Phase One

Phase One

Overview

In Phase One, you must test and implement Java classes that form the "model" of the application. In this case, the model consists of the classes that represent bus routes, bus stops, and arrival estimates. You will also be adding functionality to parse model data presented in JSON format - this is one of the two formats that TransLink uses to transmit data to clients (the other is XML, and is less easy to deal with). These classes will be used to represent the state of various components of the full *Buses Are Us* application which you will be implementing in Phase Two.

In this phase of the project, you will test the classes that you implement by running jUnit tests. This phase of the project does not use Android. We will provide you with a very limited number of tests to get you started. When you submit your work to AutoTest, additional tests will be run. You must therefore write jUnit tests of your own to check that you've implemented the specified behaviour. Note that the first thing a TA will ask to see when you request help to implement a method are the corresponding tests. Remember that by designing a test you are essentially providing examples of how a method will be used - this will help to solidify your understanding of that method's specification.

If you cannot show your TA or instructor appropriate tests for the method that you are trying to implement, you will get no help debugging that method's implementation! Note that the TAs will not help you to write code for this term project - their role is to help you interpret the specification of the classes provided and to help you debug the code that you have written.

Before you do anything, please be sure that you have read and understood the *Collaboration Policy* presented in the Project Overview. Failure to do so could lead to a grade of zero on this term project and therefore a failing grade in the course. In the past, students have failed to abide by our collaboration policy and have paid the price - don't let it happen to you!

If you believe that the specification of any of the methods that you have been asked to implement is not clear, please ask for clarification by posting a message on the edX discussion board. A message pinned to the top of the discussion board will provide updates as needed. **It is your responsibility to ensure that you check this message regularly and that your implementation conforms to the posted updates.**

You will have to wait 12 hours between successive requests for feedback from AutoTest. Every 12 hours that goes by that you do not submit an updated version of your code for grading is a lost opportunity. Start early!

Set up

Install the CheckStyle plugin

The IntelliJ CheckStyle plugin will help you check for style issues in your code. You can install it as follows:

- From the IntelliJ welcome screen, select *Configure* and then *Plugins*
- Click *Browse repositories...*
- Type *CheckStyle* in the search box and select *CheckStyle-IDEA* then click *Install*
- Click *Restart IntelliJ IDEA*

The CheckStyle plugin should now be installed and ready to use.

Phase One Tasks

Phase One - Task 1: Clone the repository containing the starter code

Your first task is to clone the repository named d7_teamxxx that contains the starter code. This project includes eight packages:

- `ca.ubc.cpsc210.translink.model` - classes that represent the "domain" of the application (e.g., Stop)
- `ca.ubc.cpsc210.translink.model.exception` - exception classes related to the model

- `ca.ubc.cpsc210.translink.providers` - classes that read data from source
- `ca.ubc.cpsc210.translink.parsers` - classes that parse TransLink data
- `ca.ubc.cpsc210.translink.parsers.exception` - exception classes related to parsing
- `ca.ubc.cpsc210.translink.tests.model` - JUnit tests for the model classes
- `ca.ubc.cpsc210.translink.tests.parsers` - JUnit tests for the parser classes
- `ca.ubc.cpsc210.translink.util` - utility classes

Note that, in the state provided to you, the project should compile. All of the included tests will fail.

