Semester Project – Object Oriented Analysis and Design – Iteration 3

Before getting started on Iteration 3, be sure to carefully review the project Introduction, Iteration 1, and Iteration 2, plus the code demos and lectures. The overall project is 40% of your semester grade and Iteration 4 is worth 40% of the project.

Iteration 3 Reminder (from Intro)

Finally, you get to the fun part – coding your implementation. You code the entire solution, test it using the CTO's test data (plus your own), and really polished up the user interface. You send your implementation to the CTO along with a detailed document explaining how to compile the project, how to run it, and all details he needs to try it out *without asking you for help*.

You really work hard on this bit because a lot of money (aka, your grade) is riding on the CTO's assessment of your work. To get paid for this iteration, the CTO expects your implementation to *compile easily and produce an itinerary for Mr. Pickles like the example*. You can expect the CTO to look at your code and step through it.

IMPORTANT! Teams of 2 must implement the XML, Test Runner, and Doxygen components in Iteration 3. See below for details.

- **Iteration 3 - 5% Bonus for Team of 1** / **Required for Team of 2** – The CTO is not comfortable being locked into a single storage format. He loves JSON, but also knows XML is far from dead in the enterprise application space. The CTO will award a bonus if the data can also be stored using XML with a quick change to a configuration file, allowing Premium to store their data in XML *or* JSON.
  IMPORTANT! *For Teams of 2, this component is* ***mandatory****, not a bonus, and is part of Iteration 3's grade.*

- **Iteration 3 - 5% Bonus for Team of 1** / **Required for Team of 2**– The CTO is a big fan of test-driven development. He will pay extra if you automate testing using either NUnit (if C#) or JUnit (if Java). He knows you are terribly busy with other things and cannot load you down with so much stuff the project fails, but he firmly believes you get a performance boost and better code using automated testing, particularly early on before your UI works well.

***IMPORTANT!** For Teams of 2, this component is **mandatory**, not a bonus, and is part of Iteration 3's grade.*

- **Iteration 3 - 5% Bonus for Team of 1** / **Required for Team of 2** – The CTO has no life and enjoys reading API documentation online on his days off. He has already required that you write appropriate comments in your project, so you're halfway there getting this bonus. Using Doxygen, automatically generate an HTML site that helps developers navigate your project's API. Doxygen can work with both C# and Java projects. ***IMPORTANT!** For Teams of 2, this component is **mandatory**, not a bonus, and is part of Iteration 3's grade.*

- **Iteration 3 - 10% Bonus** – The CTO is a *big fan* of open source projects and encourages you to share your designs and code on GitHub. He thinks it helps you build your portfolio to impress future clients – interviews are so much better when you have actual work to show. Further, it demonstrates you know a bit about source control – something so many CS graduates know little about, but employers really value. The CTO *sincerely* hopes you do a great job building a GitHub repo and readme.md that shows off your great designs. He fondly recalls a project he did in graduate school for Dr. Edward Jung's Theory of Computation class and passes along the GitHub repo link as an example:
  https://jadkisson.github.io/dfa/
  https://github.com/jadkisson/dfa

- **Iteration 3 - 10% Bonus** – The CTO thinks the web is even better than America Online. He will award a 10% bonus if the user interface is implemented in a browser using either React or Angular because these days, the best apps run on the web. The CTO does not care what web server you use, provided it follows the Model View Controller pattern and he doesn't have to install a million complex things on his computer to test your project.

UI Flows

- **When application starts**
  - Start application and choose which Agent you are. Choose Agent once.
    Show list of Agents from the TravelAgent singleton and choose one.

- **UI flow #1 – Create a new Trip (can be stopped at any point in process and resumed later)**
  This flow will be managed by the state machine pattern. Implement each state as a separate concrete class. You must be able to load a Trip into the state machine at any state and pick up the process to complete the Trip.
  - Create a new Trip and auto-assign a unique ID
    Automatically advance to next state.
  - Add 1..* Travelers to trip (from existing Travelers list)
    - You will choose each Traveler from a list of Persons. Travelers must be unique within a Trip.
      Show list of People in the Person singleton, then choose one, then repeat. If user enters "later", save Trip via Write factory and JSON strategy using serialization and go to UI flow #2. If user enters "done", go to next state (if 1.. added).*
  - Add 1..* Packages to trip (from existing Packages list)
    - One Package includes a Travels From, a Travels To, a Price, and Hours of Travel. You will choose Travels From and Travels To from a list of existing Places.
      Show list of Packages in the Package singleton, then choose one, then repeat. If user enters "later", save Trip via Write factory and JSON strategy using serialization and go to UI flow #2. If user enters "done", go to next state (if 1.. added).*
  - Choose Payment By Person.
    Show list of People in the Person singleton, then choose one. If user enters "later", save Trip via Write factory and JSON strategy using serialization and go to UI flow #2. When user picks one person, go to next state.
  - Choose Payment Type
    Cash, check, or charge. Conditionally go to Cash state, Check state, or Charge state based on payment type. *If user enters "later", save Trip via Write factory and JSON strategy using serialization and go to UI flow #2.*

