GRST Programming Challenge Aggregator

Prepared by: Matthew Griffin, Ashish Rastogi, Kiritbhai Soheliya, Martin Tiernan

Software Engineering, Fort Hays State University

https://github.com/mgriffin3/GRSTInteraction Diagrams

Table of Contents

Interaction Diagrams 3

Front-end Class Diagram 7

System Architecture and System Design 10

Project Management and Plan of Work 12

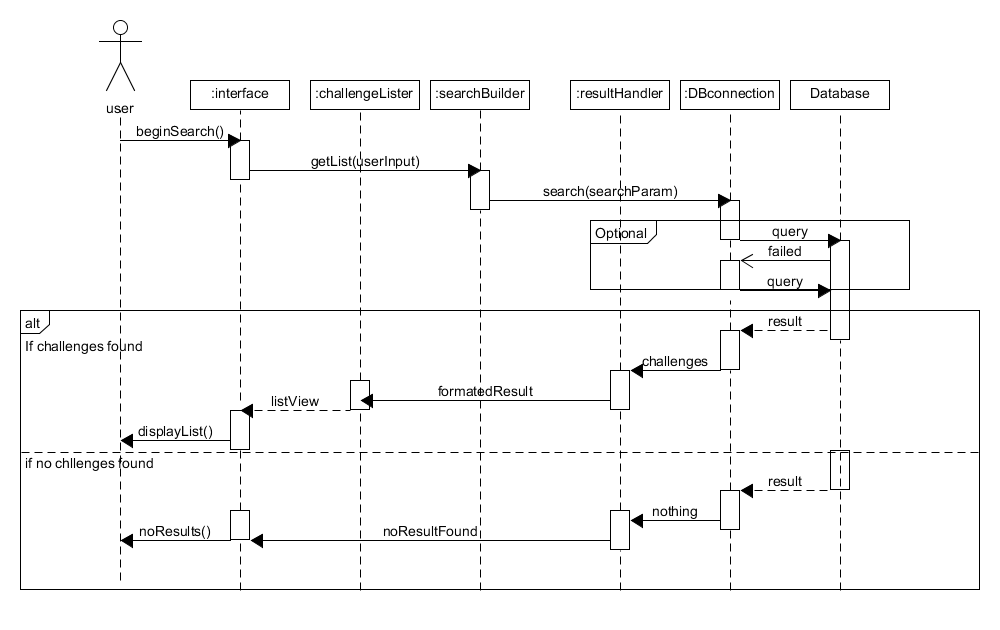
* + - Matthew Griffin: Project Management Section, System Architecture and System Design parts c-g,
    - Martin Tiernan: Front end Class Diagrams, Architectural Styles, Identifying subsystems
    - Brenton Dube: Getting up to speed on project

Interaction Diagrams

We focused on a main set of four use cases that represent the core of the program search, view, fetch, and store. This encompasses the basic functions of both our programs front and back ends. It is important that this core is well designed and understood before any future features can be added. The focus of this report will be on a “Guest” user since a registered user is expanded upon a guest user experience we can’t implement those features without this base.

**Use Case 1: Search**

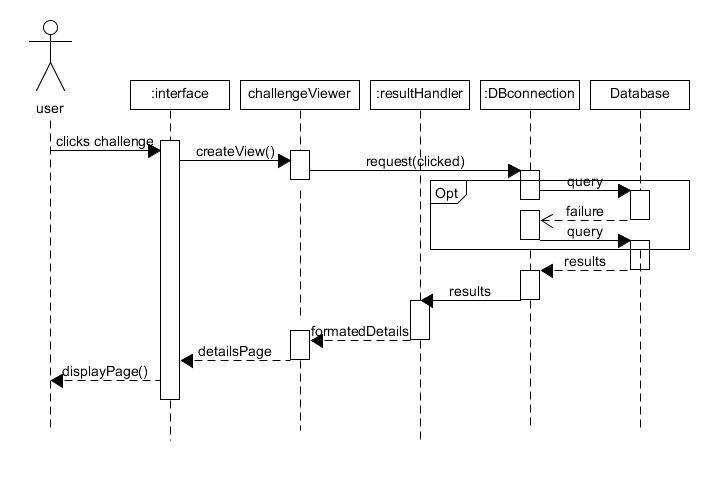
The first and primary use of the application is searching the site. This will be used by both “guest” users and registered users. This begins at the interface which then sends the users requests through all the necessary subsystems. This is similar to Use Case two but has slightly more interaction since it can return a blank result.



