Deliverable 1

Team 29: GoodbyeWorld!

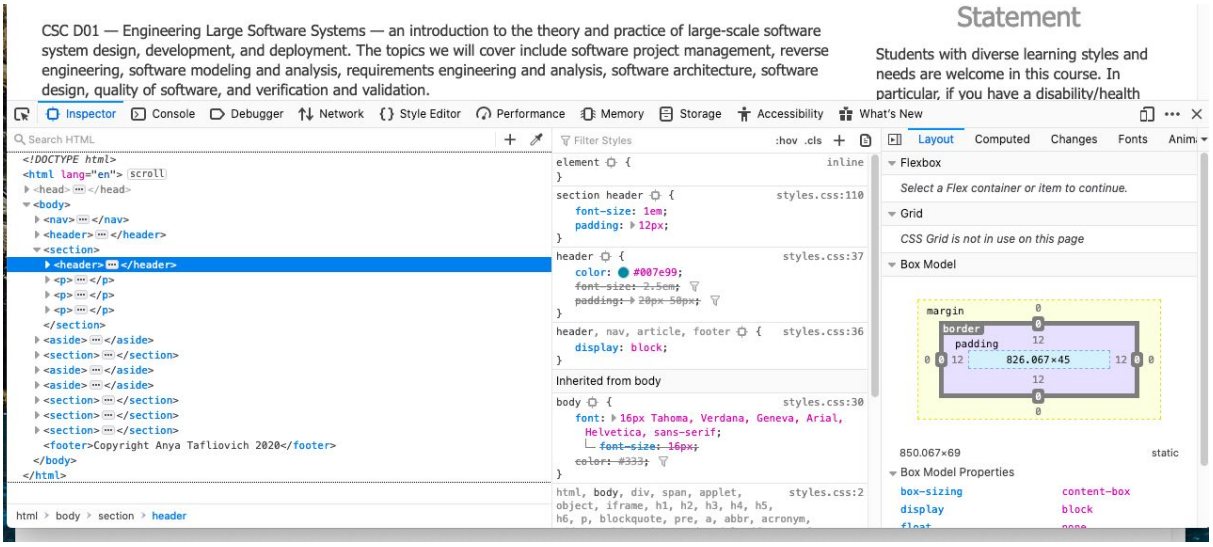
Thasitathan Sivakumaran | Ajay Rajendran | Jason Ku | Yathan Vidyananthan

For our open source project we have chosen to work on Mozilla Firefox's Developer Tools https://github.com/mozilla/gecko-dev.

