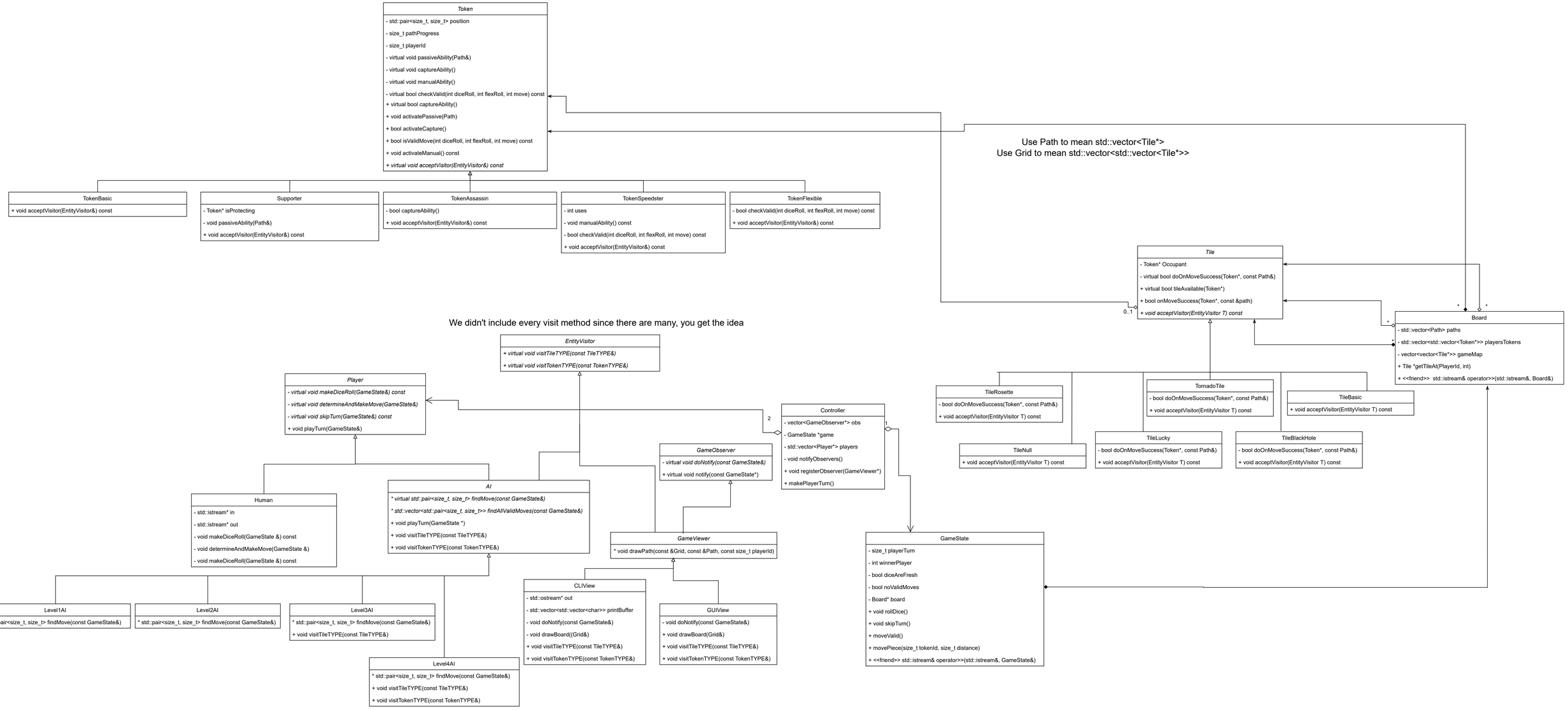
Token
- std::pair<size_t, size_t> position
- size_t pathProgress
- size_t playerId
- virtual void passiveAbility(Path&)
- virtual void captureAbility()
- virtual void manualAbility()
- virtual bool checkValid(int diceRoll, int flexRoll, int move) const
+ virtual bool captureAbility()
+ void activatePassive(Path)
+ bool activateCapture()
+ bool isValidMove(int diceRoll, int flexRoll, int move) const
+ void activateManual() const
+ virtual void acceptVisitor(EntityVisitor&) const
TokenBasic
+ void acceptVisitor(EntityVisitor&) const
Supporter
- Token* isProtecting
- void passiveAbility(Path&)
+ void acceptVisitor(EntityVisitor&) const
TokenAssassin
- bool captureAbility()
+ void acceptVisitor(EntityVisitor&) const
TokenSpeedster
- int uses
- void manualAbility() const
- bool checkValid(int diceRoll, int flexRoll, int move) const
+ void acceptVisitor(EntityVisitor&) const
TokenFlexible
- bool checkValid(int diceRoll, int flexRoll, int move) const
+ void acceptVisitor(EntityVisitor&) const
Use Path to mean std::vector<Tile*>
Use Grid to mean std::vector<std::vector<Tile*>>
Tile
- Token* Occupant
- virtual bool doOnMoveSuccess(Token*, const Path&)
+ virtual bool tileAvailable(Token*)
+ bool onMoveSuccess(Token*, const &path)
+ void acceptVisitor(EntityVisitor T) const
0..1
Board
- std::vector<Path> paths
- std::vector<std::vector<Token*>> playersTokens
- vector<vector<Tile*>> gameMap
+ Tile *getTileAt(PlayerId, int)
+ <<friend>> std::istream& operator>>(std::istream&, Board&)
TileRosette
- bool doOnMoveSuccess(Token*, const Path&)
+ void acceptVisitor(EntityVisitor T) const
TornadoTile
- bool doOnMoveSuccess(Token*, const Path&)
+ void acceptVisitor(EntityVisitor T) const
TileBasic
+ void acceptVisitor(EntityVisitor T) const
TileNull
+ void acceptVisitor(EntityVisitor T) const
TileLucky
- bool doOnMoveSuccess(Token*, const Path&)
+ void acceptVisitor(EntityVisitor T) const
TileBlackHole
- bool doOnMoveSuccess(Token*, const Path&)
+ void acceptVisitor(EntityVisitor T) const
