The Drag - Phase IV

DESIGN REPORT

GitHub Repo: github.com/53Dude/TheDrag.git
Site Link: thedrag.appspot.com

Daniel Lazcano
@: daniellazcanoplus@gmail.com **Git:** github.com/DanielLazcano

Ethan Santoni-Colvin
@: ethan.santonicolvin@gmail.com **Git:** https://github.com/ethansantonicolvin

Frank Le
@: fle734449@gmail.com **Git:** https://github.com/fle734449

Guy Sexton
@: gsexton@utexas.edu **Git:** https://github.com/53Dude

Jonathan Walsh
@: jdwalsh79@gmail.com **Git:** https://github.com/jdw4867

Kishan Dayananda
@: kishdaya@gmail.com **Git:** https://github.com/kish314

Phase I Lead: Kishan Dayananda
Phase II Lead: Ethan Santoni-Colvin
Phase III Lead: Guy Sexton
Phase IV Lead: Jonathan Walsh

Group: Morning - 4 / Pit Crew
Slip Days Used for Phase II: 1 day
Slip Days Used for Phase III: 0

Part 1a: Information Hiding

We have applied information hiding in how we access our database. Our JSP files, which display the information of the models and instances, don't need to know anything about the database. The JSP files don't make the call to the database to get the data and attributes, instead it calls a servlet which returns the requested attribute. In fact, the method in the servlet to return the desired attribute was made to return any attribute. Here is some pseudocode for how the DBServlet was utilized:

```
In ExamplePage.jsp
      Create new DBServlet instance
      Get array of names for the listing page from instance
      For each string in the listing names
            Request attributes from the DBServlet instance and
            generate a card for them
```

We anticipated that every phase we would collect more data, so we made the servlet account for this by applying the Open-Closed principle. The servlet handles getAttribute requests by only requiring the set of documents and the query term for the attribute. For example, getCarAttribute would specifically search the car collection in our database and would be passed 2 strings, an id and an attribute, which would be used as a query for the specific instance's data. This way if we wanted to add more data on our models to the database we would not need to modify our code in order to fix it. All we would need to do is add more code or extend or code to cover for the new attributes. Specifically, if we added new data to our database and wanted to display the new data on an instance page, all we would need to do is add new code in the jsp file to display the desired attribute. As a result, we don't need to go back and fix any code, we just need to add on code. This was actually setup by the end of Phase I and we were able to use it when adding attributes during Phase II to make that conversion of the website easier.

One disadvantage to the current information hiding setup we had previous to Phase IV is the size of the DBServlet class. This setup definitely gave off a "large class" code smell as the DBServlet was intended to be the main interface between the website and the database. This has been mended in our refactoring and the class has been divided up into the pieces needed for each model.

Part 1b: Design Patterns and Refactorings

Design Patterns:

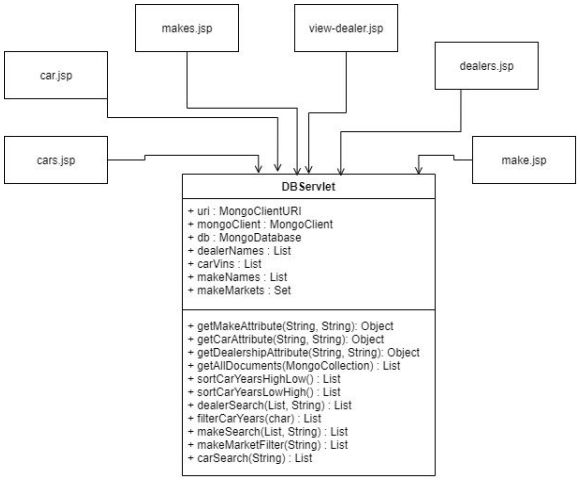
DBServlet Factory Design Pattern:

One design pattern we decided to implement in our code is the factory design pattern. Before, we had a single class handling any accesses needed by front-end to data held in our MongoDB database. This class was called DBServlet, it essentially encapsulated any calls to the database for data, be it for the attributes of a car, make, or dealer, or to do things like searching and filtering in the backend.