    - and collect appropriate details (card, check number, amount, etc.)
      If user enters "later", save Trip via Write factory and JSON strategy using serialization and go to UI flow #2. When user picks payment type and completes details, go to next state.
    - Payment must be equal to sum of Package.Price collection to become PaidInFull. If not PaidInFull, return to prior step.
    - Add 1 Payment to Trip.
    - Update Trip to add a required thank you note.
      If user enters "later", save Trip via Write factory and JSON strategy using serialization and go to UI flow #2. If user enters "done", go to next state if note is at least 5 characters. Once a valid thank you note is entered, trip is complete. Use Write Factory and JSON strategy to write Trip to disk via serialization. (plus via XML, if team of 2)
    - Show Itinerary.
      Implement decorator pattern to show itinerary.
    - Go to UI flow #2.

- **UI flow #2 – Show All Trips Owned by Agent**
    - List all trips owned <u>by the Agent</u> by status. Include the ID in each trip and its current state for use with UI flow #3.
      Load list of trips from disk using Read factory and JSON Read strategy. (plus via XML, if team of 2)

- **UI flow #3 – Load a Trip**
    - "Load Trip by Trip ID: "
        - If trip ID is invalid, return to prior step with warning.
        - If trip ID is valid:
            - If state of trip is complete, show itinerary.
            - If state of requested trip is not complete, return to editing trip.
              Load trip into TripState machine and return to proper state. Use Read factory and JSON Read strategy to deserialize the Trip from disk. (plus via XML, if team of 2)

- This is not a UI design class. The UI should be clean and functional. It does not have to look fancy. A console application like the L15_Demo_State projects in the Lecture Demos folder will be fine. If you choose to implement a more sophisticated UI, such as a form or web site, it must still meet the requirements to return later, validate input, etc.

Patterns You Must Implement (Code) In Your Project

1. **Strategy Pattern**
   You will use the Strategy pattern to implement reading and writing Trip objects to disk using JSON or XML serialization.

   When you write a trip to disk, it will store its current state and all of the details entered via the state machine.

   When you read a Trip from disk, you must be able to drop it back into the state machine (if not complete) and resume work on finishing the Trip. If it is complete, it will be ready to generate an itinerary via the decorator pattern.

   Implement the strategy pattern for JSON and XML persistence with read and write functionality.
   a. Team of 1
      Create the concrete classes for JSON and XML. Implement JSON so it works properly. Optionally implement XML for a bonus *or* throw an exception when an attempt to use the XML persistence classes is made.
   b. Team of 2
      Create the concrete classes for JSON and XML. Implement both JSON and XML so it works properly. Add a configuration variable to your application's config file (app.config if C# or config.properties if Java [or similar if using Spring, etc.]). Do not hardcode your JSON/XML configuration. You must be able to change it without recompiling.

2. **Factories (2)**
   You will use the Factory pattern to make the persistence objects used for reading and writing Trip objects to/from using JSON or XML serialization.

   a. Implement a Read factory to return the concrete persistence objects you designed in the Strategy pattern. Note that if you are a team of 1, you must still create the XML concrete classes, but you do not have to implement it when coding the project. Teams of 2 will both design and implement both JSON and XML functionality.

   b. Implement a Write factory to return the concrete persistence objects you designed in the Strategy pattern. Note that if you are a team of 1, you must still implement the XML concrete classes, but you do not have to implement it when coding the project. Teams of 2 will both design and implement both JSON and XML functionality. – 7.5%

3. **Decorator**
   You will use the Decorator pattern to implement the itinerary shown in the Project Introduction.

   Implement the decorator pattern for the Itinerary. Carefully review the itinerary from the Project Introduction and design the components needed to make each section. Remember to keep to the single responsibility principle. The Lecture 14 and 15 coding demos and slides nicely demonstrate the how to implement an itinerary using decorator.

4. **State**
   You will use the State pattern to implement the workflow to build a trip from start to finish. If done properly, your State pattern will enable you to return to incomplete trips at any point and resume your work. Once the pattern reaches the complete state, the trip should be valid and 100% ready to generate an itinerary.

