Basically, it has the following main features on the user level:

1. A player can choose among the following difficulties:

* Relaxed(2x2)
* Easy(3x3)
* Normal(4x4)
* Hard(5x5)
* Insane(6x6)

1. The board guarantees the existence but not uniqueness of its solutions
2. The board has a valid range of the number of read only squares with initially filled characters
3. The board has a timer, which can be shown or hidden by players, showing the number of elapsed seconds of solving the board
4. Players can record demo files for their gameplay and play the recorded counterparts(which also serves as integration tests)

On the implementation level, I want to practice [TDD](https://en.wikipedia.org/wiki/Test-driven_development), even though I’ve cut so many corners that I don’t think I’m really doing true TDD

1. I don’t unit test Controller at all, as I want to make it so obvious, trivial and stable that I don’t know how to unit test it nor what’s the point of doing so.
2. I don’t unit test functions that are too easy, simple and small to do so.
3. For View functions with unit testing, they’re written after rather than before the implementation details, as I still don’t know how to unit test them first.
4. I don’t write just enough to fail during the red stage of TDD, as I instead do so so that the easiest, simplest and smallest codes to pass the tests are the proper solutions.
5. I don’t use TDD to drive my design, but rather verify it as I’ve thought through it before writing the 1st test. I just think that writing test first can better verify my design before it’s too late.

With the stage being set, let’s show [what I’ve done](https://github.com/Double-X/Vanilla-Tic-Tac-Toe-Tomeks-With-Blocks).

**Requirement Analysis**

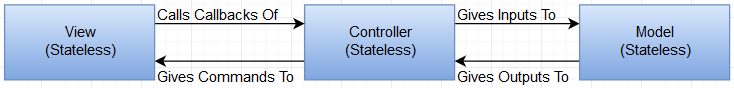
After thinking through the requirements for hours, I’ve decided to take some shortcuts:

1. While there are algorithms to generate a proper Sudoku board, it’s just too colossal, complicated and convoluted for me to implement, even though I know it’s itself easy, simple and small(It means that it’s all due to my utter inability as a programmer lol).
2. I instead try to [reinvent the square wheel](https://en.wikipedia.org/wiki/Reinventing_the_wheel) – Boards that might not have unique solutions.

Other than these, this app should be so easy, simple and small even for those having only basic knowledge of CSS, HTML and Vanilla JS and written few easy, simple and small websites.

**The Original Version**

On the top level, this version implements MVC this way:



Upon the start of the app, the View will link the callbacks from the Controller to the raw DOM events. As players interact with the UI, those callbacks will be called, causing the corresponding Model APIs to be called, which accepts inputs and gives outputs back to the View to render itself via the Controller.

On a lower level:

1. This version implements the View this way:
2. This version implements the Model this way:
3. The following’s everything combined:

There are basically events that can be triggered by players directly:

1. When a new game difficulty button’s clicked, a prompt will show to ask players to input the number of read only squares with initially filled characters. The prompt will continue to show until players give a valid input.
2. When the checkbox nearby the timer’s checked/unchecked, the timer will be shown/hidden.
3. When the play demo file input’s clicked, players can select a valid json file as the recorded demo file to have its contents played.
4. When the player inputs an inputable square, the square will either be filled with that input, or an alert will be shown to explain why the input’s invalid. If the board becomes solved, an alert will show to notify that, followed by a window letting players choose the destination to save the recorded demo file contents as a json file.
5. When an alert’s shown, the players can click ok to dismiss it.

When it comes to integration testing, the recorded demo file contents are themselves integration tests. **It’s because in order to generate those recorded demo file contents, one must actually solve the board, which is itself a manual end-to-end testing. Then if the demo file contents’ recorded, playing it is also a form of automated integration testing.** Specifically:

1. If the recorded demo file contents’ failed to be generated, it means there’s already something wrong with the “solving the board” parts of the codebase.
2. If the recorded demo file contents’ generated but failed to be played, it means there’s something wrong with the “playing the demo” parts of the codebase.
3. If the recorded demo file contents can be played from start to finish without problems now, but can’t be so later, it means that there’s [regressions](https://en.wikipedia.org/wiki/Regression_testing) in the codebase.

[**The Mini Version**](https://github.com/Double-X/Vanilla-Tic-Tac-Toe-Tomeks-With-Blocks/blob/master/appMini.js)

Basically, the **Mini Version** applies [KISS](https://en.wikipedia.org/wiki/KISS_principle) and [YAGNI](https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it) to the implementation codes, but the unit and integration tests still work. The goal’s to have a look on how much [accidental complexity](http://wiki.c2.com/?AccidentalComplexity) has been introduced by the original version, by checking the difference of the codebase size.

It’s done by removing all comments, using a more compact and terse coding style, removing redundant local variables, inlining all private functions that don’t help reducing the codebase size and putting every module implementation into a single js file. Note that the APIs and thus the external behaviors haven’t changed at all, so this refactoring won’t break any test.

While the whole process took me about an hour, the resulting codebase size are just a bit more than 80% of the **Original Version**(from roughly 490 KB to roughly 395 KB). This might mean the latter’s slightly overenginnered, as the former has the same test suites.

[**The Micro Version**](https://github.com/Double-X/Vanilla-Tic-Tac-Toe-Tomeks-With-Blocks/blob/master/appMicro.js)

Basically, the **Micro Version** applies KISS and YAGNI to throw all unit tests away while still keeping the integration tests. The goal’s to have a look on how much accidental complexity has been introduced by the unit tests, by checking the difference of the whole codebase size.

This lets me throws all the seams away, except *DIFFICULTIES*, which actually makes the codebase smaller by keeping it intact. For the rest, I basically rewrite them from scratch, using just 1 module to implement everything. As long as no integration test breaks, the refactoring should have preserved every user-level behavior.

While the whole process took me about a day, the resulting codebase size are just slightly more than 4% of the **Mini Version**(from roughly 395 KB to roughly 16 KB). While it’s because quite some of those unit tests have really large test data and they’re immensely helpful for maintaining the core nondeterministic behaviors, as the former’s codebase’s still covered by integration tests, this might mean even the latter might be still at least slightly overengineered.

**Conclusion**

While the user requirements are easy, simple and small, I’ve still learnt quite some useful knowledge. While I still clearly don’t know how to do TDD, one day I might eventually get it.

For instance, if one day I decided to use a proper algorithm to ensure a proper Sudoku board, basically I’ll have to throw nearly half of the Model modules and unit tests away, and change some of the View interfaces. While they’re the core business logic, it still means I’ve failed hard.

On a side note: This article will be used as a base of quite some of my subsequent ones :D