Software Design Coursework
Deliverable D4: bus stop displays

1. Introduction

The coursework deliverable D4 is concerned with the bus stop displays for the `BusInMetropolis` project.

Bus stops are equipped with displays which show the next 10 expected buses at any time of the day. A bus stop display is illustrated in table 1; each displayed row corresponds to a scheduled bus journey of a route; each row is sub-divided into four columns: bus number (or route number), destination, expected time of arrival, and the status — either *on time*, *delayed* by a certain number of minutes, or cancelled.

Number	**Destination**	**Due at**	**Status**
15	Gotham	15:05	2 minutes delay
21	Smallville	15:07	3 minutes delay
6	New Troy Boulevard	15:08	cancelled
12	Planet Square	15:10	on time
17	Los Malos	15:13	5 minutes delay
25	Topaz Lane	15:15	on time
3	Centennial Park	15:18	on time
15	Gotham	15:20	on time
21	Smallville	15:22	on time
6	New Troy Boulevard	15:23	on time

Table 1: An illustration of a bus stop display.

The displays are updated regularly based on internal timetable information and the current time. An expected bus is removed from the display when: (i) the bus stop display is notified that a scheduled bus has departed from the bus stop, (ii) one minute after the due time for cancelled journeys, and (iii) three minutes after the expected time of arrival of a scheduled journey (scheduled time plus recorded delay) if no departure notification from the bus stop is received until then — it is assumed that the bus has departed and no notification was received. The display receives notifications from the central

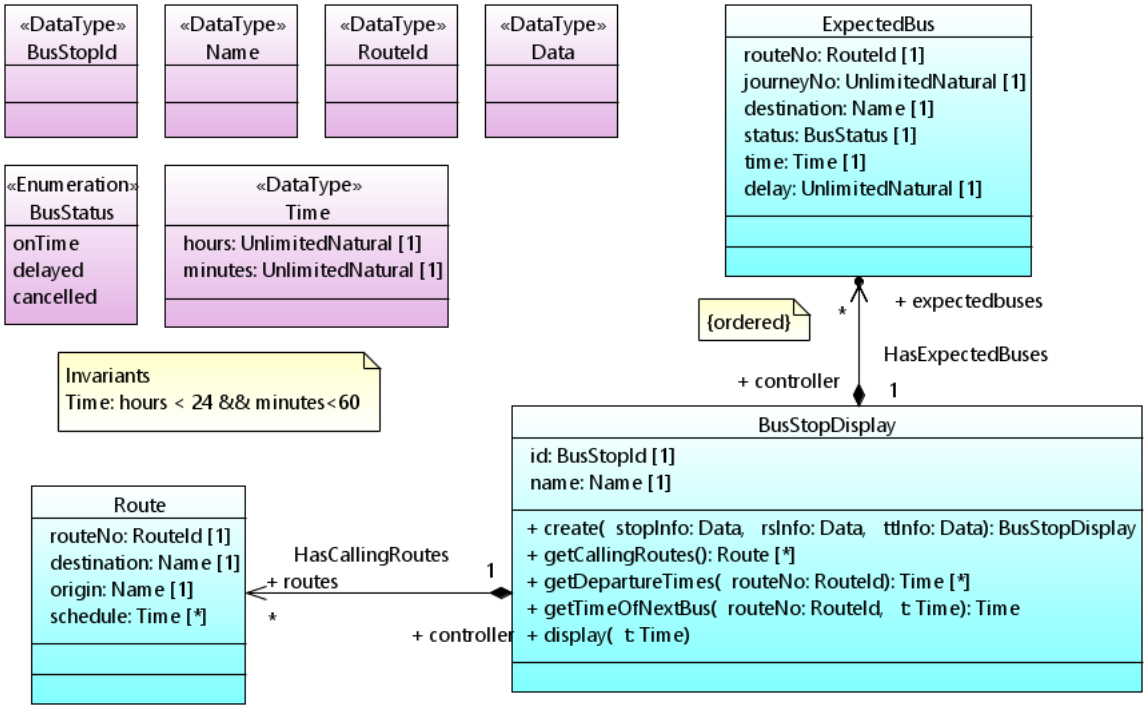

Figure 1: UML class diagram of `BusStopDisplay` system.

system on expected buses, concerning bus delays, cancellations and bus departures. The notifications scheme must adhere to the *observer* pattern [1].

