Ounce of Rust Project Design Document

Rev. A draft 1

James Richey

Oct 18, 2019

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Purpose of this Document

This is the design document for the Ounce of Rust project. This document describes in detail the objectives requirements, and design considerations of the project providing a central location for this information. This is invaluable for planning the project's tasks and milestones. Anyone wishing to understand the project and resulting deliverables is encouraged to read this document.

1.2 Scope of the Project

The main deliverable of the Ounce of Rust project is a Rust crate that provides common Tic Tac Toe logic that can be used by other Rust applications. While there are existing crates that provide similar functionality, this project provides an opportunity to gain experience with Rust, its ecosystem, and general software development practices.

1.3 Overview of this Document

There are three main parts to this document.[1] The *Overview* chapter provides a general overview of the problems this project is intended to address and how the project addresses these problems. The *Requirements* chapter specifies the requirements of the project. The *Design* describes how the project's deliverables are designed to fulfill the requirements.

Additionally, the *Glossary* defines terms that are used throughout this document.

[1] The structure of this document is influenced by [Berezin-1999].

CHAPTER

TWO

OVERVIEW

2.1 Objectives of the Project

This section describes the objectives of the Ounce of Rust project.

2.1.1 Create Reusable Library to Speed Development of Tic Tac Toe Games

The result of this project is a reusable library that provides a core set of functionality that speeds development of future Tic Tac Toe games. Core functionality includes game state management and artificial intelligence algorithms. The user interface for Tic Tac Toe games is outside the objectives of this project.

2.1.2 Make the Library Open Source and Widely Available

The resulting library is released under a permissive open source license and made widely available. This includes placing the code on a public repository such as https://github.com/ and in Rust's crate registry, https://crates.io/.

2.1.3 Learn About Rust

Rust is a modern statically typed systems programming language that has a focus on safety. Its mix of high level concepts, ability to avoid entire categories of bugs, and focus on correctness has made Rust an increasingly popular language.

Even though Rust is unlikely to replace C and C++ any time soon, Rust's concepts, such as traits, error handling, and memory management systems are worth learning to help expand ones general programming knowledge.[2]

2.2 Current System

Tic Tac Toe is a game where two players, X and O, take turns placing their mark in a gird. The first player to get three marks in a row, column, or diagonal wins the game. The game can also end in a draw, known as a "cat's game". An example of a Tic Tac Toe game is shown in Figure 2.1.

[2] The Rust official documentation is likely to be helpful for meeting this objective. For details see [Rust-Docs].

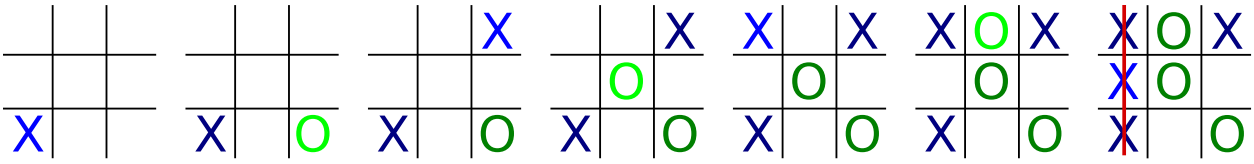

Figure 2.1: An example game of Tic Tac Toe where player X is victorious.

2.2.1 Pencil and Paper

Traditionally Tic Tac Toe is played by two players using pencil and paper. This is a quick and convenient way to play with a friend. However, this does not allow for single player games. For people stuck on an airplane this can lead to extreme boredom!

2.2.2 Computer Games

Tic Tac Toe computer games allow for single player versions of the game. There are many versions of the game created over the years. Figure 2.2 shows a screen shot from one of the classic Tic Tac Toe games.

These games can be developed as stand alone applications or can be created with the help of supporting libraries.

2.2.3 Supporting Libraries

To support the creation of Tic Tac Toe games many libraries have been developed to provide common functionality. Searching Rust's package repository, https://crates.io/, reveals several such libraries including `ultimate-ttt`, `zero_sum`, and `minimax`.

With the wide verity of libraries available, for both Rust and other languages, there is likely one that meets the needs of Tic Tac Toe application developer. However, one of this projects objectives is *Learn About Rust*. Thus it is worth going through the effort to create another Tic Tac Toe library.

2.3 Proposed System

This project creates a Rust library that application developers can use to create Tic Tac Toe games. The library contains common Tic Tac Toe functionality such as game state management and single player artificial intelligence. The library exposes this functionality through a well defined and documented public API. The library is open source and is available in Rust's package repository https://crates.io/.

By using this library application developers can focus on making flashy graphics or other unique Tic Tac Toe experience without worrying about the underlying game logic and management.

2.3.1 Game State and Board Management

A common task of Tic Tac Toe games is managing the state of the game. This includes knowing which player's turn it is, ensuring players cannot mark a previously marked square, and checking for victory or draw conditions.

