

User Interface Improvements To The Boeing Company's Virtual Flight Deck

Cindy Balacy, Eugene Cherkassky, Alex Donaldson, Anton Polinger

Abstract:

This paper will focus on the Seattle University project with The Boeing Company during the 2007-2008 school year. This project involved working on the User Interface components of the Boeing Flight Simulator. The discussion will include the background of the Virtual Flight Deck project, improvements that were made to the existing functionality, new functionality, and future improvements.

The Boeing Company was the project sponsor for the year-long Seattle University project. The Boeing Company is a major aerospace and defense corporation. Some of their commercial airplanes include the Boeing 747, which is the first wide-body commercial airliner produced, and the Boeing 777, which is the largest twin jet airliner. Some of Boeing's customers include Singapore Airlines, Air France-KLM, Emirates Alliance, United Airlines and many more. Safety and training are two key areas where The Boeing Company continues and strives to improve with each new addition to their flight family. The main focus of the project is to improve the user interface of the flight simulators.

A simulation is defined as "an imitation of some real thing, state of affairs, or process". Simulating something usually entails modeling key characteristics of a physical system. Computer simulations allow modeling of real-life situations so it can be studied to see how the system works. By changing variables, predictions can be made about how the system will behave. The discussions will be centered on the simulation of Boeing airplanes, specifically the User Interface components.

Simulation provides many benefits for the Boeing Company, including cost savings, exposure of flight environment to more personnel and error modeling. It allows lots of personnel exposure to the flight experience at a fraction of the cost.

The project that the Seattle University team worked on consisted of providing improvements for the Virtual Flight Deck project. A Virtual Flight Deck (VFD) is a simulated application of how flight deck panels are displayed and put together to resemble a realistic environment. VFD panels should generally look and behave like the real panels found in the plane that the application is trying to replicate. The Boeing Airplane Simulation Lab group supports a variety of software systems and design toolsets for producing flight visualization products. One of these products is the Virtual Flight Deck tool. True to the VFD definition, this is a tool which provides access to cockpit displays and controls as they would appear in the airplane. This tool interfaces with a running simulation for the full effect of a realistic environment. This tool helps with training and software testing.

The project requirements are very straightforward and that it aims to create an integrated Netbeans environment for panel design and use the environment to create a set of working panels. The Simulation Lab team provides us with a working example of how they design and develop the panels for the Boeing Commercial 777. Although the general and overall requirement is straightforward, technical difficulties arise as we dig into their current architecture. The VFD is made up of panels, which are further made up of widgets.

The main task of the project was to refactor the existing code to make it more extensible, robust, and easier to use. Widgets are the individual controls, buttons and knobs. When the widgets are placed and grouped together, they make up a panel. And multiple panels together make up a full replica of a cockpit. The current widget library design made it very difficult to maintain and create new widgets. Also, the property list that is displayed for each widget includes a long list of attributes, many of which never change and don't have any real value. The project requirement is to modify the design on how to create new widgets that make it easy to maintain in the future, and to declutter the property list.

The VFD code had been growing for years, expanding as necessary via copying large chunks of code and pasting them into new classes, rather than extending the old ones. This can work in a small test application, but quickly fails when things start scaling up. The easiest way to address this is to regularly refactor the code to ensure that the overall architecture still makes sense, pruning away areas where the project has become bloated and cleaning up the areas that remain.

Our approach was to classify the Widgets based on their "state" variables. Working with the client we were able to come up with 5 interfaces: IWidget, IBinaryWidget, IDiscreteWidget, IContinuousWidget, and IStringWidget. The IWidget interface is the parent of all the Widgets, IBinaryWidget handled everything with only two states, IDiscreteWidget handled Widgets with integer state values, IContinuousWidget was the parent of Widgets that required double type state values, and IStringWidgets were used to group labels that didn't have state variables.

Once the interfaces were defined we set about to clean up classes that had suffered from copy and paste coding, for example we combined SquareButton1, SquareButton2, SquareButton3, and SquareButton4 into a new class that incorporated all the necessary behavior, now there only exists a single SquareButton. As we were cleaning up we made sure to inherit from the correct interface and implement the newly designed methods. Providing not only a cleaned up version of the library but an abstracted architecture, building new Widgets in the future will be much easier, and more importantly the code that interacts with the Widgets will be reusable since it won't be tightly coupled to the base classes.