Mozilla DevTools

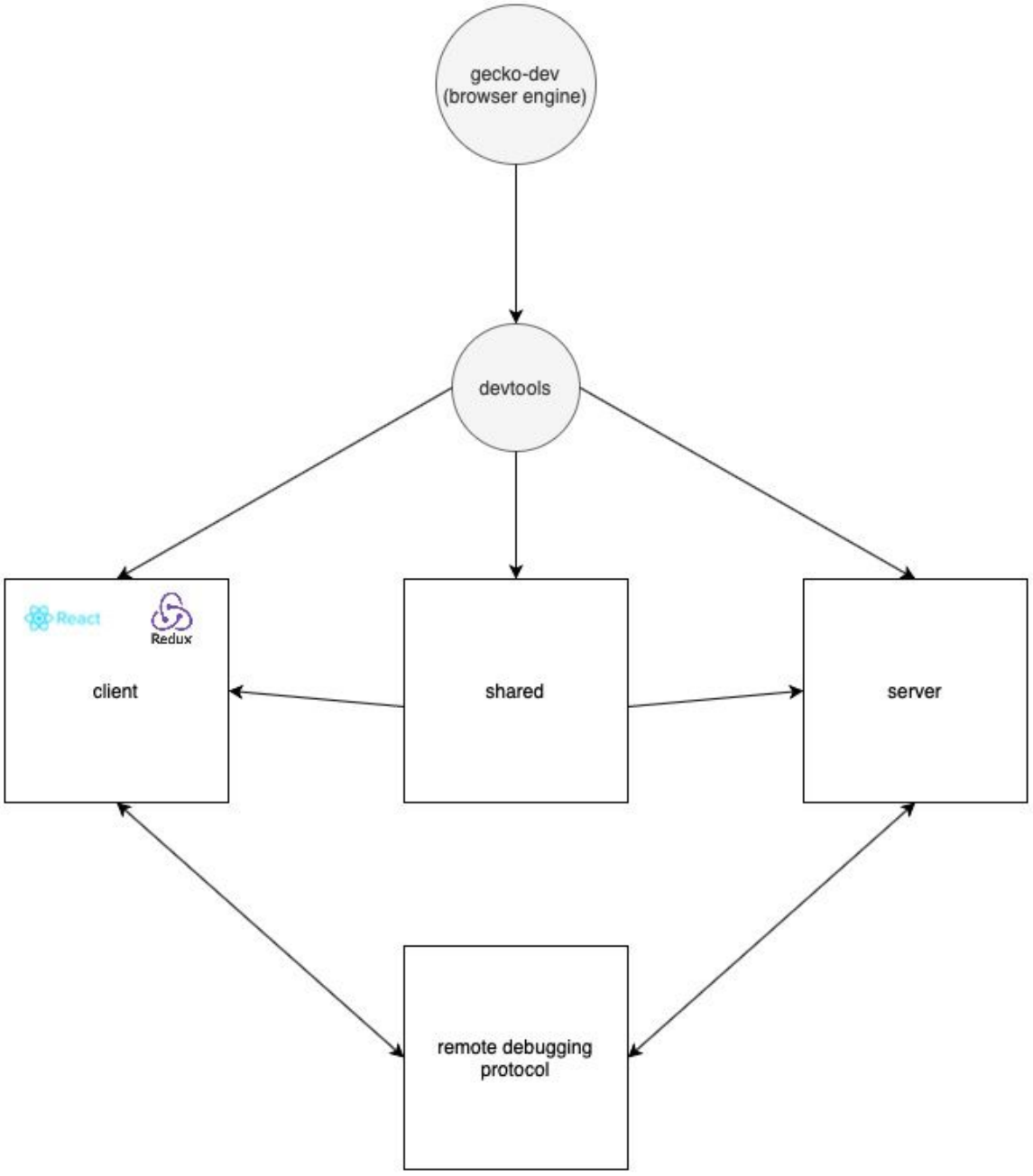
Mozilla's developer tools is a set of tools to help developers edit and debug html, css, and javascript code. You can access the developer tools by right clicking and clicking "inspect element". The following is a screenshot of how the developer tools look on our CSCD01 webpage.

The main section of the tool is the page inspector where we can view and edit page content and layout, allowing you to see box models, animations, grid layouts, etc. The web console we can display messages logged by javascript code usually done as console.log("helloworld"). There is also the javascript debugger, which allows you to step through and modify javascript code running on the page. The network monitor allows you to see the network requests made when the page is loaded. The performance tool lets you analyze your site's responsiveness. The responsive design model allows you to see how your website and app will work on different devices. The accessibility inspector allows you to see the page's accessibility tree to see what is missing.



Introduction

Gecko-dev stores Mozilla's open source code. Our project will focus on devtools which can be subdivided into three categories: client, server, and shared. Client contains the code for the front-end. Each directory under client corresponds to a panel that the user sees when they open up the inspect element tool. The server directory contains the code for the server side, which is responsible for analysing and processing the information from the front end and sending the results to the front for the user to see. Lastly, the shared directory which includes code and third-party APIs and libraries that both client and server side use.