We didn't include every visit method since there are many, you get the idea
EntityVisitor
+ virtual void visitTileTYPE(const TileTYPE&)
+ virtual void visitTokenTYPE(const TokenTYPE&)
Player
- virtual void makeDiceRoll(GameState&) const
- virtual void determineAndMakeMove(GameState&)
- virtual void skipTurn(GameState&) const
+ void playTurn(GameState&)
2
Controller
- vector<GameObserver*> obs
- GameState *game
- std::vector<Player*> players
- void notifyObservers()
+ void registerObserver(GameViewer*)
+ makePlayerTurn()
1
GameObserver
- virtual void doNotify(const GameState&)
+ virtual void notify(const GameState*)
Human
- std::istream* in
- std::istream* out
- void makeDiceRoll(GameState &) const
- void determineAndMakeMove(GameState &)
- void makeDiceRoll(GameState &) const
AI
* virtual std::pair<size_t, size_t> findMove(const GameState&)
* std::vector<std::pair<size_t, size_t>> findAllValidMoves(const GameState&)
+ void playTurn(GameState *)
+ void visitTileTYPE(const TileTYPE&)
+ void visitTokenTYPE(const TokenTYPE&)
GameViewer
* void drawPath(const &Grid, const &Path, const size_t playerId)
GameState
- size_t playerTurn
- int winnerPlayer
- bool diceAreFresh
- bool noValidMoves
- Board* board
+ void rollDice()
+ void skipTurn()
+ moveValid()
+ movePiece(size_t tokenId, size_t distance)
+ <<friend>> std::istream& operator>>(std::istream&, GameState&)
Level1AI
* std::pair<size_t, size_t> findMove(const GameState&)
Level2AI
* std::pair<size_t, size_t> findMove(const GameState&)
Level3AI
* std::pair<size_t, size_t> findMove(const GameState&)
+ void visitTileTYPE(const TileTYPE&)
Level4AI
* std::pair<size_t, size_t> findMove(const GameState&)
+ void visitTileTYPE(const TileTYPE&)
+ void visitTokenTYPE(const TokenTYPE&)
CLIView
- std::ostream* out
- std::vector<std::vector<char>> printBuffer
- void doNotify(const GameState&)
- void drawBoard((Grid&)
+ void visitTileTYPE(const TileTYPE&)
+ void visitTokenTYPE(const TokenTYPE&)
GUIView
- void doNotify(const GameState&)
+ void drawBoard(Grid&)
+ void visitTileTYPE(const TileTYPE&)
+ void visitTokenTYPE(const TokenTYPE&)

