Cyclomatic Complexity

We tried to explore different tools to calculate cyclomatic complexity, but were unable to integrate it easily. We decided to calculate the cyclomatic complexities of the different methods in the modern way:

> Assign one point to account for the start of the method
> Add one point for each conditional construct, such as an "if" condition
> Add one point for each iterative structure
> Add one point for each case or default block in a switch statement
> Add one point for any additional boolean condition, such as the use of && or ||
> With exceptions, you can add each throws, throw, catch or finally block as a single point when calculating the McCabe cyclomatic complexity metric.

The ideal McCabe complexity is 4 or 5, but 1-10 is considered well-structured and highly testable. For the sake of brevity, we will include the calculations for the Database file, as well as the number of methods with complexity over 10, and the percentage of that file's methods that are low complexity.

Overall, there were 5 complex and 1 very complex methods. This was due to list management and different cases that we had to explore. We could have split the method up further to reduce complexity, but we found the increased complexity to still be very readable and logical, so we let those methods have their higher complexity. The project has 83.3% low complexity, with an average cyclomatic complexity of 6.83 per method (when divided by methods and not by non-method sections as well, otherwise it would be 6.67)., which is not ideal, but it isn't too far off of well-structured code and still has high testability.

Java Files

Database: 2 complex, 2 very complex. 71% low complexity.

checkUser(String): Boolean	3 (start of the method, if statement, catch)
getLists(int userID, String listname): ArrayList<Info>	14 (start, 8 if/elseif, 2 while, 2 for, catch)
addToList(int userID, Boolean isRecipe, String listname, Info i): Boolean	23 (start, 21 if/elseif, catch)
removeFromList(int userID, Boolean isRecipe, String listname, Info i): Boolean	25 (start, 23 if/elseif, catch)
updateLists(int userID, Boolean add, String	3 (start, 2 if, 2 &&)

listname, Info i): Boolean	
getPrevSearch(int userID): ArrayList<Searches>	3 (start, while, catch)
addPrevSearch(int userID, String testSearch, int radius, int results): Boolean	3 (start, if, catch
changeOrder(int userID, String listname, Boolean isUp, int position): void	2 (start, if)
getrestID(int position, String listname, int userID): int	6 (start, 3 if/elseif, while, catch)
updatePos(int position, String listname, int userID, int rID, Boolean isRecipe): int	14 (start, 6 if/elseif, 6 &&, catch)
move(int userID, String listname, int posit, Boolean moveUp): void	4 (start, 3 if/elseif)
getrecipeID(int position, String listname, int userID): int	6 (start, 3 if/elseif, while, catch)
changeToDatabaseFormat(String listname): String	4 (start, 3 if/elseif)

DatabaseHelper: 0 complex. 100% low complexity.

findHighestPos(String listname, int userID): int	5 (start, 2 while, 2 if)
updateIndicesAfterRemove(String listname, int pos, int userID): void	4 (start, for, 2 if)
changeToDatabaseFormat(String listname): String	4 (start, 3 if/elseif)

ListServlet: 1 complex. 0% low complexity.

doPost(HttpServletRequest request, HttpServletResponse response): void	13 (start, throws, 3 if/ifelse, 3 &&, 4 cases, catch)

SearchServlet: 0 complex. 100% low complexity.

doGet(HttpServletRequest request, HttpServletResponse response): void	7 (start, 4 if, ||, throws)
getJSONResponse(String url): String	3 (start, while, catch)
restaurantSearch(String query, int numResults, int radius, List<Info> doNotShowList, List<Info> favoritesList): ArrayList<RestaurantInfo>	9 (start, 3 for, 3 if, while, catch)

SortLists: 0 complex. 100% low complexity

moveItemUp(ArrayList<Info> list, String itemToMove): ArrayList<Info>	3 (start, 2 if/elseif)
moveItemDown(ArrayList<Info> list, String itemToMove): ArrayList<Info>	3 (start, f2 if/elseif)

JS Files

ListClient: 0 complex. 100% low complexity

reorderResults(order, listName)	1 (start)
groceryCheckbox(item, check)	2 (start, if)

listPage: 1 complex. 0% low complexity

(not inside a function)	16 (11 if/elseif, 4 for, ||)

resultPage: 2 complex. 82% low complexity

(not inside a function)	3 (3 for)
go(search_term,search_number,search_radius)	1 (start)
assembleCollage(append, previmageURLS)	2 (start, for)
load()	1 (start)
makeList()	3 (start, 4 if/elseif)
loadRestList()	18 (start, 10 if/elseif, 2 while, for, 4 &&)
loadRecList()	18(start, 2 while, for, 10 if/elseif, 4 &&)
createNumberedRestButton(start, end)	2 (start, for)
createNumberedRecButton(start, end)	2 (start, for)
createRestButton(value)	3 (start, 2 if/elseif)
createRecButton(value)	3 (start, 2 if/elseif)
drawRestList(begin,end)	5 (start, 2 for, 2 if)
drawRecList(begin,end)	5 (start, 2 for, 2 if)

All files not included have no methods or functionality that would fall under complexity.