**High cohesion principle**: The user requests a search which is passed on to the search builder which represents the high cohesion principle. The interface only handles the interaction between the user and sends requests to other objects to fulfil the request. Also challenge lister is only responsible for creating a list page, if no results are found there is no reason for it to be called. Additionally, separating the result and the challengeLister and resultHandler a change in the format in the database when not affect the creation of a list. Creating high cohesion.

**Expert Doer principle:** By having the DBconnection be the only person that interacts with the database it allows for the other subsystems to not car about how to access the actual database. DBconnection is the only part that accesses in both the front and back ends.

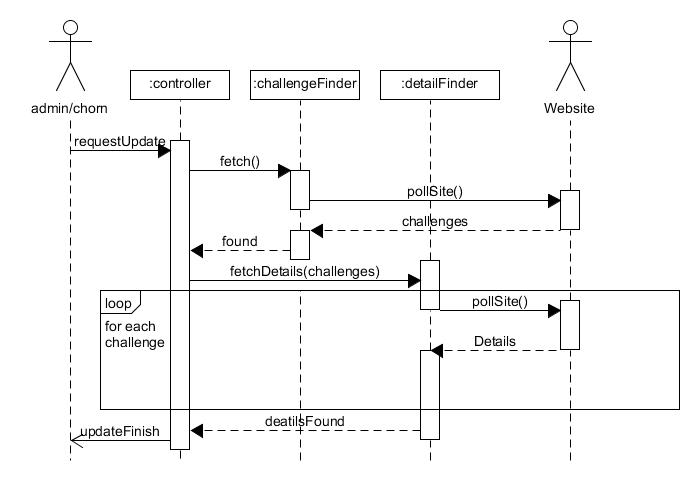
**Use Case 2: View**

 The second most important thing is for users to be able to access the details of the challenges. The interface yet again sends the request through various subsystems. It is important to note that in both use cases that the interface doesn’t manipulate or handle the data directly, but simply interacts with the user presenting the elements created by the system. Since we know that a challenge is stored with details, we know there will not be a case of a blank query, unless there is database malfunction. This is the primary difference between use case one and 2.

**High Cohesion Principle:** When a user clicks on the challenge the interface does request to build the challenge view. By separating the creation and the displaying the application becomes more cohesive.

**Expert Doer Principle:** The results handler handles results from the database no matter what it is since it already knows how to handle challenge input there is no reason to creating another subsystem to handle data from the database.

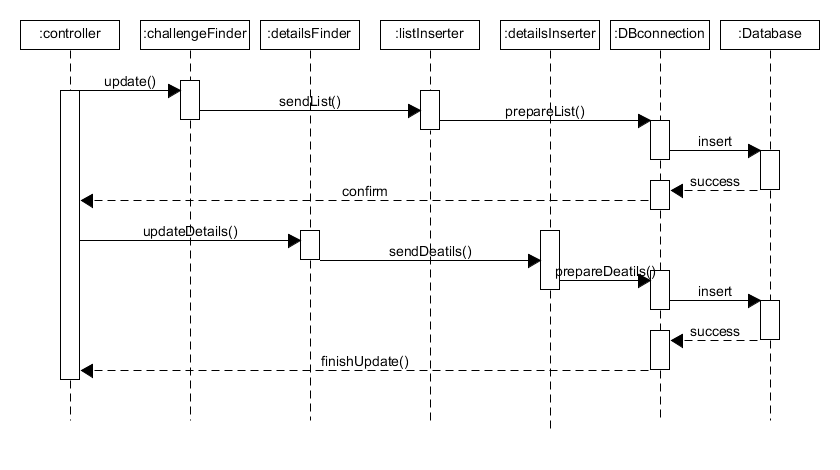
**Use case 7: Fetch**

 Use case 7 and 8 use to be a single use case, but they are both independent enough to be separated. This begins with a request from the system or an admin and then the controller takes over. The controller requests both the challengeFinder and the DetailFinder to scrape the data from the specified website.

