Solar Size: Code Quality Review

Tristan Brown-Hannibal
Karlee Fidek
Kaden Goski

Introduction **1**
Vue: Front-end Framework 2
Components 2
Components and Object-Oriented Programming 3
Single Responsibility 3
Open-Closed Principle 3
Design Pattern 5
Good Practices Followed 6
Variable naming 6
Comments 6
DRY (Don't Repeat Yourself) 7
Laravel – Back-end API 8
API 8
Conclusion 9
References 11

Introduction

Quality code helps deliver quality software. Which in turn helps achieve the current objectives of the project, and perhaps more importantly, it enables future objectives to be more easily obtained. Software is rarely ever a static product, customers and stakeholders expectations will change, as will the needs of the business. This is why software must always be developed with non-functional requirements (NFR) in mind.

The tools that allow for these NFRs to occur are: code formatting, code architecture, SOLID design principles, and object-oriented design. The Solar Size project will be compared against these aspects, in both the front-end and back-end side.

The languages used in the project are Vue, Laravel and Python. Vue is used as the front-end framework. The back-end is Laravel, with Python being used for solar calculations, and called via Laravel.

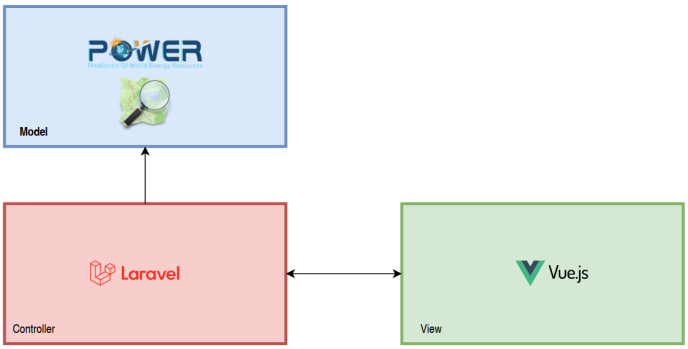


Figure 1. MVC Diagram

Vue: Front-end Framework

Vue is a tool that greatly speeds up front-end development. The purpose of using a front-end framework like Vue, is that it allows for the ‘componentization’ of common user interface elements. UI elements are often reused in multiple spots, with slight changes. Vue facilitates this reuse, by extracting these elements into discrete components. With these components, the framework also provides reactivity out of the box, meaning that data updating in the components is automatically updated in the view layer for the user. This greatly reduces the amount of Javascript code needed. The components contain CSS, Javascript, and HTML, it is standard practice to keep all these in one file, and not separate.

Components

Components in Vue, because they are discrete, can be separated into distinct files. The file structure of Vue is such that it is easy to see the purpose of each component, as seen in the figure below, which contains the Solar Size component files.

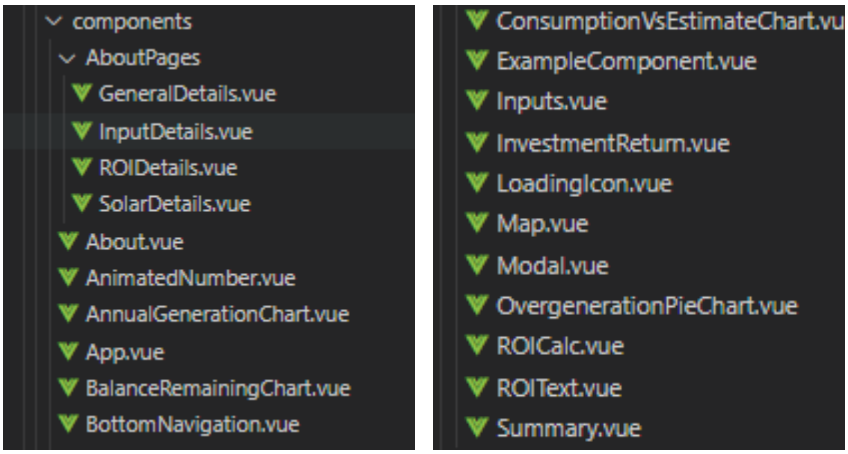


Figure 2. Component File Structure

The file naming scheme is trying to be concise, but descriptive, e.g. ***Inputs.vue*** references the inputs, ***Map.vue*** contains the map. The main improvement that could be considered is to separate the component into separate folders, such that more closely related components are grouped together. For example, ***Map.vue*** and ***Inputs.vue*** could be in a subdirectory called 'Form' which highlights that they are used together on the form page.

