Web Application User Interfaces

Patrick Toral

March 18 2019

Abstract

This document discusses a design approach, entitled Mediator Pattern, for a experiment software user interfaces. We will describe a technology named Polymer used to implement the pattern within the context of a web application as well as detail its data flow of communication with the web browser. This paper will then focus on an implementation of this design approach on an economic experiment software featuring real time data visualization and web socket communication. The software highlighted supports research of High Frequency Trading (HFT) in multiple financial exchanges and market environments within the LEEPS Lab at University of California, Santa Cruz.

1 Introduction

Web applications have become increasingly popular and interactive with the increasing power of the web browser. The user interface (UI) or front end[1] of these web applications are becoming more complex requiring larger and more complicated code bases. The development team is then tasked with designing an ideal software that is scalable, robust, maintainable, and readable within a reasonable time constraint. Without any safeguard and proper design it is easy for the code base of front end software to take on a unreadable dependency ridden clump of code developed without future iterations of the project in mind. The safeguard against hacking together UI software is to use frameworks and design patterns that promote best practices. This not only allows for the current implementation to behave correctly but also allows the seamless addition of features. We will later confirm this by detailing how the front end framework[2] Polymer utilizes the browser created HTML[3] DOM[4] in an effective and efficient manner such that it promotes best practices. When Mediator Pattern

[1]Medium in which a user interacts with software

[2]A web application development tool that prevents developers reinventing the wheel by offering means to complete high level tasks

[3]Hyper Text Markup Language is the language of the internet. The language is used to describe the structure of a web page to be interpreted by the browser.

[4]The Document Object Model is a browser created tree like structure that is a W3C (World Wide Web Consortium) standard. It defines a standard for accessing HTML

is implemented by Polymer to develop UI software we see a code base that is effectively organized and achieves the previously mentioned tasks.

This paper will focus mainly on a subset of web application user interfaces in behavioral experiment software. This type of software has been and will continue to be a crucial tool researchers use to aid in gathering data for research across multiple disciplines. The LEEPS Lab [5] at University of California, Santa Cruz has a rich history of utilizing experiment software for behavioral economic research. The experiment software this paper will focus on is High Frequency Trading (HFT). It was built on a modern economic experiment platform named oTree [6] and was developed to emulate financial markets so that different trading strategies can be tested within the confines of the given market environment and exchange. Experiment software platforms vary from Affect5[7], E-Prime[8], to OpenSesame[9] with implementation depending on field of study and once built upon creates an application to be used to test users and retrieve data.

While these interfaces share the same set of goals of regular interfaces, in our case we had to design a software that was configurable by the researcher. So in addition to the previously stated tasks we had to represent multiple interfaces within the same application at the configuration of the researcher. Related to this point, behavioral experiments are usually complex in that the UI of the experiment software greatly effects the data retrieved. With that in mind, it is necessary to create a software that is configurable and receptive to change as it most likely be modified due to testing and constant refinement.

In this paper we will not describe the psychology of user interface design but rather the implementation of the Mediator Pattern through front end frameworks to develop a product that is efficient, scalable, robust, readable, and maintainable. Mediator Pattern is an important design approach when developing software and is commonly found in the design of software deployed across the world. We will discuss grouping the user interface into components defined by responsibility so that they can can be abstracted away from the specific implementation and be used across multiple experiment designs. Not only does this decrease development time as the software functionality expands and becomes more complex, it also eradicates dependencies throughout the code base and eliminates what is normally called 'spaghetti code'.

We will first discuss the problems and solutions that the previously mentioned Mediator Pattern aims to solve, followed by an example web application that demonstrates the Mediator Pattern through a front end framework named Polymer. Ending with an analysis of an implementation utilizing all of the mentioned elements. This analysis will cover the benefits and shortcomings of applying this design in production ready software.