Client Overview

Firefox's DevTools uses several different tools for its frontend. In the documents for gecko-dev, the frontend for DevTools is referred to as "toolbox". These tools include the use of SVG files to display images, React to display user interfaces, Redux and also Telemetry for metrics gathering and display. SVG images are necessary for DevTools frontend because it allows for the creation of icons for the user interface. React is used for several reasons outlined by Mozilla which include the use of components and the React lifecycle.

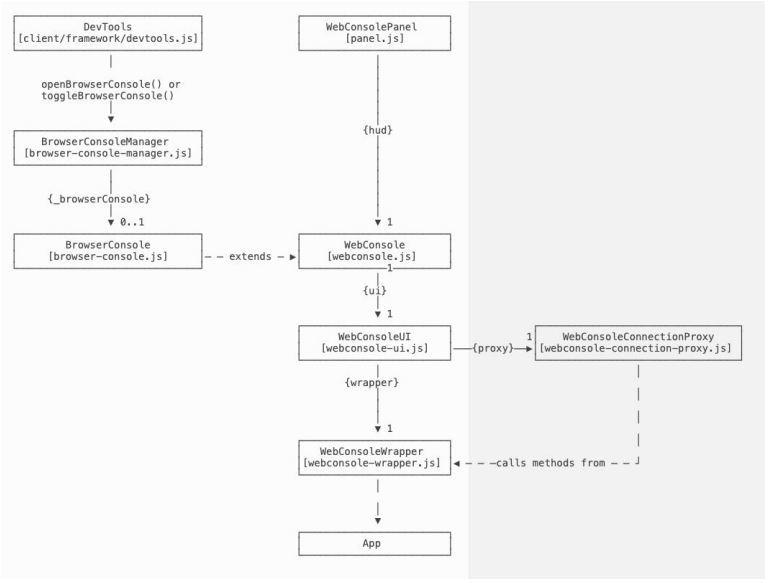
Since components can update and render, Mozilla's reasoning for choosing React is because it can update instead of rendering the whole component again. Another reason Mozilla chose React is because it uses DOM diffing. Since the component returns a "virtual DOM", a lightweight JSON structure, this allows React to compare DOMs with previous version and generate any changes necessary instead of changing everything. React also is synchronous when rendering, meaning that when a component is given some data, it will render it all in one go. This eliminates any problems that arise from race conditions, especially when promises or event emitters are used.

Although React supports local states, Mozilla opts to use Redux to save the states instead of using the state saving that React has. The reasoning they give is simply because Redux will help manage these states automatically when it binds to React instead of manually updating the states. Yes, React does support states but does not have any state management other than the developer managing them. Thus, they have elected to use Redux for this purpose. Other than simply using Redux because it manages states, Mozilla has chosen it because of the immutability to states that Redux provides, as it always returns a new state when it is updated. This removes a lot of potential errors and time saved when the states do not add up to what is being displayed to the user.

The last tool that the frontend uses is Telemetry. This isn't necessarily for the user directly, as Telemetry assists Mozilla developers by sending data of the end users' DevTools usage. With this information, it can allow the developers to prioritize different features based on popularity, in turn helping out the end users eventually.

