RESOURCES

CSC216: Project 2

Project 2, Part 2: Wolf Travel Tours

Home > Projects > Project 2 > Project 2, Part 2: Wolf Travel Tours

Project 2, Part 1: Wolf Travel Tours ▶

Project 2, Part 2: Wolf Travel Tours

Version with no table of contents (for printing).

Part 1 of this assignment laid out the requirements for the Wolf Travel application that you must create.

Deliverables and Deadlines

Part 2 details the design that you must implement. For this part, you must create a complete Java program consisting at multiple source code files and JUnit test cases. You must electronically submit your files to GitHub by the due date.

Process Points 1 Deadline: Friday, March 29 at 11:45PM

Process Points 2 Deadline: Friday, April 5 at 11:45PM

Part 2 Due Date: Friday, April 12 at 11:45PM

Late Deadline: Sunday, April 14 at 11:45PM

Set Up Your Work Environment

Before you think about any code, prepare your work environment to be compatible with the teaching staff grading criteria. The teaching staff relies on NCSU GitHub and Jenkins for grading your code. Follow these steps:

1. Create an Eclipse project named Project2: Right click in the package explorer, select *New -> Java Project*. Name the project Project2. (The project naming is case sensitive. Be sure that the project name begins with a capital P.)
2. If you have not already done so, clone your remote NCSU GitHub repository corresponding to this project. Note that each major project will have its own repository.

Working with a Partner

If you are working with a partner do the following to make sure that you both are working on the same project and to avoid collisions.

1. Identify who will be partner A and who will be partner B.
2. Partner A should create the Eclipse project as described above. Then Partner A will clone the team repo and share Project 2 to the local copy of the cloned repo. Then Partner A should commit/push the shared project to NCSU's GitHub. Verify that the project is there by navigating to the repo on the GitHub website. The project should be listed as Project2 and should contain the src/ folder, a .classpath, and .project files (at a minimum).
3. Partner B should clone the repo AFTER Partner A has pushed the project. Then Partner B will import the existing project into their Eclipse workspace through the GitHub repositories view.
4. Communicate frequently to avoid merge conflicts. If you have a merge conflict see the CSC Git Guide for help.

Note: Assigned Partner

If you were assigned a partner for Project 2 Part 2, you may work only with your assigned partner. Otherwise, you are expected to complete the project individually. All work must be strictly your own or you and your partner's own work.

Requirements

The requirements for this project are describe in Project 2 Part 1 in 13 different use cases:

UC1: Startup
UC2: Load a Wolf Travel data file
UC3: Save data to a WolfTravel data file
UC4: Open a new WolfTravel data file
UC5: Filter the tour display
UC6: Add a new client
UC7: Add a new tour
UC8: View all reservations for a particular tour
UC9: View all reservations for a particular client

Design

Your program must implement our design, which we describe here. The design consists of 16 non-nested classes and 4 interfaces. We are providing the interfaces as well as the GUI classes.

Reminder about Provided Interfaces

The project you push to NCSU GitHub must contain *unmodified* versions of the interfaces and GUI files that we provide for you.

Code that is provided for you

The following code is provided to you. You may not change it in any way.

1. SimpleList<E> interface
2. SortedList<E extends Comparable<E>> interface
3. SimpleListIterator<E extends Comparable<E>> interface
4. TravelManager interface
5. TravelGUI class
6. SpringUtilities class

The package structure of the design reflects the MVC Design Pattern. The user interface (the view) resides in the package edu.ncsu.csc216.travel.ui and the model resides in the sub-packages of edu.ncsu.csc216.travel.model. The package edu.ncsu.csc216.travel.list_utils contains utilities that could be used in many contexts.

edu.ncsu.csc216.travel.list_utils

SimpleList<E>. A modification of the java.util.List interface that is customized for the Wolf Travel application. **Do not change this code.**

SortedList<E extends Comparable<E>>. Another modification of the java.util.List interface that is customized for the Wolf Travel application. **Do not change this code.**

SimpleListIterator<E extends Comparable<E>>. A highly restricted modification of the java.util.ListIterator interface that is customized for the Wolf Travel application. **Do not change this code.**

SimpleArrayList. Implements the SimpleList interface with an array data structure.

SortedLinkedListWithIterator<E extends Comparable<E>>. Implements the SortedList interface with a data structure of linked Nodes.

