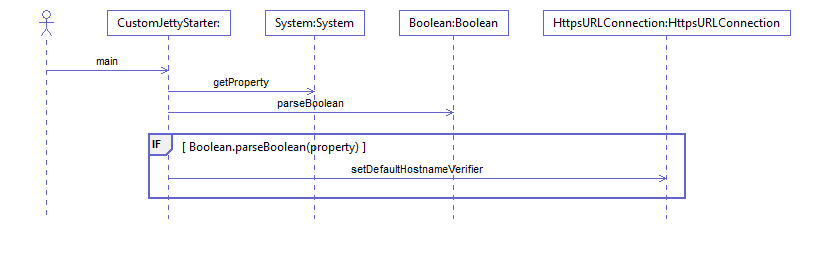
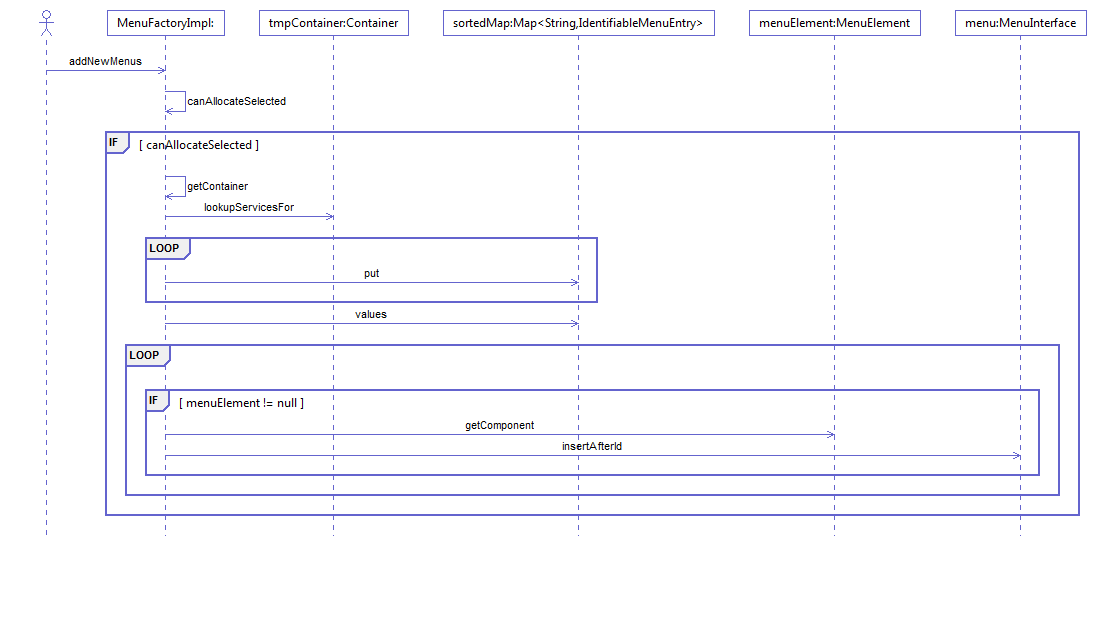
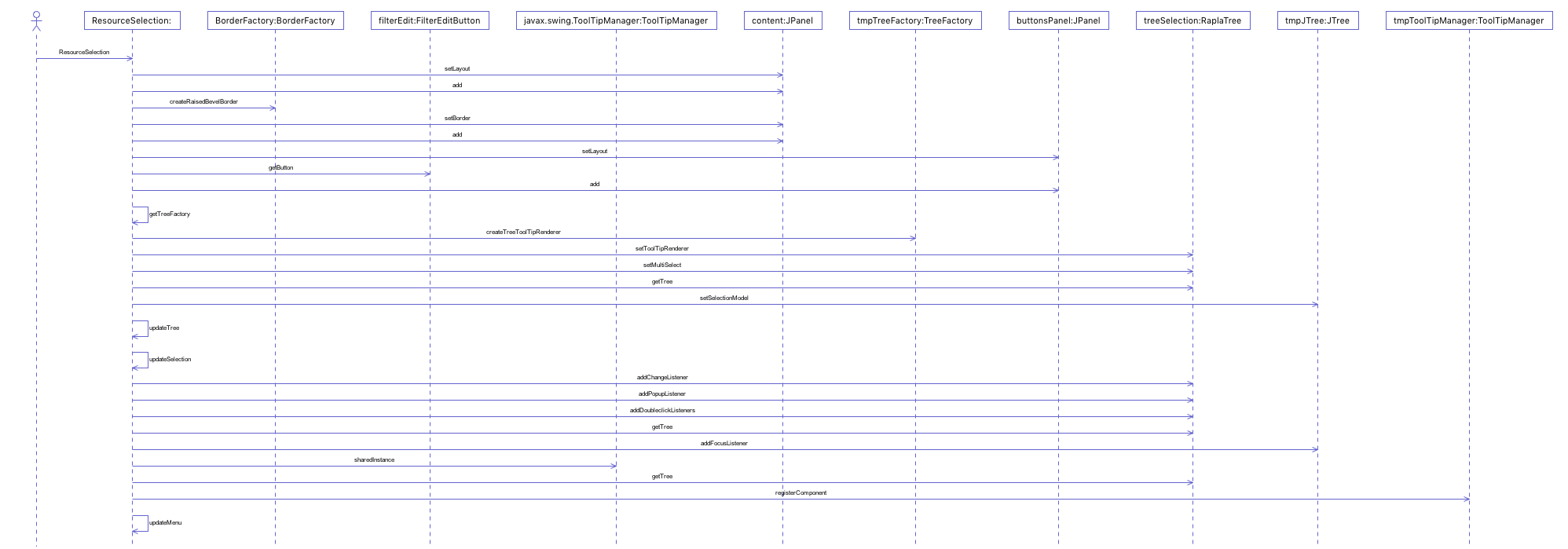
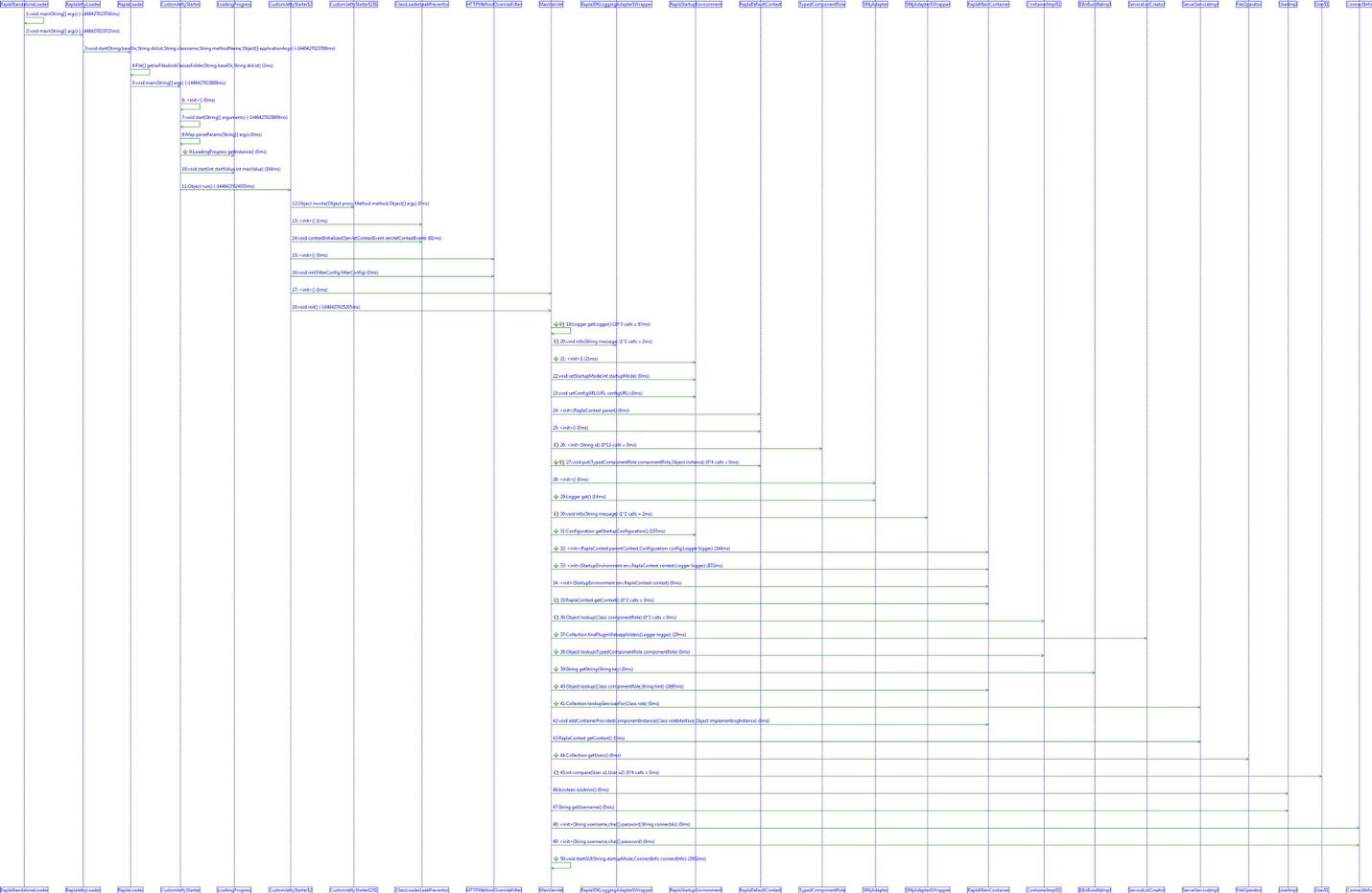
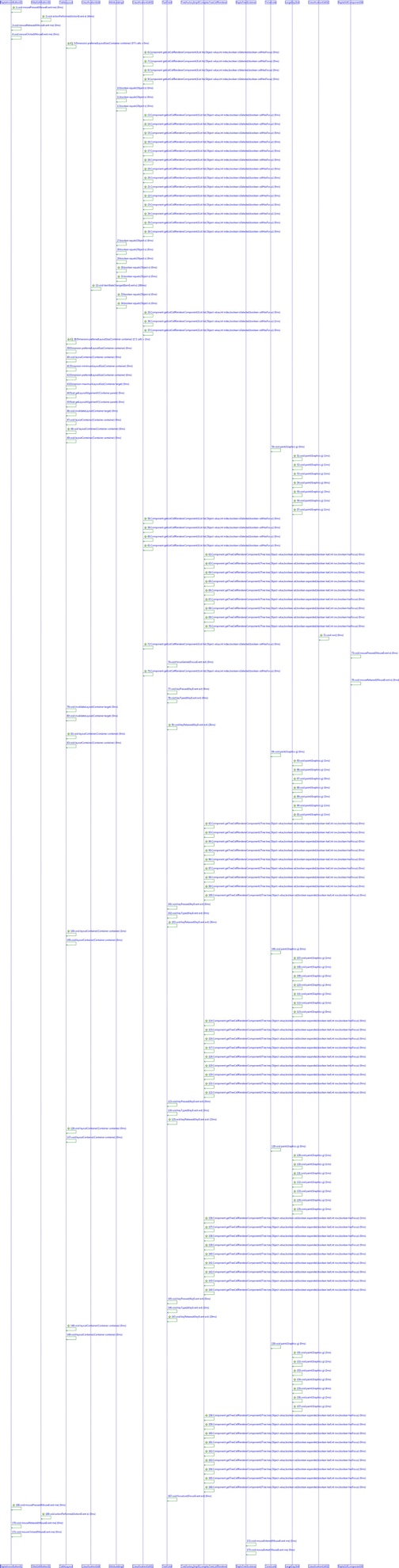