[5]`https://leeps.ucsc.edu/home/`
[6]`https://www.otree.org/`
[7]`https://ppw.kuleuven.be/apps/clep/affect5/`
[8]`https://pstnet.com/products/e-prime/`
[9]`https://osdoc.cogsci.nl/`

2 Mediator Pattern

The Problem and Solution

Without the proper precaution, as the code base of a software becomes more complex with addition of features and refactoring the code will take on a phenomenon known as 'spaghetti code'. The reference comes from the characteristic of individual spaghetti noodles coupling together in clumps when one tries to scoop out their serving from a bowl, an all too common problem both in real life and in software development. In code this problem is translated when sections of code are dependent on other sections of code in one big messy clump. Moreover, the problem becomes more complex when features are added or refactoring is beneficial. Not only will the individual section of code need to be worked on so will the other sections and modules it is dependent on. This creates an increase in development time, makes the code less reusable, harder to maintain and read.[2]

The solution to the problem stated above is to utilize the behavioral design pattern known as the Mediator Pattern[1]. This design pattern defines a set of objects and their interactions between each other. At the foundation of this design pattern are two main classification of objects, Mediators and Colleagues. This approach details how a set of objects interact within the confines of a Mediator object. The Mediator has the responsibility of encapsulating objects and facilitating their interactions between one another. These interactions include communication, special behavior, etc. Colleagues are the objects contained within and maintained by the Mediator. With this approach developers can keep objects from referring to each other explicitly and define their interactions independently. If properly implemented this essentially eliminates the 'spaghetti code' phenomena mentioned above by promoting a common practice known as loose coupling. This practice contains the knowledge of individual components to solely itself. If we abstract this concept it proves to be beneficial when structuring communication in multiple industries.

An instance of an approach similar to Mediator Pattern being used is Air traffic control (ATC). The problem that concerns air traffic is that there is a limited number of runways available at any given time and a variable amount of planes that need this resource to perform their actions safely. If there was no air traffic control and planes communicated with each other individually it is obvious to assume what the issues would be. This design works in the most simple case but fails miserably as we scale up to normal operating conditions.

Say we have an airport, named Airport Santa Cruz, with one runway and two planes that are allowed to use it. If one is taking off and one is landing the planes would communicate directly to each other to find the best time to perform their actions safely and there would be no need for an ATC. If we were to add more planes the design would result in disaster as it would become increasingly difficult to communicate to all planes at any time. If there was an emergency landing necessary then that plane would have to explicitly communicate to all the other planes within reach to use the runway. The pilot would most likely have more important problems to attend to. As we can see this approach is neither safe nor reasonable to implement.

Thankfully, there exists a mediator between the planes and the runway resource, the air traffic controller, that controls the communication with all nearby planes. All requests and changes go through this mediator and it is prohibited for any one plane to bypass the mediator. If the same plane had to conduct an emergency landing all it would have to do is notify the ATC and they would deal with the logic and organization of plane arrivals and departures. Doing this eliminates the dependencies between the individual airplanes and furthermore allows for some observer within the air traffic control tower to know exactly the status of the runway at any given time.[4]

Within the context of computer science the implementation is similar. We have to identify how the objects are communicating and organize them into responsibility defined classes so that a mediator object can facilitate the interactions. These classes, also known as participants, that are necessary are defined as Mediators and Colleagues. In our above example we recognize that the Colleagues (airplanes) were communicating with each other explicitly and that as we scale up that approach would be unsatisfactory. Building on that realization we would then place a Mediator object (ATC) to facilitate this communication so that the only interaction for a Colleague (airplane) is to and from its Mediator (ATC). The Mediator would know the information communicated between itself and all its Colleagues.

3 Polymer and Mediator Pattern

Polymer[10] is a web components framework that simplifies and expands the tree like structure of the Document Object Model (DOM). The HTML DOM is a standard object model and programming interface for HTML. This model allows for HTML elements to be interpreted as objects with properties, methods, and events[3]. Polymer allows for developers to create custom HTML elements that are accessible and mutable within the DOM. This allows the framework to be paired with JavaScript to generate robust web applications. With this library there are concepts that need to be discussed in more detail as they will be mentioned throughout the rest of the paper.