Vue best practice is for parent components to pass data (through an item called a prop), and then if the parent component needs to change based on the child's handling of that data, the child should emit an event. (Abrickis, 2019) The emission of data helps ensure that parent components don't break the idea of encapsulation, that is, a component's private data should only be altered by that component.

An example of this in the code can be found in ***Map.vue*** and ***Inputs.vue*** in the code below:

Parent Component: *Inputs.vue*

```
bus.$on("latlongAdded", (latLong) => {
        this.formInputs.latInput = latLong[0] % 90;
        this.formInputs.longInput = latLong[1] % 180;
        this.displayLatLong = true;
        });
```

Child Component: *Map.vue*

```
 addMarker(clickEvent) {
        this.markerLatLng = clickEvent.latlng;
        var latLong =
[clickEvent.latlng.lat,clickEvent.latlng.lng];
        bus.$emit("latlongAdded", latLong);
}
```

Code Snippet 1. Parent and Child Components

This code will take a click item, and then emit an event (`"latlongAdded"`) upwards, to fill in the formInputs for latitude and longitude.

Components and Object-Oriented Programming

Vue itself uses Javascript, which does support classes, but there is discussion about the usefulness of class in Javascript and Vue. (Fekke, 2021) Components can be thought of as class-like and thus SOLID principles and other good practices can be observed.

Single Responsibility

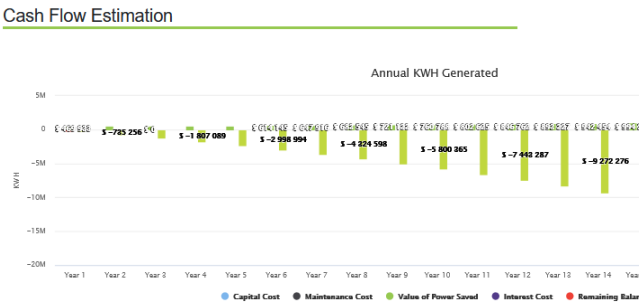
AnimatedNumber.vue is a component which takes a number as a prop, and simply animates the counting to that number. It exists for a single reason, and will only change if the number it's given changes.

Open-Closed Principle

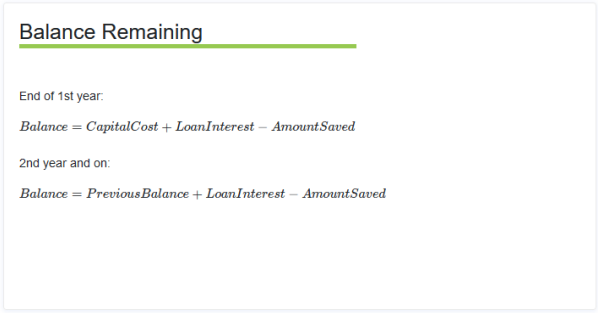
An example which follows the spirit, though not following the strict definition, of the Open-Closed principle can be found in the ***ROIText.vue*** component. This component utilizes a Vue feature called "slots" which allows for templates of HTML code to be slotted in the places

indicated by the slots. The ***ROIText.vue*** component provides styling for the header, body, and footing of the component.

```
<ROIText>
  <template v-slot:header>
    <h3>Cash Flow Estimation</h3>
  </template>
 <cashflow
  :solarPanelData="solarPanelData"
  :investmentData="investmentData"
  :formattedGenerationArr="formattedGenerationArr”
>
</cashflow>
<template v-slot:footer> </template>
</ROIText>
```




```
<ROIText>
  <template v-slot:header>
    <h3>Balance Remaining</h3>
  </template>
  <p>End of 1st year:</p>
  <div v-katex="'\\small Balance = Capital\\ Cost + Loan\\
Interest - Amount\\ Saved'"><
  /div>
  <p>2nd year and on:</p>
  <div v-katex="'\\small Balance = Previous\\ Balance + Loan\\
Interest - Amount\\ Saved'">
  </div>
        <template v-slot:footer> </template>
</ROIText>
```



Code Snippet 2. Component Templating

As seen in the code above one could swap out the template for different contents, but the overall theming will remain consistent.

Design Pattern

Publish-Subscribe Pattern

The publish subscribe pattern is useful for Vue. By creating an event bus, components are able to emit events to the bus, and then any components listening to a specific event on that bus are able to deal with the data in any way they need to.

The publish-subscribe pattern is useful as opposed to the observer design pattern, as the publishers of the event, and the subscribers of the event, do not know about each other. This leads to increased code decoupling and flexibility.