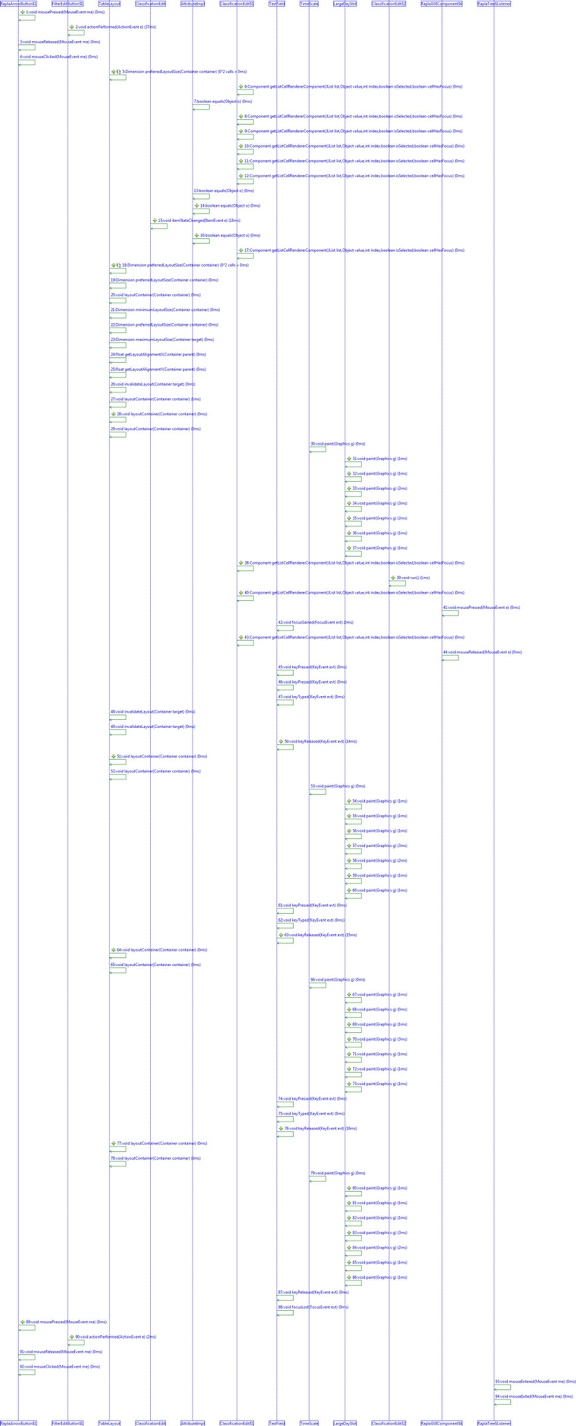
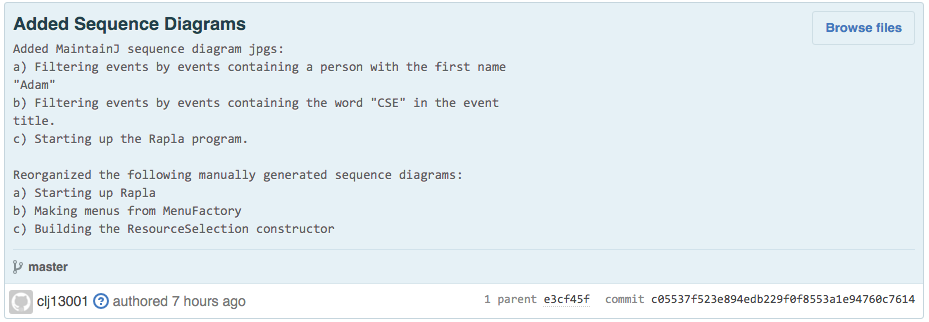
1. The first step we took in this assignment was generating static sequence diagrams using ModelGoon. First, we will describe a simple sequence diagram that was generated in the CustomJettyStarter class when we call on its main method. This class is located in org.rapla.bootstrap.  
     
