SkiMapp:

An Android App to Enhance Ski Maps via Domain-Specific Modeling

Tizzy Macgregor

Undergraduate Honors Project Report

Computer Science
School of Informatics
University of Edinburgh
Supervisor: Perdita Stevens

2019

Abstract

Traditional ski maps have numerous shortcomings making them confusing to interpret, rendering the development of a easy-to-use app essential. The goal of this project is to develop a domain-specific modeling language to describe the runs and lifts in a ski domain, and then use this as a basis for an Android app that provides skiers with helpful information and directions based on their preferences and location. The motivation behind designing a domain-specific modeling language is so that the app can work with any ski resort expressed in the DSML, a valuable feature not yet present in any other ski map app.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Tizzy Macgregor

Table of Contents

1. Introduction

The goal of this project is to develop a domain-specific modeling language to describe the runs and lifts in a ski domain, and then use this as a basis for an Android app that provides skiers with helpful information and directions based on their preferences and location.

1.1 Motivation

Traditional ski maps, like the one of Passo Tonale shown in Fig 1.1, can often be difficult to interpret. According to a report in the Austrian media [1], 400 people get lost or even go missing each year in the Austrian mountains alone. This can be even more of a problem when skiing in a foreign country, as it was for the Australian family that got lost at a Japanese ski resort and had to spend the night in a snow cave before being rescued the next day [2]. Rescues like this involve helicopters and many mountain professionals and are therefore very expensive. Even in less extreme situations, getting lost while skiing can simply ruin a skier's day.

For large resorts whose maps have to be at small scale, it can be difficult to tell if it's possible to ski to a particular lift from a run or if they are just geographically close. For short sections of a run, it can be difficult to tell its direction, especially if neither end is a lift. Ski maps around the world use different notations which can be confusing for skiers used to a certain set of symbols and color coding. Less experienced skiers need to be able to navigate the mountain whilst staying within their ability. If they get lost they could end up on part of the mountain with slopes beyond their ability which can be a dangerous and scary situation.

The motivation behind designing a domain-specific modeling language (DSML) is so that the app can work with any any ski resort expressed in the DSML. This is a valuable feature as many ski apps are bound to a particular resort.

My aim with this project is to create an app that helps skiers navigate a resort so that they can stay safe and get the most enjoyment out of their time skiing.

1.2 Achievements

I was able to define a DSML to describe ski resorts and develop an Android app with the following features:

- Works with any ski resort expressed in the DSML.
- Locates the skier on the mountain in terms of the DSML model.
- Calculates customizable routes around the mountain:
    - Tailored to the skier's ability
    - Option for only groomed runs
    - Option for route with more difficult runs
    - Option for longer routes
    - Takes into account run closures and lift opening times
- Provides quickest routes to nearest mountain facilities, eg. restrooms.

- Provides essential information in case of an emergency.
- Real-time groom status and run closure updates.
- Users can look up information about particular runs and lifts.

