Redesigning Jet

A documentation of the involved Software Processes

Ojaswirajanya Thebe

Abstract

This report documents a project involving the redesigning of Jet from a software engineering perspective. We provide an introduction to Jet, the Gadgetron toolchain and its significance. We then discuss our software engineering processes in terms of knowledge areas defined by the Software Engineering Body of Knowledge (SWEBOK) [5]. We end by summarizing our successes and a discussion for the future of an application such as Jet.

Introduction

The Gadgetron toolchain makes the design and building of gadgets a lot more accessible to the everyday individual. A traditional approach involves the arrangement of circuit boards, placement of supporting components such as resistors, routing of components and other processes, some of which may need to be performed manually, or with little automation. Somewhere along this process, hidden beneath all the technical obligations, we lose track of the essence of our gadget. Gadgetron users, however, will be able to focus on the gadget's essence: what it looks like, how the user interacts with it and what it does. Gadgetron then implements the electronics and manufactures the gadget automatically. Our goal is for anyone to build any kind of gadget they can dream of [1].

Jet is the user-facing frontend for Gadgetron, allowing them to design their gadget. Our overall goal here is to provide a simple and intuitive user-interface, the simpler the better to reduce the barrier to entry, that interfaces with our backend tools. Jet provides a graphical interface for building gadgets from components, the output of which provides schematics for processes further down the toolchain.

Jet is available as a web application built using Typescript with the AngularJS framework. We include support for touch devices, and have made an effort to focus on the UI/UX aspect as well, considering that Gadgetron is scheduled to be used for entry-level undergraduate courses in the following academic year.

A screenshot for Jet is shown below. The center portion is the graphical representation of the gadget, consisting of multiple components. Users may drag additional components to their gadget from the component catalog displayed on the right. Components may be arranged via the graphical representation, or in more detail via inspectors which can be accessed through the tabs on the top. Gadget information may also be imported or exported to text data. The current workspace is automatically saved upon modification, allowing a user to resume building their gadget at a later point. Users can switch layouts, providing a different view depending on their requirements.

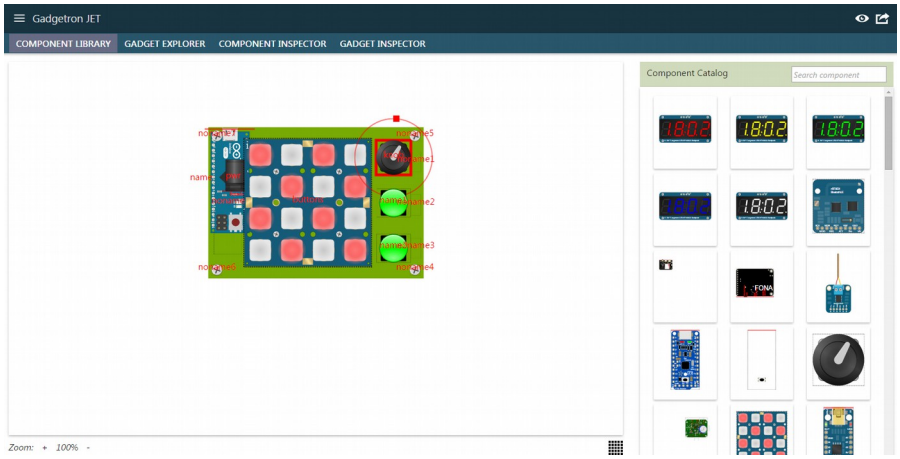

Illustration 1: Screenshot of Jet.

Software Engineering Processes

This section describes the software engineering processes we used in reimplementing Jet. Prior to this redesign, Jet existed as a single HTML page with a supporting jQuery script. The HTML page contained 94,817 lines of pre-generated (static) content, and the single script contained 1197 lines in a mix of unordered functions. The lack of a clear code structure made Jet a difficult application to deal with and coupled with the lack of any safety checks was vulnerable to new errors being introduced. Keeping this in mind, Jet was rewritten from the ground up with with a more detailed software engineering approach to aid in the design, development and maintenance of the software, by the application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of the software [2].