Node<E>. A private, static inner class of SortedLinkedListWithIterator that contains an E and a reference to the next Node in the list.
Cursor. A private inner class of SortedLinkedListWithIterator that provides a cursor for iterating forward through the list without changing the list.

edu.ncsu.csc216.ui

TravelGUI. The graphical user interface for the project, which is given to you. **Do not change this code.**
SpringUtilities. A class developed by Oracle that we customized slightly and used by the GUI for laying out components on a grid. This code is given to you. **Do not change it.**

edu.ncsu.csc216.travel.model.participant

Client. Represents a client of Wolf Travel. Every client maintains a list of reservations for that client.

edu.ncsu.csc216.travel.model.vacation

Tour. An abstract class representing a Wolf Travel tour. Every tour maintains a list of reservations for that tour.
EducationalTrip. A concrete class extending Tour representing an educational type tour.
RiverCruise. A concrete class extending Tour representing a river cruise type tour.
LandTour. A concrete class extending Tour representing a land tour.
CapacityException. Exception thrown when there is an attempt to create a reservation that would fill a tour over its capacity.
Reservation. Represents a client's reservation for a tour.

Note: The class Reservation ties together tours and clients. The reason we placed it into the package edu.ncsu.csc216.travel.model.vacation was to recognize the fact that the tour must have sufficient capacity before the reservation can actually be created.

edu.ncsu.csc216.travel.model.office

TravelManager. Declares the methods used to manage tours, clients, and reservations. **Do not change this code.**
TourCoordinator. Implements TravelManager and represents the person coordinating tours, clients, and reservations. The TourCoordinator maintains a sorted list of tours and a list of clients.
DuplicateTourException. Exception thrown when there is an attempt to add a duplicate tour to the list of tours.
DuplicateClientException. Exception thrown when there is an attempt to add a duplicate client to the list of clients.

edu.ncsu.csc216.travel.model.file_io

TravelReader. A class for reading Wolf Travel data files.

TravelWriter. A class for writing Wolf Travel data files.

UML Diagram

A UML class diagram for the Wolf Travel application is shown in Figure 1 below. This is the *minimum* set of state and behavior required to implement the project. You MUST have the public methods exactly as listed below for the teaching staff tests to run.

Note: This UML diagram does not show the private methods of the teaching staff solution. It also does not show any members of TravelGUI, or SpringUtilities or methods declared in any of the interfaces, which are given to you. You are strongly encouraged to create private methods to help implement some of the behaviors of the classes. You may also create additional private fields.