[3] https://www.imaginaryphase.com/ttt.html

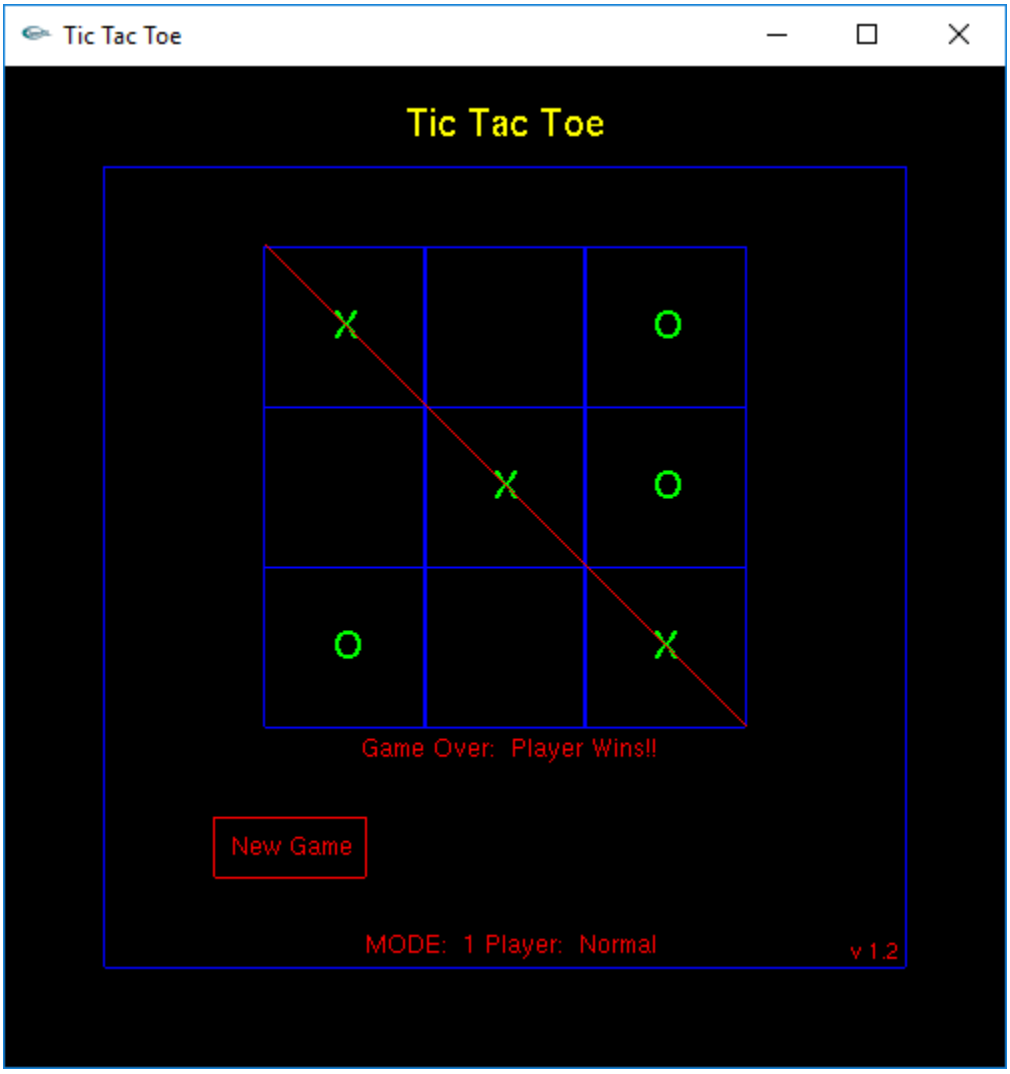

Figure 2.2: Screen shot of Tic Tac Toe[3] developed by James Richey.

2.3.2 AI Players

The library allows single player Tic Tac Toe games to be created by providing AI players. These players use artificial intelligence algorithms that pick a square to place the AI player's mark. The application developer has control over how difficult to win over the AI player.

2.3.3 Available on crates.io

The library is available on https://crates.io/, the source code released under a permissive open source license, and the API documentation is hosted on a publicly accessible website.

2.3.4 Deliverables

The project deliverables are:

- `open_ttt_lib` crate.[4]
- API documentation.[5]
- Public source code repository with a tagged release.

2.4 Interactions with Other Systems

The Tic Tac Toe library is intended to be used in other user facing applications. This includes, but is not limited to, stand alone graphical applications, command line applications, or even as a mini-game in a larger video game. The diagram in Figure 2.3 shows how the library is used by other systems.

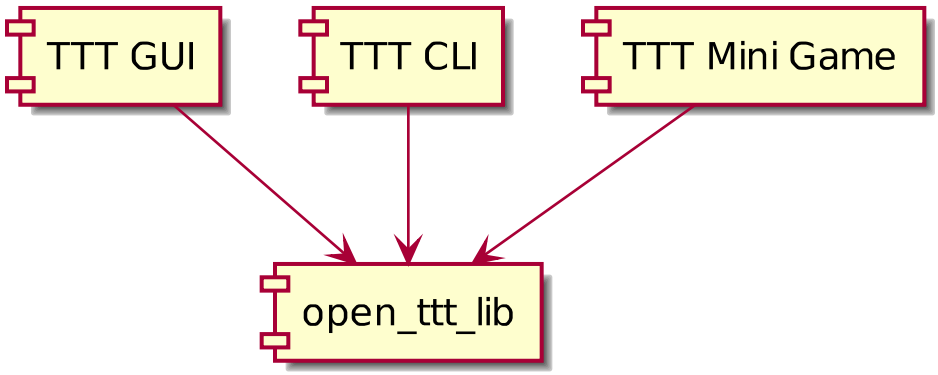

Figure 2.3: Diagram showing how the library is used by other applications.