Software Requirements

Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem[3]. It is worth noting that the software engineering industry widely acknowledges the relationship between the success of the project and the quality of this process. Here, we describe our systemic approach, aka "*requirements engineering*" to formalizing our requirements.

Requirements Sources

We drew on our stakeholders to provide the requirements for Jet. Stakeholders included the user, for whom we wanted to provide an intuitive interface; the developer, for whom we wanted to provide a well engineered solution; as well the overall goals for Gadgetron.

Elicitation Techniques

Here we summarize the techniques we used to gather and refine requirements.

- Facilitated meetings: Biweekly meetings allowed us to brainstorm, summarize and review our requirements. These meetings were held with the entire Gadgetron group including both developers and product users. Discussions with the group helped us avoid any conflict that may have arisen in working individually.
- Prototyping: Our biweekly meetings also provided us a chance to present prototypes which were valuable in clarifying our interpretation on the requirements.
- Observation: We were able to observe two undergraduates use Jet in how they interacted with it, which provided us with additional insight on what we could improve.

Project Goals

Based on our requirements, we were able to come up with the following broad goals for Jet.

- Modularization of components: A criticism of the existing version of Jet was that the entire application was written within a single jQuery script. Code boundaries were not clearly defined, which made tracking causality and the effects of code changes hard to do. As a result, modification of existing code was a daunting task since there were no clear interfaces between objects. The redesign would modularize Jet into different classes that use encapsulation to provide clear interfaces. This facilitates safer code modification and improves the ease of code searches/tracking causality.
- Separation of data, logic and views: Jet contained an architecture where its logic was tightly coupled with its view. This required changes to both logic and views when updating its presentation, which is a common task for an updating web application such as Jet. This redesign would reduce coupling of logic and views. Furthermore, decoupling data from logic would allow for reuse of modules.
- Data abstraction and hierarchy: An important goal in design is to identify a program structure that simplifies both program maintenance and program modifications made to support changing requirements [4]. Jet would use data abstraction to separate implementation from behavior. This would allow for easier maintenance by creating smaller scopes, and also support easier modification by simply substituting the implementation.
- Adding type-safety to Javascript: Javascript is a loosely typed language, and one drawback includes the possibility of introducing errors due to mismatched types, which increases as the application grows in size and complexity. Our application would add type-safety to Javascript, realized through a compiler to provide static checking to reduce errors.
- Improved user-experience in design and usability: A product requirement for Jet is to lower the barrier of entry for users designing gadgets. With that in mind, Jet would need to be designed properly from a UI/UX perspective.
- Proper software engineering tools and methods: Considering the developer as a stakeholder, Jet would be written using proper software engineering tools and methods such as documentation, task management, build and configuration management etc.

Software Design

Software design has been defined as both “the process of defining the architecture, components, interfaces, and other characteristics of a system or component” and “the result of [that] process.” [5]. Here, we describe the translation of software requirements into the structure of the application.

Architectural Design

Jet uses AngularJS to create a Model-View-ViewModel architecture [6]. Each instance of the application contains an *Application Context*, which stores data and services used throughout the application similar to a singleton design pattern. An application also contains an array of *Perspectives*, which are HTML files declaring the user-interface. The user-interface in a perspective is defined in terms of *Directives*, which can be thought of as custom HTML elements. Directives in turn, contain their own data, logic and HTML view. We pass reference to the application context to each directive, which gives it access to data and common services. When a perspective is loaded, all directives declared within the layout are initialized and displayed as well. The directive has a reference to the data objects, allowing it to run operations such as getters, setters and lookups directly. The following subsection provides a more detailed look at the various modules in Jet.