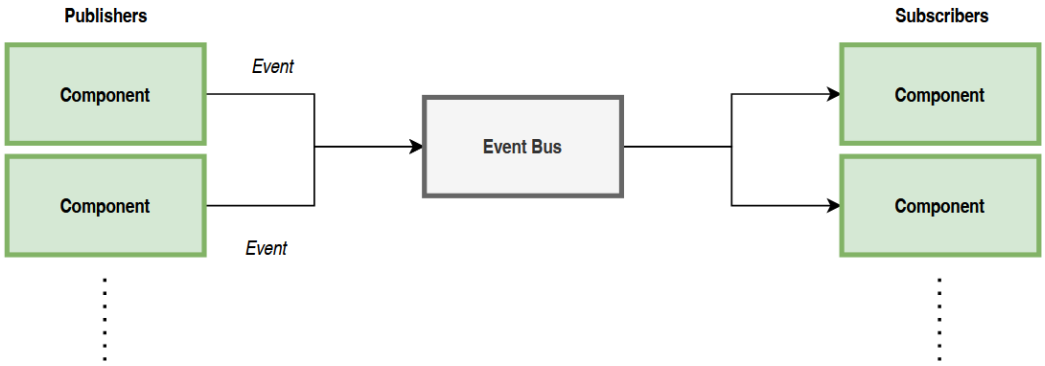


Figure 3. Pub-Sub Diagram

Good Practices Followed

Variable naming

Variable names are used such that at a glance, a reader can understand the intention. For example, in ***Inputs.vue*** the following variables are initialized in the data section of the component.

```
formInputs: {
        latInput: "",
        longInput: "",
        consumptionInput: "",
        directionInput: "",
        ...
}
```

Code Snippet 3. Variable Naming

The variables are clear in their intention, they are sections of the form that deal with latitude, longitude, consumption data, and direction. They are not misleading, and their context is clear. Another aspect that supports their quality is that they are human readable, that is, they are actual words someone can pronounce.

Comments

Comments were used in the project, rather than explaining what something does, to explain why something is written in a particular manner. An example of this can be found in *ConsumptionVsEstimateChart.vue* in the chartOptions tooltip formatting function.

```
return ("Date: <b>" + Highcharts.dateFormat( "%a, %b, %d, %I:%M %p",
            new Date(dateMS)) //Need this as this.xAxis doesn't account for timezone offset
automatically.
```

Code Snippet 4. Inline Commenting

Another way comments were used was to describe functions and their purpose. The purpose can include any side effects or returned variables.

```
/**
        * Get lat/long from address string request
        * @param Request $request The request containing the string of the address to find
        * @return  json returns lat/long information about location entered
 */
        function sendAddressRequest(Request $request)
```

Code Snippet 5. Function Commenting

DRY (Don't Repeat Yourself)

Although coding to the componentization of UI elements and logic standard of Vue limits repeat code, the project still ended up repeating code that is very similar. This is especially true for the ***Inputs.vue*** component.

Pre-Refactor

```
<div class="symbol-input">
  <VueInputUi
    :id="'panel_area' + index"
    v-model="panel.Area"
    label="Panel Area"
    :ref="'panel_area' + index"
    :dark="darkMode"
    required
    :loader="loading"
@focus="
$refs['panel_area' + index
][0].$el.nextElementSibling.classList.add('focused
')"
@blur="$refs['panel_area' + index
][0].$el.nextElementSibling.classList.remove('focu
sed')"
/>
<i>M²</i>
</div>
```

Refactored

```
<symbol-input
  :id="'panel_area' + index"
  v-model="panel.Area"
  label="Panel Area"
  :ref="'panel_area' + index"
  :dark="darkMode"
  :loader="loading"
  symbol="M²"
/>
```

Code Snippet 6. Refactoring Example

These inputs always contain the div "symbol-input" and contain the two functions @focus and @blur, which are repeated for every input with a symbol. Looking at the above code we could refactor it into its own component, removing the need to specify the @focus and @blur, as well as embedding the symbol into its own datafield. This would reduce the amount of

repeated code, and greatly improves the readability. The code in the project was written by first time Vue developers, and thus the understanding of the power components in alleviating repeat code was poor in the earlier stages of development, but was improved on as the project went along.

Laravel – Back-end API

Laravel was used as the back-end framework for the Solar Size project. It was used to serve the Vue file, as well as run a few simple API calls needed to get the data for the controller.

Controllers

The Laravel side of the code dealt with getting code from the model. Since we aren't storing any data on our server, the data comes from API calls, specifically to the NASA solar data, and Nomatim street map data.

The web page will make a request to an API route, which in turn is handled by a controller. Since a controller has similar functionality regardless of the exact details of the implementation, it is useful to inherit some functionality, as seen in the diagram below.

