

---

# Ounce of Rust Project Design Document

*Rev. A draft 1*

**James Richey**

**Oct 05, 2019**



**CONTENTS:**

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose of this Document . . . . .	1
1.2	Scope of the Project . . . . .	1
1.3	Overview of this Document . . . . .	1
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	Objectives of the Project . . . . .	3
2.2	Current System . . . . .	3
2.3	Proposed System . . . . .	5
2.4	Interactions with Other Systems . . . . .	6
2.5	User Roles and Responsibilities . . . . .	6
<b>3</b>	<b>Requirements</b>	<b>7</b>
3.1	Rules of Tic Tac Toe . . . . .	7
3.2	User Stories . . . . .	8
<b>4</b>	<b>Design</b>	<b>13</b>
4.1	Public API . . . . .	13
4.2	Artificial Intelligence Algorithms . . . . .	18
<b>5</b>	<b>Glossary</b>	<b>19</b>
	<b>Bibliography</b>	<b>21</b>
	<b>Index</b>	<b>23</b>



## 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. 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.<sup>1</sup> 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.

---

<sup>1</sup> The structure of this document is influenced by [Berezin-1999].



**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 Current System

Tic Tac Toe is a game where two players, X and O, take turns placing their mark in a grid. 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 Fig. 2.1.

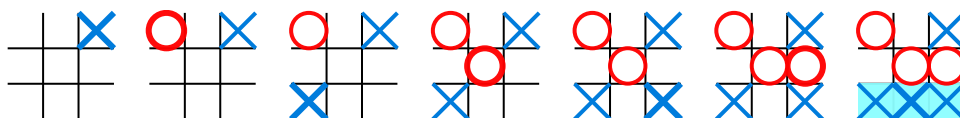


Fig. 2.1: An example game of Tic Tac Toe where player X is victorious. Image from [wikimedia.org](https://commons.wikimedia.org/wiki/File:Tic_tac_toe_game_board.png)<sup>2</sup> by user Stannered released under [CC BY-SA 3.0](https://creativecommons.org/licenses/by-sa/3.0/)<sup>3</sup>.

### 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. Fig. 2.2 shows a screen shot from one of the classic Tic Tac Toe games.

