Overview
This document describes a minor application created for a code review with ApolloTech MSI's chief engineer, Eric Nguyen. It walks through all phases of the SDLC, from scoping to release. Maintenance and obsolescence are excluded, since this is a demonstration project.

Scoping
This program will help users select vacation destinations based information they enter about themselves. The program will contain a predefined set of vacation destination options. It will accept input about a person's vacation preferences, their budget, and their passport status, and it will recommend a destination according to those inputs.

Requirements
This section contains informal requirements for the program.

Destinations:
-Each vacation destination will have the following properties:
1. Cost to visit in the destination's local currency.
2. Whether it's a foreign location or domestic one.
3. A beach rating from 0 to 7.
4. A mountain rating from 0 to 7.
5. A city rating from 0 to 7.

The software will contain the following destinations:

	Miami	Denver	Paris	Rio de Janeiro
Beach Rating	7	0	0	6
Mountain Rating	0	7	0	0
City Rating	4	1	7	6

-Each destination also has a weekly cost.
1. Domestic destinations have fixed weekly costs in US dollars. Miami costs $1,000 per week and Denver costs $500 per week.
2. Foreign destinations have fixed costs in their local currencies. Paris costs 1,000 Euros per week. Rio de Janeiro costs 2,000 Real per week.

Travellers:
-Each traveller will enter the following properties to describe themselves:
1. Beach preference (an integer from 0 to 7).
2. Mountain preference (an integer from 0 to 7)
3. City preference (an integer from 0 to 7).
4. Their maximum budget in US dollars.
5. Whether or not they have a passport.
6. Their name.

The software will compute for a traveler:
1. Which destination will provide the best overall experience, within their budget and travel ability.

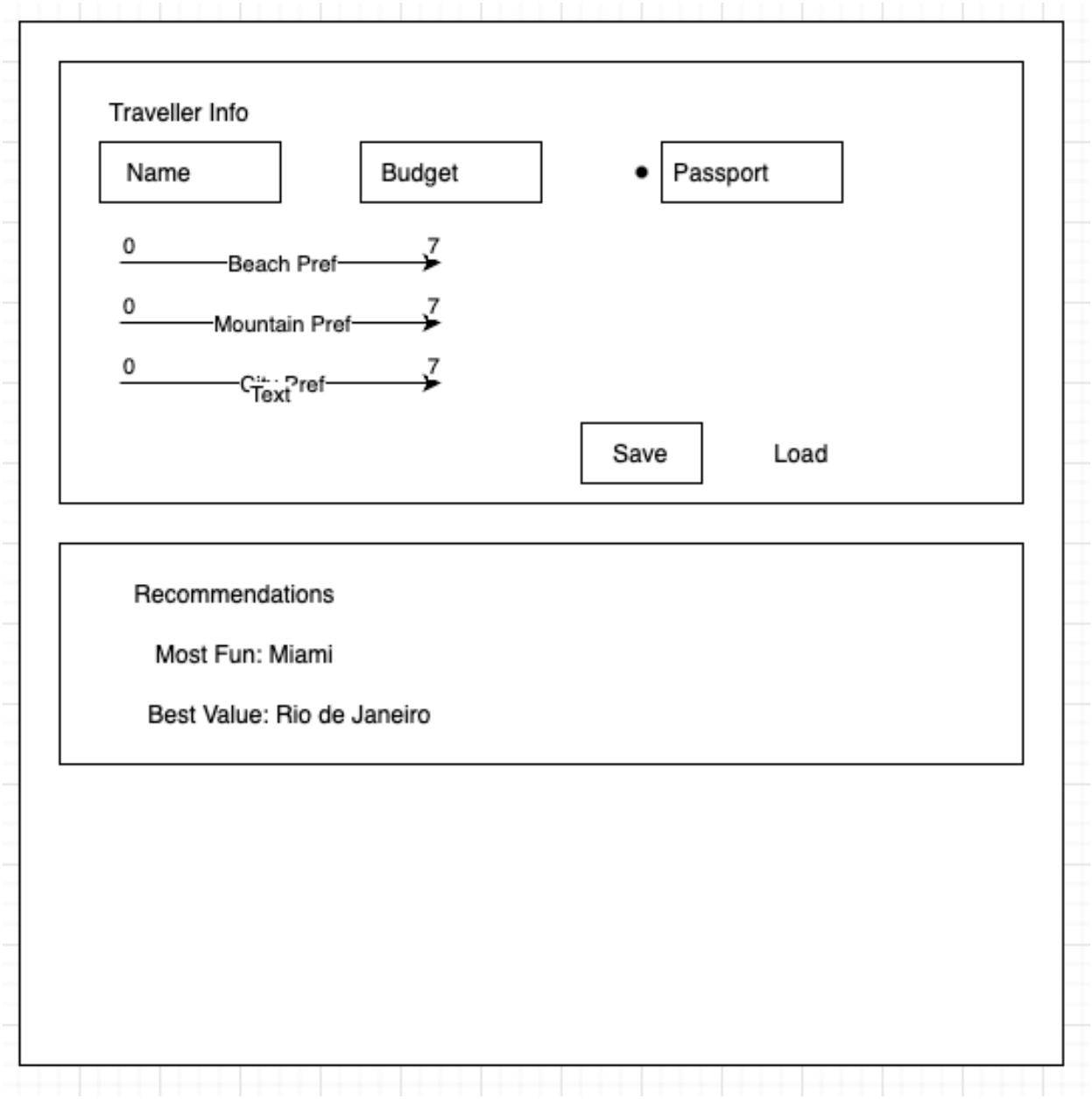
2. Which destination will provide the best experience per dollar spent, within their budget and travel ability.