# Final Design Document: UrPlusPlus

*Umut Emre, Carol Xu, Chris Fang*

## Introduction

The Royal Game of Ur is the oldest known board game dating back to 2600 BC. Although it is still unknown what the original ruleset truly was, we have created a modern adaptation of a widely accepted ruleset; alongside a multitude of additional features to spice up the game!

## Overview

### Game Rules

As a reminder, we will first quickly cover the basic rules of the game. Each player starts with an equal number of tokens off the board. The objective is to race all tokens across the board before your opponent. Tokens are moved one by one, according to the total of four two-sided dice with values 0/1. Paths will overlap, and unless otherwise stated, moving onto an opponent token will send the token back off the board. Some specifics are:

- Only one token can occupy a space at one time.
- After a roll, a player must make a move if possible.
- To complete a path, a player must move a token exactly one past the last tile of the board. Rolling over will result in an invalid move.
- Rolling a zero or having no invalid moves will pass the turn to the next player.

### Special Tiles

As required, all 3 special tiles - alongside the classic Rosette in the base game - are implemented from the proposal. This includes:

- **Rosettes:** Upon a token landing on, grants another turn to the team of that token. Also protects the occupying token from capture.
- **Black Hole:** Upon landing on, send token off the board to the start.
- **Lucky Tile:** Upon landing on, there is a small chance the token is sent off the board to the start, and a small chance the token is sent off the board to the finish. This chance is equivalent to rolling the same value on all four dice. (1/16 for each event).
- **Tornado Tile:** Upon landing on, there is a 50% chance the token is blown forward and a 50% chance the token is blown backwards. A “blown” token moves along its path until reaching either an empty tile or the start/end of a path. If the empty tile is itself a special tile, the tile ability is not activated for gameplay clarity; but this is purposely implemented to be easily extendable if desired.

## Special Tokens

Initially, we intended to implement 3 tokens as required - each of which are the favorite of a group member. These are:

- **The Assassin** - Gains another turn when an enemy token is captured
- **The Flexible** - Has access to an alternative dice roll when making its move
- **The Speedster** - Has a once-per-game ability to move one more than the regular dice roll

On top of the required 3 tokens, we decided to add **the supporter** - a token that protects the first allied token within 3 tiles in the forward direction. This token ended up being the trickiest to implement, which we will cover in more detail in the **Passive abilities** section under Game State and Game Flow, and under **Extra Credit Features**.

## Input

Although the original game of Ur is played on a set board, we allow for custom boards (with custom tiles), custom paths, and custom tokens. Further, the game can be loaded and continued from any point in time!

The input we decided on is of the form:

*Two integers specifying the height (h) and width (w) of the board, respectively*

*A (h) x (w) block of characters specifying the tiles on a board. O represents a regular tile, B represents a black hole tile, L represents a lucky tile, T represents a tornado tile, X represents a Null tile (wall), * represents a rosette.*

*For each player, two integers representing the starting row and column respectively of the path. Then a series of characters N, E, S, W to indicate the direction to move next in the path. (North, East, South, West respectively)*

*An integer specifying the number of tokens (n) each player starts with*

*For each player, A sequence of (n) tokens, each of the form (tokenType) (row) (column). Starting at the beginning is denoted with row = column = -1, and starting in the completed state is denoted with row = column = -2. tokenType is defined as follows: A is an assassin, B is a basic token, F is a flexible token, S is a Speedster, G is a supporter token.*
***Additionally**: Speedster must take in an additional fourth parameter, the number of uses of the speedster ability.*

Following this, type *roll* to roll the dice. To move a token, use *(<tokenName> <distance>).*

We make use of command-line flags to change the number of players and AI in the game. This allows us to have matches with any number of human players and any number of AI players.

Usage: [-AI <AI level>] [-AI <AI level>] [--nogui] essentially you can add up to two -AI flags followed by the level of difficulty (1 to 4 inclusive), and each -AI flag makes one of the players an AI. The nogui flag makes it CLI only. For example, `./UrPlusPlus -AI 1 -AI 2` launches the GUI and is a game between AI level 1 and AI level 2, `` `./UrPlusPlus -AI 4 --nogui` `` is a CLI-only game between a human player interacting with the command line and a level 4 AI. Note that the GUI displays the game but a human player must enter their moves and actions as prompted in the terminal.

