Recently I’m learning [React](https://reactjs.org/tutorial/tutorial.html) in my new task. After a week of explorations, I’m inspired by its Tic Tac Toe tutorial, so I’ve tried to build a vanilla [Tic Tac Toe Tomek](https://code.google.com/codejam/contest/2270488/dashboard) with blocks.

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

1. A player will win, with all the winning squares marked, if a line contains at least 1 of his/her symbols and nothing other than them and tomeks(A line having only tomeks can’t be a winning line)
2. If no player can win, the game will immediately draw(players won’t be able to continue the game)
3. This game has, in addition to tomeks, blocks as well(A line having blocks can’t be a winning line)
4. This game has configuration values that can be changed on the fly, including:

* Board Width And Height
* Player List
* Tomek positions
* Block positions

1. This game shows the list of moves made as its history that can be navigated by players and/or cleared(The move ordering can be either ascending or descending)

On the implementation level, I want to practice these things that are new to me:

1. [Lifting state up as a concept](https://reactjs.org/docs/lifting-state-up.html)
2. [Unidirectional Data Flow](https://teamtreehouse.com/library/unidirectional-data-flow)
3. [Functional Core, Imperative Shell](https://www.destroyallsoftware.com/talks/boundaries)
4. [Reducers as a concept](https://redux.js.org/docs/basics/Reducers.html)
5. [Partial Application](https://en.wikipedia.org/wiki/Partial_application) in Vanilla JS(I didn’t know I could use bind to do that)

Along with these things that I’ve practiced before:

1. [Composition root](http://blog.ploeh.dk/2011/07/28/CompositionRoot/) with [Pure DI](http://blog.ploeh.dk/2014/06/10/pure-di/)
2. Test After Development(With [unit](https://en.wikipedia.org/wiki/Unit_testing) and [integration testing](https://en.wikipedia.org/wiki/Integration_testing))
3. [Respecting Levels of Abstraction](https://simpleprogrammer.com/2017/01/27/respecting-abstraction/)
4. [MVC](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)(with skinny controller)
5. Using CSS, HTML and [Vanilla JS](http://vanilla-js.com/)only(i.e., no 3rd party framework at all)

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 noticed the following edge cases:

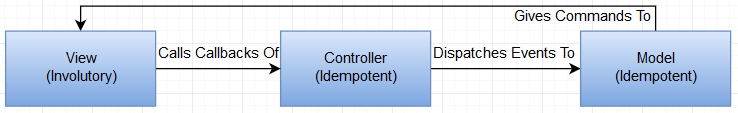
1. **It’s possible for a player to win with more than 1 winning lines.** For instance, it’s possible for his/her symbols to fully occupy both the 2nd row and column by making the middle square as the last and thus winning move. While it’s very unlikely to occur unintentionally, I should still deal with it, especially when doing so’s easy, simple and small.
2. **It’s possible for the game to draw upon its start.** It’s due to having no possible winning lines, which can be caused by all of them have blocks or fully occupied by tomeks.

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.

On a side note: I should’ve figured all these out within minutes instead of hours 😊

**The Original Version**

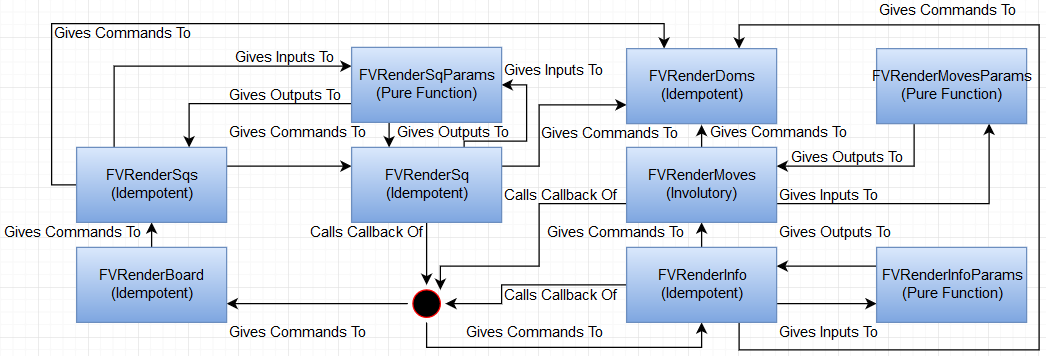
On the top level, this version implements MVC this way:



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 events to be dispatched to the Model, which then gives appropriate commands to the View to render itself via its APIs encapsulated and provided to be Model by the Controller.