The next phase of the project included improving the Panel Template Design. A generic design is crucial in a design that is easy to maintain and grow. The prototype panel template for use with Netbeans includes implementations of method developed in a generic way, so that the panel and any widgets on it can work without any hand written interface code development. The following lists the requirements of how the panel should work:

- Discover what widgets it contains, what simulation variables are included in those widgets, and build a list.
- Provide the list of simulation variables that the panel is interested in via the `VariableRequest` method.
- Provide code to update the widget state in response to simulation variable changes via the `changedValue` method.
- Provide processing of widget change events and generate simulation variable updates via the `stateChanged` method.

The client was very interested in extending the library to embrace graphical programming features. These allow the users to build their entire user interface without having to understand the Java code that is involved, greatly increasing the accessibility of the product and theoretically cutting development time. It was requested that we use Netbeans as the framework tool with built in palette feature. This allowed us to focus on building the library and not have to worry about creating the tool that the end user would use to build their interface.

Netbeans framework provides a quick and easy way of building user interfaces, but forces the user to look into the source code and provide behavior for the panel manually. Boeing wanted us to research the possibility of bypassing this step, building on a prototype panel they had provided. The panel that was provided attempted to fix this by providing a large if/else block with a handful of widgets, some of which weren't used in the panel. It is easy to see that this method, if fully developed, would have surely masked the complexity from the user, but would have made any future development complicated.

Our approach to this problem was to push this behavior further down in to the Widgets. This would allow for easy maintenance of the template, since it wouldn't have to be updated every time a new Widget was developed, and would consolidate the behavior of the Widgets solely in the Widgets themselves.

Now, to understand the next steps we should explain the general communication architecture between the flight deck and simulator. The simulator mimics data coming from the airplane's sensors. Each Widget is assigned a number of String variables that correspond to inputs from the simulator. For example a dial may have a variable that corresponds to its light and a variable that corresponds to the position of the dial. When the dial is spun it sends the key (variable) and a value to the simulator. In a real

airplane this may adjust the air conditioning. Then if the light variable is adjusted in the simulator the light would be turned on the Widget, perhaps notifying the pilot that the air conditioner is now on.

To make this work the template needed to detect the interactive components it contained to allow communication between these Widgets and the data simulation engine. Because of the earlier refactoring we were able to utilize the high level interfaces to call the methods and didn't have to create a scary, difficult to maintain if/else block. Doing a look up of the variable and its corresponding component, then ensuring that the event was passed to that component the template panel became a sort of triage so that Widgets didn't have to listen on every method, but the panel wasn't responsible for behavior, it routed the events to the right place and let the Widgets take care of the necessary actions.

Some other improvements included the Fraction Layout Manager and Event Passing. Developers do not find it difficult to inadvertently drop a widget or other objects into another object layout, rather than the panel layout. The FractionLayout will not correctly resize all the objects on the panel. The requirement is to have FractionLayout operate recursively on all sub-layouts. Panels can be placed on top of other panels to create a larger panel. In this scenario, the event passing mechanism does not work as expected and state changes are not passed through to the simulator. The events of children panels are lost and are held back from the large panels.

A problem that was encountered in the previous version of the VFD was that events made it only as far as the top level panels, but more often, users wanted to embed a panel inside another one. When using a built in JPanel this behavior is not automatically generated, even in the prototype template panel Boeing had implemented this component was missing, forcing them to only develop single layer panels.

Luckily this was an easy fix once the Template was implemented. By expanding the methods that take care of communicating events to and from the panel we were able to not only pass to sub elements that were widgets, but also to sub elements that were panels.

The layouts have proved to be a problem of a different sort. Because of the drag and drop approach to the new VFD Panel Builder standard Java layouts couldn't be used. Typically Layouts like Grid Bag or Border Layout have to be tweaked manually to get the results you want and automatically generating code to do this was well beyond the scope of our project. What Boeing did want was to be able to resize panels and not a) lose the position of widgets relative to background images and b) not have the widgets distort.

The original VFD implementation used a custom layout in order to support resizable panels which also cause resizing of the controls on the panel. This layout worked very well for plain panels, but had problems when backgrounds were added into the mix. Resizable panels with backgrounds have controls that must be placed in fairly specific places. The problem is that by their very nature panels (and thus their backgrounds) can be resized without maintaining their proportions, whereas proportions of widgets on the panels must not change.