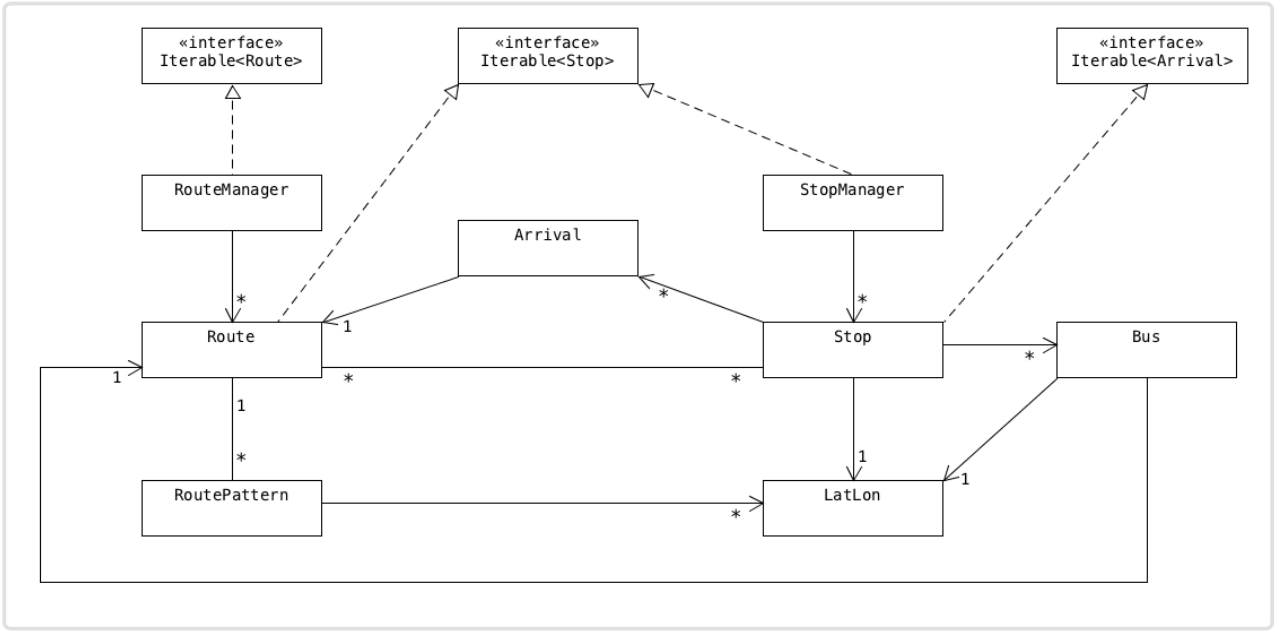
The code will also conform to the style guidelines in effect for this project. To run the CheckStyle tool, select *View -> ToolWindows -> CheckStyle*. A new panel will open (at the bottom of the IntelliJ, if you are using IntelliJ's default layout). In the *Rules* drop-down list box select *CPSC 210 Checkstyle*. Hover over the buttons on the left edge of the panel until you find the one that has *Check Project* as its pop-up tool tip. Click this button to run a check of all the code in your project (note that tests are not included in this check). Be sure to run this check prior to pushing your code to GitHub. If your code does not conform to the style guidelines, AutoTest will refuse to grade it.

Start by familiarizing yourself with the code that has been provided to you. In particular, note the classes in the `util` package which will make implementation of other classes easier for you, so be sure to use it!

Phase One - Task 2: Test and implement classes in the `model` package

Your second task is to test and implement classes in the `ca.ubc.cpsc210.translink.model` package using the design provided in the following UML class diagram and the specification provided with the methods in each class.

Note that if you correctly complete just this task (which includes designing tests to cover your code), you will earn a grade of at least 50% from AutoTest for this phase of the term project.



You must be careful to adhere to the given specification - do not modify the names of any of the classes/interfaces or modify the signature of any of the methods that have been provided, otherwise your code will not compile against the reference tests on AutoTest, resulting in a grade of 0 when you submit your work. Do not ignore the class diagram - it provides information about the design of the system that is not captured in the method specifications!

For each class in the `model` package:

- design jUnit tests to test methods based on the given specification
- add any necessary fields and implement the methods
- debug until all tests pass

It is recommended that you design your classes in the following order: `Bus`, `Arrival`, `RoutePattern`, `Route`, `Stop`, `StopManager`, `RouteManager`. Keep the following in mind:

- AutoTest will give you hints on failed tests for classes in the order in which they are specified above. So, if you see hints that are related to `RoutePattern`, you know that all the tests that we run on your `Bus` and `Arrival` classes have passed.
- Some classes have some of their fields provided. Be sure to use them without modification!
- `Arrival` implements the `Comparable` interface. This is done so that instances of this class can be sorted. In this particular case, we want to be able to sort arrivals so that the first bus to arrive at a stop is listed first. Note that the implementation of the method specified in this interface (`compareTo`) has been provided for you.
- `Stop`, `Route`, `RouteManager`, and `StopManager` implement the `Iterable` interface. This allows us to iterate over the corresponding collection using a for-each loop. For example, note that `Stop` has a collection of `Arrival` objects. Given

that `Stop` implements `Iterable<Arrival>`, we can iterate over that collection as follows:

```
for (Arrival next : myStop) {
    // do something with next
}
```

assuming that `myStop` is an instance of `Stop`. This is an application of a design pattern known as the iterator pattern - we'll talk more about it later in the course. An implementation of the method specified in this interface (`iterator`) has been provided for you - be careful not to modify it.