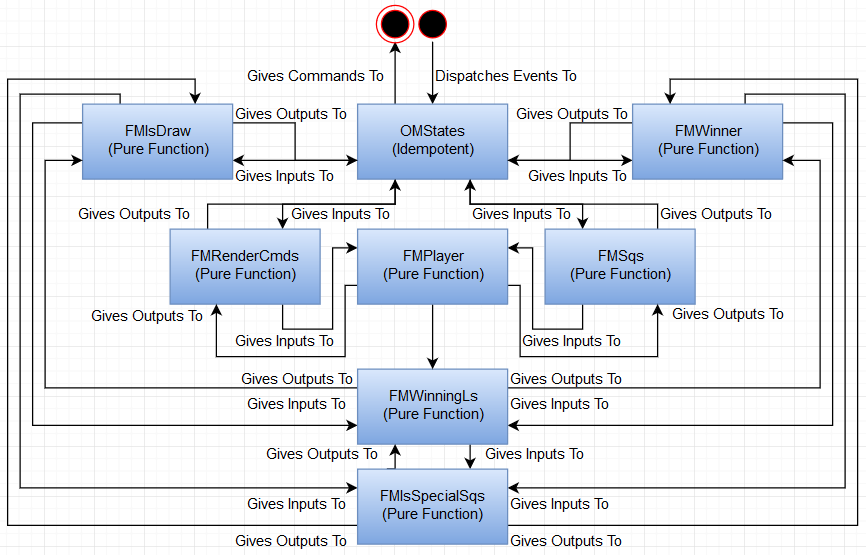
On a lower level:

1. This version implements the View this way:



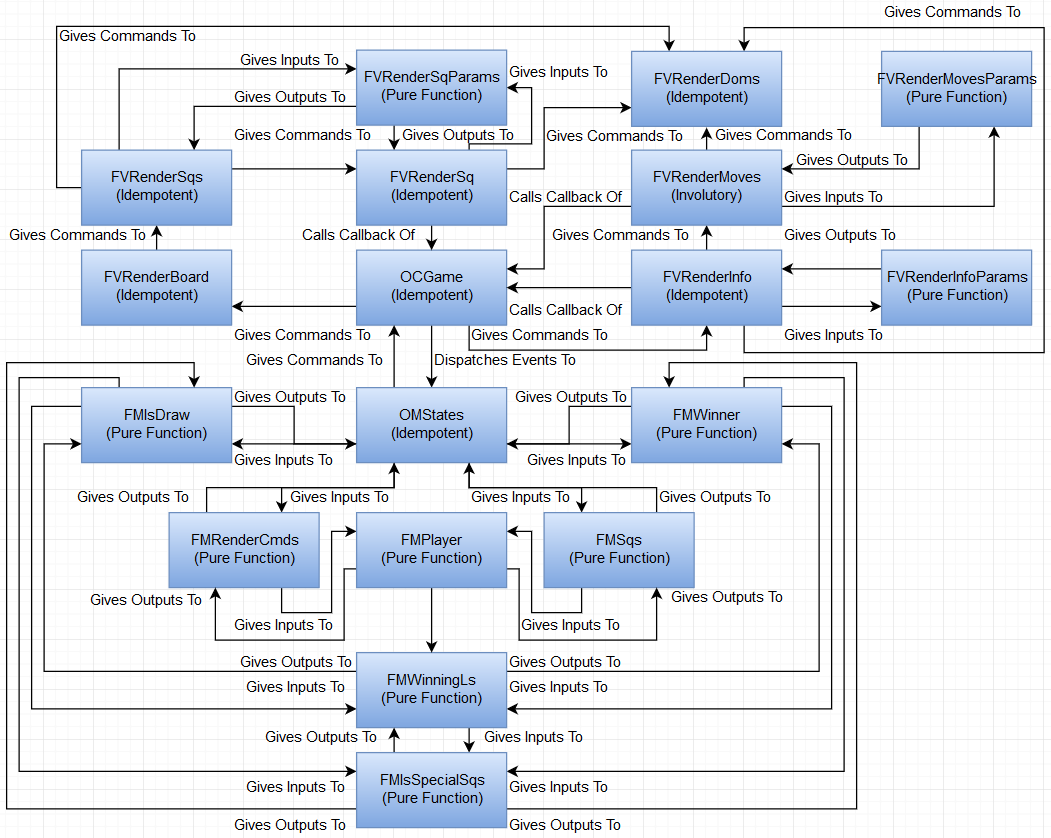
So basically, *FVRenderBoard*, which is responsible for rendering the 2D board, and *FVRenderInfo*, which is responsible for rendering the game information, are the high level Views that resolves and delegates the high level “what to render” commands received from their APIs to the lower level Views, which includes *FVRenderMoves*(renders the list of moves made), *FVRenderSqs*(shapes the 2D board) and *FVRenderSq*(renders a square). The latters converts the “what to render” commands to the “how to render” commands that are delegated to the lowest level View - *FVRenderDoms*, which renders the DOM. Finally, *FVRenderInfoParams*, *FVRenderMovesParams* and *FVRenderSqParams* are helper functions encapsulating the display constants and formats.