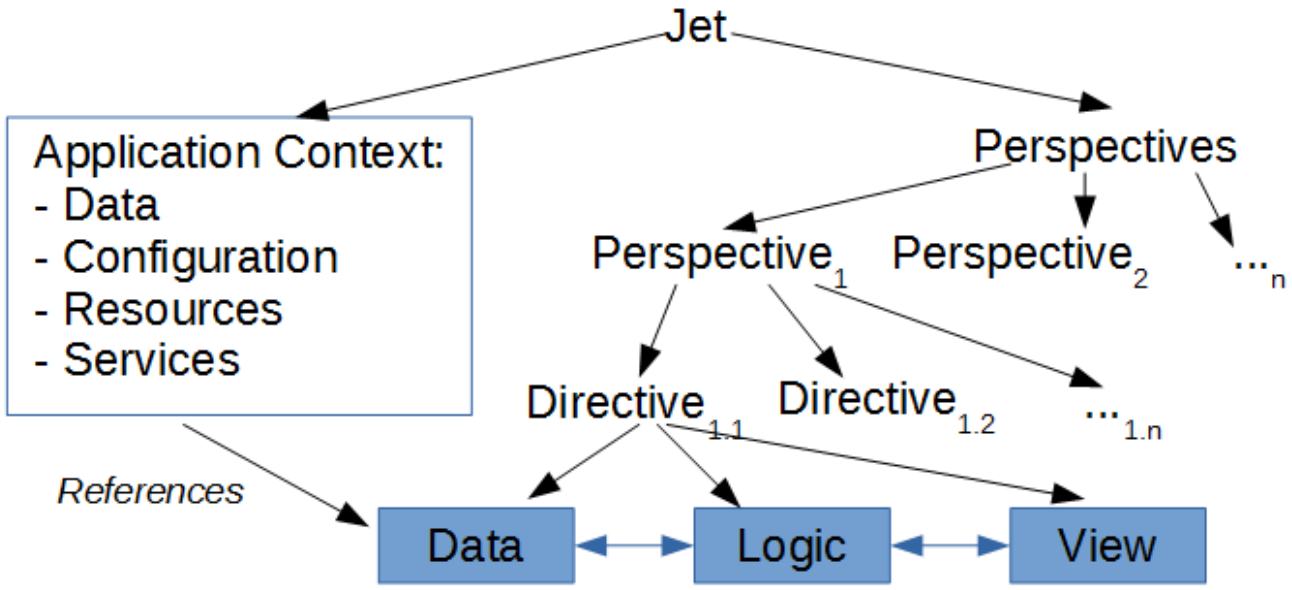

Illustration 2: Architectural design for Jet.

Detailed Design

Here, we provide a detailed description for each component in Jet.

Application Context

The application context contains contextual information for each instantiation of Jet. Objects available within the application context are available as singletons, such data objects, configuration and resource objects and access to services. The application context can be passed as reference to other components.

Gadget data object

The gadget data object contains information on gadgets currently being built within an instance of Jet. It includes general information like gadget dimensions, name etc. as well as information on every gadget component that consists within the gadget. Read and write access is provided via a set of functions.

Catalog data object

The catalog data object contains information on the catalog of components available within Jet. Upon initialization, the catalog data object reads in data for each individual component such as Gadgetron specific IDs, SVG artwork etc.

Configuration and Resources

Strings and constants are declared as objects, requiring the implementation to use the reference instead of directly creating new values. This helps isolate the scope of modification. Examples of strings include names and error strings; and padding, borders and colors as constants. Initial values such as dimensions are also written as constants.

Eagle ↔ Display mapping Service

Eagle is a publicly available graphics software used within Gadgetron, with its own set of rules for rendering images. Images in Eagle have different origin points, and have a mirrored y-axis and rotation as compared to how browsers render SVG images. The images generated within our toolchain are built for Eagle and thus do not render properly out of the box in Jet. This service provides functions to convert

coordinates from Eagle into browser coordinates, which lets them display correctly. It also provides for converting browser coordinates to Eagle coordinates which are required to interface properly with Eagle.

SaveState Service

This object provides persistence via a key-store mechanism for storing data in Jet. This service is currently used by Jet to store the state of the gadget data object when a change is detected. The last saved data is then loaded whenever Jet is re-initialized, allowing a user to auto-save and resume their work. The implementation uses the *LocalStorage* mechanism in HTML5, although the use of information hiding means the actual implementation can be replaced to any other mechanism such as cookies or server-based solutions.

Key handler

The key handler offers a wrapper around a browser's *onkeyup, onkeydown,* and *onkeypress* event handlers. A key handler object is initialized with three functions that handle the *keyup, keydown* and *keypress* events, while providing an abstract set of functions to query the event.