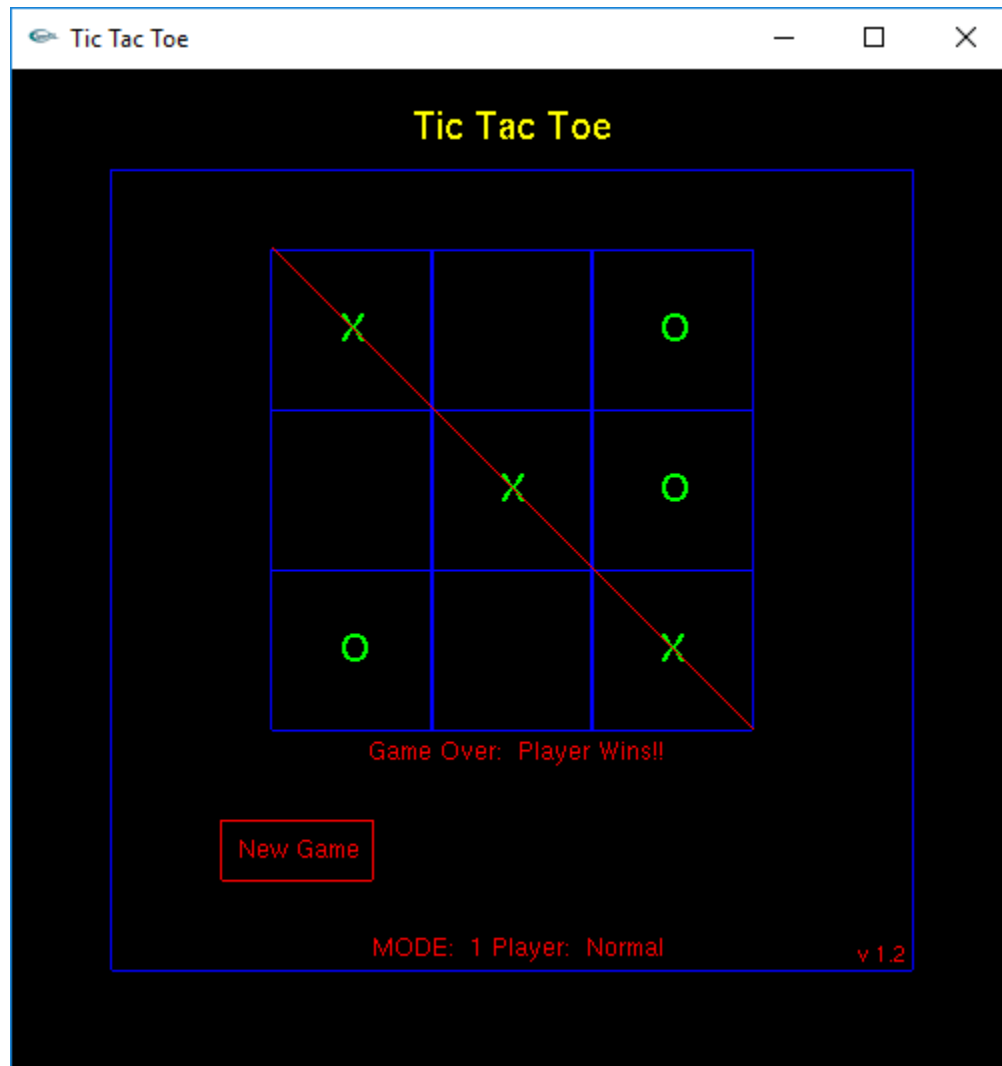


Fig. 2.2: Screen shot of Tic Tac Toe<sup>4</sup> developed by James Richey.

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

---

<sup>2</sup> <https://commons.wikimedia.org/wiki/File:Tic-tac-toe-game-1.svg>

<sup>3</sup> <https://creativecommons.org/licenses/by-sa/3.0/deed.en>

<sup>4</sup> <https://www.imaginaryphase.com/ttt.html>



### 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.

### 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.<sup>5</sup>
- API documentation.<sup>7</sup>
- Public source code repository with a tagged release.

---

<sup>5</sup> The naming conventions for crates is described in the *Rust Style Guidelines*<sup>6</sup>. Future projects built around `open_ttt_lib` may use `open_ttt` as part of their crate name.

<sup>6</sup> <https://doc.rust-lang.org/1.0.0/style/style/naming/README.html>

<sup>7</sup> Popular places to host Rust API documentation include <https://docs.rs/> and <https://pages.github.com/>.

## 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 Fig. 2.3 shows how the library is used by other systems.

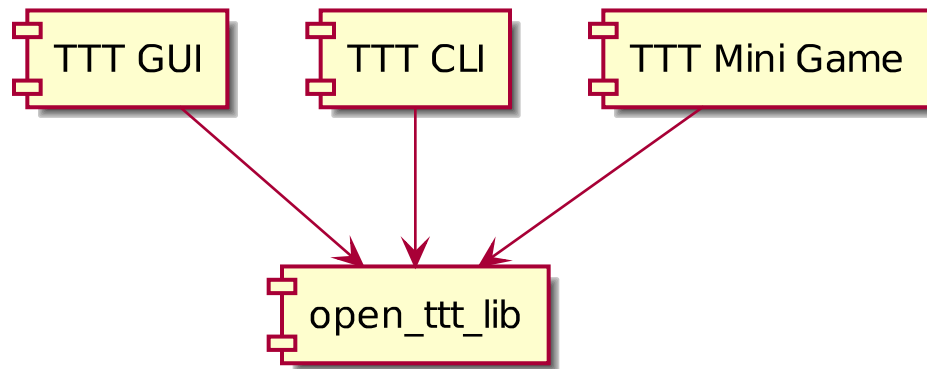


Fig. 2.3: Diagram showing how the library is used by other applications.