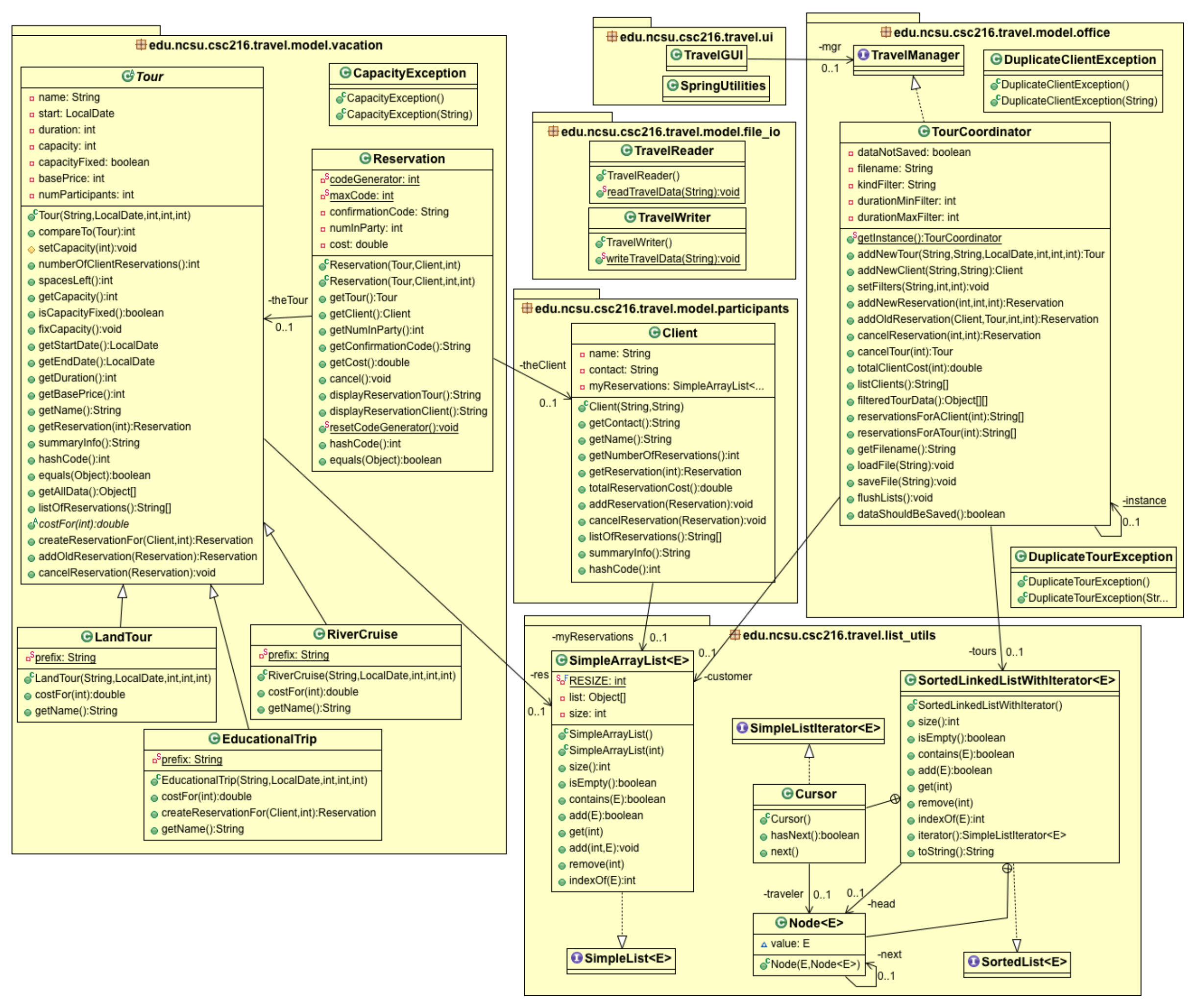


Figure 1: Wolf Travel Application Class Diagram

UML diagram

UML Diagram Notations

UML uses standard conventions for display of data, methods, and relationships. Many are illustrated in the UML diagram above. Here are some things you should note:

- or a red square in front of a class member means private.
+ or a green circle in front of a class member means public.
# or a yellow triangle in front of a class member means protected.
Static members (data or methods) are underlined.
Methods that are declared but not defined (abstract methods or those declared in interfaces) are in italics, unless they are in an interface.
The names of abstract classes are in italics.
Dotted arrows with triangular heads point to interfaces from the classes that implement them.
Solid arrows with triangular heads go from concrete classes to their parents.
Solid arrows with simple heads indicate *has-a* relationships (composition). The containing class is at the tail of the arrow and the class that is contained is at the head. The arrow is decorated with the name and access of the member in the containing class. The arrow is also decorated with the "multiplicity" of the relationship, where 0..1 means there is 1 instance of the member in the containing class and 0..* means there are many (usually indicating a collection such as an array).
A circle containing an "X" or cross sits on a the border of a class that contains an inner (nested) class, with a line connecting it to the inner class. (See the class Cursor and its corresponding outer class SortedLinkedList for example.)

Our UML diagram has some additional graphic notation:

A square (empty or solid) in front of a name means private. (See Client,name.)
A green circle in front of a name means public. (See Reservation.getTour().)
A triangle in front of a name means package-private (See Node.value.)
SF in front of a name means static, final. (See SimpleArrayList.RESIZE.)
Methods embellished with C are constructors. (See Client.Client().)

You can read this UML class diagram and see the names and types of all the class members. The method signatures in your program ***must*** match the above diagrams and the provided interfaces ***exactly*** for the teaching staff tests to run.

Access Modifiers

Note: You can modify the names of private variables, parameters, and methods. However, you MUST have the non-private data and methods (names, return types, parameter types, and order) exactly as shown for the teaching staff tests to run. YOU MAY NOT ADD ANY ADDITIONAL PUBLIC OR PROTECTED CLASSES, METHODS, OR FIELDS.

Implementation Process

Professional software development is more than building working software. The process of building software is important in supporting teams of developers to build complex systems efficiently. We expect you to follow good software engineering practices and processes as you create your software to help you become a better software developer and to help with understanding your progress on the project.

Project-wide Process Points

There are certain practices that you should follow throughout project development.

Commit with meaningful messages.

Meaningful Commit Messages

The quality of your commit messages for the entire project history will be evaluated for meaning and professionalism ***as part of the process points***.

Process Points 1 Milestone

Process Points 1 Milestone gets you started with the project and ensures that you meet the design and that you start commenting your code.

Compile a skeleton. The class diagram provides a full skeleton of what the implemented Wolf Travel program should look like. Start by creating an Eclipse project named **Project2.** Copy in provided code and create the skeletons of the classes you will implement, **including the test classes**. Ensure that the packages, class names, and method signatures match the class diagram *exactly!* If a method has a return type, put in a place holder (for example, `return 0;` or `return null;`) so that your code will compile. Push to GitHub and make sure that your Jenkins job shows a yellow ball. A yellow ball on Jenkins means that 1) your code compiles, 2) that the teaching staff tests compile against your code, which means that you have the correct code skeleton for the design, and 3) you have a skeleton of your test code in the `test\` folder..

Comment your code. Javadoc your classes and methods. When writing Javadoc for methods, think about the inputs, outputs, preconditions, and post conditions.

Fully Commented Classes

Fully commented classes and methods on at least a compiling skeleton program are due ***at least two weeks before*** the final project deadline to earn the associated process points.