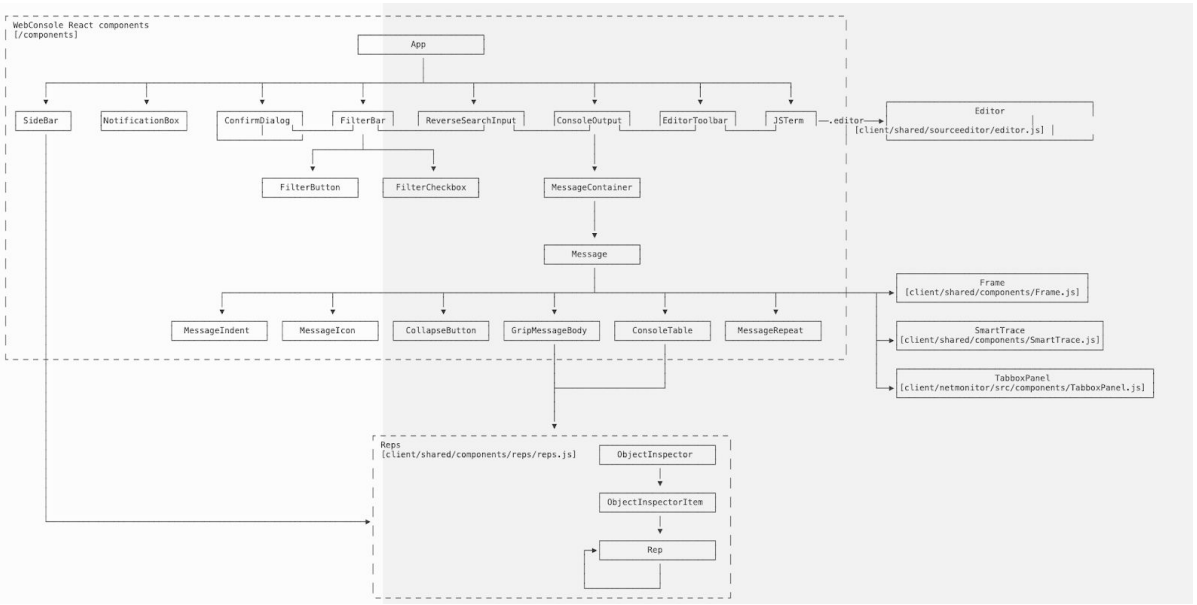
Client - Console Tool

Another UML model we will go more in depth into is the Console Tool from the firefox developer tools, this is what is responsible for rendering the logs from the page that is loading. The DevTools class represents the developer tools, which keeps track of what toolboxes are open in the browser and the set of tools available. The web console shows you all the api calls made by scripts and it can also display network logs.



References: https://firefox-source-docs.mozilla.org/devtools/tools/console-panel.html

The diagram below shows the components which the console panel UI is built on top of which uses react. The app uses components from react such as SideBar, NotifcationBox, FilterBar, etc. It also uses external components from the WebConsole such as a Frame which is used to display location of messages, as well as SmartTrace, a tool used to display the stack trace of messages and errors.



References: https://firefox-source-docs.mozilla.org/devtools/tools/console-panel.html

This panel implements actions which can be divided into three groups Filters that are content filtering , Messages message rendering, and UI state. The states are managed using redux; filters.js has a state for panel filters, messages.js has a state for messages rendered.

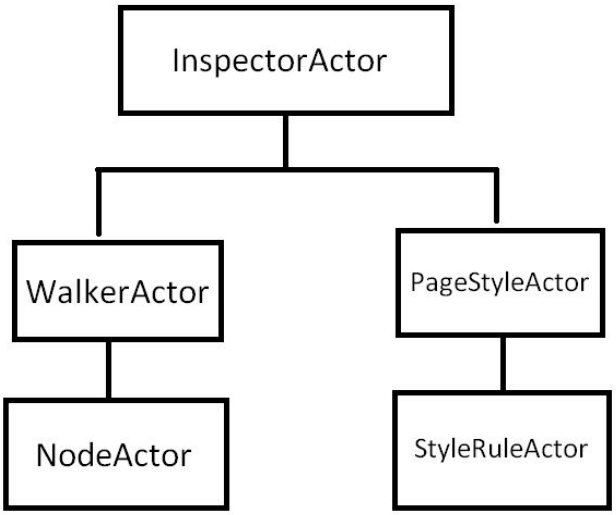
Server Overview

The backend of devtools is responsible for processing the information present in the front end web application and returning the debugging results in order to display it to the client. The client side is known as the debugger and the back end is the browser which processes the information. There are many servers, and each server is responsible for a single component of devtools, for example, inspector, console, and memory have their own server. The backend servers use a protocol to connect to the front end, actors to relay information, and APIs such as Client and Debugger to monitor and manipulate information.