Persistence:
1. The software will save and load travelers.
2. Travelers will be identified by their name for saving and loading purposes.

[We can definitely talk about why these aren't great software requirements for a real system. I'm just writing these out here informally to help guide my creation of the codebase.]

Functional Specification
This section contains the UXD for the program.

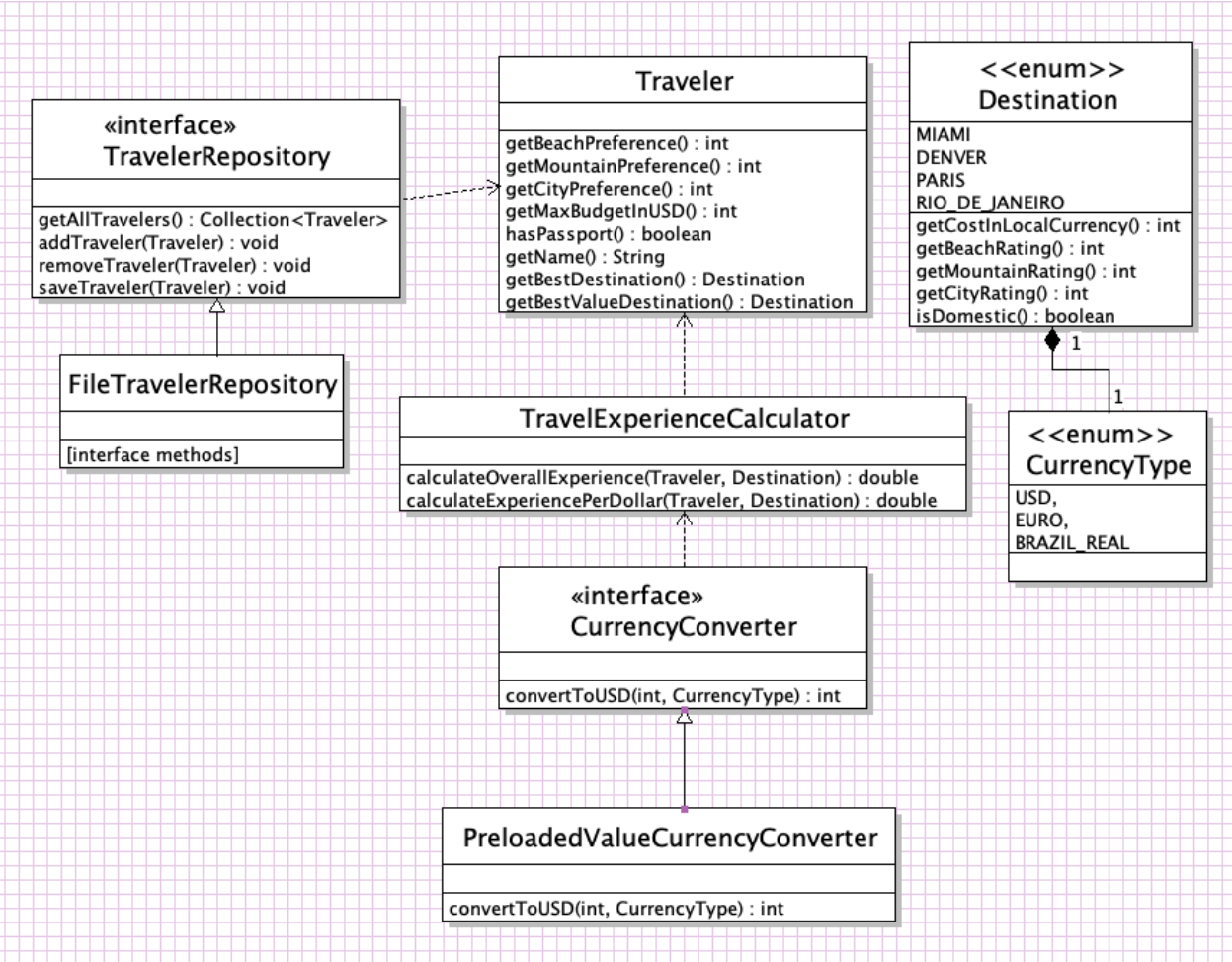
This is a quick / low-fidelity wireframe made using draw.io.



For real software, I'd want to use a tool like Figma. I'd start by defining a user persona, I'd then make higher fidelity wireframes and (maybe) flow charts. It might also be worth using prototyping tools, depending on the time and resources that can be dedicated to UXD.

Technical Design
This section contains a brief technical design for the application.