Run your CheckStyle and PMD tools locally and make sure that all your Javadoc is correct. Make sure the tools are configured correctly (see configuration instructions) and set up the tools to run with each build so you are alerted to style notifications shortly after creating them.
Practice test-driven development. Start by writing your test cases. This will help you think about how a client would use your class (and will help you see how that class might be used by other parts of your program). Then write the code to pass your tests.

Process Points 2 Milestone

The Project 2 build process is set up differently than the builds on earlier projects and labs. Instead of requiring that you have unit tested all of your classes with high quality tests that executed 80% of statements (i.e., 80% statement coverage) before revealing teaching staff tests, the new build will provide teaching staff test feedback in a scaffolded, incremental manner. This means that you can focus on one class at a time and as you complete each class more information will be revealed.

The build process will consider the classes in the design in the following order:

SimpleArrayList
Node
Cursor
SortedLinkedListWithIterator
CapacityException
Tour
LandTour
RiverCruise
EducationalTrip
Client
Reservation
TravelWriter
DuplicateClientException
DuplicateTourException
TourCoordinator
TravelReader

The classes are considered in order and PMD notifications, insufficient coverage, and teaching staff test failures will remove teaching staff test feedback on the classes below it on the list. This will help you focus your work on the more foundational classes and increase confidence that the classes work before they are used by other classes.

If a class like Tour has a JUnit related PMD notification or insufficient coverage, teaching staff test feedback for Tour (and all classes below it on the list) will not be provided. If a class like Tour has a teaching staff test failure, the teaching staff failures for Tour are revealed, but teaching staff test feedback for all classes below it on the list will not be provided.

To earn full credit for Process Points 2, you should have 80% statement coverage from high quality unit tests, no PMD JUnit notifications, and be passing all of your unit tests and the teaching staff unit tests for roughly half of the project. That means you should be passing all unit tests in the following 4 teaching staff test files: TS_SimpleArrayListTest, TS_SortedLinkedListWithIterator, TS_TourTest (which tests the entire Tour hierarchy), and TS_ClientTest.

You should be passing the rest of the teaching staff tests by the final deadline which include the following 4 additional teaching staff test files: TS_ReservationTest, TS_TravelWriterTest, TS_TourCoordinatorTest, and TS_TravelReaderTest.

If any of the teaching staff test feedback is not revealed to you, your build's console output will contain the following string NOT ALL TS TESTS HAVE RUN. You will see the number of teaching staff tests that have run and are displayed a few lines higher up in the console output. While the build *should* be configured to only go to green ball when all teaching staff tests are revealed and you have sufficient coverage and no static analysis notifications, the system is in beta.

It is your responsibility to ensure that all EIGHT teaching staff test files are reporting results and that you're passing the tests to ensure completion of the project. You can check see if the teaching staff tests have run by exploring the Test Results links in on the build and checking that all eight teaching staff test files are listed.

If you run into any problems with the build that you think are scaffolding related, please email Dr. Heckman directly (sarah_heckman@ncsu.edu) or @ her in a Piazza post so she will receive the notification about the issue. Dr. Heckman will work to resolve issues within 24 hours of notification, but an issue may not be resolved after 4pm on the Process Points 2 deadline until the following day. Please plan accordingly and start early!

Implementation

Just as for Project 1, we suggest you attack Project 2 Part 2 one part at a time, starting with the collection classes that provide a basis for the rest of the project. Practice test-driven development. The rest of this section contains implementation details that are not revealed directly in the UML class diagram.

edu.ncsu.csc216.travel.util

The package edu.ncsu.csc216.travel.util provides all the list utilities required for the model. Many of the operations are identical to the ones you had to implement for labs. Classes in this package are totally independent of the rest of classes in the solution. That makes it especially attractive to define and test them first.

SimpleArrayList<E> details

SimpleArrayList<E> has a field which is an array of Objects, a size to indicate how many array elements belong to the list, and a RESIZE constant with the value 12. The initial length of the array is 12. The length of the array should be increased by 12 whenever needed to accommodate new list items.

The SimpleList<E> interface contains 8 methods that must be implemented by SimpleArrayList<E>. The Javadoc of the SimpleList interface describes what each method should do, including when exceptions must be thrown. Note: The provided SimpleList interface is a *custom* version of the java.util.List interface specifically for this project. Details of this custom list interface vary from the Java API’s List interface.

SortedLinkedListWithIterator<E extends Comparable<E>> details

SortedList<E extends Comparable<E>> is an interface containing 8 methods that must be implemented by SortedLinkedListWithIterator<E extends Comparable<E>>. SortedLinkedListWithIterator should be implemented as a singly linked list, with an ordinary nested class Node (which you can declare as static or non-static).

