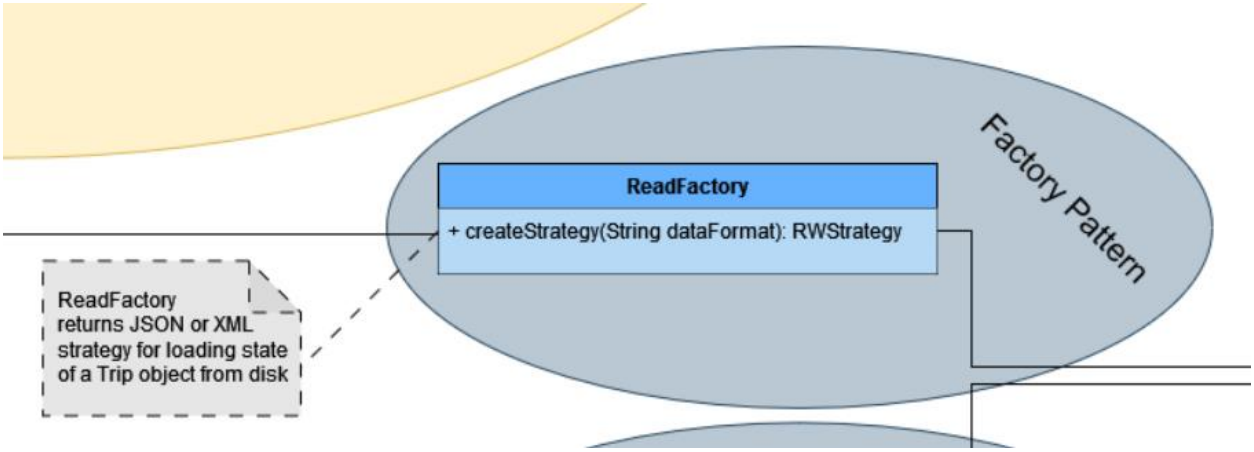
Audrey Allen & Ed Exley

SWE 4743

Section W01

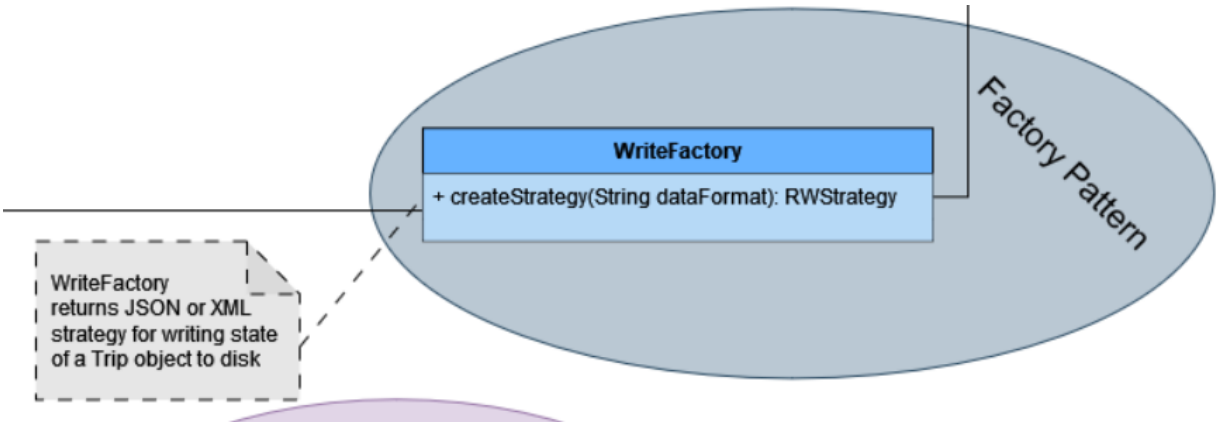
Iteration 2 – Writeup for Updated Class Diagram