We decided the factory design pattern would work well here because we could instead make a DBServlet for each of the three model types instead of using one super generic one. Now, we have ServletFactory which takes “cars”, “dealerships”, or “makes” as an input, and then generates an instance of one of DBServlet’s newly-made subclasses based on it. The pieces of code calling for a DBServlet are abstracted from knowing how the specific Car/Make/DealerServlet object was created, which follows examples of the Factory Pattern as intended. So, for example, a page that needs information about a car can call for ServletFactory to return a CarServlet and then use CarServlet’s methods for whatever information is needed.

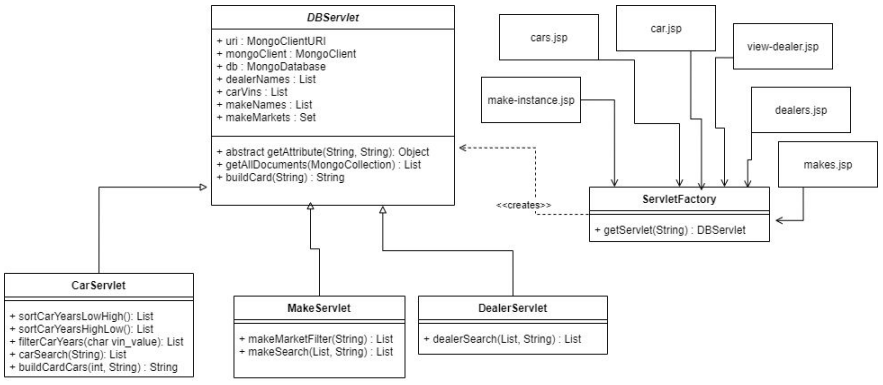
There are multiple advantages to switching to this pattern. First off, the large DBServlet class is now broken into smaller subclasses, making the code more legible. Additionally, because these subclasses have methods more specifically parted out, pages don’t automatically have access to a large number of database accessing methods that they don’t need. One disadvantage of this pattern is that multiples pages require access to more than one type of Servlet. For example, a make--instance.jsp page needs an instance of each of the three types of DBServlet to get all the info required for the page. This means 3 objects instead of one general one. However, we think the benefits of implementing this design factor outweigh that downside because it increases the modularity of our code.

Original (make/dealer/car attribute getters, etc. all in DBServlet)




```
public Object getMakeAttribute(String makeName, String attribute) {
        MongoCollection col = db.getCollection("makes");

        Document doc = (Document) col.find(eq("_id", makeName)).first();

        if(doc == null)
                return null;

        return ((Document)doc.get("query")).get(attribute);
}

public Object getDealershipAttribute(String dealershipName, String attribute) {
        MongoCollection col = db.getCollection("dealerships");

        Document doc = (Document) col.find(eq("_id", dealershipName)).first();

        if(doc == null)
                return null;

        return ((Document)doc.get("query")).get(attribute);
}

public Object getCarAttribute(String vin, String attribute) {
        MongoCollection col = db.getCollection("cars");

        Document doc = (Document) col.find(eq("_id", vin)).first();

        if(doc == null)
                return null;

        return ((Document)doc.get("query")).get(attribute);
}



private static List<Document> getAllDocuments(MongoCollection<Document> col) {
        List<Document> allDocs = new ArrayList<Document>();

// Performing a read operation on the collection.
FindIterable<Document> fi = col.find();
MongoCursor<Document> cursor = fi.iterator();
try {
    while(cursor.hasNext()) {
```