The Javadoc of the SortedList interface describes what each method should do, including when exceptions must be thrown. Note: The provided SortedList interface is a *custom* version of the java.util.List interface specifically for this project. Details of this custom sorted list interface vary from the Java API’s List interface.

Iterators are objects that can travel through lists. SortedLinkedListWithIterator has an additional nested class, Cursor, which is for traveling through the list, one item at a time. It implements SimpleListIterator<E extends Comparable<E>>, which requires only two simple behaviors: 1) tell what the current item is then travel down the list to the next item, and 2) tell you whether there is a next item. Such a simple iterator cannot travel backwards, it cannot change, add, or remove list items.

Two SortedLinkedListWithIterator methods are not declared in SortedList:

SortedLinkedListWithIterator.iterator(). Returns a SimpleListIterator. Since the inner class Cursor implements SimpleListIterator, this method should simply return an instance of Cursor.

SortedLinkedListWithIterator.toString(). Returns a string representation of the list in the format [A,B,...,X] where A is the first list item, B is the second, ..., and X is the last. For example, if the list contains “Apple”, “Betty”, “Claude”, and “Matthew,” then the string representation would be “[Apple, Betty, Claude, Matthew]”, with a space after each comma that separates one element from the next.

Custom Iterators

> ***Note:*** A *simple iterator,* which represents a cursor into a list, provides only two behaviors: (1) a method to determine if there are additional items in the list for the iterator to visit; and (2) a method that returns the next element to be visited and pushes the cursor beyond that element. These are far fewer behaviors than java.util.Iterator.

edu.ncsu.csc216.travel.model.participants and **edu.ncsu.csc216.travel.modelvacation**

Client details

Client represents the Wolf Travel customers. Details of its less obvious methods are listed below. If any actual parameters to a Client method are illegal (such as names not starting with alphabetic characters or indexes out of bounds), the method should throw an IllegalArgumentException.

Client.myReservations. A private data member of type SimpleArrayList<Reservation>.
Client.Client(). The first parameter corresponds to the client's name and the second corresponds to the client's contact.
Client.summaryInfo(). Returns a string that contains the client name followed by contact inside parentheses. The lines shown in Figure 5 of [UC2,S5] provides examples of appropriate formatting of the return value. The space between the last character of the name and the opening parenthesis is optional.
Client.listOfReservations(). Returns an array of strings representing the client's reservations. The lines shown in Figure 14 of [UC9,S2] illustrates appropriate formatting for the strings in the array. The return value should not be null, even if the client has no reservations.
Client.getReservation(). Returns the client's reservation at the given position, where position numbering starts at 0.
Client.addReservation(). Adds a reservation to the client's list. Throws an IllegalArgumentException if the client for the reservation is not this client. Note: This method should not attempt to add the reservation to the corresponding tour.

Tour class hierarchy details

Tour is an abstract class, with children RiverCruise, LandTour, and EducationalTrip. Details on that hierarchy are listed below. If any actual parameters to a Tour method or a method of one of its child classes are illegal, the method should throw an IllegalArgumentException or a CapacityException.

Tour.res. A private data member of type SimpleArrayList<Reservation>.
Tour.Tour(). The first parameter corresponds to the name, the second to the start date, the third to the duration, the fourth to the base price, and the fifth to the capacity. The constructor should throw an IllegalArgumentException if any parameters are not valid (including the parameter corresponding to capacity).

Tour.getName(). Overridden in each child class to return *prefix-name,* where *prefix* varies according to class type: RC for RiverCruise, ED for EducationalTrip, LT for LandTour. See Figures 11 and 12 in [UC7].

Tour.capacityFixed, isCapacityFixed() and Tour.fixCapacity(). Required for the capacity to change. Tour.capacityFixed should be initialized to false for an EducationalTrip, since it can double its capacity once. See P2P1 Problem Overview and [UC7, S7].

Tour.addOldReservation() and Tour.createReservation(). Should throw a CapacityException if the tour cannot accommodate the number of people in the reservation party ([UC10, E1]) and an IllegalArgumentException if any parameters are illegal. Both of these methods should ask the corresponding clients to add the reservation to their lists of reservations. EducationalTrip.createReservation() should override the parent's method in order to attempt expanding the capacity when needed. Such is not needed for Tour.addOldReservation() under the assumption that any capacity expansion would have already taken place when the reservation was initially created.

Tour.costFor(). Child classes should define this method, which is abstract in the parent class. See [UC10, S3].

`Tour.spacesLeft(). Returns the number spaces on this tour not yet reserved. If the tour has no reservations, this should be the capacity of the tour.

Tour.compareTo() and Tour.equals(). Should be consistent with [UC2, S4] and [UC7, E2].

Tour.listOfReservations(). Returns an array of strings representing the client's reservations. The lines shown in Figure 13 of [UC8,S2] illustrates appropriate formatting for the strings in the array. The return value should not be null, even if the tour has no reservations.