2. The Given Design

Figure 1 presents the UML class diagram (CD) of `BusStopDisplay`; it is as follows:

- Datatypes `Name`, `Data`, `BusStopId` and `RouteId` represent names, raw data, and identifiers of bus stops and routes respectively.
- Enumeration `BusStatus` represents the status of a bus expected at the bus stop — either `onTime`, `delayed` or `cancelled`.
- Datatype `Time`, which represents time in terms of `hours` and *minutes*, is subject to an invariant: `hours` must be between 0 and 23 and *minutes* between 0 and 59.
- Class `BusStopDisplay`, the system façade or interface to the environment, holds information on: `name` and `id` of the bus stop, routes calling at the bus stop (association `HasCallingRoutes`) and an *ordered* collection of expected buses (association `HasExpectedBuses`).

Operation	Description	Contract
`create`	The façade constructor builds objects holding necessary information for the operations of the display from route and timetable files	Table 4
`getCallingRoutes`	Returns a collection of `Route`s (preferably unmodifiable) served by the bus stop	Table 6
`getDepartureTimes`	Gets the bus stop's timetable for the given route as a collection of times (preferably unmodifiable).	Table 7
`getTimeOfNextBus`	Gets time of next expected bus of a given route after a given time t as per bus stop timetable.	Table 8
`display`	Displays 10 next expected buses starting from a given time t (current time under normal circumstances)	Table 9

Table 2: System operation of system `BusStopDisplay`.

- Class `Route`, which represents the routes served by the bus stop, holds the route's number (`routeNo`), `destination`, `origin` and `scheduled` times of departure (as a list or sequence) with respect to the bus stop (the stop's timetable for the route).
- Class `ExpectedBus` represents buses expected at the bus stop as derived from the stop's timetable; this holds the route number of the expected bus (`routeNo`), the number of the bus journey with respect to the route (`journeyNo`) — bus journeys are numbered in ascending order from the earliest expected bus of a route to the latest (1 is first expected bus) —, the bus's `destination`, `status` (either `onTime`, `delayed` or `cancelled`) and number of minutes of `delay`.
- The system operations, identified in the façade class, are described in table 2, which refers to contracts from appendix A[1].

3. The Assessed Task

The design outlined in section 2 above would display all buses as being on time as there is no information on bus delays and cancellations coming from the central system (bus info). For example, suppose that an expected bus reaches the bus stop "Constantinople Boulevard" with a delay of 5 minutes, in the given design the bus stop display remains unaware of this as there is no incoming channel of information to enable notifications concerning bus journeys of interest.

The assessed task involves the following: (i) implementing the given design in the Java programming language, (ii) supplementing the design with notifications coming from the central system following the observer pattern [1], and (iii) implementing the augmented

[1]The contracts assume that internally the datatype `Data` (raw data) is a matrix made up of rows and columns; for a data matrix d, $d(i)(j)$ gives the piece of data at row i and column j starting from 0.

Name	Description
A UML class diagram	The given UML class diagram augmented with the bus notifications scheme following the observer pattern.
Contracts	To describe the operations added to the given design to support the notifications scheme.
UML communication diagrams	To describe the interactions involved in the setting up of notifications with the 'bus info' simulator (registration) and actual notifications coming from the 'bus info' simulator.
A Java implementation	The implementation of given design, bus notifications scheme, and simulation of central system with the purpose of testing the bus stop display. This should include a demo highlighting the interaction between the display and the simulator.
JUnit tests	The JUnit tests developed to test the implementation.

Table 3: Required artefacts of deliverable D4 on the bus stop display.

design supporting notifications, which involves building a bus info simulator whose sole purpose is to test the bus display.

Table 3 gives required artefacts for deliverable D4.