-Travel Software Class Diagram:



Commentary:
These classes are very simple and maybe I should've decided to build a somewhat more complicated app to demonstrate more OOP principles. Well designed classes encapsulate both structure and behavior. The traveler class exposes some meaningful behavior, in that it's the source of the best destination and best value destination that the rest of the system needs.

TravelerRepository is this app's use of a design pattern. In this case, I'll use the repository pattern to persist travelers.

CurrencyConverter is the interface that defines how values are converted from foreign currencies to US dollars. There's one implementation, the Preloaded one.

Note: For more complicated software, I'd create a more detailed design. This might feature more UML class diagrams and sequence diagrams, depending on the complexity of the software I'm trying to make.

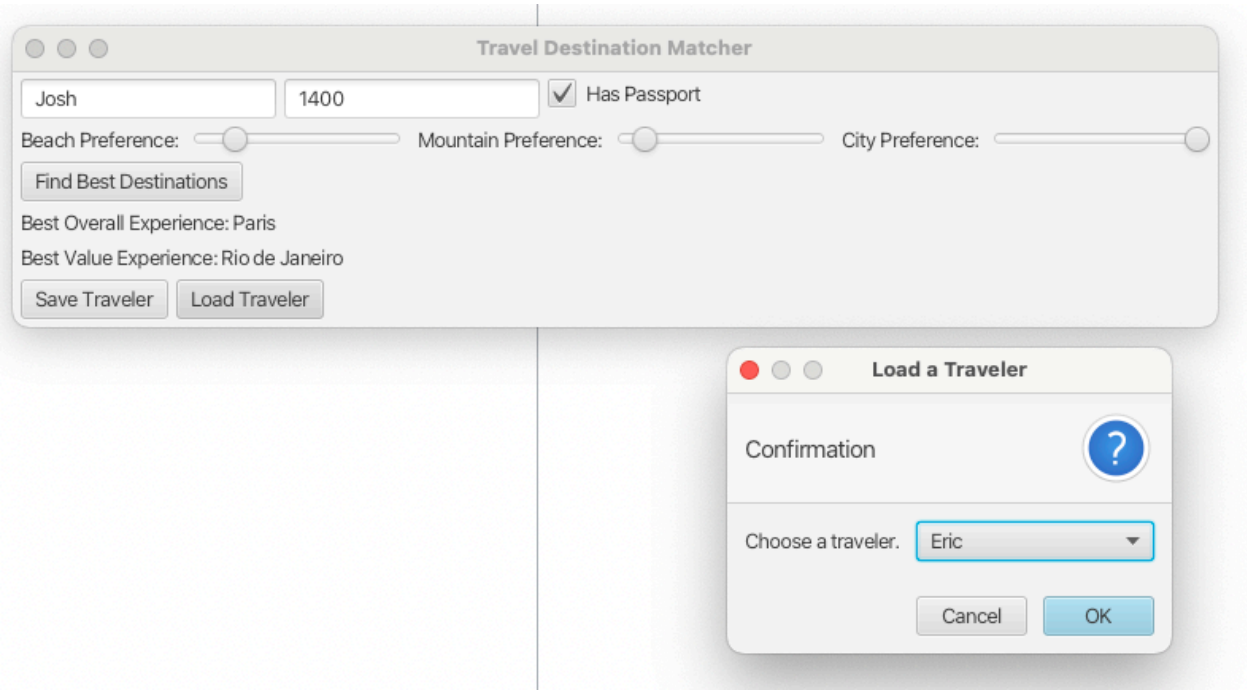
If I were really setting up the entire architecture of a more complicated app, I'd perform some kind of architecture analysis for choosing the languages and frameworks I'd use. In this case, I'm just going to use core Java for creating the business logic, Maven for the build, jUnit for writing unit tests, and JavaFX to create the UI. I'll probably use GSon just to serialize the Traveler instances to Json when I store them.

For the sake of brevity, I'm going to proceed with just this one class diagram, just because I'm a bit time-crunched this weekend.

Code Guide
The code for this software can be found on my GitHub repository, here:
https://github.com/SRJavaDeveloper/TravelSoftwareDemo/tree/master

The most important classes / files are as follows:
1. TravelApplication: This is the primary front end class, using JavaFX to set up the UI for the app.
2. Traveler: This is the business class to model a traveler.
3. Destination: Describes a travel destination.
4. TravelExperienceCalculator: This houses the algorithm to find ideal destinations for travelers. It uses the currency converter in its figuring.
5. TravelerRepository / FileTravelerRepository: These are the persistence / JSON-serialization mechanisms.
6. TravelApplicationTests: This class houses the unit tests for the project.
7. pom.xml: I use Maven to build the project and manage dependencies.

App Screenshots, Discussion Points, Conclusion
Below is a screenshot of the app running.



This shows calculations for a traveler named Josh and the load dialog to load data for another saved traveler.

It's a straightforward and simple program, but it demonstrates some object oriented principles, features unit tests, has a working front end and implements a persistence feature. Hopefully this provides enough substance for review and discussion, and I'll be happy to talk about any parts of the project deemed fit for discussion.

Thank you for taking the time to review my work!