a. ReadPersistence Factory Pattern



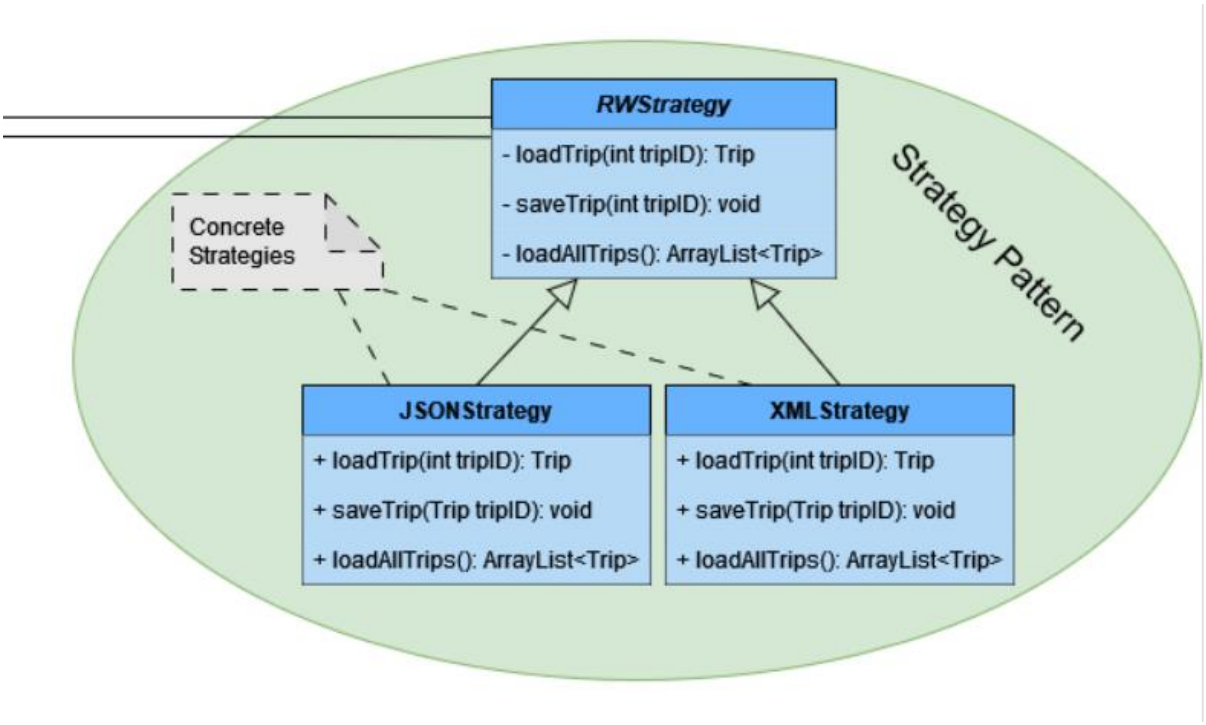
The application's read factory is designed to work in tandem with the strategy pattern. It is meant to return the strategy for 'reading' (loading) a trip object that has been saved to the disk using the write factory.

b. WritePersistence Factory Pattern



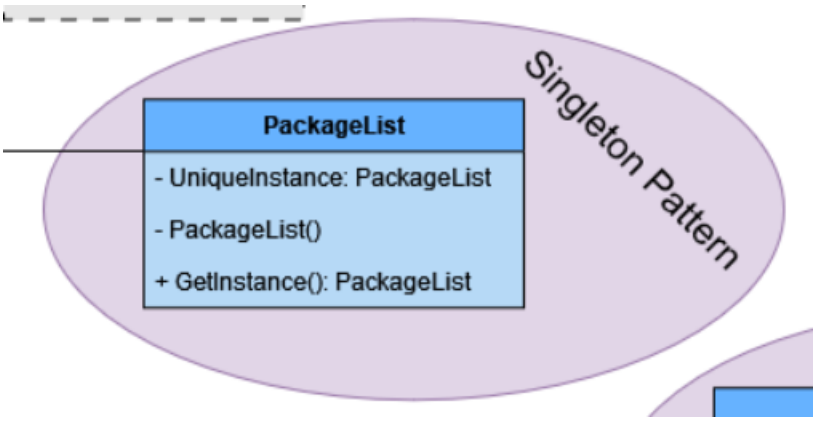
The write factory is also meant to work in conjunction with the strategy pattern. In this case, it returns the strategy for 'writing' (saving) a trip to the disk.

c. Strategy Pattern



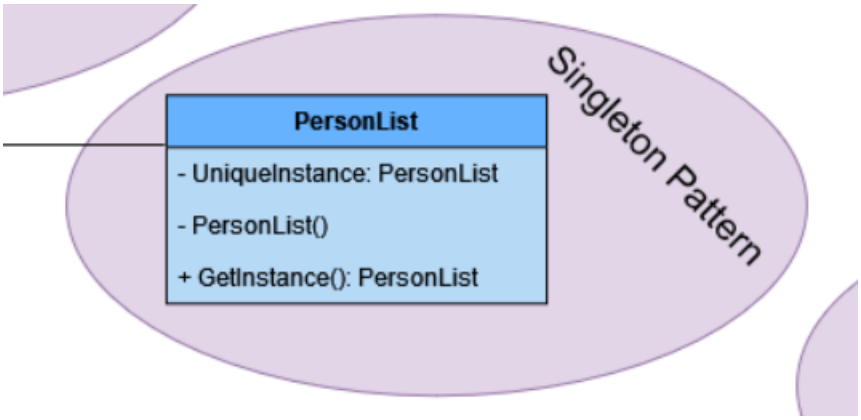
The strategy pattern is designed to allow trips to be loaded and saved using either JSON or XML. These two concrete strategies are interchangeable; changing the configuration between JSON and XML will determine which strategy is used at runtime.