Tour and its children (RiverCruise, LandTour, and EducationalTrip) constitute part of the Factory Design Pattern. There are more details on this follow in the *TourCoordinator details* section below.

Reservation details

A Reservation connects a Client to a Tour. Details for the less obvious Reservation members are described below. If any parameters to a Reservation method are illegal, the method should throw an IllegalArgumentException.

Reservation.codeGenerator determines the confirmation code of the next reservation. See [UC10, S2].

Reservation.Reservation(). Creates a "temporary" reservation with the parameter information (client, tour, number in party, then optional confirmation code). It is the responsibility of the Tour to determine whether it has the capacity to accommodate the reservation.

Reservation.displayReservationTour() and Reservation.displayReservationClient() should return strings that are consistent with the displays in Figure 13 of [UC8,S2] and Figure 14 of [UC9,S2] respectively. -Reservation.resetCodeGenerator(). Sets codeGenerator to 0. Used for testing only.

edu.ncsu.csc216.travel.model.file_io

TravelWriter.writeTravelData(String filename) writes the data currently in the TourCoordinator to the given file. The file is created as described in UC3. The method throws an IllegalArgumentException if any error occurs.

TravelReader.readTravelData(String filename) populates the TourCoordinator with data read from the given file. The file is processed as described in [UC2]. This method throws an IllegalArgumentException on any kind of read error or inconsistency detected.

edu.ncsu.csc216.travel.model.office

TourCoordinator details

TourCoordinator provides the glue to coordinate the data for the backend model classes. Details that are not explained in TravelManager, which TourCoordinator implements, include the following.

> TourCoordinator.customer. A private data member of type SimpleArrayList<Client>.
> TourCoordinator.tours. A private data member of type SortedLinkedListWithIterator<Tour>.
> TourCoordinator.dataNotSaved. Should be set to true whenever the TourCoordinator data has not been saved to a file and false otherwise.
> TourCoordinator.TourCoordinator(). A private constructor, used to implement the Singleton Pattern.
> TourCoordinator.flushLists(). Clears all client and tour data from the customer and tours lists, sets TourCoordinator.dataNotSaved to false, and notifies observers (see *The Observer Pattern* below). Note that the other method that sets TourCoordinator.dataNotSaved to false is TourCoordinator.saveToFile().

TourCoordinator is part of three major design patterns:

1. Singleton Pattern (also used in Project 1)
   The pattern is implemented through a private constructor and the method TourCoordinator.getInstance(), which returns the only instance of TourCoordinator.
2. Observer Pattern.
   The pattern is implemented through the Observable class, which TourCoordinator extends, and the Observer interface, which TravelGUI implements. Details on the Observer Pattern follow below.
3. Factory Pattern.
   The pattern is implemented through the method TourCoordinator.addNewTour() and the Tour class hierarchy. The method *creates* a concrete Tour according to the method's parameters. Both TravelGUI and TravelReader call this method for Tour creation. Figure 2 illustrates how the Factory design pattern applies to this project.