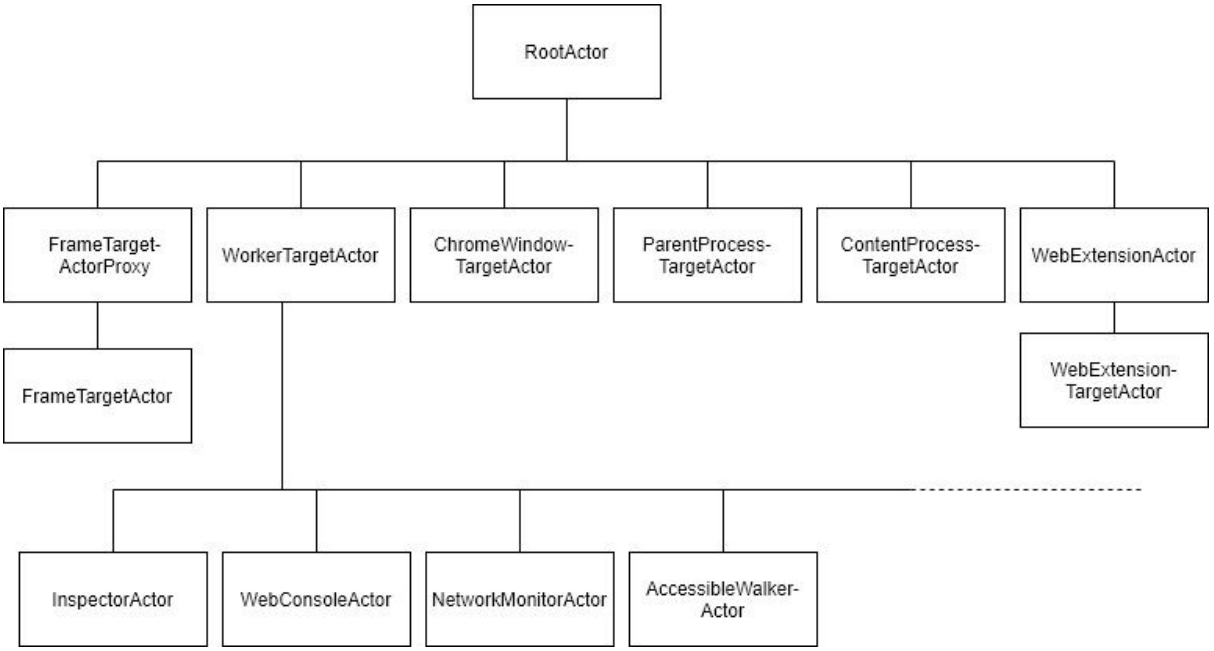
Mozilla's devtools uses a protocol known as Remote Debugging Protocol (RDP) in order to connect a debugger (client side) to the browser (server side) to discover debuggable objects. The protocol is responsible for providing a unified view of technologies that are present in the client-side web such as DOM nodes, CSS rules, Javascript, etc. As stated, the client side and the server side are decoupled, which means that a single client can connect to multiple servers at a time. In order for the client to communicate with the server, they send requests in the form of JSON objects. One of the main components that RDP uses are Actors. Actors reside in the servers and are responsible for exchanging JSON objects with the client. This is how the client and multiple servers can dynamically communicate. The protocol is in place in order to ease the process of implementing new features and allows for creating mock implementations for testing.

Actors are the backbone behind the communication between the client and the servers. Actors receive and respond with JSON packets. Client sends JSON packets to the actors, which is then used by the javascript code to process, and the results are then sent back using JSON as well. There are multiple actors for each individual server so the JSON packets include information as to which actor they need to communicate with, and the responses also include the client they should be sent to. The interesting architectural design of the actors allow for ease of dynamic communication. The actors are designed in a hierarchical tree order, with a root actor called "root" which is the root of the tree. The root actor is mainly used to instantiate target actors that are required for different components.