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.

### 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.

## 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 [Fig. 3.1](#).

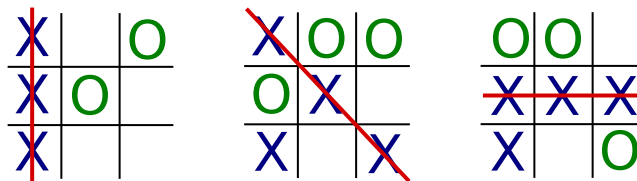


Fig. 3.1: Examples of winning Tic Tac Toe games showing player X winning by getting three marks a row, diagonal, and column. The red line shows the squares that contributed to the win.

6. The game ends in a draw if no more empty squares remain. Examples of games ending in a draw are shown in [Fig. 3.2](#).

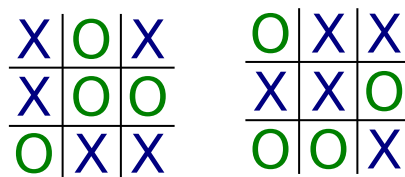


Fig. 3.2: Examples of Tic Tac Toe games ending in a draw. No player managed to get three marks in a line.

## 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.
- There are integration tests that help library developers detect if the library's API changes.

### 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 winning possibilities than the second player.<sup>11</sup>

### 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.

---

<sup>11</sup> The player with the first move has about double the number of winning possibilities. For details see Wikipedia's Tic-tac-toe page<sup>12</sup>.

<sup>12</sup> <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](https://docs.rs)<sup>8</sup>.

### 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.

---

<sup>8</sup> <https://docs.rs>

### Acceptance Criteria

- The Rust API Guidelines are consulted when designing the libraries API.<sup>13</sup>
- 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](https://crates.io/)<sup>9</sup>,  
so that I can easily incorporate it into my Rust based application with Cargo.

### Acceptance Criteria

- The library can be downloaded from [crates.io](https://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](https://github.com/)<sup>10</sup>  
so I can view the source code to get a better understanding of how the library works.

---

<sup>13</sup> See the [\[Rust-API-Guidelines\]](#) for details.

<sup>9</sup> <https://crates.io/>

<sup>10</sup> <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.



## 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 [Fig. 4.1](#) is used for the type diagrams in this section.

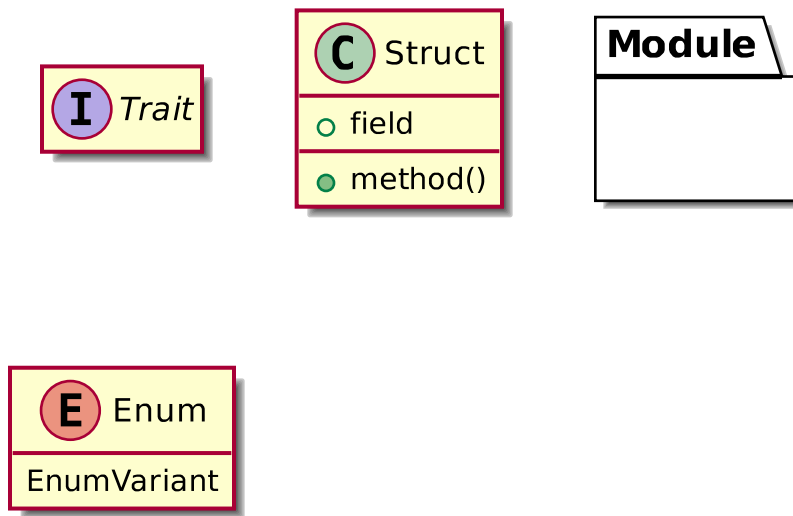


Fig. 4.1: Legend used for the type diagrams in this section.

An overview of the public types is shown in [Fig. 4.2](#).

The library contains a single public module that holds the public types.

### 4.1.1 Struct Game

The Game structure is one of the central types provided by the crate. It contains the state machine logic and the underlying game board. [Fig. 4.3](#) show the Game structure.