     
   Notice it starts with the user calling on the main method in CustomJettyStarter. It then calls on other methods throughout the main process, such as System.getProperty(), Boolean.parseBoolean(), etc. CustomJettyStarter is one of the first classes involved in starting up this program, so we would think that more calls would be made to start up the program. Static diagrams restrict the method calls to just ones until the main class ends – we will observe later through MaintainJ that more calls are actually performed to completely start up the Rapla application.  
     
   Next we can analyze another sequence diagram for MenuFactoryImpl’s method addNewMenus, which is found in org.rapla.gui.internal.  
     
     
   We liked using this example because it’s a bit more complex and it shows examples of if statements and loops. It begins by calling on a canAllocatedSelected method inside the MenuFactoryImpl class, and performs an if statement on that Boolean instance. If true, it mostly loops through a bunch of entries and inputs them into a sorted map. We thought this would be a useful sequence diagram, because we are working with the GUI a lot in our search addition – this will add to our overall GUI understanding of Rapla.  
     
   Finally, we analyzed the ResourceSelection constructor, located in org.rapla.gui.internal. It’s a pretty large one.  
     
     
     
   This is one of the more important diagrams to our add-on because this is specifically the location where we will be adding our fast search text field. This diagram shows that the constructor performs multiple methods such as setting layouts (setLayout to JPanel), making buttons (such as getButton to FilterEditButton), and allowing these features to have click/pop-up listeners (all to the RaplaTree class). In our search addition, calls will be added to this diagram that will allow us to place a text field in the resource selection menu and effectively search with it.  
     