After (getters, etc. in more specific classes, DBServlet much shorter):




```
public abstract class DBServlet {
        public MongoClientURI uri;
        public MongoClient mongoClient;
        public MongoDatabase db;
        public List<String> dealerNames;
        public List<String> carVins;
        public List<String> makeNames;
        public TreeSet<String> makeMarkets;

        public DBServlet() {
        }


        public abstract Object getAttribute(String name, String attribute);

        protected static List<Document> getAllDocuments(MongoCollection<Document> col) {
                List<Document> allDocs = new ArrayList<Document>();

        // Performing a read operation on the collection.
        FindIterable<Document> fi = col.find();
        MongoCursor<Document> cursor = fi.iterator();
        try {
            while(cursor.hasNext()) {
                allDocs.add(cursor.next());
            }
        } finally {
            cursor.close();
        }

        return allDocs;
    }
```

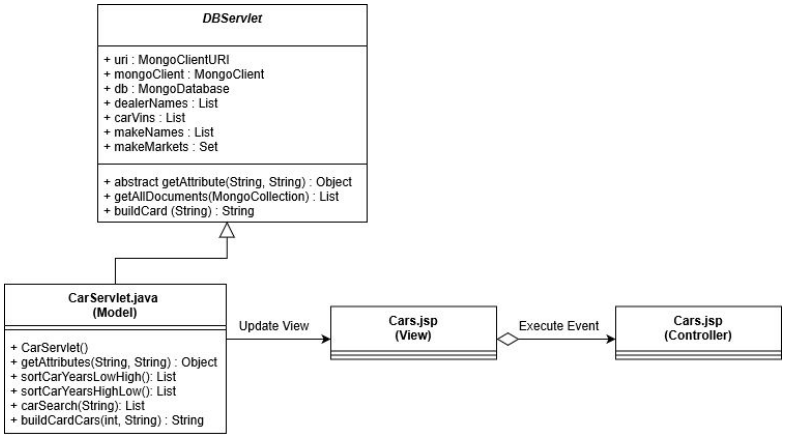
Model-View-Controller Design Pattern:

Another design pattern we decided to implement in our project is the Model-View-Controller (or MVC) Design Pattern. This framework provided a good structure to how we segregated code for modelling our website (aka the java interactions with mongoDB), presenting/displaying information (the html used for graphics/display), and controlling our website (the html/java used for UI/user-input).

Working with a website naturally lends itself to the MVC design pattern because it brings together user interaction with back end data interfacing and front end graphical display. To maintain a streamlined code framework, we would need to seperate each of these three components into separate classes and functions. For example, in cars.jsp, we have explicit interactions with the MongoDB database that exists between the Controller and View components of our project. These interactions reference our servlets which are compartmentalized away from the other code. These servlets are what interface with our backend to return information relevant to the user input, and eventually dynamically generate a new page. This example of the car.jsp file relies on user input to query the back end interface and return relevant components that need to be updated in the View component. The View component is the following HTML code, which takes the Model generated data from user-input and dynamically updates the instance page. This pattern of MVC is similarly replicated in make-instance.jsp and view-dealer.jsp. The Controller portion of our code involves essentially all of the buttons and forms that the user can select, and is primarily encapsulated by each of cars.jsp, dealers.jsp, and makes.jsp. Each has a unique Controller aspect because each of these three models has different types of filtering and sorting that can be applied, but all three have the ability to select page numbers and search.

There are multiple advantages we found to the MVC design pattern. For one, it allowed multiple programmers to be working at the same time on the project due to the compartmentalization. For example, the View component didn't care how it got the variables it was trying to display to the user, and the Model component didn't care what the variables it was returning would be presented. This allowed for simultaneous workflow. We also have the option of displaying the same information any way we want, and changing an individual component of MVC with ease without disrupting the other two, because they have been separated out. We could run into disadvantages in the future of complexity and performance. Each component being isolated means that development is stagnated by one group not "keeping up" per-say. But, we believe the benefits of fast development, and ease of collaboration/de-bugging are worth it.