1. This version implements the Model this way:



So basically, *OMStates* receives events dispatched through its APIs and updates its states accordingly, with the aid of helper functions including *FMIsDraw*(checks if the game’s a draw), *FMIsSpecialSqs*(check if a square’s a special one), *FMPlayer*(returns the player of the given move number), *FMSqs*(returns the current and last square states), *FMWinner*(returns the list of winning squares with the winner) and *FMWinningLs*(returns the list of winning lines). Finally it gives commands, which are delegated to *FMRenderCmds*, to callbacks passed to it using the updated states.

1. The following’s everything combined:



There are basically 5 events that can be triggered by players directly:

1. When a player clicks an empty square before the game ends, the *FVRenderSq* representing the Dom being clicked will call a callback of *OCGame*, indicating that that square’s updated.
2. When a player goes to a move made, the *FVRenderMoves* representing the Dom being clicked will call a callback of *OCGame*, indicating that the game’s to restore the state when that move has just made.
3. When a player clears the game histories, the *FVRenderInfo* will call a callback of *OCGame*, indicating that the game’s to restore its initial state.
4. When a player toggles the sorting order of the list of moves made, *FVRenderInfo* will call an API of *FVRenderMoves* to reverse the said order.
5. When a player uses the *EditCfgs* API call, the *CompositionRoot* module, which encapsulates the whole composition root, will be run with the new configuration values again, causing the game to be reset with the said configuration values.

When it comes to unit testing, **I constantly ask myself “How am I going to test this?” whenever I design an API for a module.** While I practice Test After Development here, **I’ll still pretend that the implementations aren’t there and only test against a module’s APIs.** To verify that the tests are indeed testing something, **I’ll comment out the top level implementations to see the tests fail.** If I feel the desire or need to test a supposedly private function, **I’ll instead consider extracting it and its lower level implementations into a new module.** It’s because such desire/need might actually indicate that the module’s still too large to be highly testable.

For instance, *FMRenderCmds* are indeed an afterthought, which is extracted from *OMStates* so I can write unit tests to ensure that the correct sates always lead to the correct rendering commands. **Also, as *OMStates* are idempotent, which can only be tested by ensuring that the same initial states and input sequences always lead to be same output sequences, it’s beneficial to extract a pure function, which is trivial to test, out of it to improve testability.**

Similarly, *FVRenderDoms* are also an afterthought to improve testability. In this case, it acts as a seam, rather than an abstraction, so that **I can stub it out when testing *FVRenderInfo*, *FVRenderMoves* and *FVRenderSq*, meaning that I don’t have to touch with the Doms, which are a bit hard to unit test directly.**So unit testing them now just means verifying whether the same “what to render” commands always lead to the same “how to render” commands under a test fixture, which is the initial state of the app in this case(So a downside of this approach is that the unit tests for the Views only work when they’re run with that state).

On a side note: Running the unit test suite’s as straightforward as using the *UnitTest* API call.

When it comes to integration testing, I use the Doms to dispatch relevant events and check against the new Dom states directly. This idea motivated me to extract *FVRenderInfoParams*, *FVRenderMovesParams* and *FVRenderSqParams* out of *FVRenderInfo*, *FVRenderMoves* and *FVRenderSq* respectively. With this, **I can access the needed Doms to dispatch the events and check the new Dom states against the expected ones without needing to know the display constants and formats as they’re now accessible from the APIs of those parameter modules. So whenever I change those implementation details, the integration tests won’t break.**

On the implementation level, each integration test will edit the configuration values to be the ones intended to be tested. After the test’s over, the old ones will be restored.

On a side note: Running the integration test suite’s as straightforward as using the *IntegrationTest* API call.