d. Package Singleton Pattern



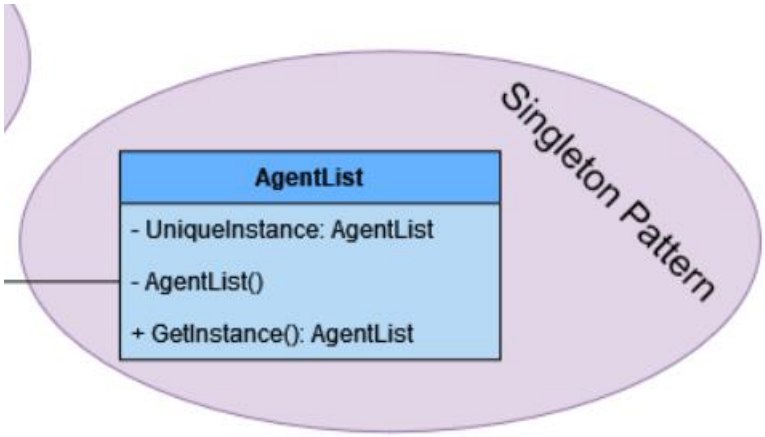
All package objects are consolidated into a single list of packages, which will act as the proxy by which a travel agent interacts with package objects while using them to build a trip, rather than accessing package objects individually.

e. Person Singleton Pattern



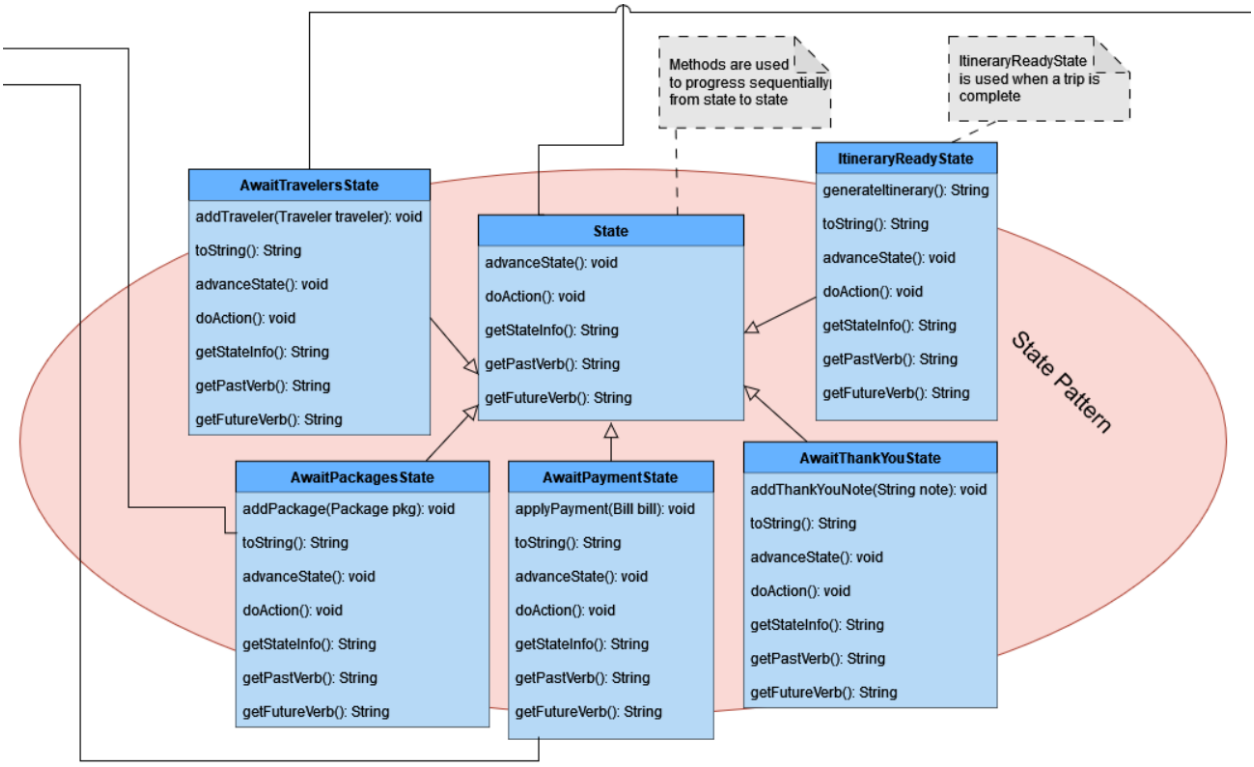
All person objects available for agents to add to a trip are consolidated into a single list. This is not to be confused with the singleton for agent objects, even though an agent is an extension of the 'Person' class. This singleton includes only customers and includes both travelers and non-travelers (such as payors). This list is the proxy by which the travel agent will access person objects whose data can be used to build a trip.