We first have to discuss Custom Elements which is the foundation of the Polymer framework. This element defines a reusable custom component model for the web that is accessible in the DOM. This presents high level abstraction that comes with using Polymer. If you want to define a custom DOM component that serves a responsibility driven purpose then we would create a Custom Element. The Shadow Root provides a local encapsulated DOM tree that belongs to a Custom Element. In this Shadow Root we would write the HTML markup necessary for the Custom Element. In theory this promotes decoupling because the Shadow Root is only accessible through its Custom Element and the Custom Element only knows that it itself exists. So if anything needs to

[10]`https://polymer-library.polymer-project.org/`

happen to a Custom Elements Shadow DOM the Custom Element should be notified first to deal with the appropriate action within its component. No outside influence should interact directly with a Custom Element's Shadow Root as that would increase dependencies and create the problem that Polymer serves to eliminate.

Polymer's data system is also central to its functionality. Properties are defined by the data that belong to the Custom Element. Observers are functions linked to properties that are triggered when the value on the field it is linked to changes. Computed functions are similar to Observers except they are triggered when the input parameters, which are properties, change instead of being triggered to a change to a linked property. This is useful because Computed functions can be repeated with different property value changes.

Data binding is a crucial feature within the data system of Polymer as it allows for parent and child communication. If we were to define a Custom Element with a set of properties we would use data binding to communicate to it's Shadow Root. There are two types of binding, one way and two way. One way binding simply linking a property of a parent to a child. As the parent manipulates that property the child is also updated with the new value. Two way binding has the same functionality as one way binding but additionally will notify the parent when the child manipulates the bound property value.

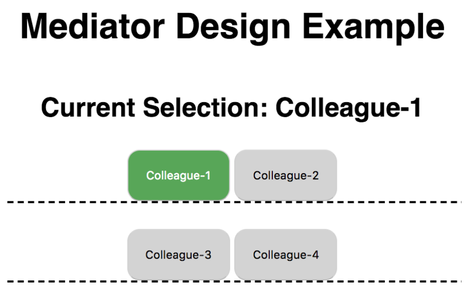
Since DOM interprets HTML in a tree structure there is an opportunity to utilize parent and child relationships. Polymer expands on this idea as it implements the parent child as a Custom Element and Shadow Root relationship. Polymer's data system provides the communication between the two objects. This ensures that the DOM and the browser are being used correctly, efficiently, and effectively when creating these elements. With communication from Custom Elements to its Shadow Root central to Polymer functionality we can build on this and structure communication between separate Custom Elements to represent Mediator Design.

An Example

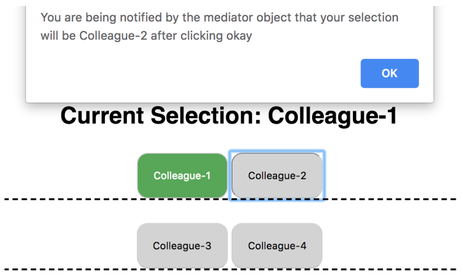
The example used is a simple web application demonstrating the use of Mediator Pattern on a user interface using Polymer[11]. At a glance this basic application allows you to toggle buttons with text and style responsiveness depending on which button is being pressed. Image 1 displays the application start state. When the user clicks any of the four buttons they are presented the transition state in which you are notified your selection is going to be changed, as seen in Image 2. Image 3 presents the unique behavior of special buttons being clicked. The application responds to these special button clicks but this feature abstractly represents unique behavior between a set of objects.

Figure 2, shows the architecture of the application in greater detail. Because our example is using Polymer we organize our Mediator Pattern participants into Custom Elements. Our Mediator, named mediator-participant, contains

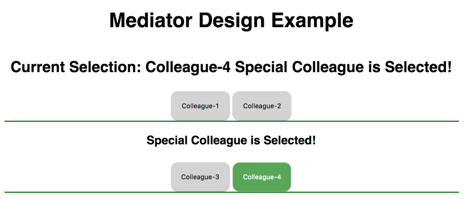
[11]`https://github.com/tatrickporal/Mediator-Polymer-Example`



(a) Image 1 - Start State


You are being notified by the mediator object that your selection will be Colleague-2 after clicking okay

(b) Image 2 - Notification


Mediator Design Example
Current Selection: Colleague-4 Special Colleague is Selected!
Colleague-1
Colleague-2
Special Colleague is Selected!
Colleague-3
Colleague-4