Touch handler

The touch handler is similar to the key handler, but handles the *mouseup, mousedown*, *mousemove* and the similar *touchstart, touchend* and *touchmove* events. It keeps track of the input events to provide an abstract set of functions that calculate values such as the translation between inputs and the rotation around a given point. It also unfies the handling for mouse and touch events.

Directive base class

A directive is a construction in AngularJS for custom components. These can be placed within HTML as normal elements, ex. ** <catalog-component></catalog-component> ** to populate the catalog component within the div in the DOM. We use an abstract directive base class to hide the actual boilerplate code and to enforce our constraints, such as requiring a directive to have reference to the application context. All Jet directives extend this base class, and override its abstract methods to handle implementation. A more detailed description of AngularJS and directives is perhaps out of scope for this document, and better referred to from the online AngularJS manual.

Catalog directive

The catalog directive is responsible for representing the available components in the catalog data model. The directive currently renders the artwork and name:

The catalog directive is responsible for representing the available components in the catalog data model. The directive currently renders the artwork and name as shown in the screenshot. With the help of the touch handler, users can use their mouse or finger for touch devices to drag a component into their gadget. They may also use the search bar to filter the components shown.

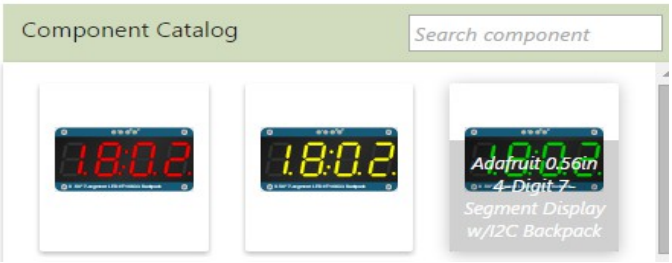

Illustration 3: Component Catalog directive.

Gadget directive base class

The gadget directive is responsible for rendering the container for the gadget, within which users may add their components. Based on changing requirements and code commits, the gadget directive is the most updated module in Jet, and as such we have separated this into a base class and extending classes to help with modification. The base class has access to all data objects and services within the application context, but requires an extending class to handle the display/presentation. It contains a set of functions that define the behavior of the gadget such as adding and deleting gadget components, among others. An

implementing class would then manage how the adding and deletion are handled visually. Since implementing classes are responsible for rendering, they are allowed to be creative by adding user-facing details such as selection boxes, input handling, while keeping in mind that their interaction with the actual data is limited to the set of functions exposed by the base class.

Gadget-Component directive base class

While the gadget directive represents the container, a gadget-component represents an actual instance of a component added to the gadget. Similar to the gadget base class, extending classes add new UI/UX elements such as transformation controls for translation and rotation, display of supporting text messages etc. Pictured is an example of a knob component on a gadget. The extending class is responsible for how the artwork is displayed, along with handling the rotation (via the touch-handler), and other user-facing aspects.

Illustration 4: Knob component.

Component inspector directive

Although gadget component properties can be manipulated directly from the board via the transformation controls, the component inspector provides an alternate text-based option to achieving this in a more fine-tuned manner. Changes made here affect the gadget data, which is then reflected in all other directives (such as the gadget-component) as well.

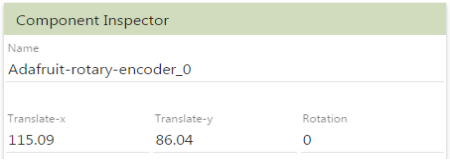

Illustration 5: Component inspector.

Gadget inspector directive

Similar to the component inspector, this directive instead allows the user to manipulate properties related to the actual gadget itself such as the height, width and name of the gadget.

Gadget explorer directive

This directive displays a tree of the different components placed on the gadget. Users may directly interact with entries in the tree for relevant actions such as selections and deletions.

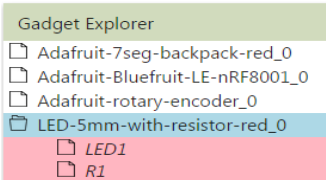

Illustration 6: Gadget explorer.