(MVC UML, shown in this case for our “car” model. Essentially the same for both makes, and dealerships, our other two models)



Model

```
public class CarServlet extends DBServlet {

    public CarServlet() {
        uri = new MongoClientURI(
                "mongodb+srv://jdwalsh21:BI6SfPDyhGX8ihAU@thedragapiscrapes-2duen.gcp.r

        mongoClient = new MongoClient(uri);

        db = mongoClient.getDatabase("The_Drag");
        carVins = new ArrayList<String>();
        MongoCollection col = db.getCollection("cars"); //"cars"
        List<Document> allCarDocs = getAllDocuments(col);
        for(Document doc : allCarDocs) {
            carVins.add((String)doc.get("_id"));
        }
    }

    public Object getAttribute(String name, String attribute) {

    public List<String> sortCarYearsLowHigh() {

    public List<String> sortCarYearsHighLow() {

    public List<String> filterCarYears(char vin_value){

    public ArrayList<String> carSearch(String searchTerm) {
        ArrayList<String> searchResults = new ArrayList<String>();
        MongoCollection col = db.getCollection("cars");
        List<Document> allCarDocs = getAllDocuments(col);

        String lc_searchterm = searchTerm.toLowerCase();

        for(Document doc : allCarDocs) {
```

View

```
<%

String slug = dealership.replace('&','$').replace(' ','_').replace("'",".")+"~";


String listing = "<body><div><ul class=\"list-inline offset-xl-2 col-xl-8\"><li class=\"list-inline-item\"><h2 id=\"CarName\"
listing += "<div class=\"np-img-wrapper col-10 offset-1 \" style = \"width: 100%; height: 50%;\" ><div class=\"card np-element
listing += "<br><div class=\"row\"><div class=\"offset-xl-1 col-xl-4\"> <a href=/html/make-instance.jsp?make=" + make + "><di
listing += "<h5 class=\"card-title\">Manufactured by " + make + "</h5><p class=\"card-text\">Click here to learn more about "
listing += "<div class=\"card np-element\"><div class=\"card-body\"><h5 class=\"card-title\">Price</h5><h1 class=\"card-text\"
listing += "<div class=\"card np-element\"><div class=\"card-body\"><h5 class=\"card-title\">Body Class</h5><h1 class=\"card-t
listing += "<div class=\"card np-element\"><div class=\"card-body\"><h5 class=\"card-title\">MPG</h5><h3 class=\"card-text\">"

listing += "<div class=\"col-xl-4 offset-xl-2\"><a href=\"/html/view-dealer.jsp?dealership=" + slug +"\"><div class=\"card np
listing += "<a href="+ url + "><div class=\"card np-element np-hover\"><div class=\"card-body text-center\"><h5 class=\"card-t
listing += "<div class=\"card np-element\"><div class=\"card-body\"><h5 class=\"card-title\">Horsepower</h5><h1 class=\"card-t
listing += "<div class=\"card np-element\"><div class=\"card-body\"><h5 class=\"card-title\">VIN</h5><h3 class=\"card-text\">"

out.print(listing);

%>
```

Controller

```
<h1 class="offset-1 col-9 np-text-accent">Cars</h1>
 <body class="navbar-dark">
 <br>
      <div style = 'text-align: left'>
          <form action = "/html/cars.jsp?page=1&amp;search=">
              <span class= "col-1 offset-1">
                  <a class="np-element ">Sort by year: </a>
                  <a href="/html/cars.jsp?page=1&amp;sort=year_lowhigh" class="np-element np-hover">Low - High</a>
                  <a href="/html/cars.jsp?page=1&amp;sort=year_highlow" class="np-element np-hover">High - Low</a>
              </span>

              <span class= "col-1 offset-1">
                  <a class="np-element">Filter by year: </a>
                  <a href="/html/cars.jsp?page=1&amp;filter=H" class="np-element np-hover">2017</a>
                  <a href="/html/cars.jsp?page=1&amp;filter=J" class="np-element np-hover">2018</a>
                  <a href="/html/cars.jsp?page=1&amp;filter=K" class="np-element np-hover">2019</a>
                  <a href="/html/cars.jsp?page=1&amp;filter=L" class="np-element np-hover">2020</a>
                  <a href="/html/cars.jsp?page=1&amp;filter=M" class="np-element np-hover">2021</a>
              </span>

              <span class= "col-1 offset-1">
                      <%
                          String search = "";
                          if (request.getParameter("search") != null) {
                              search = request.getParameter("search");
                          }
                      %>
                      <input type="text" class="np-element" id = "user_search_input" name ="search" value="<%=search%>">

                      <button type="submit" style="width: 100px" class="np-element np-hover" >Search</button>

              </span>
```