(c) Image 3 - Unique Behavior

Figure 1: User Interface of Example

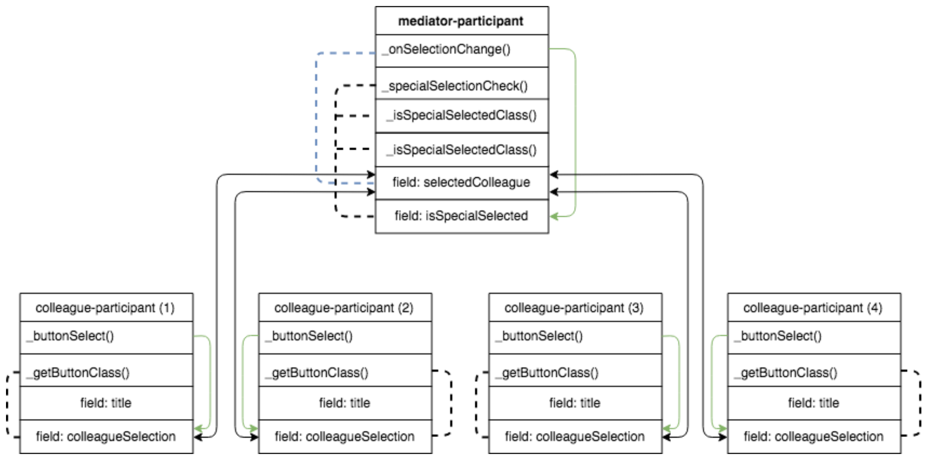


Figure 2: Architecture of Example. Two way arrows are two way data bindings. Green arrows are functions that set property values. Black dashed lines are computed functions from property to method, simularly blue dashed lines are observer functions.

the whole application within its Shadow Root. The Colleagues are implemented with a Custom Element named colleague-participant and is placed within the mediator-participant Shadow Root.

The mediator-participant has a String field to hold the value of the currently selected colleague, named selectedColleague, and a Boolean field, named isSpecialSelected, that is computed when selectedColleague changes. The colleague-participants have String title and String selectedColleague fields. The selectedColleague fields are two way bounded from mediator-participant to each colleague-participant present in its Shadow Root.

Here is an example of the data flow resulting in Image 3:

1. The user triggers the onclick listener of the button within the colleague-participant Shadow Root firing the buttonSelect() method. Which sets the uniquely clicked colleague-participant selectedColleague field to its title.

2. Two way binding of the selectedColleague property triggers an update to the bound property linked to its paired Mediator.

3. The Observer function to the selectedColleague property in mediator-participant, onSelectionChange(), is triggered which alerts the user of the change.

4. The computed function specialSelectionCheck(selectedColleague) is then triggered with the new value returned in step 3. The function sets isSpecialSelected to true if selectedColleague is a special colleague, in our case colleague-participant (3) and (4).

5. The computed functions, isSpecialSelected() and isSpecialSelectedClass() are called with isSpecialSelected as the parameter which set the specialized text and styling.

6. The new value for selectedColleague is then set and is sent from Mediator to Colleagues where the Colleagues check to see if they are selected or not and act accordingly

Implementing this application without the use of Polymer and Mediator Design would be undesirable for two main reasons. For one, there would be no organization of communication between the set of objects that require it. Buttons would have to directly contact all other buttons to notify its change of state. Values would have to be explicitly set by another object which in almost all cases is not good coding practice. Additionally, we would have to explicitly define each button. If we wanted to implement a behavior on the buttons we would have to add it to each implementation of the button. These issues would continue to grow as more features are added and changes to the user interface are done. These are the exact problems Polymer aims to get rid of. By implementing this application using Polymer and Mediator Pattern it has become easier to maintain, read, reuse, and configure.