The quality of the Laravel code is easier to reckon with than Vue. Since there is less code written, it is inherently more readable, and the file sizes are small (around 80 to 150 lines).

A small thing to notice is using the Laravel features to reduce the amount of code. For example, they allow for one line routing, which specifies a controller, and then a function. This makes it extremely easy to read, and to change at a later date, as we can either change the function itself, or call a different function. The code can be found in ***api.php.***

```
Route::get('/estimate', [EstimateController::class, 'executeNoPanels']);
Route::get('/estimateOptimized', [EstimateController::class, 'executePanels']);
Route::post('/uploadCSV', [FileController::class, 'uploadFile']);
Route::delete('/uploadCSV', [FileController::class, 'deleteFile']);
Route::get('/getCoords', [AddressController::class, 'sendAddressRequest']);
```

The code in Laravel also follows the common variable quality indicators such as having useful variable names, short live time for variables, and proper text formatting ('camelCasing'). Examples of this can be found in ***AddressController.php***

```
if ($this->overLimit()) {
        $limiter = $this->getLimiter();
        $timeUntilAvailable = $limiter->availableIn($this->getKey()) + 1;
        sleep($timeUntilAvailable);
        return $this->sendAddressRequest($request);
        }
```

```
        $query = $request->address;
        $requestUrl = "https://nominatim.openstreetmap.org/search.php?q=" . $query .
"&countrycodes=ca&limit=1&format=jsonv2";
        $response = Http::get($requestUrl);
        $latitude = $response->json()[0]["lat"];
        $longitude = $response->json()[0]["lon"];


        $this->getLimiter()->hit(
        $this->getKey(),
        1 //1 request per second
        );
        if ($response->successful()) {
        return $this->respondWithData(['lat' => $latitude, 'long' => $longitude], $response->status());
        } else {
        return $this->respondWithError($response->body(), $response->status());
        }
```

As can be seen, a query string is initialized, then immediately used, indicating short live time. As well as variable names such as $requestUrl, and $latitude are clearly understood, without being cumbersome to read.

Conclusion

The quality of the code follows guidelines in some respects. The code is separated into distinct files, with each having a purpose, though sometimes, that purpose has morphed into separate responsibilities, without being a separate class/component. Components are a large part of the benefit of Vue, and in future development, an even greater breakdown of pages into components would be warranted.

Object-oriented programming was a harder thing to grasp when implementing the front-end with Vue. Most tutorials do not use SOLID principles. Since learning Vue was occurring at the same time as developing Vue, some aspects of SOLID programming were neglected, but refactoring has helped to implement those ideas, to create better code, e.g. templates and the open-closed principle.

Design patterns are used to make maintainable code, reducing the specificity of code, that is, they are easily understood, and solve problems in a robust and generalized way. The design patterns we used were the MVC, which is standard for web development. Another extremely useful pattern that was used was the publish-subscribe pattern. It was nice to use this pattern, as it made reactivity driven by events, which is easy to extend, or alter.

Some code formatting things were done well, such as variable naming and standardized formatting. While other code formatting was less than stellar. Thinking specifically of how some component files in Vue are many lines long, and repeat code. Breaking code into smaller

chunks is supremely useful on larger projects, and it was easy to understand the justification for small code files by the end of the project.

Overall the project was a great learning experience. With a larger codebase, and multiple people working on it, it was different from almost all projects done throughout the engineering coursework. As the project developed, the true understanding for why code quality is important was reinforced, and helped to create insight. If the project were to continue developing, it would highlight that quality code is an ongoing process, and can always be improved upon.

References

Abrickis, A. (2019, August 28). Vue communication patterns: An intro to “props-down” and “events-up” pattern. *Quick Code*. https://medium.com/quick-code/vue-communication-patterns-an-intro-to-props-down-and-events-up-pattern-d53340d2c94

Fekke, D. (2021, April 19). *Why You Should Not Use Classes in JavaScript*. Medium. https://javascript.plainenglish.io/why-you-should-not-use-classes-in-javascript-ca960d13c625

Gutha, S. R. (2015, August 31). Code Review Checklist – To Perform Effective Code Reviews. *Evoke Technologies*. https://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/

Importance of code quality. (2018, October 8). TatvaSoft Blog. https://www.tatvasoft.com/blog/importance-code-quality/

Rome, P. (n.d.). *What are Non Functional Requirements—With Examples*. Perforce Software. Retrieved March 21, 2022, from https://www.perforce.com/blog/alm/what-are-non-functional-requirements-examples

Why is code quality important? (2019, July 24). *Codegrip*. https://www.codegrip.tech/productivity/why-is-code-quality-important/