The Game structure uses a state machine to determine which player has the next move or when the game is over. The state diagram is shown in [Fig. 4.4](#).

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.

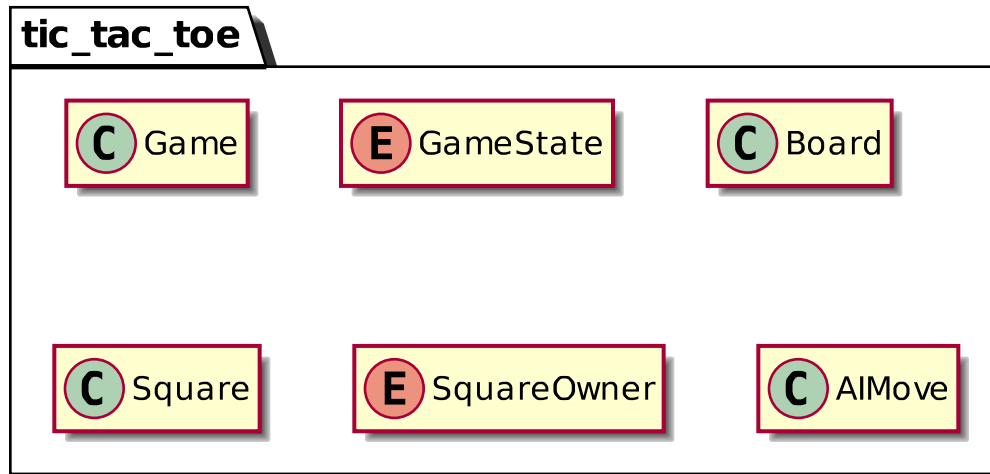


Fig. 4.2: Public modules, structures, and other types.

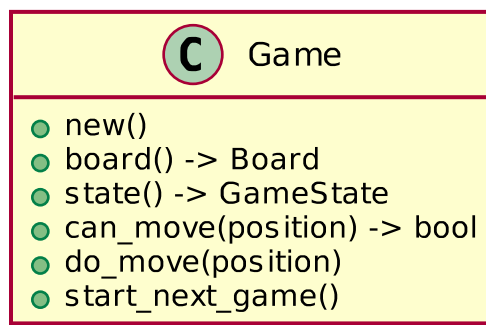


Fig. 4.3: The Game structure.

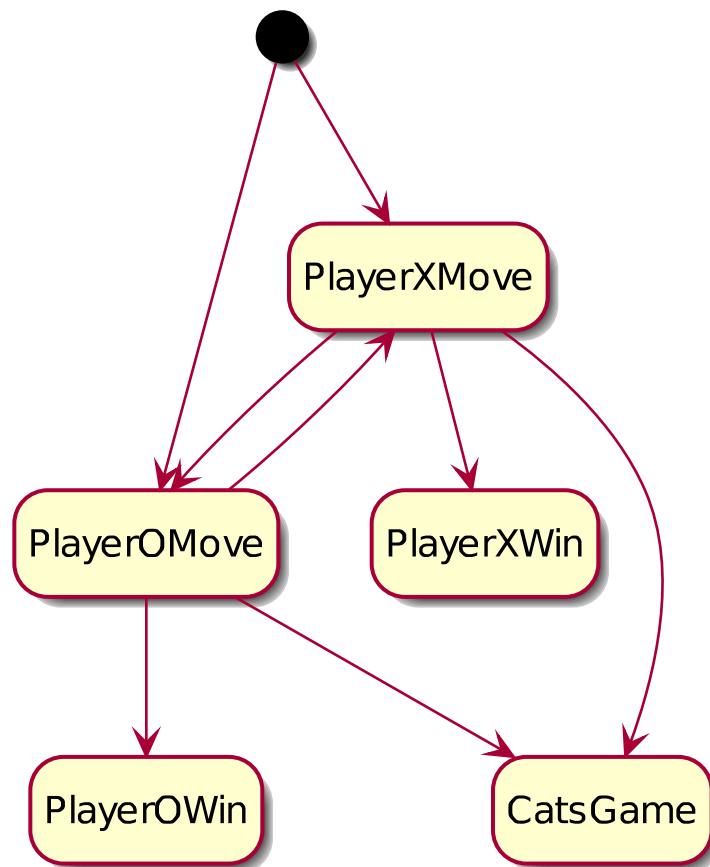


Fig. 4.4: State diagram of a Tic Tac Toe game.