Here is sample input to start a game:

```
3 8
*OOOXX*O
BOT*LOLT
*OOOXX*O
0 3
WWWSEEEEEEENW
2 3
WWWNEEEEEEESW
4
B -1 -1
A -1 -1
S -1 -1 1
A -1 -1
B -1 -1
A -1 -1
S -1 -1 1
A -1 -1
0
```

And here is sample input for a game in progress:

```
3 8
*OOOXX*O
BOT*LOLT
*OOOXX*O
0 3
WWWSEEEEEEENW
2 3
WWWNEEEEEEESW
4
B -1 -1
A -1 -1
S -1 -1 1
A -1 -1
B -1 -1
A -1 -1
S -1 -1 1
A -1 -1
1
```

## AI

As required, we implemented the three AI levels:

- **Level 1** picks a random available move.
- **Level 2** prioritizes finishing tokens and capturing enemy tokens.
- **Level 3** (along with developments in level 2) also considers tile effects.

For extra credit, a **level 4** was added. Alongside level 3 improvements, token abilities are factored into the calculation with sophisticated expected distance tests. For more details, please see the AI section under “Extra Credit Features” or refer to the file level4ai.cc

## CLI and GUI

We implemented both a CLI and GUI display that visualizes the game at any turn. Human players always type their input through the terminal.

# Architecture Overview

## Model controller view architecture

As a guiding principle we followed a model controller view architectural pattern for the game. We abstracted the actual game state (rules for how the state can be atomically modified through player moves, rules for parsing valid states) as the model, and we abstracted a controller which handles game flow. The controller is responsible for requesting input from either human or AI players and conveying the input to GameState. The aim was to have minimal coupling between these three abstractions beyond the necessary interfaces by which they communicate.

As discussed in following sections we used a number of OOP design patterns to implement the game logic as well as to realize the model controller view architecture (observer and visitor design patterns namely).

Another general guiding principle we used was to embed the rules and game logic as much as possible into tokens and tiles to allow us to easily add different tiles and tokens with interesting properties. For example, a token should know, given a potential amount of distance to move and the values of the dice roll for a turn, whether it is a valid move (see Token::isValidMove in Token.cc, and see the same method in its derived class implementations, namely TokenSpeedster.cc has a different implementation since a speedster token may move one greater than the amount shown on the dice once per game). Similarly, see in Tile.cc the methods Tile::tileAvailable and Tile::onMoveSuccess, and the overridden implementations in the children TileRosette or TileTornado, these functions dictate whether its possible to land on the token and any side effects of landing on it.

Then, in gamestate.cc you can see GameState::movePiece, which consults tokens and tiles and the board to see if an attempted move is valid by calling GameState::moveValid, and then carries out the move, and invokes any side effects by calling methods on the tokens and tiles - all done polymorphically so GameState is agnostic to what types of Tiles and Tokens are

actually on the board. Following this principle means that we can trivially implement new types of Tiles and Tokens with a wide variety of different side effects and behaviors.

Finally, we followed the NVI Idiom. All abstract base classes (Player, Token, Tile, GameObserver, EntityVisitor, etc.) were designed to provide a non-virtual public interface, with derived classes implementing inherited virtual private functions, wherever it made sense (see Token and its derived classes for an example). We made use of multiple inheritance over creating more public methods when we needed a derived class with a public interface that requires extra functionality (see ai.h and gameviewer.h for examples of where a class needed to make use of multiple inheritance).

As a note, the Token abstract base class can be found under the files token.h and token.cc. Derived classes of Token can be found under any header and implementation files with the word `token` followed by the class of the token. Similarly, the Tile abstract base class and derived classes can be found under files with the word `tile`.

# Design Considerations

## GameViewer

The view abstraction in our game is meant to be an observer of a controller, and it takes a const reference to a game state so it can display it whatever way it wants. Namely, we have a CLI and a GUI for displaying the game state that inherits the interface of our GameViewer abstract class. This makes use of both the observer pattern and the visitor pattern (which we nicknamed the “voyeur” pattern) to gather information from the GameState.

## Game State and Game Flow

One of the major goals of the implementation was to make our entire application entirely stateless, in the sense that we should be able to load any game state and begin playing from any point. Doing so imposed the challenge that every action and turn must be calculated and handled solely from the previous game state. This approach enables this application to be integrated with some kind of network layer for easy online multiplayer.

This mainly caused complications when combined with the various timings that were required; especially with token and tile abilities. We ultimately chose to split these such timings into 4 different categories, which are generalizable to future tokens/tile additions.

**Passive abilities** (e.g. Supporter ability) need to be calculated when the board is loaded in, and must be refreshed for every subsequent turn, action, or other board event. This must occur for Tokens owned by all players, as passive abilities mainly grant benefits during opponent turns. Further, since we want to be able to load any game state, additional cases were considered and

handled such as ensuring passive abilities were properly updated when loading a game which had already made a dice roll but not a move (and vice versa).