The other applications link directly to `open_ttt_lib`. The library does not provide any ways of remote access, e.g. via a network interface and it does not save any state the computer's persistent storage.

2.5 User Roles and Responsibilities

This section describes the users of the Tic Tac Toe library.

[4] Future projects built around `open_ttt_lib` may use `open_ttt` as part of their crate name.
[5] Popular places to host Rust API documentation include https://docs.rs/ and https://pages.github.com/.

2.5.1 Rust Application Developer

The Rust application developer uses the library to create awesome Tic Tac Toe games.

Responsibilities

- Connect the library to a Tic Tac Toe user interface.
- Read the library's documentation to determine how to use the library.
- Debug the application when it does not run as expected.

2.5.2 Tic Tac Toe Player

The Tic Tac Toe player is an indirect user of the library. They use the application created by the application developer to play an exciting game of Tic Tac Toe.

Responsibilities

- Challenge a friend to a game of Tic Tac Toe.
- Attempt to win against the computer.

CHAPTER

THREE

REQUIREMENTS

This chapter describes the detailed requirements of the Tic Tac Toe library. This includes the rules for playing Tic Tac Toe and user stories describing how the library is used.

3.1 Rules of Tic Tac Toe

The rules for Tic Tac Toe are as follows:

1. Play occurs on 3 x 3 grid. The grid starts empty with no marks.
2. The first player places their mark in one of the grid's squares. Traditionally, the mark is the letter *X*.
3. The second player places their mark in one of the grid's empty squares. A square that already contains a mark cannot be updated or altered. Traditionally, the second player uses the letter *O* as their mark.
4. Turns alternate between the players until the game is over.
5. The game ends in a win when one of the players gets three of their marks in a line. That is: they have three marks in a row, column, or diagonally. Examples of winning games are shown in Figure 3.1.

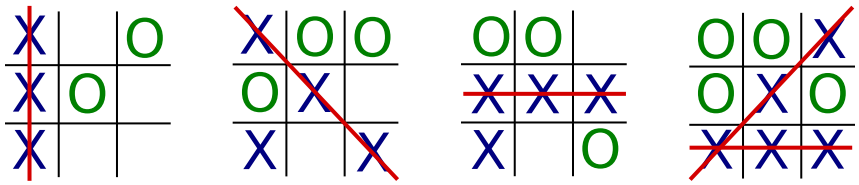

Figure 3.1: Examples of winning Tic Tac Toe games showing player X winning by getting three marks a row, diagonal, and column. The red line shows the squares that contributed to the win. Notice that it is possible to get multiple sets of three marks in a row.

6. The game ends in a draw, known as a cat's game, if no more empty squares remain and a player has failed to get three marks in a row. Examples of cat's games are shown in Figure 3.2.

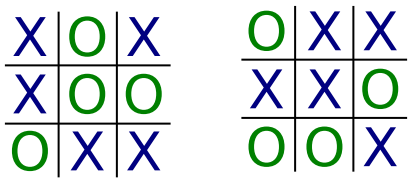

Figure 3.2: Examples of Tic Tac Toe games ending in a cats game. No player managed to get three marks in a line.

3.2 User Stories

User stories are informal descriptions of the software system features. These stories are written from the perspective of the users roles described in *User Roles and Responsibilities*. The general format is:

> As a <user> I want <goal/desire> so that <benefit>.

This section contains the user stories identified for this project.

3.2.1 Game State Management

As a Rust application developer,
I want the library to contain functionality for managing the state of the game,
so I can focus on creating engaging user experiences.

Acceptance Criteria

- The library exposes APIs for getting the current state and updating the state of the game. This includes functionality for checking the victory condition and determining the winner of the game, if any.

Notes

The design details of the APIs are outside the scope of this requirements chapter.

3.2.2 Know Cells that Contributed to Player's Victory

As a Rust application developer,
I want to know what cells contributed to the player's victory,
so I can draw a line through them or mark them in a special color.

Acceptance Criteria

- When a player has won the game there is a way to obtain the board's cells that contributed to the victory.

3.2.3 Stable Library API

As a Rust application developer,
I want the library to have a stable API,
so that my application does not unexpectedly break if I use a different version of the library.

Acceptance Criteria

- The library uses semantic versioning to clearly communicate when there are API changes.[9]
- There are integration tests that help library developers detect if the library's API changes.

[9] See https://semver.org/ for details on semantic versioning.

3.2.4 AI Player

As a Tic Tac Toe player,
I want to play against the computer,
as I do not always have a friend to play with.

Acceptance Criteria

- The library provides an AI player that Rust application developers can incorporate into their applications.

3.2.5 AI Difficulty Settings

As a Tic Tac Toe player,
I want different AI difficulty settings,
so I can play a challenging yet winnable game of Tic Tac Toe.

Acceptance Criteria

- The difficulty for AI players can be configured by the Rust application developer.

Notes

The difficulty can be thought of as a probability of how likely the AI will make a mistake.

3.2.6 Players Take Turns Having the First Move

As a Tic Tac Toe player,
I want to have the first move on the next game if I did not have the first move this game,
so I have a better chance of winning the next game.

Acceptance Criteria

- The game logic ensures the starting player alternates between games.

Notes

The player who takes the first move has more wining possibilities than the second player.[10]

3.2.7 Maximum AI Update Time

As a Rust application developer,
I want the AI to block for less than one frame when picking its next move,
so it does not block my rendering thread making my animations choppy.