Each component of devtools (ex: inspector) has their own actors, and each part that the component is responsible for has their own actors. They are placed as parent and child actors, and if the client uses a particular part of that section, they would be communicating with the corresponding child actor. When the client first loads up devtools by clicking on inspect, they initially connect to the root actor. Once they are connected, the connection is then attached onto the corresponding actor of the component that they are using. If the user wishes to use the Network section, the connection will go to the actor representing Network. When the user uses one of the parts of Network, it would then send the request packets to the corresponding child actor of the Network actor. This architectural design also allows servers to dynamically close a connection with the client if the client is not currently using its services. Any branch of the tree that is not currently connected with the client will be closed. This eases the overhead allows faster response and connection.



As an example of a branch of the Tree Hierarchy for actors, here is an example of a branch of the tree for the Inspector element of devtools. When the user clicks on the inspector tab, the connection goes to the inspectorActor actor, whose job is to create and return the child actors, WalkerActor and PageStyleActor. The WalkerActor is responsible for walking the Entire DOM of the current page, and nodeActor is responsible for a single DOM node. Depending on which part the user is currently debugging, they would connect to the corresponding child node. The PageStyleActor and StyleRuleActor are responsible for responding with the result of the computed view.



The way hierarchy of the actors is set up is that we have a root actor at the top and it has multiple target actors connected below it. The target actors are general components in the browser including tabs, workers, and addons. Below the target actors there are target-scoped actors which have particular data associated with them which is what is reflected of the target actor above them. Examples of the target-scoped actors are actors such as InspectorActor and the WebConsoleActor. The target actors contain a list of ids which correspond to the target-scoped actors below it. This is set up this way so that target-scoped actors are only instantiated by the root actor only when they are needed and the target actors act as a mapping to the target-scoped actors associated with them.

The client api is provided by DevTools so that we can create applications that can debug and inspect web pages. We can create applications that can be attached to tabs so that they can monitor data related to the tab. Once these applications are attached to a tab on the browser, they can then attach themselves to the thread that the tabs are running on. By being able to attach to the thread of the tab, the application can they utilize the debugger on that thread in order to observe the running javascript code. Furthermore, it is able to utilize the DevTools client in order to receive events that are triggered in the tab that it is attached to.

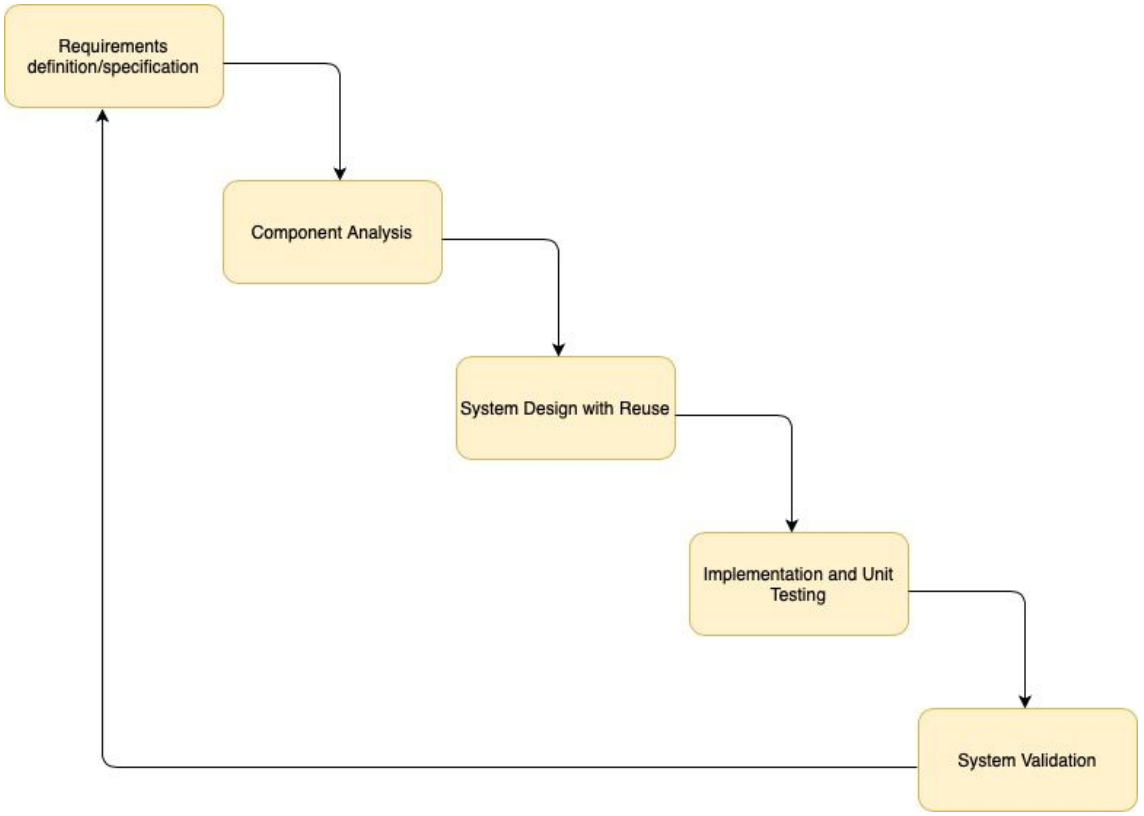
Mozilla uses the javascript engine SpiderMoney, which comes with an interface named debugger. The debugger allows applications to observe and manipulate other javascript code. The debugger is also a source level language written in javascript which gives it the advantages of being able to utilize tools and resources that are available to any other application written in javascript. However, it is not an exclusive api, it can be utilized by any javascript code running on the same thread.