- `StopManager` and `RouteManager` are singletons. Singleton is another example of a design pattern. For now, all you need to know about a singleton is that it is designed in such a way that there is only ever a single instance of the class that is globally accessible throughout the code. Notice that the constructor is private but a static `getInstance()` method is provided that allows clients to access the one (and only) instance of the class. We have provided you with a skeleton class named `StopManagerTest` that you can use to design tests for the `StopManager` class and a skeleton class named `RouteManagerTest` that you can use to design tests for the `RouteManager` class. Pay particular attention to the `setUp` method - it retrieves the one and only instance of the corresponding `WhateverManager` and then clears the list of managed objects so that the manager is in the same state at the start of each test.
- The `Route`, `RoutePattern` and `Stop` classes must use a subset of their fields to test for equality and to determine appropriate hash codes for use with a collection, such as `HashSet`, as follows:
  - the equality of two `Route` instances must be based only on the route number
  - the equality of two `RoutePattern` instances must be based only on the route pattern name
  - the equality of two `Stop` instances must be based only on stop number

An explanation of how to use a Java collection is found in the *Best Practices for Using the Java Collections Framework* section of the Project 1 readings on edX.

Phase One - Task 3: Test and implement the `RouteMapParser` class

Bus Routes are identified by a number, and have a number of different patterns depending on the time of day, day of week, etc. Open the file `/data/allroutemaps.txt` and look at the format of the data. Each line describes one pattern of one bus route. Each line begins with a capital N, followed by the bus route number, a dash, the pattern name (which may contain a dash), a semicolon, and a series of 0 or more real numbers corresponding to the latitude and longitude (in that order) of a point in the pattern separated by semicolons. The "N" that marks the beginning of the line is not part of the bus route number. We use this data in the app to plot the bus routes on a map. The `RouteMapParser` has a single method that you are to implement: `parseOnePattern`. It takes a string of the format described above and extracts the route number, the route pattern name, and a list of `LatLon` objects - one for each of the points defined in the string. If the series of real numbers is empty, the corresponding list of `LatLon` points must be empty. It uses the `storeRouteMap()` method which we have provided for you to store the newly parsed `RoutePattern` in its corresponding `Route` object. Be sure to test this method carefully. You may assume that the data always has the format indicated above - your route pattern parser is not expected to deal with data that is not formatted correctly. This information does not contain the destination or direction of the `RoutePattern` object; these values will be replaced with real values when the routes are parsed (Task 4).

If you have taken CPSC 121, you may wish to approach this problem using regular expressions. If you take this approach, you will find the `Pattern` class in the Java library useful.

If you are not familiar with regular expressions, you can also come up with a good solution to the problem through judicious use of the `String indexOf(String str, int startIndex)` and `String substring(int startIndex, int endIndex)`, and `String[] split(String regex)` methods of the `String` class. The online documentation for the `split` method also refers to regular expressions but it provides some very simple examples that you will find helpful.

Phase One - Task 4: Implement parsers to parse Translink data from JSON format

Many applications need to access and consume data obtained from various sources. To make use of the data, applications need to know its structure. A common format for representing and exchanging data is JSON and this is the one used by the TransLink Open API.

A useful tool to examine the structure of JSON data is: http://jsonviewer.stack.hu/. You can paste sample JSON data into the tool and explore its structure. Use it to examine the structure of the sample JSON data provided with this project.

In general, there are different JSON parsers that can be used. We have linked one such parser to the project - you *must* use it! Documentation can be found here: http://www.docjar.com/docs/api/org/json/JSONObject.html and http://www.docjar.com/docs/api/org/json/JSONArray.html

It is recommended that you consult the JSON parser example provided on GitHub (`JsonParserExample`). It parses a simple JSON file containing library data. As you read through this example, be sure to refer to the documentation linked to above. Keep in mind that this is a very basic example. It does not cover all the features of JSON parsing that you will need to use in this project.