[10] The player with the first move has about double the number of winning possibilities. For details see Wikipedia's Tic-tac-toe page[11].
[11] https://en.wikipedia.org/wiki/Tic-tac-toe

Acceptance Criteria

- There is a benchmark that measures the worst case time the AI blocks while picking the next move.
- How to run the benchmark is documented so developers can quickly evaluate this library to see if it meets their needs.

Notes

Frame times can vary greatly depending on platform and application. For example, an update time of one second might be just fine for a casual Tic Tac Toe game. However, a Tick Tac Toe mini-game in a modern FPS is expected to take just a fraction of the 1/144 second frame time. Therefore, providing the tools to allow the Rust application developer to see if this library meets their needs is sufficient to fulfill this requirement.

3.2.8 Getting Started Example

As a Rust application developer,
I want an example of getting started with the library,
so I can quickly start integrating the library into my application.

Acceptance Criteria

- There is a runnable example of using the library.
- The example is in a prominent location such as library's documentation.

3.2.9 Detailed Library Documentation

As a Rust application developer,
I want detailed and thorough library documentation,
so I can determine how to use the library from my specific needs.

Acceptance Criteria

- All public modules and their members are documented using Rust's documentation comments.
- The documentation contains the typical sections such as **Panics** and **Errors**.
- The documentation is accessible from the internet, such as being hosted on Docs.rs[6].

3.2.10 Idiomatic Rust APIs

As a Rust application developer,
I want the library to provide idiomatic Rust APIs,
so the library works in natural and familiar way.

[6] https://docs.rs

Acceptance Criteria

- The Rust API Guidelines are consulted when designing the libraries API.[12]
- An experienced Rust programmer code reviews and signs off on the library's API.

Notes

This can be a subjective subject subject. However, providing an idiomatic Rust API is important to fulfilling the *Learn About Rust* objective.

3.2.11 Cross Platform Support

As a Rust application developer,
I want the library to work on a variety of platforms,
so I can make Tic Tac Toe games for a wider use base.

Acceptance Criteria

- The library is tested and verified on two different platforms such as Windows 10 and Linux.

Notes

The use of platform specific code is minimized, however, the number of platforms the library is tested on may be limited due to resource constraints.

3.2.12 Available on crates.io

As a Rust application developer,
I want the library to be on crates.io[7],
so that I can easily incorporate it into my Rust based application with Cargo.

Acceptance Criteria

- The library can be downloaded from crates.io.
- The library can be obtained by simply specifying it as a dependency in Cargo.toml.

3.2.13 Source Available on GitHub

As a Rust application developer,
I want the library's source code to be available on GitHub[8]
so I can view the source code to get a better understanding of how the library works.

[12] See the [Rust-API-Guidelines] for details.
[7] https://crates.io/
[8] https://github.com/

Acceptance Criteria

- The library's source code is hosted on a public GitHub repository.
- The library's tags match the releases on crates.io.

3.2.14 Permissive License

As a Rust application developer,
I want the library to be licensed under a permissive open source license,
so that I can incorporate the library into my application without worrying about legal issues.

Acceptance Criteria

- The library is released under a permissive open source license. The MIT license fulfills this requirement.

CHAPTER

FOUR

DESIGN

4.1 Public API

This section describes the public API of the library. The provided types and functions are used by other applications to create Tic Tac Toe games. The legend shown in Figure 4.1 is used for the type diagrams in this section.

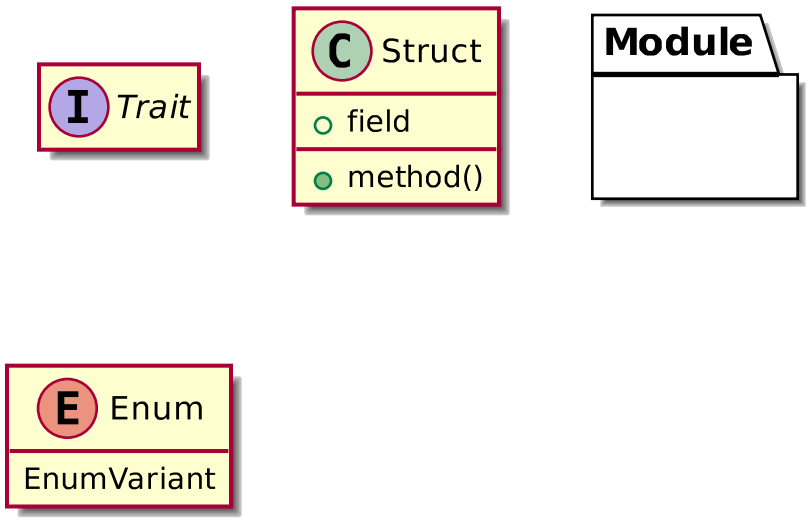

Figure 4.1: Legend used for the type diagrams in this section.

An overview of the major public types is shown in Figure 4.2.

The library contains a single public module that holds the public types. The naming conventions used in this library follow those described in the Rust API Guidelines[13] per the *Idiomatic Rust APIs* user story.

Each of the major and supporting types are described below.

4.1.1 Game Management

Game management is handled by the Game structure. This structure is one of the central types provided by the crate. It contains the state machine logic and holds the underlying game board. Figure 4.3 shows the Game structure along with the GameState enumeration.