4. The Implementation

The configuration data of the 'Sweetspot' bus station, available on Moodle, is given as testing data. You are required to implement the bus stop display according to the augmented design, bearing in mind that the design extension to support the notifications should have a minimal impact with respect to the design of section 2 and, hence, avoid substantial changes to the given classes and operations. The implementation needs to simulate the bus tracking functionality of the central system with the sole aim of feeding the bus display.

The implementation involves the following:

1. Construction of the class structure in Java according to the UML CD of Fig. 1.

2. Loading and parsing of the information contained in the configuration files as part of the system operation `create` (contract in table 4) representing the constructor of `BusStopDisplay`. This involves creating routes and their timetables and involves parsing the given configuration files[2]. It is suggested that you create classes `RoutesAndStopInfoParser` and `TimetableParser` to parse the data contained in the configuration files.

3. Creation of the list of expected buses maintained by the bus stop display (association `HasExpectedBuses` in the CD of Fig. 1) as part of the effort on the system

[2]This step corresponds to what was done in week 9 of the module *object-oriented programming* (OOP).

operation `create` (table 4) and the operation `addScheduledToExpected` (table 5).

4. Implementation of system operations `getCallingRoutes`, `getDepartureTimes` and `getTimeOfNextBus` (contracts of tables 7 to 8).

5. Implementation of system operation `display` (contract of table 9).

6. Design and implementation of a notifications scheme following the observer design pattern [1], which involves building a simulator of 'bus info'.

7. Implementation of a demo highlighting an interaction between the bus stop display and the 'bus info' simulator.

The 'BusStopDisplay' demo should comprise of an infinite loop that interleaves between displaying and simulating with refreshes every 15 seconds:

```
while (true) {
   d.display(Time.now());
   bi.simulate();
   TimeUnit.SECONDS.sleep(15);
}
```

5. The Submission

The submission should include:

- Java implementation of bus stop display according to augmented design. This should include code of bus display, simulator of central system, a demo showing the interaction between bus stop display and simulator, and JUnit tests.

- Report with augmented UML-based design, including the augmented UML class diagram, the new contracts and the communication diagrams that describe the operations involved in the notification scheme, and an explanation of the implementation.

The report should cover all artefacts of table 3 and include three main sections: augmented design (with sub-sections for UML CD and UML communication diagrams), implementation and testing. The design section should explain how the given design of section 2 was augmented to enable support for notifications of bus journeys coming from the central system by applying the observer design pattern. The implementation section should explain how the implementation is related to the design and provide a table describing the mapping from design types (datatypes, enumerations and classes) into implementation types. The testing section should include a table for the JUnit tests with a column for the name of the test (name of the method in the testing class), the objective of the test, and whether the test passed or not.

The submission must consists of a zip file containing the following:

i. The report in word or pdf format.

ii. The zip with the Java code, which can be obtained using the export functionality of Eclipse.

iii. The zip of the Papyrus model (produced using the export functionality of Eclipse).

A. Contracts of System operations

Operation: `create (stopInfo, routesInfo, ttInfo) : BusStopDisplay`
Pre-condition:

Post-condition:
• rs a collection of `Route` objects is created
• For each row i in $routesInfo$ (or lines in routes file), a `Route` object r is created:
  • $r.routeNo = routesInfo(i)(0)$
  • $r.destination = routesInfo(i)(1)$
  • $r.origin = routesInfo(i)(2)$
  • $r.schedule$ = route times obtained from ttInfo
  • r is added to rs
• ebs a list (or sequence) of `ExpectedBus` objects is created and then filled through operation `addScheduledToExpected` (see contract in table 5)
• d, a new `BusStopDisplay` object is created:
  • $d.id = stopInfo(0)(0)$
  • $d.name = stopInfo(0)(1)$
• Object d is linked to collection rs via association `HasCallingRoutes`
• Object d is linked to collection ebs via association `HasExpectedBuses`
• Object d is returned

Table 4: Contract of system operation `create`.