Note the use of helper methods in the sample parser. In particular, note that we are parsing a collection of books. We have one method for parsing the collection of books (`parseLibrary`) and another to parse a single book (`parseBook`). Note that you saw a similar decomposition in CPSC 110: when you designed a function that consumes `(listof X)`, you also designed a helper function (or used a built-in function) that consumes `X`. **Be sure to use helper methods when designing your parsers. By doing so, you will find your code easier to debug. Note that the TAs will not help you find bugs if you show up with a monolithic method that attempts to cover all the tasks involved in parsing the JSON data. You will first be asked to decompose your method into appropriate helpers. Also keep in mind that CheckStyle enforces a maximum method size of 20 lines (including the header and the line containing the closing brace).**

We provide you with sample JSON data for the buses in the Greater Vancouver area (`allroutes.json`), bus stops (`stops.json`), for arrivals of the 43 eastbound at a stop near the hospital around 3:00pm (`arrivals43.json`), bus location data for buses passing through a certain stop (`buslocations.json`) and all arrivals at a stop near the hospital around 3:00pm (`arrivals.json`) in the folder `/data`. TransLink provides documentation of its API on the web. The sample data for the bus routes was obtained by querying all the bus routes (using `http://api.translink.ca/rttiapi/v1/routes/351`) and aggregating the replies. The route 43 arrivals data was obtained using `http://api.translink.ca/rttiapi/v1/stops/51479/estimates?timeframe=120&routeNo=43`.

There are three more classes in the parsers package that you need to test and implement. We suggest you do so in the following order, as this is the order in which AutoTest will provide hints on failed tests:

1. `BusParser` - a parser for bus location information.

   The method `parseBuses` receives a `Stop` and a string that represents the JSON data encoding information about buses on routes that pass through that stop. The data in the file `/data/buslocations.json` is representative of such data - open

it to familiarize yourself with the structure of the data. To successfully parse this data and add the buses to the given stop you must:

- parse the route number, latitude, longitude, destination and recorded time of each bus, construct the corresponding bus object and add it to the stop.

Note that if a bus is operating on a route that does not service the given stop, it is silently ignored and not added to the stop.

2. `RouteParser` - a parser for route information

   The method `parseRoutes` receives a string that represents the JSON data encoding information about multiple bus routes. The data in the file `/data/allroutes.json` is representative of such data - you should have that file open and consult it as you read the following description. To successfully parse this data and produce the corresponding `Route` objects, you must:

   - parse the information about the bus `Route` and store it in the `RouteManager` - this includes the `RouteNo`, `Name`, and `Patterns` (see below) fields in the JSON data
   - parse the information about the `RoutePatterns` and add each of them them to the route
     - this includes the `PatternNo`, `Destination`, and `Direction` fields in the JSON data

3. `StopParser` - a parser for bus stop information

   The method `parseStops` receives a string that represents the JSON data encoding information about multiple bus stops. The data in the file `/data/stops.json` is representative of such data - you should have that file open and consult it as you read the following description. To successfully parse this data and produce the corresponding `Stop` objects, you must:

   - parse the information about each bus stop and store it in the `StopManager` - this includes the `Name`, `StopNo`, `Latitude`, `Longitude`, and `Routes` fields
   - add any routes mentioned in the stops file to the `RouteManager`
     - included in the information about each stop is a list of the bus route numbers that service this stop

4. `ArrivalsParser` - a parser for estimated arrivals at a particular stop

The method `parseArrivals` receives a string that represents the JSON data received in response to a query to get the arrivals at a stop. It also receives a stop to which the parsed arrivals must be added. The data in the files `/data/arrivals43.json` and `/data/arrivals.json` is representative of such a response - you should have the latter file open and consult it as you read the following description. To successfully parse this data and add the arrivals to the stop you must parse the response as an array. Each element in the array represents a bus route that has an arrival at the stop. Each bus route object has a `RouteNo:` field to identify the route, and a `Schedules:` field that contains a number of upcoming arrivals, stored in a JSON array. For each arrival, use the following fields to construct an `Arrival` object and add it to the corresponding `Stop`:

- `ExpectedCountdown`
- `Destination`
- `ScheduleStatus`

Use the `RouteNo` to lookup the route on which the arrival will occur. Note that the route must be specified when the arrival is created and added to the stop.

In general, when parsing data, you must be careful to avoid adding duplicate objects to the system. So, for example, at no point should you have two route objects in the RouteManager with the same number, nor should you have two stop objects in the StopManager with the same number. This means that, except perhaps for testing, you should be accessing all Route objects via the RouteManager and all Stop objects via the StopManager. Note that many bus routes (for which data has been provided) have stops in common (the 43, 41, and 480 all use stops along Wesbrook Mall, for example), so you can design a test to check that you have implemented this correctly.