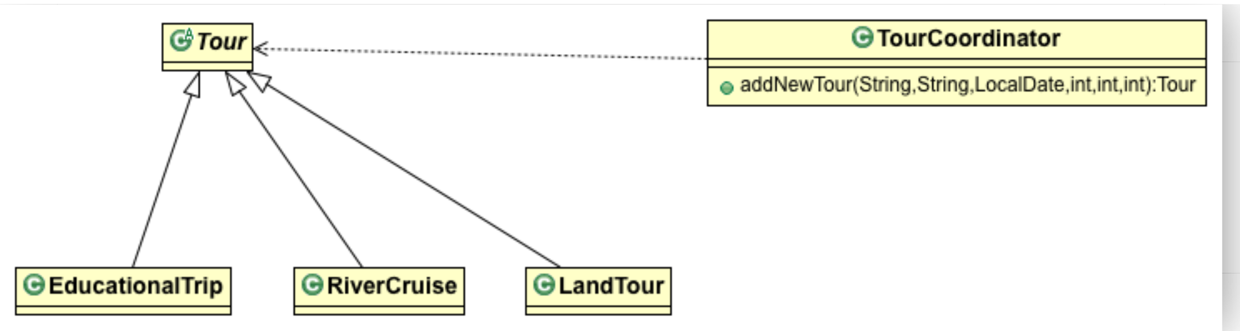
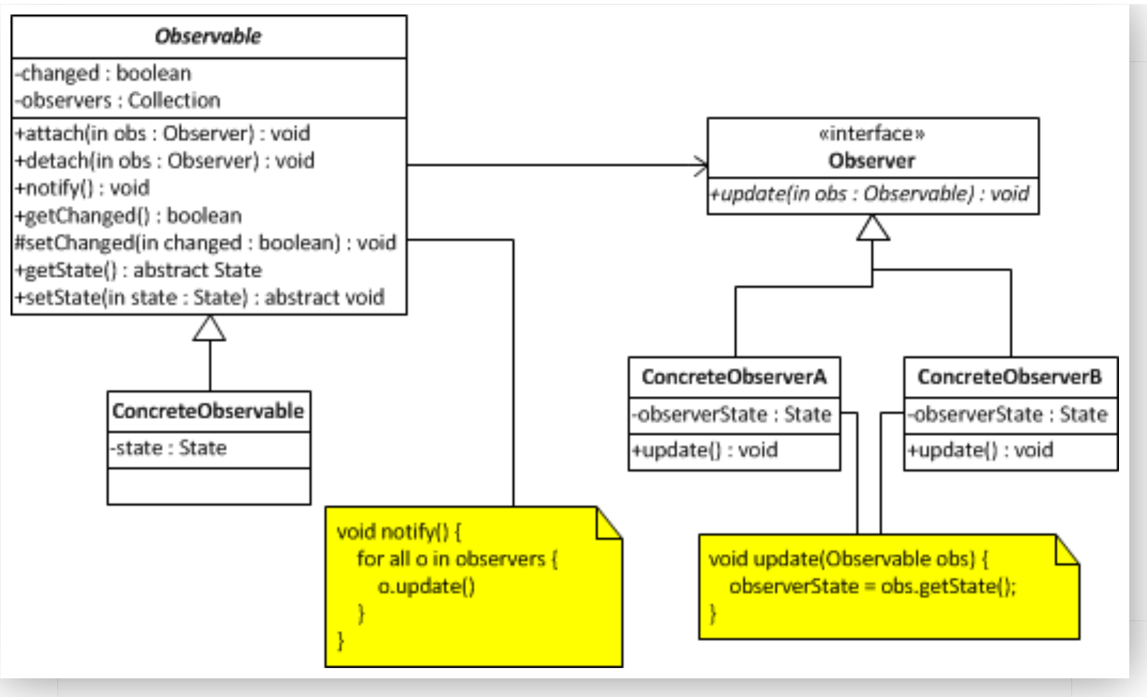


Figure 2: The Factory Design Pattern.

The Observer Pattern

In object oriented systems, there are many times when certain objects in the system need to know about changes that occur in other objects. Object Oriented Design (OOD) best practices require that we decouple as much as possible the associations between objects in a system to reduce dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers (or listeners).

The Observer Design Pattern so common in Java that the Java8 API supports it. For example, you can define an observer to act as a listener for a button in a user interface. If the button is clicked, the listener is notified and performs a certain action. But the Observer Pattern is not limited to user interface components. Another example of the Observer Pattern is a system that monitors a temperature sensor. One part of the system listens for changes in temperature in order to update a display that shows the current temperature. Another part of the system listens for when the temperature exceeds some preset value, triggering another action then that occurs. Figure 2 below illustrates the basic objects and their relationships in the Observer Pattern.



```
void notify() {
  for all o in observers {
    o.update()
  }
}
```

```
void update(Observable obs) {
  observerState = obs.getState();
}
```

Figure 3: The Observer Design Pattern.

The classes participating in the Observer Pattern of Figure 1 are:

Observable - The class that defines the operations for attaching and detaching Observers to the client that will be observed. In the GOF book, this class/interface is known as Subject. The Java Libraries include the java.util.Observable class, an abstract class that implements the core observer management functions.
Observer - The interface that defines operations used to notify observer objects when changes occur in observable objects. The Java Libraries include the java.util.Observer interface that defines a method to be implemented that allows an observable object to notify its observers of changes.
ConcreteObservable - A concrete class that maintains the actual state of the object. When a change occurs in that state, it notifies the observers that are attached to it.
ConcreteObserverA, ConcreteObserverB - Concrete classes that are interested in changes to an observable object and have the means, through the Observer interface, to react to changes in the objects that are observed.

The Observer Design Pattern is appropriate when the change in state of one object must be reflected in another object without requiring the objects to be tightly coupled. The pattern also enables the system to be enhanced with new observers without making dramatic changes to the system. Two classic examples of implementations of the Observer Design Pattern are the Model-View-Controller (MVC) Pattern and Event Management frameworks for graphical user interfaces. In the MVC, the View represents the observer and the Model represents the observable, enabling the View to be updated whenever the Model changes state.