f. Travel Agent Singleton Pattern



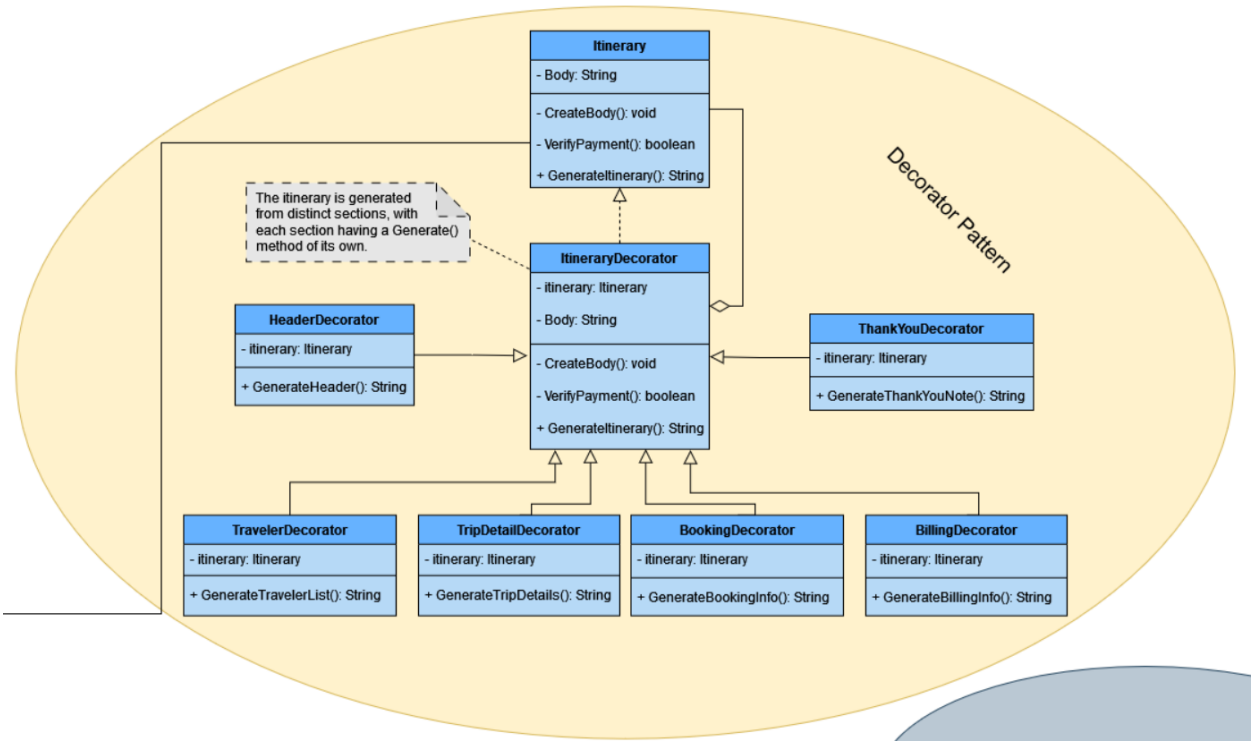
Every travel agent object is grouped into a single list. This list represents the identities of users who have access to the system, not the identities of people who will be added to trips as customers. Selection of a user from this list grants access to the ability to build a trip.

g. State Pattern



States exist for each step of building a trip. The advanceState() method guides sequential progression through the state machine, ensuring that there is only one linear route that can be used to build a trip, and that the final state, the 'ItineraryReadyState,' may only be accessed once the trip is complete. This state also does not advance to another state, emphasizing its finality and requiring another trip to be created before the process can begin again. The method also verifies that valid input has been given before allowing the user to progress to the next state. Because the trip exists within a specific state until advancement requirements have been completed, when that trip is saved and loaded again, it will load in the exact state it was last in. This ensures the user can quit and resume the project at any point.

h. Decorator Pattern



The primary itinerary decorator has concrete decorators for each separate section of the itinerary, modeled after the sample itinerary given. All classes but one pull data from input methods used during each stage of building a trip. There is an additional decorator for the itinerary's header, which includes the booking agent's name, number, and opening message.