Refactorings:

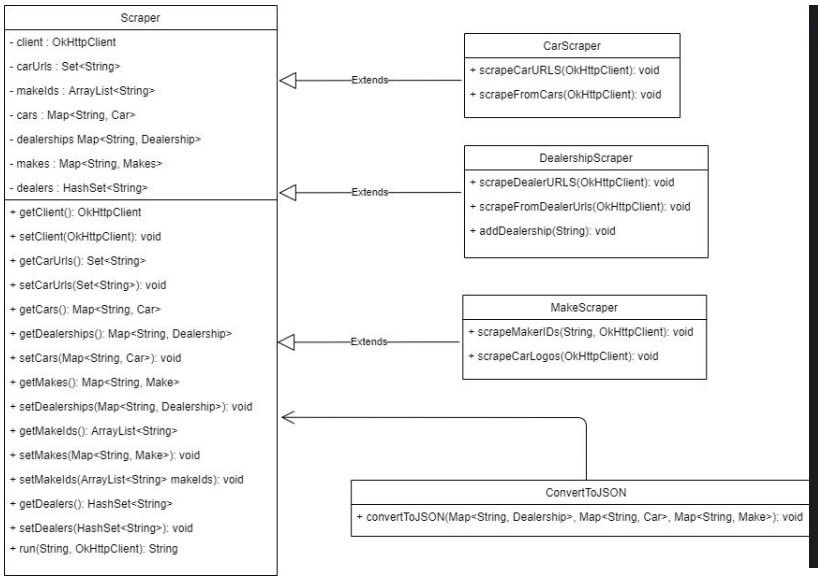
Web Scraper Refactoring:

Initially, the web scraping functionality of TheDrag consisted of many different scraping methods contained within one java file called Scraper.java. This corresponds to the *large class* code smell and led to numerous calls to the particular java file across other files as well as numerous attributes to accommodate the methods that led to bloated code. To resolve this code smell and promote better divisions/usability of similar scraping code we applied the ***extract and move method*** refactoring Scraper.java. This involved breaking up Scraper.java into the different classes ConvertToJson.java, DealershipScraper.java, MakeScraper.java, and CarScraper.java. Due to the different attributes of each of the models (Dealer, Make, and Car), this refactoring helped divide attributes and methods into model-specific classes for more purposeful specific calls in the other files of TheDrag.

Original (Most scraping functionality within main and public attributes in one file)


```
public static void main(String[] args) {
        client = new OkHttpClient.Builder().readTimeout(30, TimeUnit.SECONDS).callTimeout(30, TimeUnit.SECONDS).build

        try {


                //Scrapes all makeIds
                String mdoc = run("https://www.cars.com/", client);
                Elements options = Jsoup.parse(mdoc).getElementsByAttributeValue("name", "makeId").get(0).children();
                String makeId = null;
                for (Element option : options) {
                        makeId = option.val();
                        makeIds.add(makeId);
                }

                //scrape carlogos.org
                for(int i = 1; i < 9; i++) {
                        String makeDoc = run("https://www.carlogos.org/car-brands/list_1_" + i + ".html", client);
                        Elements elements = Jsoup.parse(makeDoc).getElementsByClass("logo-list").select("li");
                        for (Element e : elements) {
                                String url = "https://www.carlogos.org" + e.select("a").attr("href");
                                String name= e.select("a").first().ownText();
                                String market = e.select("a").select("span").eq(0).text();
                                String years = e.select("a").select("span").eq(1).text();
                                String image = "https://www.carlogos.org" + e.select("a").select("img").attr("src");
                                Make m = new Make();
                                m.url = url;
                                m.name = name;
                                m.market = market;
                                m.years = years;
                                m.img = image;
                                makes.put(name, m);
                        }
                }
```