When we test your code, we will check that your parsing is robust to the errors in the JSON data identified below, so be sure to include these cases in the tests that you write! To do so, you will need to create additional JSON data files. We suggest you copy the ones that we have provided (*do not edit the originals*) and edit the copies to produce the errors described below. You can also copy and edit the tests that we have provided. This part of the project is difficult. *Approximately* 15% of the Phase 1 grade will be applied to correctly handling the errors in JSON data described below. So it is still possible to get a good grade if you don't manage to implement this part of the project correctly.

The TAs have been instructed not to give *any* help testing, implementing or debugging code that handles the exceptional cases identified below. If you wish to earn the portion of the grade allocated to this section, you must do the work *entirely on your own*.

In the following discussion when it is stated that JSON data is "missing" an element, this means that the entire key/value pair is missing. Again, it is recommended that you approach errors in data in the order in which they are presented below, as this is the order in which AutoTest will provide hints when tests fail.

`BusParser.parseBuses`

- must *not* throw an exception if a bus is missing any of the data necessary to construct a bus object (route number, latitude, longitude, destination, recorded time) or if a bus is found that is operating on a route that does not pass through the stop. In such cases, the bus is simply ignored and the parser continues to parse remaining buses.
- if any other error occurs parsing the JSON data (for example, the data to be parsed does not have the correct format for a JSON object or array), a `JSONException` must be thrown

`RouteParser.parseRoutes`

- must not add route to the route manager and must throw a `RouteDataMissingException` when
  - any route is missing *any of*:
    - `RouteNo`
    - `Name`
    - `Patterns`
- if a pattern is missing any of PatternNo, Destination or Direction it must be silently ignored and not added to the route
- if any other error occurs parsing the JSON data (for example, the data to be parsed does not have the correct format for a JSON object or array), a `JSONException` must be thrown
- all routes for which the JSON file contains the required information must be added to the RouteManager, even if some other routes are missing information

`StopParser.parseStops`

- must throw a `StopDataMissingException` when
  - any stop is missing *any of*:
    - `StopNo`
    - `Name`
    - `Latitude`
    - `Longitude`
    - `Routes`
- if any other error occurs parsing the JSON data (for example, the data to be parsed does not have the correct format for a JSON object or array), a `JSONException` must be thrown
- all stops for which the JSON file contains all the required information must be added to the StopManager, even if some other stops are missing information.

`ArrivalsParser.parseArrivals`

- must throw a `ArrivalsDataMissingException` when the reply contains no arrivals with complete information, that is, all the arrivals obtained for a stop are missing *at least one of* the following:
  - `ExpectedCountdown`
  - `Destination`
  - `ScheduleStatus`
- an arrival that is missing any one of the fields listed above must not be added to the stop
- if a route is missing the `RouteNo` or `Schedules` field, it is silently ignored and no arrivals are added to that route
- if any other error occurs parsing the JSON data, a `JSONException` must be thrown

Phase One Deliverables

You must submit your code for automated testing and grading to AutoTest by the deadline specified on the project overview page. You will have to wait 12 hours between successive requests for feedback. On each submission you will receive at most 2 hints. Keep in mind that we will be running approximately 100 tests against your code, so don't rely on AutoTest to help you find all your bugs!

Your grade will be a combination of the scores earned for testing and for coverage of code found in the `model` and `parsers` packages (with the exception of the classes provided for you in the exceptions packages). Note that you are not required to achieve 100% code coverage to earn full marks for Phase 1 of the term project but you have to get close. We will scale your AutoTest grade out of 98% (to a maximum of 100%). So overall scores of 98% or higher will eventually convert to 100% when we compute your Phase 1 grade.

Before you submit your code to AutoTest for grading, ensure that:

- you have no unused import statements
- you have run checkstyle and resolved any issues

When you are ready, commit and push your code to GitHub. If AutoTest automatically responds to indicate that your code did not compile or failed the CheckStyle test **do not** request a grade - doing so will simply result in you getting the same feedback and you will then have to wait 12 hours before you can resubmit your request.

Assuming AutoTest does not automatically respond with an error (give it a few minutes to do so), you can then request a grade by making the following comment on your commit:

@autobot #d7

Remember that you must wait 12 hours between successive requests for feedback from AutoTest and that, if your grade is to count, you must submit the request to AutoTest prior to the deadline. *This means that any request you make within 12 hours of the deadline will be your last opportunity to have your work graded - so use this last attempt wisely.* As long as the request is submitted prior to the deadline, the grade will count, even if AutoTest doesn't respond until after the deadline.