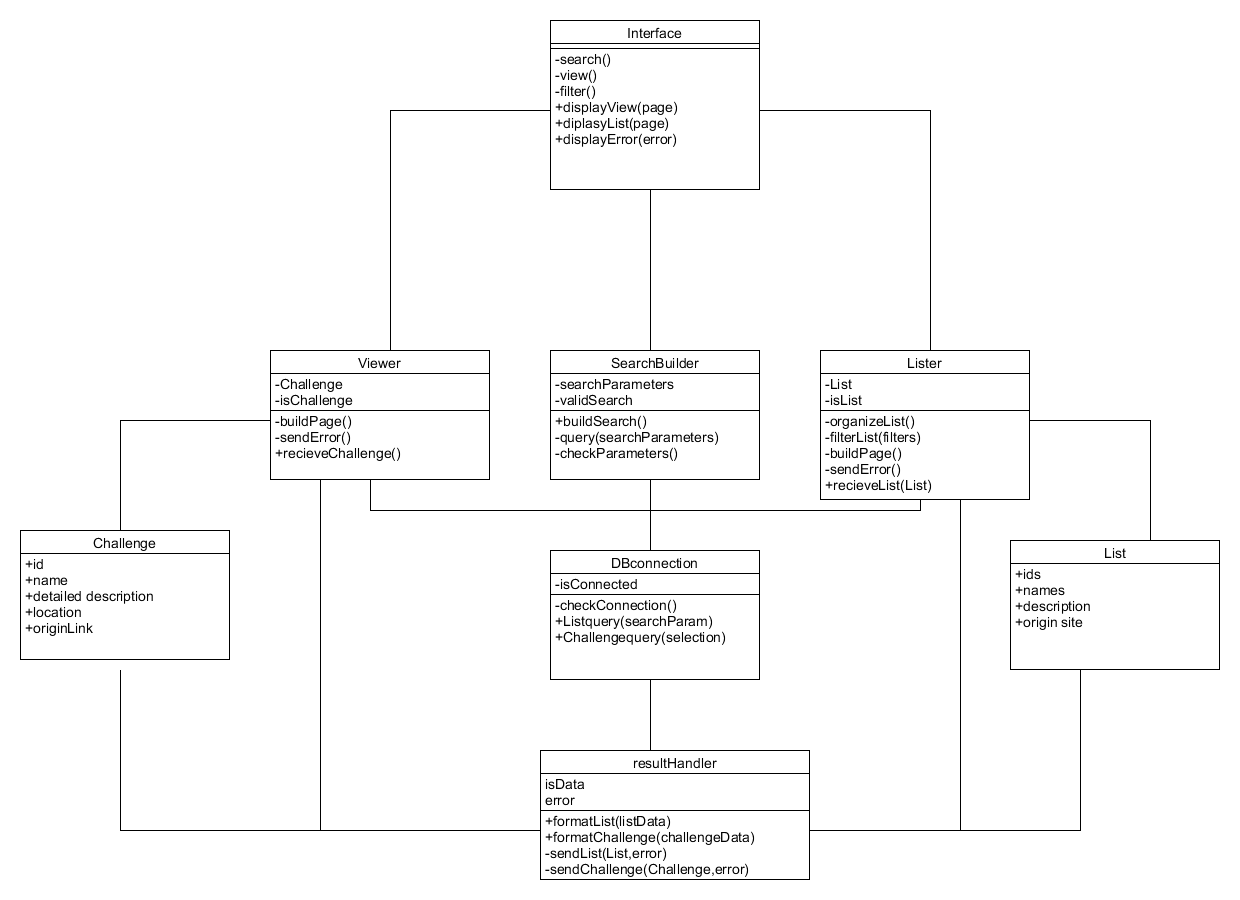
**High Cohesion Principle**: Separating the challenge finder and detail finderallows for formatting for two different types of data the location of challenges and the details of the challenge they can be manipulated as needed since they are two very different pieces of data. The controller doesn’t actually manipulate data, but rather control timing and pass messages along to other subsystems.