## Member Details

**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 if the game is over.

**start\_next\_game** Starts

## Trait Implementations

- Debug
- Clone (or Copy?)

## Related Requirements

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

### 4.1.2 Enum Game Sate

The game state enumeration

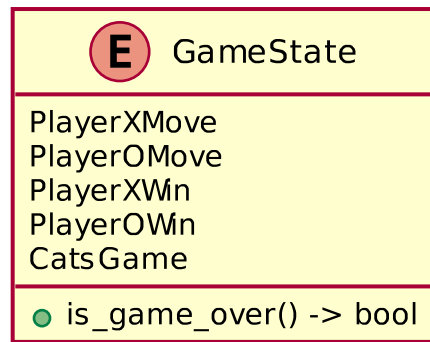


Fig. 4.5: GameState enum

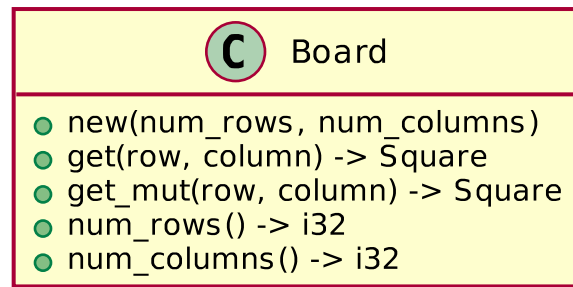


Fig. 4.6: The Board structure.

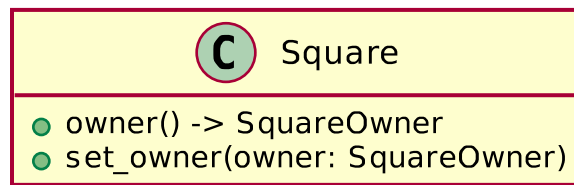


Fig. 4.7: The Square structure.

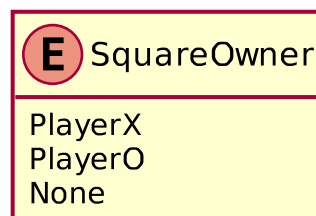


Fig. 4.8: Enumeration for indicating a square's owner.

### 4.1.3 Struct Board

### 4.1.4 Struct Square

### 4.1.5 Enum Square Owner

### 4.1.6 Struct AI Move

See *Artificial Intelligence Algorithms* for details on the algorithms.

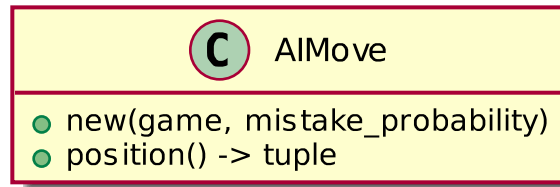


Fig. 4.9: AIMove structure.

### Related Requirements

- *AI Player*
- *AI Difficulty Settings*

## 4.2 Artificial Intelligence Algorithms

## 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<sup>14</sup>)
- [Rust-API-Guidelines] *Rust API Guidelines* retrieved from <https://rust-lang-nursery.github.io/api-guidelines/> (source on GitHub<sup>15</sup>)

---

<sup>14</sup> <https://web.archive.org/web/20190922201012/https://eecs.ceas.uc.edu/~cpurdy/sefall10/ReqsDoc.pdf>

<sup>15</sup> <https://github.com/rust-lang-nursery/api-guidelines>



## INDEX

### C

Cargo, [19](#)  
Cargo.toml, [19](#)  
cat's game, [19](#)

### I

idiomatic coding, [19](#)

### R

Rust programming language, [19](#)

### S

semantic versioning, [19](#)