|  |
| --- |
| Tables: |
| 1. Table must be identified by ONE WORD ONLY.  And must end with word "table".  EX: session\_table EX: owner\_table EX: trial\_table  2. If table utility accesses a table,  that table name should appear in the utility name.  EX: TokenTU <--Transaction utility that only has access to token\_table. EX: OwnerTokenTU <--Transaction utility that has access to Owner and Token tables.  3. Solve naming arbitration using alphabetical order. EX: OwnerTokenTU, NOT TokenOwnerTU |

|  |
| --- |
| Naming conventions: |
| 1. Battle against "Hedonistic Clarity"   |  |  |  | | --- | --- | --- | | Hedonistic: | Compromise: | Concise and Clear in long-term: | | SessionTableTransactionUtility | SessionTransUtil | SessionTU | | TokenTableTransactionUtility | TokenTransUtil | TokenTU | | OwnerTableTransactionUtility | OwnerTransUtil | OwnerTU | | TrialTableTransactionUtility | TrialTransUtil | TrialTU | | This is very easy to understand at first glance. But will slow you down with visual clutter in the long run. Only 25% of the classname is used to actually tell us what is UNIQUE about this class.  It is hedonistic because people are more comfortable with it at the beginning. When they are looking over a code base they have never seen before. | This is better. Using abbreviations, about 50% of the identifier name tells us what is unique about it. | This is best in long run. Rules of clean code it follows:  1. The "TU" is on end, so it won't muck with your auto complete.  2. The names are pronounceable. You can still call it "Session Transaction Utility" when you speak of it in conversation. |   1.5: If a class uses an abbreviation. That abbreviation should be explained in the header file. SessionTU is okay, if it is explained in the header that  TU == transactionUtilities. That and it should be in a namespace like: transactions.utilities.  2. Research and create concise words for your unique problem domain: It may seem silly. But making up your own words for ideas unique to the problem domain of your application will lead to cleaner and more concise code. It will also make it easier to think abou the code. Imagine having to think about linear algebra, but you don't have words for matrix or vector. Instead matrix is: "**TwoDimensionalArrayOfFloats**" and Vector is "**PairOfFloats**". **Longer words slow down your thinking. Short, descriptive words are best.** |

|  |
| --- |
| Error handling: |
| **Program like you are a paranoid idiot who will do everything wrong.** Better to spend 6 hours coding, putting error checks into the code as they come to you, than 3 hours coding and 3 hours debugging. The frustration will eat away at your quality of life.  When something does go wrong, if you are doing it correctly, 95% of the time, the error should be caught by your custom error code. And your custom error code should offer some possible solutions to the error.  Reasoning: Task switching. **It can take a long time to get your head into a problem.** How many times have you come back to relatively trivial code to realize you don't know what it does any more? You wish you wrote more documentation? I do it a lot. It is the same thing. The solution to the error is MORE OBVIOUS at the time of writing the error checking code. So you should inline the possible solutions that come to mind in the error message while you write it. So that 6 months later, you'll see the error message and have a great jumping off point for where to fix the problem. |

|  |
| --- |
| Coding: |
| Baby Steps: Even the most trivial change can break stuff. Write the smallest amount of code you can that keeps everything unbroken. Then test. Then keep writing. Sometimes figuring this smallest unit can be very hard.  Get it on the screen QUICK: Get something up and working and tangible as soon as possible. Until you have something tangible, you are working with a hypothesis. Einstein once said something like it takes exponentially more thought to fully work out an idea than the thought it takes to think of the initial idea. If you don't have something tangible, you are NOT in a position to grasp the magnitude of the problem at hand.  A plan is great: **But don't use that as a waterfall template!!!** Figure out the minimal viable product to get a complete working product. For example, with gauntlet, the first step is just to be able to dispatch a token and confirm the token.  This step requires: 1. A database. 2. Back-end logic. 3. Api calls. 4. UI front end.   |  |  | | --- | --- | | **Admin View: Just a blank page with input and submit button.** | **Candidate View: Just a blank page with input and submit button.** | | |  | | --- | | **XGYPWX** |   **[Launch Token]** | **Enter Token:**   |  | | --- | | **XGYPWX** |   **[Submit Token]** |   This is all the first step should have been!! You then confirm that the data is in the database! Then you figure out the next step. You add the tables and columns as you go. I have documentation from months ago on how this should have been. |