4 An Implementation

High Frequency Trading

In this paper we describe a software architecture that extends a relatively new experimental platform, called oTree, to allow for sophisticated financial environments and market formats to be studied experimentally. The markets emulated in the lab environment consist of several components of a system where oTree is the master, unifying element (and referred to as experiment server). Our architecture allows the experimenter to work with arbitrary market engines/mechanisms (including external exchanges, if desired) and to implement experiments in continuous-time environments. Importantly, our architecture is appropriate for environments where communication latencies are an important element of the environment (see application below). With this architecture, a wide variety of experiments can be implemented under the oTree framework. Our architecture facilitates experimental research in modern financial market environments. A similar approach can be taken for experiments in other areas of economics and other disciplines where interactive user interfaces and continuous-time settings are required along with complex decision spaces. Most components can be replaced with or connected to other pre-existing programs which reduces duplication of efforts in the development of experimental environments.[12]

User Interface Architecture

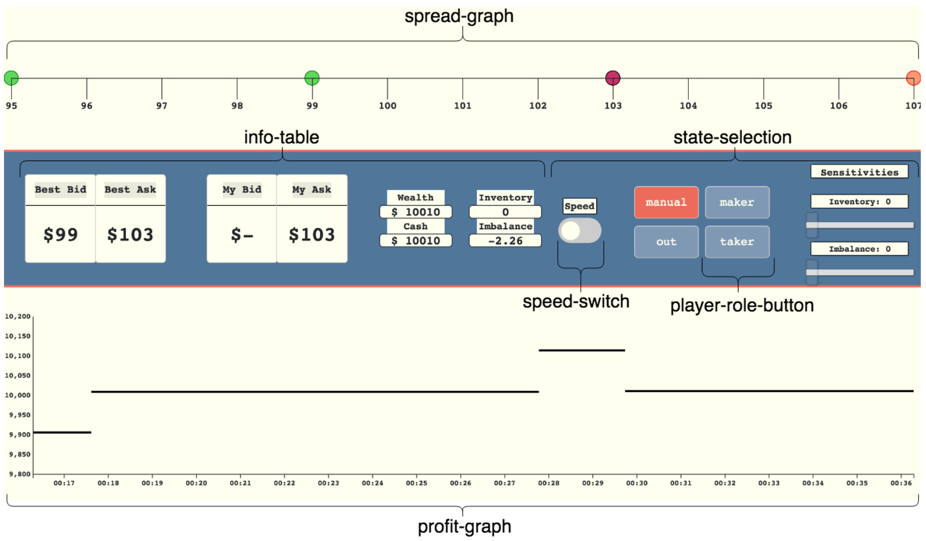
The main objectives for our user interfaces were to accurately display real time financial market data, respond to user input, and to be configurable by the researcher to display different market environments. The technologies that would best accomplish these goals was to implement the user interface using web socket protocol for communication with back end services[13], D3 a JavaScript graphing framework to display market data, and Polymer with Mediator Design for organization of components and configuration. This paper will mainly focus on the use of Polymer with Mediator Design and how it accomplishes the goals we set out to accomplish. The UI of the HFT experiment is displayed in figure 3. Spread-graph and state-selection are the user interactive components that are responsible for listening to user input. Additionally all components shown respond to data passed from the oTree application under the facilitation of the market-session Mediator. The architecture of the experiment software is designed for configuration of experiment details from the researcher via a configuration YAML[14] file. The configuration file defines session specific details read by the oTree application. The client receives the correct configurations at session start by the oTree application and runs the corresponding environment and exchange interface from there. While this paper will be focusing on the ELO[15]

[12]`https://github.com/Leeps-Lab/high_frequency_trading`

[13]Data manipulation and server interaction not presented by the front end

[14]YAML is a data serialization standard file

[15]Exogenioius Limit Order



Best Bid	Best Ask	My Bid	My Ask
$99	$103	$-	$103

Wealth: $ 10010
Cash: $ 10010
Inventory: 0
Imbalance: -2.26
Inventory: 0
Imbalance: 0

Figure 3: CDA financial exchange in the ELO market environment. What you see is contained within the Shadow Root of the market-session Custom Element. All the data passed to and from the oTree application is displayed on the user interface.

market environment in a CDA[16] exchange, the software design allows for the plug in of multiple market environments to represent corresponding financial exchanges.