- Content coupling – Comp1 directly affects Comp2
- Common coupling – Comp1 shares data with Comp2
- External coupling – communication via external medium
- Control coupling – Comp1 controls execution of Comp2
- Stamp coupling – complete data structures exchanged
- Data coupling – only simple data exchanged

Complexity (want less complexity)

- Inter-module: Looks at the complexity of the dependencies between modules
    - 1. A contains B 2. A precedes B 3. A uses B
- Intra-module: inside one module

Hierarchy Graphs

listpage.jsp → loginChecker.js (checkLoggedIn())
→ resultPage.jsp
→ searchPage.jsp
→ function setStoredItem()
→ function(ind)
dropdown.js
ListClient.js
parseQueryString

recipePage.jsp → loginChecker.js (checkLoggedIn())
→ resultPage.jsp
→ recipePagePrint.jsp
→ function addToList→ update back to results button → search resultPage
dropdown.js
ListClient.js
parseQueryString.js
recipePage.js

recipePagePrint.jsp → loginChecker.js (checkLoggedIn())
dropdown.js
parseQueryString.js
recipePage.js

restaurantPage.jsp → loginChecker.js (checkLoggedIn())
→ resultPage.jsp
→ restaurantPagePrint.jsp
→ function addToList→ update back to results button → search resultPage
dropdown.js
ListClient.js
parseQueryString.js
recipePage.js

restaurantPagePrint.jsp → loginChecker.js (checkLoggedIn())
dropdown.js
parseQueryString.js
recipePage.js

resultPage.jsp → loginChecker.js (checkLoggedIn())
→ listPage.jsp
→ searchPage.jsp
→ create collage
dropdown.js
parseQueryString.js

Database.java
→ JDBC driver
→ function checkUser(String username) → executeQuery()
→ function getPasswordInfo(String username) → executeQuery()
→ function getUserID(String username) → executeQuery()

→ function createUser(String username, String passwordHash, String salt)
→ function getLists(int userID, String listname)
→ function addToList(int userID, Boolean isRecipe, String listname, Info i)
    → add recipe
    → add restaurant
→ function removeFromList(int userID, Boolean isRecipe, String listname, Info i)
    → for recipe
    → for restaurant
→ function updateLists(int userID, Boolean add, String listname, Info i)
→ function getPrevTests(int id)
→ function addPrevSearch(int i, String testSearch, int i1, int i2)

loginPage.jsp → function sendData()
→ function hexString(buffer)

signupPage.jsp → function sendData()
→ function hexString(buffer)

Imhungry.sql → groceries
→ prev search
→ recipe and lists
→ restaurant and lists
→ users

ListServlet.java → doGet(HttpServletRequest request, HttpServletResponse response) (fetch list contents)
→ doPost(HttpServletRequest request, HttpServletResponse response) (add/remove list items)

SortLists.java → function sortAlphabetically(ArrayList<Info> list) → return list
→ function sortByDriveTime(ArrayList<Info> list)
→ function sortByPrice(ArrayList<Info> list)

SearchServlet.java → function doGet() data from Search class
→ function getJSONResponse(String url) → API
→ function recipeSearch(String query, int numResults, List<Info> doNotShowList, List<Info> favoritesList) (Given a String query and number of results expected, return an ArrayList of RecipeInfo. The function uses 2 Spoonacular's recipe APIs and refer to Do Not Show List and Favorites List)
→ function restaurantSearch(String query, int numResults, int radius, List<Info> doNotShowList, List<Info> favoritesList) (Given a String query and number of results expected, return an ArrayList of RestaurantInfo. The function uses several Google Maps APIs and refer to Do Not Show List and Favorites List)

→ function getDistances (ArrayList<RestaurantInfo> restaurants) (grabs the distances of each restaurant and stores in a map)
→ function getDriveTimes(ArrayList<RestaurantInfo> restaurants) (Create a request using place_id of all RestaurantInfo and send one request to obtain all drive times.)
→ function getPhoneAndURL(ArrayList<RestaurantInfo> restaurants) (separate request is needed to get detailed information including phone and URL)
→ function getImageURLs(String query) (Given a String query, return an ArrayList of String storing URLs for images to present in the collage. The function uses Google Custom Search API. The search engine is configured to search for images.)

searchPage.jsp → loginPage.jsp
→ resultPage.jsp
→ function happyToSad()
→ function sadToHappy()

LoginServlet.java → doPost(HttpServletRequest request, HttpServletResponse response)t (POST method used to add and remove items from a list)
→ Database → checkUser(username) → createUser(username, PasswordHashing.hashPassword(password, salt), salt) → searchPage.jsp

1. List the classes/components that provide its implementation
2. List the dynamic diagrams that show how its classes/components interact to implement its required runtime functionality
3. Provide natural language explanations of this mapping

I know that for our descriptions for simple design we described how our design
- had low coupling and high cohesion
- how our call graph looked like with words a little vaguely, albeit we lost points. But from that we found out that we had to do a call graph
- we would argue why the complexity was simple from that (line count, branches, etc. per feature)