**The limitations of these unit and integration tests are that they only test a single configuration value set, meaning that the code and state coverage are very low.** While I know property-based testing are called for here, I don’t know how to write them in this case yet.

To sum up, I’ve built the original version with MVC, unit and integration testing, unidirectional data flow, all states lifted up, and functional core with imperative shell. It took me a full week to finish all these. While I feel that I’m at the danger of [overenginnering](https://en.wikipedia.org/wiki/Overengineering), it doesn’t matter much even if that’s the case. **After all, my goal here’s to practice new things in order to learn them.**

[**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 1 hour, the resulting codebase size are just a bit more than two-thirds of the **Original Version**(from roughly 210 KB to roughly 143 KB). This might mean the latter’s indeed 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 *EditCfgs*(which lets players change configuration values on the fly), and those needed by the integration tests, namely *FVRenderInfoParams*, *FVRenderMovesParams* and *FVRenderSqParams*. 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 1 day, the resulting codebase size are noticeably less than one-third of the **Mini Version**(from roughly 143 KB to roughly 44 KB). This might mean the latter still has too much overhead induced by the unit tests, given the fact that they don’t have high code nor state coverage. As the former’s codebase’s still covered by integration tests, this might mean even the latter might be still at least slightly overengineered.

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

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

This lets me throws all the seams needed by the integration tests, namely *FVRenderInfoParams*, *FVRenderMovesParams* and *FVRenderSqParams*. Their functionalities are inlined into the single module implementing everything. Now **I’ve to manually test the app.**

While the whole process took me about 1 hour, the resulting codebase size are less than a quarter of the **Micro Version**(from roughly 44 KB to roughly 10 KB). This might mean that the latter still has too much overhead induced by the integration tests, given the fact that they don’t have high code nor state coverage. As the codebase’s so easy, simple and small, manual testing might be feasible. This might mean even the latter might be still at the edge of overengineering.

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

Basically, the **Pico Version** applies KISS and YAGNI to cut corners on the user level requirements. The goal’s here to have a look on how much accidental complexity has been introduced by those very requirements themselves, by checking the difference of the whole codebase size. Again, I’ve to manually test the app.

This lets me throws the following features away:

1. The clear history button – It behaves exactly the same as refreshing the page
2. The toggle move list order button – Players just have to scroll a bit so that’s ok
3. The vertical move list – Making it horizontal will simplify implementations
4. The disabled redundant buttons – Letting players click them is ok
5. Changing configurations on the fly – Changing the js file content’s already suffice

While the whole process took me about 1 hour, the resulting codebase size are just a bit more than two-thrids of the Nano Version(from roughly 10 KB to roughly 7 KB). This might mean that it’s worth pushing back and simplifying the user requirements, even if it might be perceived as cutting corners. After all, overengineering can occur on the user requirements as well.

**The Kilo Version?**

While I didn’t make this version(at least not yet), I still wonder what the codebase would be if I further improve the testability by extracting yet more modules – Going for the opposite directions of KISS and YAGNI in this case. Also, a bit of future coding might pay off in the future.

For instance, I might extract the functionality of converting old states to new states from *OMStates* into a reducer that’s a pure function, extract the functionality of converting the “what to render” commands to “how to render” commands given the current dom states as FVRenderCmds, and let players save/load the histories with the configurations to/from json files.

I didn’t make this move(at least not yet) as I just don’t know how to do it well yet.

**Conclusion**

While the user requirements are easy, simple and small, I’ve still learnt quite some useful knowledge. While my code clearly sucks like hell, I’m still [sucking less and less](https://blog.codinghorror.com/sucking-less-every-year/).

For my own projects, I’ll usually stick to the approach illustrated by the **Original Version**, while at work I’ll try to stick to the existing practices. For instance, my current supervisor prefers to be able to view all details without jumping up and down nor scrolling, so he tends to write very long functions(he’ll also group codes that would be a function by comments showing their goals just like what function names do). While I prefer short functions to make my code read like plain English, I’m ok for the long functions approach as well, as I’m used to write codes that way.

As for why I don’t go for [TDD](https://en.wikipedia.org/wiki/Test-driven_development) already, it’s because I still want to focus on writing high quality tests first. While I’m already used to writing unit and integration tests, I still think that I’m constantly screwing them up. Even in this app, those tests have very low code and state coverage. Maybe projects like Sudoku, which can be written using almost pure functions only, might make me feel comfortable enough to finally try TDD. I might do just that if I’ve time.

Maybe, just maybe, my codebases can serve as ideal examples of how to screw things up XD

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