A state machine to determine which player has the next move or when the game is over. The state diagram is shown in Figure 4.4.

[13] See the [Rust-API-Guidelines] for details.

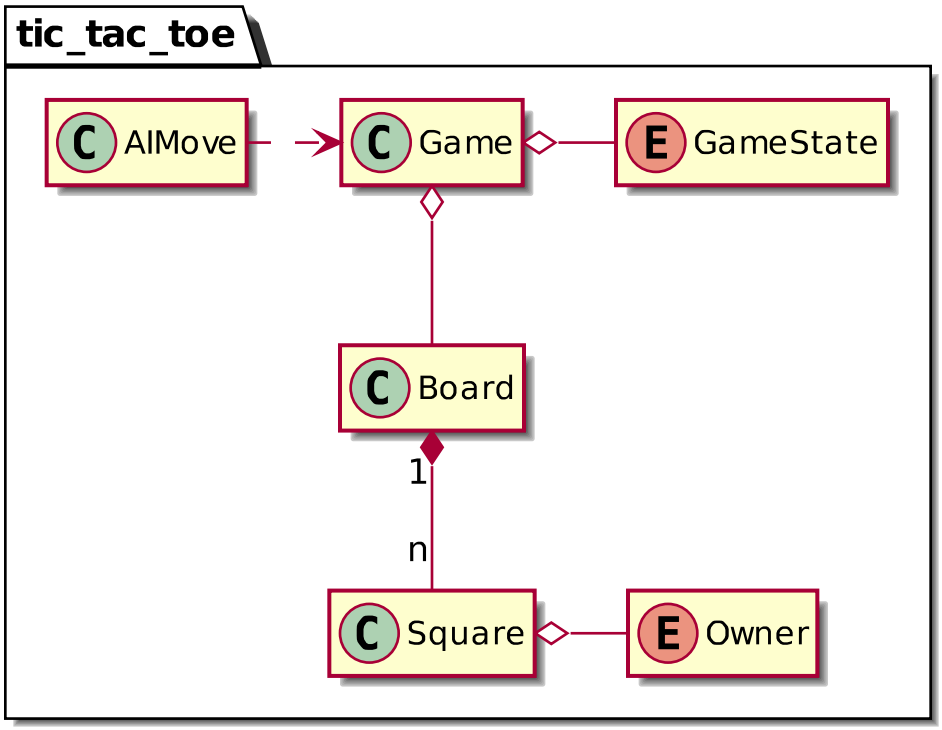

Figure 4.2: Major public modules, structures, and other types. Note: the module contains additional supporting types.

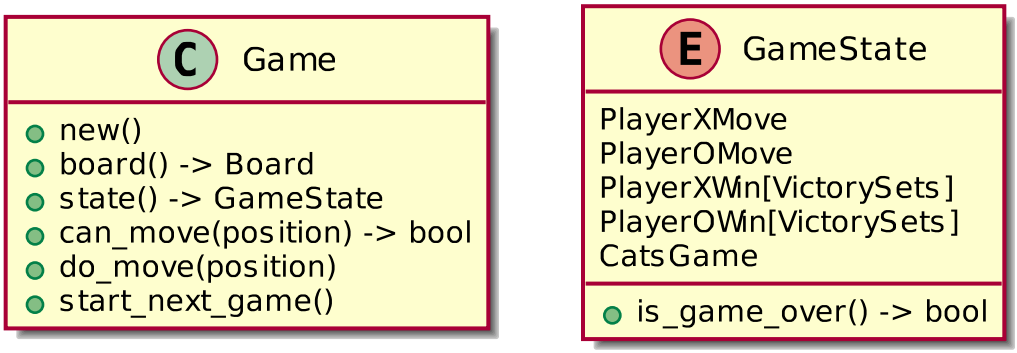

Figure 4.3: The Game structure and GameState enumeration.

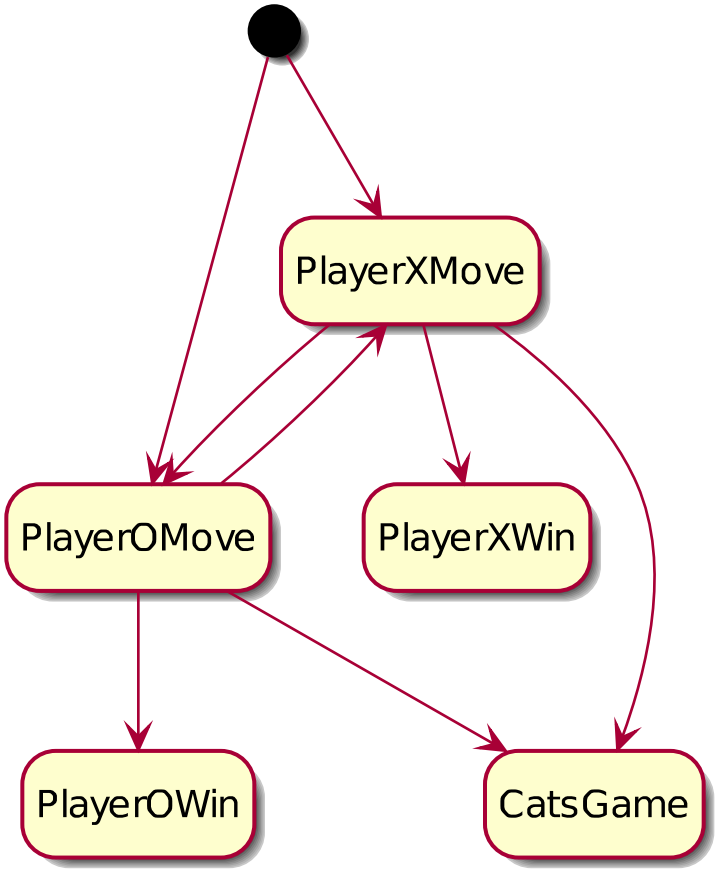

Figure 4.4: State diagram of a Tic Tac Toe game.