Interesting Aspects

In Mozilla's DevTools some of the interesting aspects is the architectural design is broken into 3 components client (front-end), server (back-end) and shared (shared components). This allows for contributors to have a better understanding of the code structure as it is clear what each section's purpose is. The degree of coupling between the server and the client is strong because of the usage of actors which allows for one client to connect to several servers inside devtools. The quality of the architecture is also well done, as it allows for ease of implementation for new features, requiring just the creation of a new actor that communicates with the client.

Software Development Process

The software development process our team will be utilizing is the Reuse-Oriented plan driven model based on the principles of Waterfall. We will be using a modified version of this, indicated in the diagram drawn below. The steps of this modified version are as follows; requirements and requirements definition/specification, component analysis, system design with reuse, implementation and unit testing, system validation.



Pros of our Software Process

Since we will be contributing to an existing project, we will be able to use existing components for the project. Especially since our task is to fix existing bugs it is essential that we utilize tools already created for the project. Mozilla DevTools is a very complex project with numerous different contributors that have helped create existing apis. It is essential that we utilize all the available tools in order to fix issues/bugs so that we can contribute effectively. As aspiring software developers we need to be able develop efficiently and effectively because there is a cost associated with development, which can be time or money.

Furthermore, we will have predefined requirements since we will be working on issues/bugs that have been reported by users of Mozilla Devtools. These predefined requirements have very little chance of changing because of the time frame we will be working with, which is why it will be more effective for us to utilize the waterfall development process in conjunction with Reuse-Oriented development. When it comes time to select the issues we will be fixing, it is possible that they are not related to each other so we have not need to develop a solution in an iterative process. Our deliverables require us to fix issues by the end of the given time frame so we will need to complete the development/testing process for the issues at once. This deliverable requirement forces us to work on this project in a plan driven process instead of using an iterative development process.

Explanation of our Software Process

The first step of our modified software development process is **requirements definition**; and specification; our requirements will be listed by the bug documentation on Mozilla's open source website. We can use the Mozilla open source documentation to get a general understanding of what we are working with. Overall, this should give us a good understanding of the bug or feature that we will be fixing or implementing if we require more details we will be able to comment on the open source community and ask for extra details or guidance as necessary.