The front end architecture is designed as a plug in for the oTree application. This design is displayed in figure 4. Meaning if oTree selects a market environment and broadcasts the correct messages the front end should be able to receive those messages, interpret them, and display them accurately on the user interface. This eliminates the dependency of having different sets of communication for each market environment. While there does exist specialized environment specific messages the majority of the messages are agnostic to the market environment. This approach puts responsibility on the front end to define the inbound/outbound messages and events it expects. These environment fields are defined within the market-environments JavaScript file. For ELO, within market-environments.js there exists an object with these fields defined that is used to initialize the market-session in the experiment JavaScript file. This initialization ensures that the front end will expect certain messages from the back end.

[16]Continous Double Auction

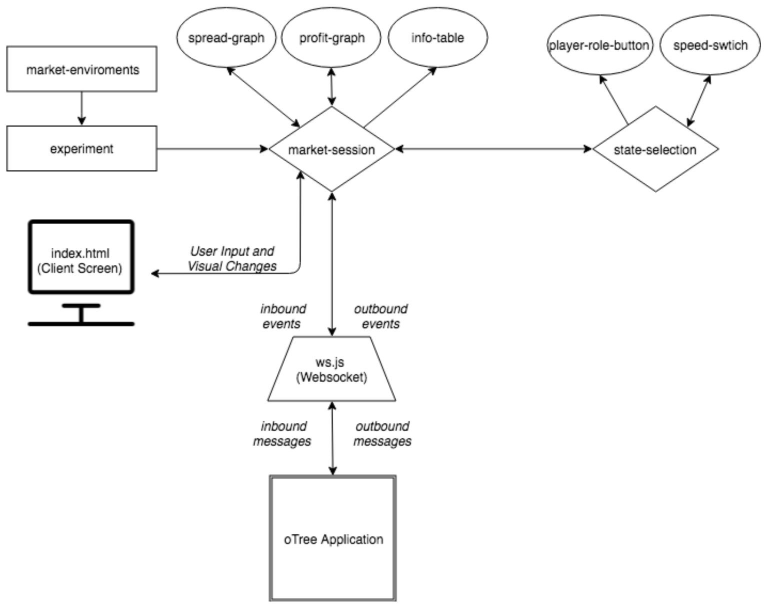


Figure 4: Arrows denote directions of communication. These are the main custom elements within the ELO user interface. We see that market-session sits at the center of communication for our user interface

Use of Polymer and Mediator Pattern

The figure 5 presents the structure of communication through responsibility driven web components. Utilizing Mediator Patter and Polymer when designing the user interface seemed intuitive for two main reasons. One, our experiment contains complex communication so it is necessary to utilize Mediator Pattern. This communication ranges from dispatched events to and from the oTree application to value passing triggered by user inputs. Having designated points of communication simplifies the process. Since our experiment runs as a web application we implement the pattern through Polymer. Two, as the project grows and the complexity of the code base increases using Polymer allows for easier development during phases of feature additions and refactoring.

As mentioned above in the Polymer section, Polymer is a web component framework in which Custom Elements can be implemented. In our ELO implementation we use a total of nine main Custom Elements. These elements are highlighted in figure 5. Here is a list of each component with the purpose it serves:

1. **market-session (Mediator)** - This element is considered our Mediator for our user interface. This component is responsible for passing values to and from each of its Colleagues, which is then displayed to the user, as well as two way communication with the oTree application. If one

of its Colleagues i.e spread-graph or state-selection requests a change, the market-session is notified, the appropriate event is handled and the respective message is sent to web socket to dispatch to the oTree application.

2. **state-selection (Colleague/Mediator)** - The concept of a user state or player role is used in this experiment. A user can choose to be one of a set of states and the state-selection element is where this communication is handled. Essentially this element is considered a Colleague of market-session but a Mediator to player-role-button and speed-switch. Since the representation of the user's current state required some complex logic to code we felt it necessary to organize it within its own Custom Element. This element is a direct Colleague of the market-session Mediator and can also be thought of a Mediator to player-role-button and speed-switch Colleagues.

3. **ws (websocket)** - The websocket component takes the dispatched messages that market-session receives, sanitizes, then sends them over socket to the oTree application.

4. **experiment** - This component creates the market-session experiment Shadow Root. For ELO the experiment component initializes the elo-experiment element which is displayed on the users screen. The initialization takes the respective environment inbound/outbound messages and events it expects from market-environment and creates the DOM to be displayed on the web page. Within the elo-experiment Shadow Root exists the market-session instance with the environment specific configurations.