**Use Case 8: Store**

This case will be automatically called by the system when there is information in finder classes to store. It will then began formatting the data in the inserter objects and the pass it to DB connection to store within the database.

**High Cohesion Principle:** By separating the Challenge locations and the details of those challenges we can format them appropriately without having one big formatting class. This is shown in the separation of detailsInserter and detailsInserter.

**Expert Doer Principle**: Again here we see that DBconnection is the one source that interacts with the database. This makes easy to change interactions with the database since it is all contained within one subsystem.

Front-end Class Diagram

Front-end data and operator types

**Interface:**

Operations

1. -search(): trigger an html event that calls to SearchBuilder and passes html forum information
2. -view(): trigger an html event that calls the Viewer and passes challengeID
3. -filter(): calls lister to filter
4. DisplayView(page): displays new html page received from viewer
5. DisplayList(page): displays a new list that is returned from Lister
6. DisplayError(error): displays error message based on type

**Viewer:**

Attributes:

1. -Selection: a challenge datatype
2. -isChallenge: bool to check if it has an actual challenge

Operations:

* -buildPage(): build html from Challenge info and call interface to display
* -sendError(): error detected when trying to create page
* +recieveChallenge(challenge): receive a challenge data, assign to selection and begin calling either buildPage or sendError

**SearchBuilder**:

Attributes:

* -searchParameters: String
* -validSearch: Bool

Operations

* +buildSearch(): creates search parameters based on passed forum information
* -query(searchParameters): send a search to db connection
* -checkParameters(): check to see if valid parameters or return error.

**Lister**:

Attributes:

* +CurrentList: List datatype
* -isList: bool

Operations:

* -organizeList(): sort and organize list for the optimal viewing
* -FilterLIst(filters): filter List based on filters
* -buildPage(): build html page and call the interface to display
* -sendError(): send error to interface
* +recieveList(list): assing list as currentList and call buildPage() or sendError()

**Challenge**

Attributes:

* +id: int
* +name: string
* +detailedDes string
* +originSite string
* +originLink string

**DBconnection**

Attributes:

* -isConnected bool

Operations:

* -checkConnection(): check to see if there is a response from database
* +listQuery(searchParam): called by searchBuilder query database and then calls resulthandler
* +viewQuery(selection): called by viewer returns data to resulthandler

**List**

Attributes:

* +ids: int array
* +names: string array
* +shrotDes: string array
* +orignSites: string array

**resultHandler**

Attributes:

* isData : bool
* error: int

Operations:

* +formatList(ListData): turns database data into List data structure
* +formatChallenge(challengeData): turns database data into Challenge data structure
* -sendList(list,error): send a List to listbuilder and include any errors that occurred
* -sendChallenge(challenge,error): send a Challenge to viewer and include any errors that occurred

System Architecture and System Design

**Architectural styles**

The most benefiting style for are system would be a multitiered architecture. Specifically, a three-tiered architecture. In this style we can separate all the components into three main systems, presentation logic and application. This is the classical design for web-based apps and websites, so it is the most fitting for our programming solution. In this style the presentation front-end is responsible for showing user the data that the logical application layer calculated or retrieved from the data tier.

This benefits us the must because it allows us to format the data as needed, as we are pulling from multiple sites. It is beneficial to have the database separate in these layers we only need to make changes in one subsystem to accommodate the change, as opposed to more centralized styles which would be more work to adapt and accommodate multiple sites.