The second step of our modified software development process is **component analysis**; we will need to understand if there are other dependencies in the code that are affected by the component we are working on. We will also need to understand if there are external apis or databases/servers that will be impacted by our fix. We can use existing code and documentation to have a clear understanding of these components. By the end of this step we should have a clear understanding of the various components of the code that we will be working with and modifying.

The third step of our modified software development process is **system design with reuse**; after analysis and having a clear understanding of the components and the code that we will be working with. We can start to design a workflow of our solution and break all features up into smaller subtasks. Since every member of the team will have a good understanding of the project from the first two stages, we can distribute all subtasks to each individual.

The fourth step of our modified software development process is **implementation and unit testing**; we will initially create failing test cases for the bugs that we are fixing. We will then implement code that will then make the tests pass. Each individual will work on their own branch fixing the subtasks that they are assigned. After they have completed their tasks and have a working branch, they will create a merge request which will then be peer reviewed with another member before merging the working code to master.

The fifth step of our modified software development process is **system validation**; After the previous stage, we should have the working code from each branch merged into master. We will then verify that the cumulation of the different components has completed our task and the bug has been fixed without altering any of the other constraining components. We should be able to run the entire test suite, along with previously existing tests and verify that they still pass with our code changes. After we have validated that our project has finished, we will create a merge request to close the bug we were working on. We will use the same process from the beginning for each bug and feature that we will complete.

Cons of other Software Process

One of the software development practices that our group will not be using is the **incremental process**. There are several reasons for this decision, mainly because that one of the reasons people choose to use the incremental process is for its ability to be flexible when the client wants change. In our case we are working on an open source project; Mozilla’s DevTools and the bugs that are needed to be worked on are already posted. In some cases, these postings have been up for several months, with requirements clearly outlined already. It is unlikely, or even impossible for the client to change the requirements of the bug as we are working on the fix. Moreso, the requirements are concrete and clear, which means change is unlikely to be necessary during the development. Incremental also allows developers to have rapid delivery and deployment of their software. Although this is good, it is not necessary for our situation since we are working on a few bugs and are not required to have rapid delivery. In fact, we do not have enough deliverables to take advantage of this aspect of incremental.

We also decided to not use the **scrum/agile** process. Like the incremental process, agile offers us much flexibility in the case of the client wanting change. Unfortunately, we do not need this aspect of the software development process since the requirements are clearly outlined before we start. Scrum also requires us to have constant meetings with group members, as well as the client to check in on the progress. In addition to this, there are many weekly activities such as sprint planning, grooming, etc.. that the agile process encompasses. As we are on a shorter period for completion of this project there are no benefits to following a scrum/agile approach. All of these aspects will only create more overhead for our group, which would be fine if the benefits outweigh the costs. In our case, the additional planning and organization would create more work in our situation.

We also decided to not use **kanban**, which uses a billboard type system which has three categories requested, in progress and done. For our next deliverable we will be focusing on 3 bugs, with this breaking it down into subtasks which are independent of each other and using kanban wasn't in track with how our tasks will be done. This allows us to focus on a specific set of tasks until they are done, and eliminates context switching and multitasking, overall in a short time frame for this project kanban is not a good tool to use for it.

Conclusion

The decision for our group to use the modified waterfall and reuse oriented software process for our project was well thought and thoroughly researched. We examined the alternatives that we could have used, such as incremental, and scrum/agile. From these processes, we analyzed the cost-benefit of using each of them and decided to incorporate a few aspects of reuse-oriented into our waterfall process. Due to the nature of incremental and scrum/agile allowing flexibility to accommodate client’s demand for change, it requires planning and constant feedback with the client on the progress. This creates much overhead for us, which is not necessary since the requirements are clear from the start. One of the disadvantages to waterfall is that it is not very flexible to change, which does not apply to our situation. Thus, because changes are unlikely to occur, it would make sense for us to choose this modified version which combines waterfall and reuse oriented software process as the software process for our group.