5. **spread-graph (Colleague)** - The spread-graph is a user interactive representation of the exchange order book[17]. This component has the responsibility of reacting to user input and to accurately display market orders along with other market details the user is in.

6. **profit-graph (Colleague)** - The profit-graph is a real time graph written using D3 that displays the users profit with respect to time.

7. **info-table (Colleague)** - The info-table displays real time exhange information i.e users endowment, best market bid and offer.

8. **player-role-button (Colleague)** - This element is a Colleague to the state-selection Mediator. It is essentially a button representing a state or role in the experiment that dispatches a request to change state event with a user click and changes its value based on the confirmation value passed by its parent element, in this case the state-selection Mediator.

9. **speed-switch (Colleague)** - The speed-switch plays a similar role to the player-role-button except it is a checkbox representing speed investment. The user click dispatches a request to invest in speed and the visual

[17]The active orders within an exchange at any given time

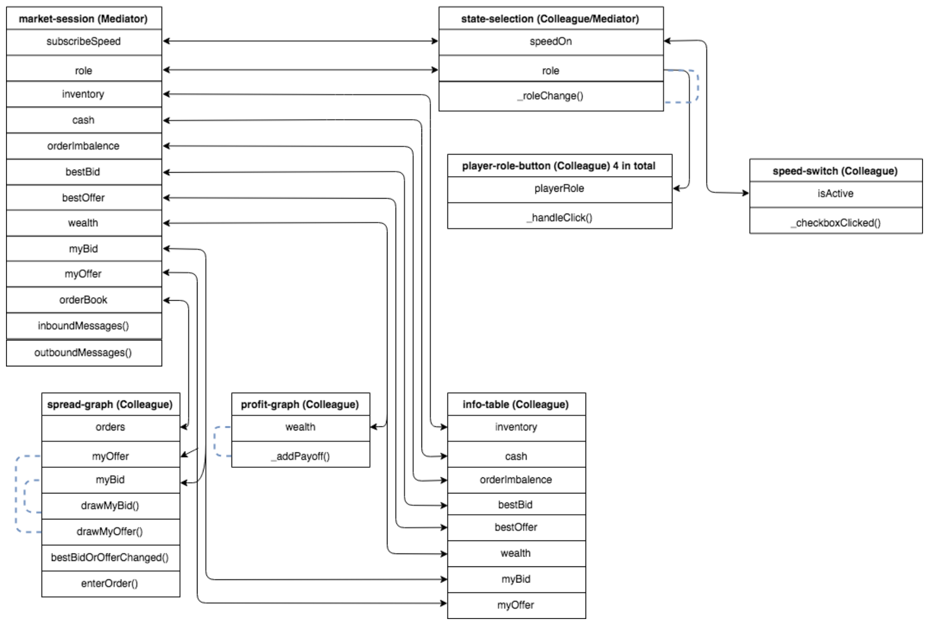


Figure 5: This is the structure of Polymer and Mediator Pattern in the ELO environment. Directional arrows denote two way and one way property binding. Blue dotted lines denote observer functions on properties. See the output of this structure presented in a browser in figure 3.

representation of confirmation is shown once the state-selection Mediator confirms the request.

The flow of data for application consists of two techniques, dispatching events and passing data bound properties. User interective Custom Elements like speed-switch and player-role-button dispatch an event entitled 'user-input' with a payload that is meant to be sent over web socket. This event is considered a request to send over the web socket since sending it directly from inside the component would be flawed and would go against our one entry point of communication approach. Our Mediator market-session is initialized with an event listener on the 'user-input' event. Once triggered it sanitizes the payload making sure it is an expected event and that it is in correct format. Once these conditions are met market-session dispatches an event on the ws Custom Element that sends the message over socket if the socket is connected and ready.