   Implement the state pattern to handle creating a single Trip. Carefully review the state diagram from Iteration 1 and review the Lecture 15 slides and coding demos.

5. **Singleton (3)**
   You will use the Singleton pattern to the reference data used by your application. All singletons will be lazy-load double-checked thread safe singletons and will return read-only data.

   a. Package singleton
      Implement a singleton that an Agent will use when choosing the packages to add to a Trip. Load the data from disk.

   b. Person singleton
      Implement a singleton that an Agent will use when choosing the People to add to a Trip and the person who is paying for the trip. Load the data from disk.

c. Travel agent singleton
   Implement a singleton that an Agent will use to choose himself/herself when the application starts. Think of this as a login function, though we are not bothering with passwords and usernames – the user will just choose an agent from a list in the singleton. Load the data from disk.

Test Data

For test data, you need to be able to recreate the itinerary from the Project Introduction document.

To prove that your application isn't hardcoded to just that itinerary, you need to add additional people, travel agents, and packages to ensure that when I test your application, I am confident your application can handle every UI case including 1 traveler, many travelers, three different billing types (check, cash, card, etc.).

Iteration 4 – Presentation

The final component of the project is your presentation to the entire class. As you are working on Iteration 3, think about what you want to show. Part of being a professional developer is demonstrating your work with confidence and convincing your customer they received excellent value. Keep that in mind.

Compilation Warning

As seniors, I should not need to mention this, but there have been problems this semester and I want to be very clear on this.

Your project must compile. If it does not compile, the highest grade you can expect is 50%.

To avoid problems with compilation, carefully document the steps to unzip, build, and run your project in your "how to build and execute document". Test your steps thoroughly. Given that everyone is doing JSON (and perhaps XML), there will be Nuget, Maven, Gradle, etc. dependencies that should auto-install in VS or Intellij.

I should not have to install additional software other than Intellij 2018 or VS 2017 to run your project (aside from auto-loaded dependencies).

Team of 1 Grading Rubric

1. Read/Write Factories – 5%
2. JSON Persistence Strategy – 10%
3. State Machine – 20%
4. Decorator Pattern – 15%
5. All Singletons – 5%
6. UI flow meets all requirements – return later works for each state, input validation ensures only good data reaches the Trip object, etc. – 25%
7. Project structure demonstrates good coding practices – commenting, small class files, good design using SOLID, etc. – 10%
8. Project was properly submitted, including writing a short “how to build and execute” document – 10%

Team of 1 Deliverables – penalty of no less than 25% if not followed

1. Zip your VS2017 or Intellij 2018 project and name it *your_name_project_source.zip*.
2. *your_name_how_to_build_and_execute.doc* file describing how to open, build, and execute your project.
3. Note any bonuses you attempted in the D2L submission details.
4. Submit the .zip file and the .doc file to D2L. You will be submitting **two** separate files.

Team of 2 Grading Rubric

1. Read/Write Factories – 5%
2. JSON/XML Persistence Strategy, plus configuration file to choose without recompilation – 10%
3. State Machine – 20%
4. Decorator Pattern – 15%
5. All Singletons – 5%
6. UI flow meets all requirements – return later works for each state, input validation ensures only good data reaches the Trip object, etc. – 20%
7. Project structure demonstrates good coding practices – commenting, small class files, good design using SOLID, etc. – 10%
8. Write at least 2 automated unit test using NUnit or Junit, including writing a short "how to execute unit tests". Test 1 should verify that your singletons contain data and Test 2 should verify that you can reload a trip from disk to the correct state. – 5%
9. Use Doxygen to produce HTML documentation using your C# or Java project. This obviously assumes that your code is properly commented. – 5%
10. Project was properly submitted, including writing a short "how to build and execute" document, short "how to execute unit tests" document, and Doxygen output – 5%

Team of 2 Deliverables – penalty of no less than 25% if not followed

1. **EACH TEAM MEMBER MUST SUBMIT!**
2. Zip your VS2017 or Intellij 2018 project and name it *your_name_project_source.zip*.
3. Zip your Doxygen HTML output and name it *your_name_doxygen.zip*.
4. *your_name_how_to_build_and_execute.doc* file describing how to open, build, and execute your project.
5. *your_name_how_to_execute_unit_tests.doc* file describing how to run your two automated unit tests.
6. Note any bonuses you attempted in the D2L submission details.
7. Submit the two .zip file and the two .doc file to D2L. Each team member will be submitting **four** separate files.