The Observer Pattern is implemented in the Wolf Travel application through the

ConcreteObservable, TourCoordinator, and its only *ConcreteObserver*, TravelGUI. TourCoordinator should be declared like this:

```
public class TourCoordinator extends Observable implements TravelManager  {
```

Each observable object notifies its observers when there is a change. Changes occur when an observable's field is set or when an object is added to or removed from an observable's collection. An observable class notifies its observers by calling the following methods of Observable.

```
setChanged(); // Marks the Observable as changed
notifyObservers(this); // Sends a message to any Observer classes that the object has
```

TourCoordinator should directly call these two methods when there is a change to its internal state. So for the second statement above, the current instance (this) is passed to notifyObservers().

Each observer is notified when there are changes to the objects they observe. Observer classes must implement the interface Observer, which requires they define the update() method. The details of what the update() method should do are specific to each observer class.

The last step in implementing the Observer Pattern is to connect the observable to its observer by adding the observer to the observable. Here is how this is done in the TravelGUI constructor:

```
public TravelGUI() {
     // Observe the Trip Coordinator
     ((TourCoordinator) mgr).addObserver(this);  }
     // More code follows
 }
```

And now here is TravelGUI.update(), which reflects the changes in the TourCoordinator:

```
/**
 * Updates the GUI in response to the TourCoordinator.
 * @param  o  the TourCoordinator
 * @param  arg  the GUI (this)
 */
@Override
public void update(Observable o, Object arg) {
   if (o instanceof TourCoordinator) {
      itemSaveFile.setEnabled(true);
      itemNewFile.setEnabled(true);
      this.refreshAllClients();
      this.refreshAllTrips();
      refreshReservationsForSelectedClient();
      refreshReservationsForSelectedTour();
   }
}
```

Testing

For Part 2 of this project, you must do white box testing via JUnit and report the results of running your black box tests from Part 1.

White box testing

Your JUnit tests should follow the same package structure as the classes that they test. You need to create JUnit tests for *all* of the concrete classes that you create. At a minimum, you must exercise every method in your solution at least once by your JUnit tests. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system. Start by testing all methods that are not simple getters, simple setters, or *simple* constructors for all of the classes that you must write and check that you are covering all methods. If you are not, write tests to exercise unexecuted methods. You can test the common functionality of an abstract class through a concrete instance of its child.

When testing void methods, you will need to call other methods that do return something to verify that the call made to the void method had the intended effects.

For each method and constructor that throws exceptions, test to make sure that the method does what it is supposed to do and throws an exception of the appropriate type when it should.

Testing Reservations

Note: The Reservation class is a little tricky to test because Reservation.codeGenerator is static, so its value propagates from one test to another, potentially starting with some value besides 0 in any given test. The order of execution of JUnit tests on your system may differ from their execution order on Jenkins. Whenever a JUnit test checks the actual value of confirmation codes, the test should call Reservation.resetCodeGenerator() prior to creating any reservations in order to set the value of Reservation.codeGenerator back to 0. You can reset the code generator in setUp() or at the beginning of the JUnit test as appropriate.

At a minimum, you must exercise at least 80% of the statements/lines in all ***non-GUI*** classes. You will likely cover the simple getters, setters, and constructors as part of testing more complex functionality in your system. You must have 95% method coverage to achieve a green ball on Jenkins (assuming everything else is correct).

We recommend that you try to achieve 100% condition coverage (where every conditional predicate is executed on both the true and false paths for all valid paths in a method).

Black box testing and submission

Use the provided black box test plan template to describe your tests. Each test must be repeatable and specific; all input and expected results values must be concrete. All inputs required to complete the test must be specified either in the document or the associated test file must be submitted. Remember to provide instructions for how a tester would set up, start, and run the application. (What class from your design contains the main method that starts your program? What are the command line arguments, if any? What do the input files contain?) The instructions should be described at a level where anyone using Eclipse could run your tests.

If you are planning for your tests to read from or write to files, you need to provide details about what the testing types, test case lists, and test cases would be so that the teaching staff could recreate them.

Follow these steps to complete submission of your black box tests:

1. Run your black box tests on your code and report the results in the Actual Results column of your BBTP document.
2. Save the document as a pdf named **BBTP_P2P2.pdf**.
3. Create a folder named **project_docs** at the top level in your project and copy the pdf to that folder.
4. Push the folder and contents to your GitHub repository.

Deployment

For this class, deployment means submitting your work for grading. Submitting means pushing your project to NCSU GitHub.

Before considering your work complete, make sure:

1. Your program satisfies the style guidelines.
2. Your program behaves as specified in this document. You should test your code thoroughly. Be sure to know what messages should be displayed for each major scenario.
3. Your program satisfies the gradesheet. You can estimate your grade from your Jenkins feedback.
4. You generate javadoc documentation on the most recent versions of your project files.
5. You push any updated project_docs, doc, and test-files folders to GitHub.

Deadline

The electronic submission deadline is precise. Do not be late. You should count on last minute failures (your failures, ISP failures, or NCSU failures). Push early and push often!

Project 2, Part 1: Wolf Travel Tours ▶