Refactored (Methods moved and divided between respective models and getters/setters)




```
//Run main to web scrape
public static void main(String[] args) {
    client = new OkHttpClient.Builder().readTimeout(30, TimeUnit.SECONDS).callTimeout(30, Time

    try {

        //Scrapes all makeIds
        MakeScraper.scrapeMakeIDs("https://www.cars.com/", client);

        //scrape carlogos.org
        MakeScraper.scrapeCarLogos(client);

        // Scrapes dealer urls
        DealershipScraper.scrapeDealerURLs("https://www.cars.com/dealers/buy/78705/?rd=30&sort

        DealershipScraper.scrapeFromDealerUrls(client);

        // Scrapes car urls
        CarScraper.scrapeCarURLs(client);

    } catch (IOException e) {
        e.printStackTrace();
    }

    System.out.println("\nSUCCESSFULLY SCRAPED ALL MAKES\n");

    //Scrapes from car URLs
    CarScraper.scrapeFromCars(client);

    //Writing to a JSON file with Jackson Library
    ConvertToJSON.convertToJSON(dealerships, cars, makes);

}
```

Instance Listing Pages Refactoring:

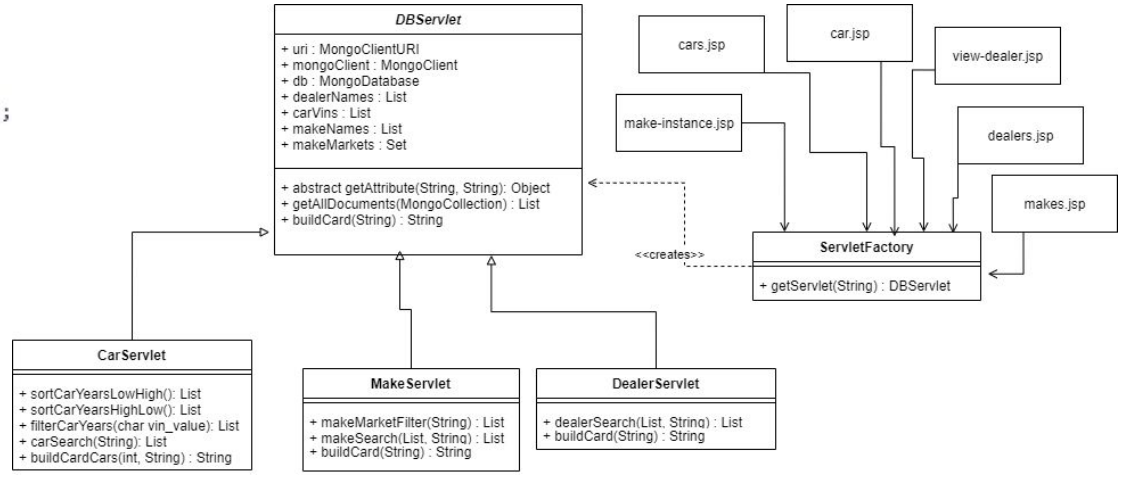
In Phase III and before, our listing JSP pages would generate the cards for each instance locally. We have implemented the ***extract method and move Method*** of refactoring and moved this function into the servlet class, allowing the site to have more models in the future and overall be more scalable as a whole. Now the strings for the instance cards are requested from the JSP and generated in the corresponding servlet class. (*See our Factory Design pattern for the Before UML)*