**Active/DiceRoll abilities** (e.g. Speedster, Flexible) need to interact prior to or with the dice roll. In the case of one-time use abilities like Speedster, additional trackers needed to be updated within the token. One particularly tricky situation to handle was abilities influencing dice rolls, since the user must not be prompted to make a move when no valid moves exist. That means we needed to check every possibility for ability usage before prompting the user for a decision.

**Capture abilities** (e.g. Assassin) need to activate when capturing an enemy token. The main challenge ended up not necessarily being with the capture timing itself, but rather storing and checking whether or not a player had gained an extra turn. This had also had to be handled in a way where the information was completely abstracted from the controller- so it was solved through a multitude of actions and checks performed between the player/AI and the game state class.

Finally, **Tile effects** activate last and occur when a piece lands on a tile and all timings above finish processing. Aside from also potentially adding a turn, tile effects also often manipulate token positions, which could also potentially end a game.

All of these timing decisions ultimately allowed us to handle each “type” of ability independently in a sequence, while preserving proper storage and updating of the GameState one timing at a time.

## Paths

Paths were originally thought of as a simple wrapper we could make on top of a vector, but upon further investigation turned out this didn't make as much sense as almost all of the interface and functionality we wanted would be identical to the interface of a vector. We ultimately opted to expose a game state's "paths" as const vectors to maintain encapsulation while eliminating an unnecessary layer of abstraction. (Which differs from the UML)

The second major consideration was whether or not to have start and end tiles as specific tiles. After switching path to a vector, the path indices would automatically give the start and end tiles, so that was also removed from the UML.

Finally, for gameplay design, we note a couple of extra restrictions that are enforced:

- Paths should not revisit the same tile.
- Paths cannot contain the start tile of another path.

## EntityVisitor and AI classes

In our original design, we did not have the EntityVisitor class. Each Tile and Token had a `draw` method in the original design. Upon consideration, we realized implementing the visitor design

pattern over top of Tokens and Tiles would lend itself well for CLIView and GUIView (drawing and printing the board). In addition, this design pattern ended up being very handy when implementing AI which needs to scan the whole board and make decisions based on the concrete types of Tiles and Tokens. Since AI needs both the functionality of EntityVisitor and Player but Human does not need to parse the game state, we made the decision to have AI inherit both from Player and EntityVisitor.

### GameObserver

The GameObserver class was also not in our UML design. It was created in the later stages of our project as we realized we needed to make abstractions for functions that GUIView and CLIView share, but also keep the observer pattern implementation separate from these functions to increase cohesion. It should be noted that the descendants of this class are not strictly observers as we defined them (they don't have any field that references what they observe), instead their notify method has an argument that contains the state object it cares about.

## Resilience to Change

Throughout our project, we always made sure to consider: “is this generalizable?” before making a change. Many of our design decisions were driven by considering possible changes. These design considerations include:

- Having superclasses for commonly-used collections of features (Player, EntityVisitor, Token, Tile) that provide a mostly-fixed static interface for descendants allows for generalizability and Liskov substitution.
- Tokens and Tiles provide a public interface for different classes of abilities and GameState handles the order of execution of all of them. This means that new Tokens and Tiles can be added easily without changing GameState logic. This also means that Tokens and Tiles are not limited to one class of ability each - for instance, if we wanted to implement a Token with a passive ability, a manually activated ability, and a capture ability, we would not have to change the implementation beyond simply adding a new Token type.
- Easily generalizable to any number of AI/Players - right now our game only supports 2, but we designed our classes and chose data structures with multiplayer in mind so that it would be natural to add this functionality if we so wished. Considerations included choices such as using a vector to represent the players and allowing our GameViewer classes to take any number of players, using a vector of playerIds to represent turn order (also allowing us to implement custom turn orders in the future, if needed), and allowing the Board class to hold any number of paths on one Board.
- Being easily extended to being completely stateless would enable this application to be integrated with some kind of network layer for actual online multiplayer.
- Support for variable numbers of ability usage, for instance, a user can determine whether a token can use a limited-use ability 3 times per game or stick with the default.

Our project specification was not given any additional questions after DD2, so we will skip the “Answers to Questions” section. We answer the final questions later in the document.