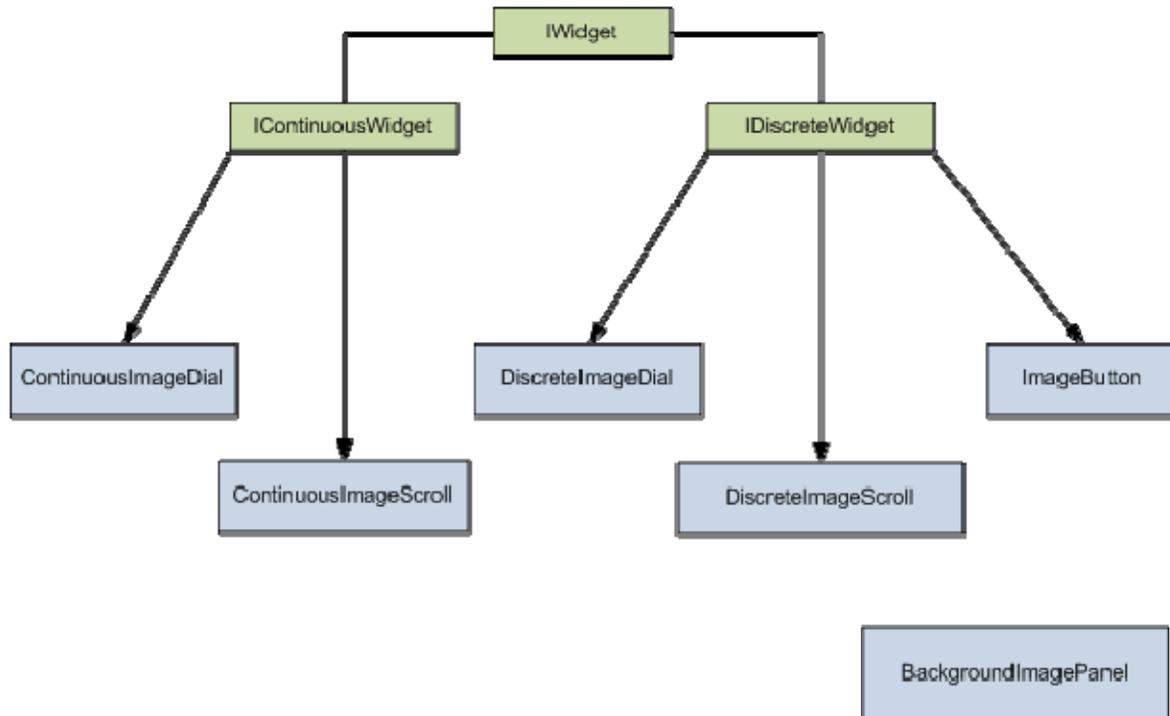
The VFD implementation of the fraction layout dealt with control resizing by calculating the factor by which the panel size has changed, and modifying both the width and height of the child widgets by the same factor. However, since the widget origins are by default located in their top left corner, the controls can slide around with respect to the backgrounds.

The solution to this problem is a fairly simple modification that pre-calculates the centers of the widgets, and resizes them around that center. As a result all four corners of a widget move, allowing the controls to stay in the same location relative to the background. Barring any extreme changes to the panel proportions, the widgets will thus stay in the same positions. The only problems appear when controls extreme proportions are used, magnifying one or two pixel alignment problems many times.

Up to this point, we have discussed the improvements of the existing functionality in the VFD. The main new feature that was added is the image widgets. The original implementations of the VFD controls were all drawn programmatically. Each circle, each line of image that represented a switch, for example, had to be defined in terms of coordinates and images on the screen. While this vector based approach is very easy to resize, it is also extremely difficult to scale to more complex switches. In addition, the resulting controls appear to be a very flat and abstract, and are hard to extend or modify. In order to correct this problem, a decision was made to design and implement a set of image-based widgets, which use photos of existing controls on control panels.

This approach has several advantages. First of all, it makes the control panels with such image-based controls appear to be a lot more real and thus produce a much closer representation of the final panel design. Second, and possibly more importantly, using pictures and photographs allows the developers to create and modify existing controls with greater speed, significantly shortening development time. For example, to modify the shape of a knob they need to simply take or modify a photo of a real-world control, and recompile the project. This allows the use of more complex shapes with higher realism and flexibility.

In an attempt to provide the main widget types that may be needed for control panel development, we have used the same infrastructure as the rest of our VFD implementation. Here is the architecture that we've chosen:



Since the image widgets all inherit from the same basic interfaces as the original widgets, the image controls can be safely used together with the older vector implementations on the same panel without any modifications.

The image widgets are all based on pictures, and use no vector graphics. In order to indicate various on-screen locations, unique and specific colors are used on the images. This allows the use of interesting effects, such as placing the axis of a knob at off center from the background image, allowing for more complex backgrounds.

Overall the project was very challenging and rewarding. It gave us a chance, as students, full exposure to the software development lifecycle. This project closely mirrored a real software project in terms of starting with a real project, improving existing functionality and adding new functionality. Some of the main features that were implemented included refactoring the code, fixing template, layout, event passing and adding image based functionality. Throughout the last year we were able to work through the project and solve the challenging issues that came up.