When a new game starts either player X or player O takes the first turn. The players alternate making their moves until one of the end game conditions is encountered. The player that did not have the first turn last game takes the first turn next game.

Struct Game

Members of the Game structure are as follows:

new() Creates a new Tic Tac Toe game.

board() Gets the board associated with the game.

state() Gets the current state of the game.

can_move() Indicates if the square at the indicated position can be marked as owned. That is, if `can_move()` returns `true` then `do_move()` is guaranteed to not panic.

do_move() Marks the indicated square as being owned by the current player. The state of the game is updated as a side effect of `do_move()`. Panics if the indicated position is already owned or if the game is over.

start_next_game() Starts the next game by resetting the state machine ensuring the player who went second last game goes first next game.

Trait Implementations

- Clone[14]

Related Requirements

- *Rules of Tic Tac Toe*
- *Game State Management*
- *Players Take Turns Having the First Move*

Enum GameSate

The game state enumeration contains a variant for each possible game state described in Figure 4.4 along with some additional helper methods.

PlayerXMove Player X's turn to mark an empty square.

PlayerOMove Player O's turn to mark an empty square.

PlayerXWin[VictorySets] Player X has won the game. The victory sets that contributed to the win are provided as the enum value.

PlayerOWin[VictorySets] Player O has won the game. The victory sets that contributed to the win are provided as the enum value.

CatsGame The game has ended in a draw where there are no winners.

is_game_over() Indicates if the state represents one of the game over states. That is, if either player has won or it is a cat's game true is returned; otherwise, false is returned.

[14] Rust's clone and copy traits both serve to duplicate an object but each goes about duplication in a different manner. Copy performs an operation similar to *memcpy* where it just copies the bits of the object. Alternately, Clone explicitly duplicates the object giving the programmer control over what parts are cloned. For details see the discussion in Trait std::clone::Clone[15].

[15] https://doc.rust-lang.org/std/clone/trait.Clone.html

Trait Implementations

- Copy
- Eq
- Debug

Related Requirements

- *Know Cells that Contributed to Player's Victory*

4.1.2 Board Data

The board structure models a Tic Tac Toe game board. It holds the individual squares of the board and provides functions to access and iterate over the squares. The board and square structures along with supporting types are shown in Figure 4.5.