Perspective

A perspective is an HTML file containing the layout for Jet's UI. The directives we built can be declared in the HTML, allowing multiple layouts for testing, prototyping and different device types. An example of a perspective HTML file looks like: ** <gadget-explorer class="fill-width"></gadget-explorer> ** to build a gadget-explorer directive with the given class. We currently have two main perspectives for mouse and touch devices respectively.

User Interface and User Experience Design

The UI/UX design evolved as a product of continuing prototyping. Layouts were inspired from Microsoft Visual Studio and Unity. The concept of perspectives was taken from the Eclipse IDE, where users are allowed to switch between different perspectives such as coding, debugging etc. The touch perspective was influenced by the general design of mobile web applications. The overall look and feel was themed using Google's material design.

Control and Handling of Events

Isolated scopes

Although the application context contains data that is meant to be available throughout the context of the application, it is not a good practice to expose the entirety of the data. Access to directives can be controlled by passing it in as an HTML attribute and setting the data inherited by default to be null. For ex.

<gadget-explorer catalog-model="catalogModel"> gives the directive access to the *catalogModel* object. This is a feature provided by AngularJS, and one we have enforced to ensure that directives only have access to the objects they need, as the default behavior makes them inherit everything from their parent. Non-directive classes are instantiated normally and their access can be controlled via standard means such as the constructor.

AngularJS - Event handling and Data binding

Jet would typically be written as a set of event fires and handlers. We chose AngularJS as an alternative to manually setting up this event framework. As a brief overview, AngularJS allows for objects to be watched for changes instead, while updating any HTML the object may be bound to. This reduces a lot of code overhead.

Error and exception handling

Components in Jet throw errors under necessary conditions which are then handled at the top level.

Quality analysis and Evaluation

Biweekly Gadgetron meetings provided an opportunity for software design reviews. We spent the initial four weeks of this project on group-based critiques and refinement of the architecture, components and other design artifacts.

Prototyping was also part of the design process, where we built smaller versions of Jet in other frameworks such as Google Closure, and experimented with external libraries and modularizations to evaluate our overall design.

Software Construction

Software construction refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing and debugging [5]. This section details our efforts to improve the quality of this process to deliver better quality code.

We made an effort to follow coding standards to produce code that was easier to read and identify causality. The source code directory structure has been organized according to modularization. The source contains documentation including a description of the file, its functions and attributes. Variable naming schemes and namespaces were inspired from industry experience, such as the use of prefix underscores in Javascript to denote private members at Google. We used an *evolutionary prototyping* software development lifecycle model, where we refined our prototypes that we presented at every biweekly meeting.

An effort was made to properly incorporate the Jasmine testing suite into Jet, but this has not been fully realized due to time and manpower constraints, instead relying on informal technical reviews. The use of Typescript and the strongly-typed flavor it provides Javascript provided a level of static checking.

Software Engineering Tools

Software Construction Tools

AngularJS and Typescript were the main tools used for this reimplementation of Jet. AngularJS provides a simplified event handling system and data binding which let us focus more on the essence of Jet instead of the technical overhead.

Typescript is a pseudo-language built over Javascript that adds a number of features such as object-oriented programming and type-safety. Its strong integration with Microsoft Visual Studio provides an efficient environment for working with Typescript projects like Jet, with features such as code completion and IntelliSense. The Typescript compiler performs static type and functional checking and handles

dependencies of multiple Javascript files.

Release and Build tools

We used the Node Package Manager to handle package level dependencies for Jet. The required packages are TSD for typescript dependencies, Bower for Javascript dependencies and Grunt for build management.

TSD handles dependencies for Typescript definitions, which are similar to header files, allowing Javascript libraries to be used within Typescript. Bower handles dependencies for Javascript libraries. The use of such package managers has allowed Jet to be written as a smaller, portable application. Grunt provides a configurable Javascript task runner, i.e build manager, and is currently used for tasks such as the proper importing assets into Jet.