**Identifying Subsystems**

**Presentation/Interface:**

The first subsystem that is present within is the presentation. This contains both our user interface and our back-end controller, this subsystem is responsible for requesting the builder subsystem to retrieve/store data and present the users with the outputted data. This subsystem doesn’t care about what or how retrieved data was handled, calculated or stored.

**Application/Worker:**

Are worker subsystem being what creates data to display to user. This includes are finders, and builder classes. This layer is both responsible for bridging the users with the database. It both formats for display and storage. There should be no direct interaction with the database here. It only manipulates data that has been acquired already. If a change to the database should occur this subsystem should mostly be unaffected.

**Data/Database:**

The third and final subsystem is the database. This also includes inserters and the result handler classes. In this stage the defining feature is if we were to change the database system would the classes need to be changed. Only classes that have direct interaction with the database here.

**Mapping Subsystems to Hardware**

Our system will run from two locations: the user’s computer and a server. The server will contain our database, web scraper, and the parts of our front-end that connect to the database. The actual user interface will be run client-side as a web

**Persistent Data Storage**

We are using a relational database for persistent storage. The database will contain all collected information about challenges and will also store user account information.

The DESC of the challenge table will look like:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| id | int(11) | NO | PRI | *NULL* | auto\_increment |
| name | text | NO |  | *NULL* |  |
| text | text | NO |  | *NULL* |  |
| origin | text | NO |  | *NULL* |  |
| difficulty | text | NO |  | *NULL* |  |
| url | text | NO |  | *NULL* |  |

User accounts are not being implemented until demo 2. Tentatively, there will be a user accounts table with user id and user name. Each account will get a custom table for keeping track of favorite challenges and status changes. This may change based on security or sprred/size concerns.

**Network Protocol**

Our systems will use HTTPS.

**Global Control Flow**

Our system is partailly procedure and partially event driven. Our back-end is procedure driven. The only option for the back end is whether or not to get challenges from a given site – everything else is automatic. The front end is event-driven. It does nothing until the user tells the system to search. Once a list of challenges has been returned, they can either modify the search and search again or select any returned challenge to view more information. Our system uses no timers, has no time dependency or requirements, and only uses single threads.

**Hardware Requirements**

Our back-end and database will require a server with a high-speed internet connection. The front-end will require a computer, a monitor, a contemporary web-browser and a high-speed internet connection. Our system is not resource intensive; any computer capable of running the current version of firefox should have no issues.

**7. Project Management and Plan of Work**

1. Merging the Contributions from Individual Team Members  
   Compiling the final copy of the report from everyone’s work, ensuring consistency, uniform formatting and appearance.  
     
   Currently Matt is merging the contributions from Team Members.  
     
   Describe what issues were encountered and how they were tackled.
2. Project Coordination and Progress Report  
   What use cases have been implemented?   
   What is already functional, what is currently being tackled?  
   List and describe other relevant [project management](http://www.ece.rutgers.edu/~marsic/Teaching/SE/projects.html" \l "TEAMS) activities.
3. Plan of Work  
   List the projected milestones and dates by which you plan to accomplish them. Preferably, you should use [Gantt charts](http://www.ganttchart.com/) for planning and scheduling your project.

This Gantt chart shows the milestones required in order to meet a November 1st deadline. This will allow for several days to record demo 1. The database, web scraper, and User Interface can all be created independently. The uploader is dependent on the Web scraper. Details are dependent on results which are dependent on search, which is dependent on both the user interface and the uploader.

Additional features and requirements, most notably related to user accounts, will be implemented for demo 2. As demo 2 approaches, updates will be made to the Gantt chart and plan of work.

1. Breakdown of Responsibilities
   * List the names of modules and classes that each team member is currently responsible for developing, coding, and testing
     + Matthew Griffin: Will finish back-end documentation and implementation.
     + Martin Tiernan: Will continue working on front-end and documentation.
     + Brenton Dube: Just joined project. Will study existing documentation to get up to speed on project direction. Will create basic non-functional UI for front end.
   * Who will coordinate the integration?
   * Who will perform and integration testing? (The assumption is that the unit testing will be done for each unit by the student who developed that unit.)