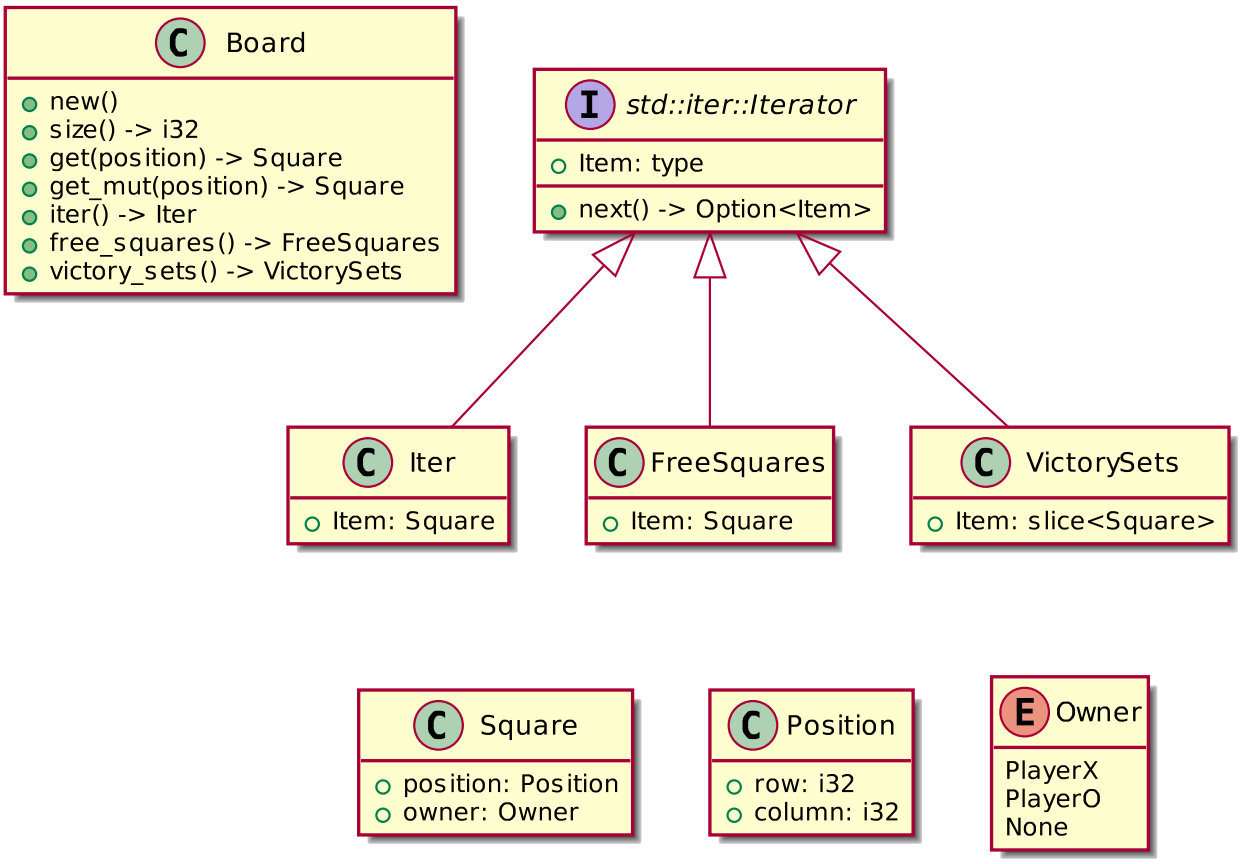

Figure 4.5: The Board structure and supporting types.

Struct Board

Data structure representing the Tic Tac Toe board. Provides multiple ways to access individual squares.

new() Constructs a new board.

size() Gets the size of board, that is the number of rows and columns. Note: boards are always square.

get() Gets the square at the indicated position. Panics if requested position is outside the size of the board.

get_mut() Gets a mutable square at the indicated position. Panics under the same situations as get().

iter() Gets an iterator that iterates over all the squares in the board.

free_squares() Gets an iterator that iterates the squares in the board that do not have an owner.

victory_sets() Gets an iterator that iterates over all the sets of squares that, if all owned by a player, would make the player victorious. E.g. this gets all the rows, columns, and both diagonals as slices.

The board structure also implements the Display trait. This provides a formatted output of the board and is suitable for use in simple console applications or debugging purposes. An example of the boards display is shown in Listing 4.1.

Listing 4.1: Example board display output.

```
+---+---+---+
| X | O | O |
+---+---+---+
| O | X |   |
+---+---+---+
| X |   | X |
+---+---+---+
```

Trait Implementations

- Display
- Clone

Struct Square

Represents an individual square of the game board.

position The position the square is located at on the board.

owner The owner of the square.

Trait Implementations

- Debug
- Clone
- Copy
- Eq

Struct Position

The position structure represents a specific board position denoted by row and column.

row The row associated with the position.

column The column associated with the position.

Trait Implementations

- Debug
- Copy
- Clone
- From<(i32, i32)>
- Eq
- Hash

Enum Owner

The owner enumeration indicates which player owns a square, if any.

PlayerX Player X owns the square.

PlayerO Player O owns the square.

None No player owns the square.

Trait Implementations

- Debug
- Copy
- Clone
- Eq

Iterating Over Squares

The board structure provides several ways to iterate over board's squares.[16] Helper types that implement the Iterator trait are used to provide this support.

Iter Iterates over all the squares in the board.

FreeSquares Iterates over squares that do not have an owner.

VictorySets Iterates over all the sets of squares that, if all owned by a player, would make the player victorious.

4.1.3 AI Moves

The AI move structure represents a move by an AI player. The AI move structure is shown in Figure 4.6.

See *Artificial Intelligence Algorithms* for details on how the AI selects a position.

[16] Rust's standard library documentation states "Iterators are heavily used in idiomatic Rust code, so it's worth becoming familiar with them." For details see [Rust-Crate-std].

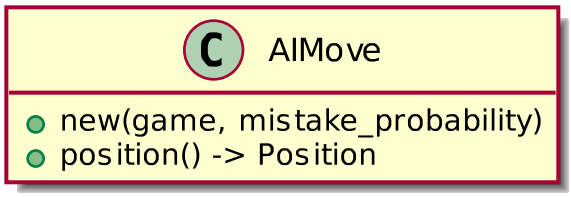

Figure 4.6: AI Move structure.

Member Details

new() Constructs a new AI move using the provided game and a given probability of making a mistake. Panics if the game is over.

position() Gets the position selected by the AI player based on the previously provided game.

Trait Implementations

- Debug

Related Requirements

- *AI Player*
- *AI Difficulty Settings*

4.2 Artificial Intelligence Algorithms

The AI algorithms are responsible for picking a position to place the AI player's mark.[17] It does this by doing a depth first search for all possible game outcomes based on the current state of the game board. An example is shown in Figure 4.7.

The algorithm selects a free position then traverses the tree looking one of the end game conditions: win, loss, or cat's game. Once the end of the game is found, the result is propagated up the tree. The algorithm assumes the opponent will play a perfect game. That is, if a specific position leads for both a win for the opponent (a loss for the AI player) and a cat's game, the algorithm marks that position as a loss.

Once all possibilities have been searched, the algorithm picks the position that lead to the best outcome. Winning positions are picked over cat's games, and cat's games are picked over losses. Additionally, if there are multiple positions with the same outcome, one is picked at random to ensure the game appears more natural to human players.[18]

The dept search algorithm can see to the end of the game, thus it cannot be beat. The best possible outcome is a cat's game. Therefore, to give human players a chance to win, the algorithm sometimes fails to consider one of the outcomes. The probability of making a mistake is configurable.

4.2.1 Example Code

[17] The name of the AI player is Robbert, or Bob for short.