For an inbound message the ws element listens on an incoming message over socket. Once this is triggered ws dispatches a 'inbound-ws-message' event that market-session is listening for. This triggers market-session to parse the message and call the corresponding function defined in experiment.js based on the message type. This function will in most cases change a property of market-session and whatever Colleague is bound to that property is updated. For instance this is the flow of communication to change user role in the experiment:

1. A user click triggers the onclick listener on player-role-button within state-selection that dispatches a 'user-input' event. With the payload being a JavaScript object that defines the message type and the value. In our case the type is 'role-change' and the value is the given name of the player-role-button.

2. market-session is listening to this event and triggers its corresponding function that sanitizes the payload.

3. Once sanitized market-session dispatches an event to ws that translates the JavaScript object to JSON format and sends the message over socket.

4. A confirmation of this request comes in the form of JSON message sent from the oTree application server. Once the ws is notified it dispatches an 'inbound-ws-event' event.

5. market-session is listening to this event and translates the payload to a function that updates its role property.

6. The data bound property found in player-role-button and the respective custom actions are then set.

Analysis

Utilizing Mediator Pattern and Polymer has proven to be especially beneficial when extending the application to different market environments. Because the environments are mostly formations of the Custom Elements, with possibly some additions or subtractions, there is opportunity to duplicate these web components. If made to be general enough a component can be reused within the Shadow Root of market-session. It would be essentially plugging in already implemented elements and giving it the correct inputs via its encapsulating Mediator. For example, player-role-button can be duplicated as such:

```
<player-role-button
      role-name='$MEDIATOR_GIVEN_NAME'
      player-role='$MEDIATOR_BOUND_VALUE'
>
</player-role-button>
```

All the logic of sending a request is handled within the component of player-role-button so there would be no additional logic needed to be placed besides CSS[18] styling to match the aesthetic of the market-environment. This clearly would save time when developing a new market environment as opposed to explicitly coding each environment interface from scratch.

[18]Cascading Style Sheets meant to define the style and layout of markup languages, in this case HTML

Another benefit using these technology brings is that they make the code base more readable and easier to maintain. Polymer's aim to eliminate dependencies and create uniform code allows multiple people work on the same components without having just one developer able to work on the specific element they created. Polymer is optimized for browser and HTML DOM interaction, if it is used correctly then we are ensured that the browser is efficiently used.

Conversely, this gave rise to new challenges. Because Polymer is a framework to interact with the native browser and HTML DOM we saw some limitations with this interaction. For example, the data flow between Boolean DOM native attributes and Boolean Custom Element properties proved to complicated when bounded to one another. This is because the DOM treats Boolean attributes to be true by the attribute's presence, such as [19]

```
<button
    class='$CLASS_NAME'
    disabled
>
</button>
```

rather than,

```
<button
    class='$CLASS_NAME'
    disabled=true
>
</button>
```

While there are intelligent Polymer hacks to solve this issue what this does highlight is a bigger problem and that is Polymer is at the mercy of the HTML DOM. Since this is a framework to interact and customize this DOM it is clear that there will be some complexity when fitting Custom Elements and native DOM elements. The good thing is that Polymer does an excellent job of fixing these incompatibilities and in most cases does not hinder functionality if the proper Polymer fix is used. For the above example, Polymer allows for the mix Custom Element properties with native attributes by adding a special reflectToAttribute field to the property and tagging the attribute with a question mark. While this is a solution to the problem it does show the flaw. However, even with this draw back we found that using Polymer and Mediator Design resulted in tangible benefits. We were able to design and implement new market environments in a quicker time by reusing Custom Elements. The code base is more readable and easier to maintain due to the intuitive standard of using Mediator Pattern through Polymer. Moreover, we are ensured that scheduling of updates to properties and handling of events to and from the HTML DOM and the browser is optimally handled by Polymer.

[19]button is a native DOM element and disabled is a native button attribute

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, Inc., Reading, Massachusetts, 1995.

[2] M. Pizka. Straightening spaghetti-code with refactoring. 204. doi: http://itestra.com/wp-content/uploads/2017/08/04_itestra_straightening_spaghetti_code_with_refactoring.pdf.

[3] W. Schools. The html dom (document object model), 2019. URL `https://www.w3schools.com/js/js_htmldom.asp`.

[4] A. Shvets. Mediator design pattern, 2019. URL `https://sourcemaking.com/design_patterns/mediator`.