Miscellaneous tools

We initially used a Javascript library called Fabric.JS to handle image transformations on the gadget directive. It offered high level functionality such as transformation controls and selection boxes, but it worked by rasterizing the SVG images leading to a loss in quality and native SVG functionality. Another library called Snap.SVG provided good wrappers for SVG functionality, but none of the high level functionality such as Fabric.JS. Our first builds worked with Fabric.JS until we required the use of the SVG functionality. We then wrote a library that bridged Fabric.JS and Snap.SVG called Snabric.JS, effectively providing Snap.SVG functionality to Fabric.JS images. This worked well until we required more customization from Fabric.JS and required better quality images especially when zooming in, something still missing due to rasterization. As our third build, we implemented our own SVG image transformation solutions. This is integrated into Jet, and offers transformation controls, selections, text and mouse/touch handling.

Artwork images used in Jet are generated in a different part of the Gadgetron toolchain and are written for Eagle. Images are sized at 300x300 units or more, which includes space around the actual component artwork. This was visually confusing and also complicated the transformation and other operations in the Eagle<->Display Mapping module. As a solution, we created a tool to automatically resize component artwork for Jet. This has been included as part of our build system.

Successes

Jet has been rewritten while keeping software engineering standards in mind. We have been able to meet our project goals and write software that is approachable by an engineer other than the developer. Jet is better organized, easier to read, and provides the framework to safely change the look and behavior of the application.

Jet has also met its functional requirements of integrating with Gadgetron. From a user's perspective, Jet has been successful at designing gadgets and generating the proper schematics to be used with the rest of Gadgetron. Shown below is a picture of a countdown clock gadget designed in Jet.

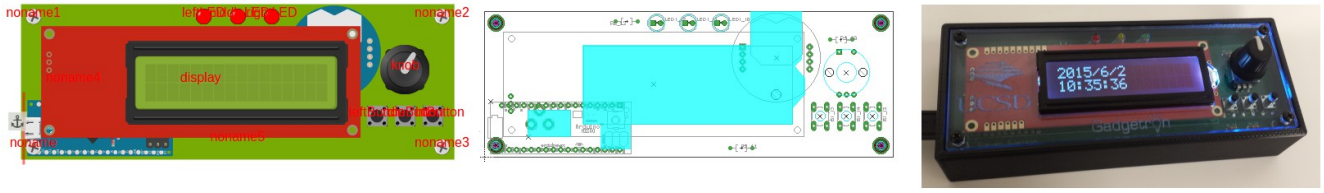

Illustration 7: Pictured: A design from Jet to schematics, to ultimately a gadget.

Jet will also be used as a teaching aid for an entry level undergraduate course in the Fall of 2015. The course will introduce students to hardware design and programming, and potentially have them build a robot with Gadgetron as a project.

Discussion

Jet, along with the rest of Gadgetron has a grand goal of making gadget design and building much more accessible to everyone. There are still many issues to be addressed and discovered to realize this goal both in terms of UI/UX and functionality. At its current state, Jet will benefit from regular UI/UX refinements and interfacing with more of the backend tools, preferably automatically. Such enhancements have been scheduled to add a final polish before making its debut at the undergraduate course next academic year.

Gadgetron was meant to allow a user to focus on the essence of their gadget, and this ideology even applies to this redesign of Jet. A well planned and engineered application lets the developer focus on efficiently improving the essence of their application without being bogged down by code overhead. This version of Jet provides a stable base for future development even as code size and complexity increases.

Acknowledgements

I would like to thank Professor Steven Swanson for the mentorship and opportunity to work on Jet, and for helping me hone my software engineering practices, Devon Merrill for help in coding the Gadget data model, and the rest of the Gadgetron team for bringing this project to fruition.

References

[1] Gadgetron: Synthesizing Electronic Gadgets (http://nvsl.ucsd.edu/index.php?path=projects/gadget)

[2] IEEE Standard Glossary of Software Engineering Terminology, IEEE std 610.12-1990, 1990.

[3] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques, John Wiley & Sons, 2000.

[4] Data abstraction and Hierarchy, Barbara Liskov, MIT Laboratory for Computer Science, May 1998.

[5] Guide to the Software Engineering Body of Knowldege, Alain Abran, James W. Moore, 2004 pp 4-1.

[6] Smith, Josh (February 2009). "WPF Apps with the Model-View-ViewModel Design Pattern". MSDN Magazine.