# Extra Credit Features

1. As discussed in our project specs, three types of special tokens were needed. However, we added an additional special token: the Supporter token (see supporter.cc and supporter.h). This token implements a passive ability, which was a challenge as a passive ability could activate or deactivate upon any event on the Board. This was the source of many design considerations and bugs (e.g. we needed to carefully choose when to refresh this ability in GameState; some sources of bugs included not reversing a passive ability’s effects when the token with that ability was taken off the board). However, we implemented it regardless, and adding this new class of Token ability made us really consider how to make our design easily extendable to include tokens with many different types of abilities without much modification.
2. All memory management is handled using the RAII idiom, namely using STL containers and smart pointers. Any raw pointer in our codebase is strictly a non-owning pointer, and ownership is only ever implied through owning a unique pointer. Also the only thing that leaks is SDL so all is well.
3. The sophisticated AI Level 4 (level4ai.cc and level4ai.h) has 4 main capabilities that distinguish it from Level 3. We chose to implement an AI that could use Ur strategies effectively, because we thought this would be a complex but fun task to take on - and it was. These are in the form of different strategies that we found effective while playing our game and wanted to incorporate in an AI:
    - Tornado tiles are dynamically scored based on the position of other tokens on the board and the position of Tornado itself on the board. This allows the AI to utilize strategies such as using Tornado to travel large distances on the board and/or land on an end tile.
    - Lucky tiles are dynamically scored based on their position in the board. Since Lucky tile has such a low activation rate, the benefit and cost of Lucky tiles are scaled down to the probability of the activation rate. Early Lucky tiles are slightly favored by the AI, while Lucky tiles that appear later in the path are slightly penalized.
    - This AI strongly prefers occupying a Rosette on a path shared with an opponent, and is slightly penalized for moving off of a Rosette on a shared path. (This is in contrast to the AI’s behavior when occupying Rosettes on its own path, where it is rewarded for moving both on and off a Rosette.) This is a tried-and-true strategy in the original game of Ur - occupying a shared Rosette early in the game gives a noticeable advantage to one player, as one of the player’s tokens gains invulnerability while slightly limiting the range of possible moves the opponent’s tokens can make.
    - The AI is slightly penalized for moves that result in a piece landing 1, 2, or 3 squares in front of an opponent, as those moves make up ⅞ possible dice roll outcomes. To do this, the AI looks ahead and first calculates if the token to be

moved would be protected at its new position or not. It also calculates if the token to be moved is currently protected. It then searches the tiles behind it for occupants at both positions. Based on these conditions it decides which move would be safer to make.

Other features of Level 4 AI:

- Strongly prefers using Assassin to capture other tokens over using others.
- Prefers moves that capture other tokens.
- Prefers moving tokens in front of Supporter when possible.
- Prefers choosing the higher flexible tile move unless it moves it 1, 2 or 3 squares in front of an enemy token - the opponent's most likely rolls - on a shared path.
    - (avoids doing so even not as a flexible)
- Avoids Black Hole tiles if possible.

# Final Questions

*What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?*

Chris - I learned the importance of code review- I can recall countless times edge cases being caught during pull reviews that would have otherwise been extremely difficult to catch in testing. This allowed us to have much more reliable code which ended up being crucial when integrating everything.

Carol - Everyone has their own ideas and their own schedules and you have to work around both things to be able to work as a team. And everyone brings more knowledge to the table than one person alone.

Umut - You not only have to gaslight yourself into thinking your design is sane, but you now have to gaslight others too. Thankfully, in the cases where your design is actually bad, other people will catch it and improve it. PRs, discussion and whiteboarding out the approach really helped in terms of helping us converge to a working, extensible and maintainable final product. As well, we accomplished way more together than at least I could have done on my own in the time span.

*What would you have done differently if you had the chance to start over?*

Chris - In the planning phase, I felt we tried to make too many design decisions at once rather than making one decision at a time and working around it. I'm happy with the final design we ended up with, but I think we could have potentially come to it faster. Personally I should have clarified the entire gameflow and logic of making moves before trying to implement it- it ended up taking much longer than expected because I had not.

Carol - I probably would have tried to spend more time on “designing while coding”: what I mean is implementing a little bit of code first, then going back to the drawing board and restructuring the class to incorporate the new ideas, then going back to coding. I tend to only code while coding and design before coding, which means in my design I miss a lot of the edge cases I think about during implementation. I think this would have helped me modularize my code more quickly instead of having a lot of code at the end to sift through.

Umut - I would have wanted to spend more time on the design process to flush things out, and maybe even get some more pseudocode implementations of certain things. Having to make design decisions or have design reconsiderations during implementation definitely slowed things down because sometimes that meant a lot of refactoring or game logic changing - whereas catching any of these potential bumps in the design phase saved time and it was virtually free to redesign in that phase.