Original Front-End Code in dealers.jsp with a lot of java code inside jsp file

```
for(String s:pageDealers){
        String name = db.getAttribute(s, "name").toString();
        String slug = name.replace('&','$').replace(' ','_').replace("'",".")+"~";

        String listing= "<li class='card np-element np-hover col-4 dealer-card' style='margin: 20px;height:275px;' >"+
                        "<a href='/html/view-dealer.jsp?dealership=" + slug + "' style='margin:0px;display:block;wi
                        "<h3 style='text-align: center;'>" + name + "</h3>";

      String image = db.getAttribute(s, "img").toString();
      String address = db.getAttribute(s, "address").toString();
      String phoneNum = db.getAttribute(s, "phoneNum").toString();
      String website = db.getAttribute(s, "website").toString();

      listing += "<div class='np-img-wrapper' width='50px' height='50px' style='display:block;'>";
      if(!image.equals(""))
              listing += "<img class='np-img-expand' src='" + image + "' width='inherit' height='inherit' style='margin:
      listing+="</div>";

      if(!address.equals(""))
              listing += "<p><strong>Address:</strong> " + address + "</p>";
      else
              listing += "<p><strong>Address:</strong> N/A </p>";

      if(!phoneNum.equals(" "))
              listing += "<p><strong>Phone:</strong> " + phoneNum + "</p>";
      else
              listing += "<p><strong>Phone:</strong> N/A </p>";

      if(!website.equals(""))
              listing += "<a href='" + website + "'><strong>Visit Dealer Website</strong></a>";
      else
              listing += "<p><strong>No Dealer Website Listed</strong></p>";

      listing += "</a> </li>";

      out.print(listing);
}
```

Refactored using extract and move method so java code executes in java class

```
<%

for(String s:pageDealers){
        out.print(db.buildCard(s));
}

%>
```

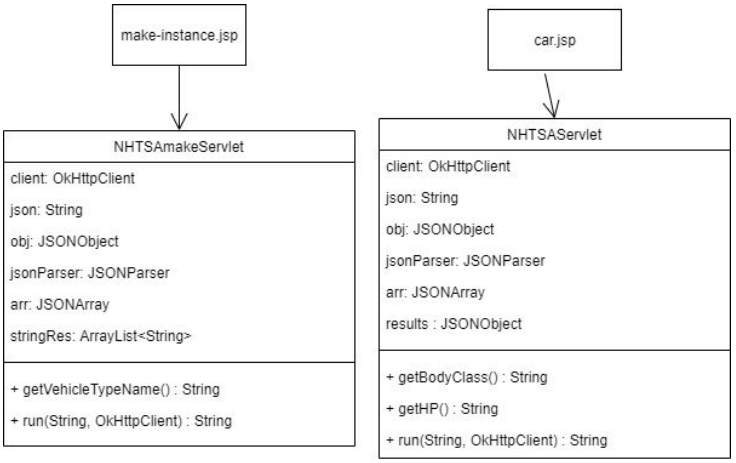


NHTSA API Calls Refactoring:

Previously, two of our models, cars and makes made calls to the NHTSA API with seperate NHTSA Servlets. Both calls were similar in nature so we have organized our code better to avoid style smells using techniques which deal with generalization. We have implemented ***extract superclass*** on NHTSACarServlet and NHTSAMakeServlet to create the Superclass NHTSAServlet. Both classes now extend the super class and inherit similar methods like run() and the getNHTSAAttribute. Since both NHTSA attributes needed for cars were similar we used ***pull up*** so all subclasses share the same call. Since the call for a make was different we used ***push members down*** to override the API calling function for makes. Implementing these refactorings allows for future extensions of code if we want to add a new type of NHTSA call. All we need to do is make a new subclass.

Original NHTSA API Calls

```
public String getBodyClass() {
        try {
                return (String)results.get("BodyClass");
        }
        catch(Exception e){
                return "NHTSA API Server Error";
        }
}

public String getHP() {
        try {
                return (String)results.get("EngineHP");
        }
        catch(Exception e){
                return "NHTSA API Server Error";
        }
}
```



Refactored NHTSA API Call and renamed classes for better code style

```
public String getNHTSAAttribute(String attribute) {
        try {
                return (String)results.get(attribute);
        }
        catch(Exception e){
                return "NHTSA API Server Error";
        }
}
```