[18] One of the challenges of creating AI for video games is the AI needs to be fun. An AI that picked the same square every time would not be fun! See [Buckland-2004] for a discussion on making game AI fun.

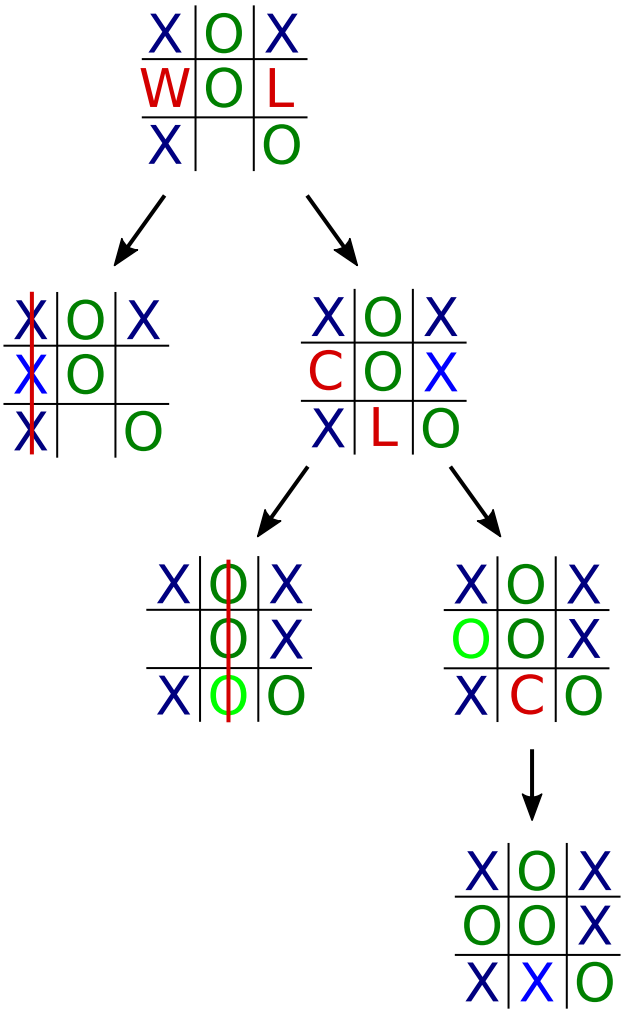

Figure 4.7: Example dept first search algorithm looking for possible Tic Tac Toe game outcomes. The AI player is playing as X and the opponent is playing as O. The possible outcomes of selecting a particular square are marked with W for win, L for loss, and C for cat's game.

Listing 4.2: searching the tree

```
def ai_move(game, player_turn):
    if game.state.is_game_over():
        return game.state

    for square in game.board.free_squares():
        square.set_owner(player_turn)
        # TODO: switch who's turn it is
        # TODO: clone the game or the board.
        result = ai_move(game, player_turn)
        square.set_result(square)
```

4.3 Other Considerations

This section contains other design considerations for the Tic Tac Toe crate.

4.3.1 Benchmarking AI Update Time

The *Maximum AI Update Time* user story requires there be a benchmark for the AI update time. Luckily cargo makes this as easy as running *cargo bench*.[19]

Listing 4.3 shows an example of benchmarking the worst case AI move.

Listing 4.3: Example worst case AI move benchmark.

```
#[bench]
fn worst_case_ai_move_benchmark(b: &mut Bencher) {
    let game = Game::new();
    b.iter(|| AIMove::new(game, 0.0));
}
```

The worst case is time is for a new game and a zero percent mistake probability. Under these situations the *Artificial Intelligence Algorithms* have to evaluate the entire problem space.

[19] See Benchmark tests[20] for details.
[20] https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html

CHAPTER

FIVE

GLOSSARY

Cargo Cargo is *Rust's* build system and package manager.

Cargo.toml The configuration file for *Cargo*. This allows developers to specify their projects metadata and dependencies.

cat's game Term used when a game of Tic Tac Toe ends in a draw where there is no winner.

idiomatic coding Idiomatic coding is using familiar conventions, techniques, and practices of a particular programming language.

Rust programming language Rust is a systems programming language with a focus on safety and speed. Website: https://www.rust-lang.org/

semantic versioning A set of rules governing how API version numbers are managed and updated with the goal of clearly communicating when incompatible changes, new features, and bug fixes made. For details see https://semver.org/.

BIBLIOGRAPHY

[Berezin-1999] Tanya Berezin (1999) *Writing a Software Requirements Document*. Retrieved from https://eecs.ceas.uc.edu/~cpurdy/sefall10/ReqsDoc.pdf (archived on archive.org[21])

[Buckland-2004] Mat Buckland (2004) *Programming Game AI by Example*

[Rust-Docs] *Rust Documentation*. Available online at https://www.rust-lang.org/learn and on any system with Rust installed by running *rustup doc*.

[Rust-Crate-std] *Crate std - The Rust Standard Library*. Retrieved from https://doc.rust-lang.org/std/index.html

[Rust-API-Guidelines] *Rust API Guidelines*. Retrieved from https://rust-lang-nursery.github.io/api-guidelines/ (source on GitHub[22])

[21] https://web.archive.org/web/20190922201012/https://eecs.ceas.uc.edu/~cpurdy/sefall10/ReqsDoc.pdf
[22] https://github.com/rust-lang-nursery/api-guidelines

INDEX