   All ModelGoon diagrams will be uploaded to Husky CT along with this assignment since some of them are unclear in this document.
2. Next, we used MaintainJ to generate Sequence Diagrams during runtime. The first diagram we chose to analyze is the one generated while the program is built and started.  
     
      
     
   Notice how much bigger this one is than the static diagram we generated before – that’s because this one actually contains EVERY single call while the program is starting, and not just the calls restricted by CustomJettyStarter’s main method. This shows MaintainJ’s full use in action, as we can observe what *actually* occurred at runtime, rather than ModelGoon’s *prediction.*To explain some methods: call 4 is in the beginning where RaplaLoader calls on getJarFilesAndClassesFolder; as the name highly suggests, all the needed jar and class files needed to build this program are collected in this method call. Call 9 is when CustomjettyStarter calls on getInstance on LoadingProgress; this is most likely a call that updates the loading progress bar that shows while the program is booting up. Call 20 is where MainServlet called on the info method to pass string parameters to RaplaJDKLoggingAdapter$Wrapper, where this string will probably be used later on in building the program.   
     
   Then we chose to trace what happens when we perform a resource filter. We chose to filter the person resource, by the first name attribute when it contains the name “Adam.”  
     
     
   We chose to use this action as a part of our MaintainJ traces because in order for us to apply our program addition, we need to understand how the filter feature works so that our implementation can reuse the existing algorithms. While we were able to search through the code to find a lot of the filter features during one of the previous assignments, this sequence diagram shows us some extra actions that we overlooked, but are actually just as important to understand.  
     
   To explain some methods: call 1 shows that mousePressed was called on RaplaArrowButton; simply put, this explains that the filter button was pressed and a popup was thus shown. Call 3 shows quickly after to describe the call mouseReleased on RaplaArrowButton to show that we released the mouse, and now a popup should show after we release our click (if you hold a click down for a long time, an action usually holds and isn’t performed until you release.) Then we see that calls 10, 11 and 12 call an equals method on AttributeImpl; we are guessing that this call has something do with comparing what was typed in as a filter attribute with all existing resource entities.  
     
   Lastly, we chose to trace what happens when we perform an event filter, rather than a resource filter. The difference is that we use a filter based on the title of an event, such as (in our example) only showing events containing the keyword “CSE.”  
     
     
     
   We chose to use this trace as part of a comparison with the resource filter. If the two are similar enough, then we know that trying to implement our third filter feature shouldn’t be too difficult to implement using the existing code. If there are obvious differences that exist, then it will point out where they are and we should be able to interpret why those differences are needed – this will help us to apply whatever differences are necessary for our addition.  
     
   To explain some methods: we notice that most of the calls are similar to the resource filter. Call 6 shows getListCellRendererComponent called on ClassificationEdit; we’ve deduced early on in this project that ClassificationEdit is strongly associated with filtering results, so it’s likely that this method call has something to do with grabbing either the input, or the values that the input should be compared to. Call 31 shows the method paint called on LargeDaySlot; this suggests that a comparison was made based on our input, and one of the sections where events can exist was “painted” over, or more so updated to either filter out or filter in that specific event instance. On call 45, keyPressed was called on TextField; this exhibits that someone pressed a key in the input text field, which in this case was when we typed in the keyword “CSE” into the filter textfield.
3. Our third static sequence diagram from ModelGoon shows how ResourceSelection builds up the ResourceSelection menu, and performs some constructor calls such as making a new BorderLayout and a new FilterEditButton. Interestingly enough, this “creation call” doesn’t show up as a part of the sequence diagram, but subsequent calls made after it’s constructed (such as when the filter edit button performs getButton) show up normally in the sequence diagram.  
     
   This makes sense because in a sequence diagram, when an instance of a class is created, it is denoted by a vertical dotted line which we call the object’s lifeline. As the name suggests, it shows at what point that object was created. If we included every single constructor method call, we feel like that would be highly redundant considering the lifeline already encapsulates that idea.
4. We already committed the sequence diagrams before taking a screenshot, so here is a screenshot of the update on our Github repository:  
     
     
     
   We will upload all of these files to the Husky CT submission.