Operation:
`addScheduledToExpected ()`

Pre-condition:

Post-condition:
• Let d be the `BusStopDisplay` instance
• Let ebs' be a list (or sequence) of `ExpectedBus` objects
• Let rs be collection of `Route` objects linked to d ($d.routes$) through association `HasCallingRoutes`
• For each `Route` object r in rs, and for each `Time` t in $r.scheduled$:
  • eb, a new `ExpectedBus` object is created:
    • $eb.routeNo = r.routeNo$
    • $eb.journeyNo = i+1$, i is position of `Time` t in list (or sequence) $r.scheduled$
    • $eb.destination = r.destination$
    • $eb.status = onTime$
    • $eb.time = t$
    • $eb.delay = 0$
    • eb is added to list ebs'
• List ebs' is ordered such that for any two position i, j in the list we have that $i < j \implies ebs(i).time \leq ebs(j).time$ (`ExpectedBus` objects in ebs are in ascending order with respect to their departure times)
• Let ebs be a list (or sequence) of `ExpectedBus` objects associated with d ($d.expectedBuses$) through association `HasExpectedBuses`.
• All objects of `ExpectedBus` eb in ebs' are added to list ebs by preserving the order of ebs'

Table 5: Contract of operation `addScheduledToExpected`.

Operation:
`getCallingRoutes () : Route [*]`

Pre-condition:

Post-condition:
• Let rs be collection of `Route` objects associated with the `BusStopDisplay` object through association `HasCallingRoutes`
• An unmodifiable copy of object rs is returned

Table 6: Contract of system operation `getCallingRoutes`.

Operation:
`getDepartureTimes (routeNo) : Time [*]`
Pre-condition:
•There exists a `Route` object r such that $r.routeNo = routeNo$
Post-condition:
• Let r be the `Route` object r such that $r.routeNo = routeNo$:
• An unmodifiable copy of collection $r.schedule$ is returned

Table 7: Contract of system operation `getDepartureTimes`.

Operation:
`getTimeOfNextBus (routeNo, t : Time) : Time`
Pre-condition:
•There exists a `Route` object r such that $r.routeNo = routeNo$
Post-condition:
• Let r be a `Route` object r such that $r.routeNo = routeNo$
• Let t' be a `Time` such that $t' > t$ and $t' \in r.schedule$ and for any other t'' such that $t'' \in r.schedule$, $t' < t''$ (t' is earliest possible time after t from $r.schedule$)
• t' is returned

Table 8: Contract of system operation `getTimeOfNextBus`.

Operation:
`display (t : Time)`

Pre-condition:

Post-condition:
• Let *ebs* be list of `ExpectedBus` objects of *BusStopDisplay* *d*, linked to *d* (*d.expectedBuses*) through association `HasExpectedBuses`
• For each `ExpectedBus` *eb* in *ebs*:
  • if $eb.status = cancelled$ and $eb.time < t$, then *eb* is removed from *ebs*
  • if $eb.time +_{min} eb.delay +_{min} 3 < t$, then *eb* is removed from *ebs* (expected departure added with known delay and 3 minutes tolerance is less then current time, hence, it is assumed that bus has passed)
• If $ebs.size() < 10$ (size of *ebs* list is smaller than 10) then the list is refreshed with the bus stop's scheduled departures according to operation `addScheduledTo-Expected` (see contract in table 5)
• Consider the display *di* to be a matrix made up of rows (display lines) and columns. Each row of *di* is filled from the first 10 `ExpectedBus` objects in list *ebs* (see table 1); for each i, such that $0 \leq i < 10$, and *eb* obtained from position i in *ebs* ($= ebs(i)$), we have that:
  • $di(i)(0) = i + 1$
  • $di(i)(1) = eb.routeNo$
  • $di(i)(2) = eb.destination$
  • $di(i)(3) = eb.time$
  • $di(i)(4) = eb.status$ if $eb.status = delayed$ or $eb.status = cancelled$; otherwise $di(i)(4) = eb.delay$ added with string "minutes delay"

Table 9: Contract of system operation `display`.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.