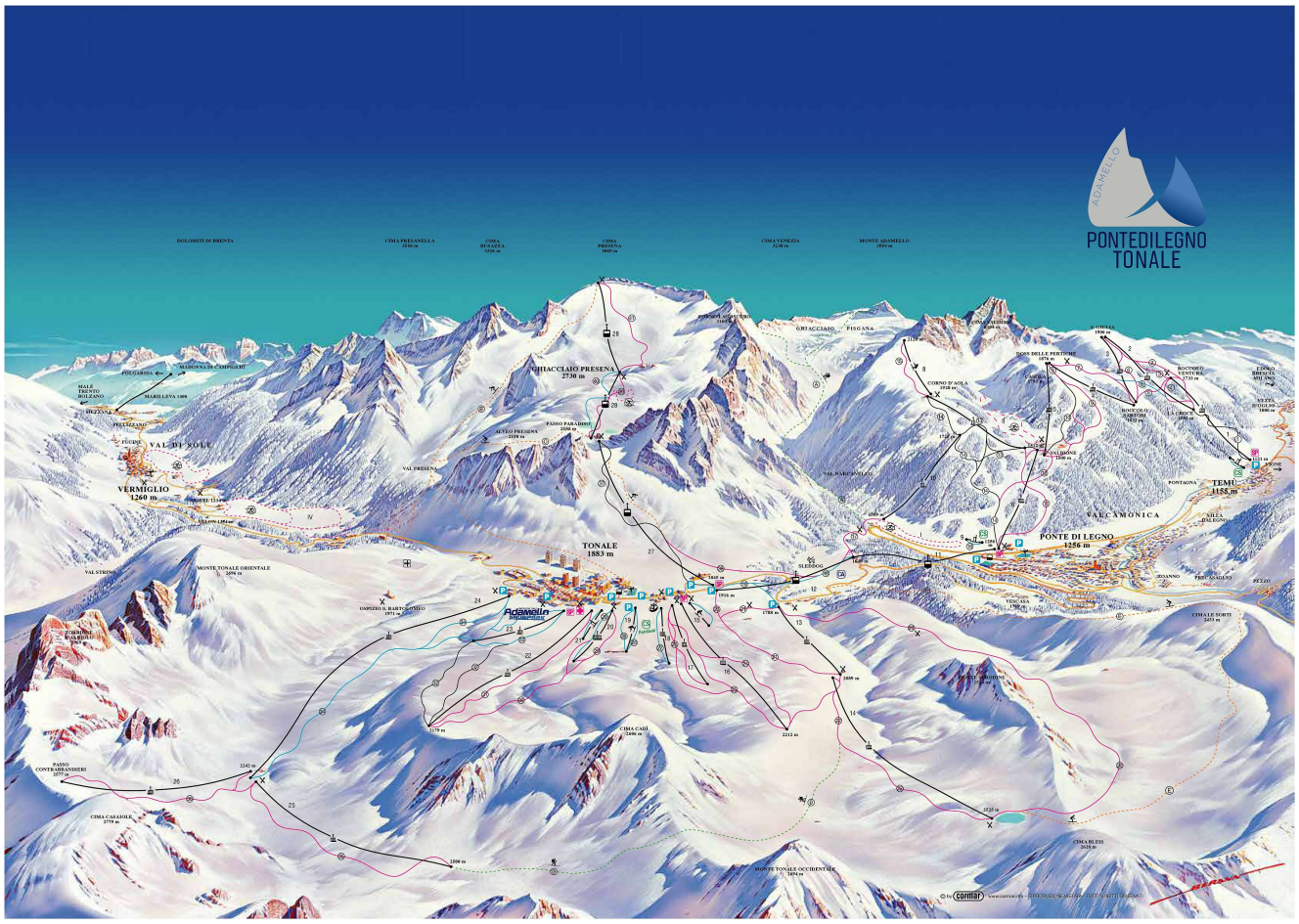

Fig 1.1. Map of Passo Tonale (image from [44])

2. Background

2.1 DSMLs

Ulrich Frank defines a domain-specific modeling language in the following way:

> " A DSML is a modeling language that is intended to be used in a certain domain of discourse. It enriches generic modeling concepts with concepts that were reconstructed from technical terms used in the respective domain of discourse. A DSML serves to create conceptual models of the domain, it is related to." [3]

In this project, I design a DSML for describing ski resorts. The main motivation behind this is the ability to describe any and all ski resorts with this language. This means that any ski resort written in this DSML can be easily used with the Android app without having to change anything else in the app. DSMLs make modeling more productive; a ski resort wishing to use my app won't have reconstruct the technical terms themselves. Furthermore, a DSML is the result of a rigorous development process. This results in a high quality domain model which can be used as the basis for developing the app itself.

2.2 Android Development

The first version of Android was released in 2008 and since then Android has become one of the world's most popular operating systems with 88% of all smartphones sold in 2018 being Android devices [4]. Android is a Linux-based operating system that runs on myriad different devices, not just smartphones. Android software is free and open source making it both accessible and customizable.

The most common approach to developing Android applications is the native approach. This means applications developed with Java and for devices running Android 5.0 or higher. These devices by default use Android Runtime which translates the app's bytecode into native instructions to be executed by the runtime environment. I use this approach in this project because native apps are able take full advantage of the capabilities of the Android OS and run with minimal overhead, resulting in excellent performance.

2.3 Analysis of Existing Systems

There are many ski apps available with many different focuses. A lot of them track a skier's stats such as their fastest run or the number of days they have skied; many are for checking conditions like snowfall or avalanche bulletins. Another common app is one specific to a particular resort.

By contrast, the emphases of my project are helping skiers navigate and the ability to do this for any resort in a single app, so I wanted to review apps that focused on navigation and also worked for multiple locations.

2.3.1 MY EVALUATION CRITERIA

Functionality:
How well does the app help a skier navigate the mountain? Does it work in multiple resorts? How accurate are the directions?

Usability:
Is the app intuitive and easy to use? Is it easy to navigate as a skier using this app?

Performance:
How well does the app do this? Is the app responsive? It is fast?

I now evaluate and discuss three different apps relevant to my project.

2.3.2 ASPEN SNOWMASS APP

The Aspen Snowmass app[5] exemplifies a resort-specific app. This app also displays a traditional map on a phone screen.

Fig 2.1. Screenshot of Map feature of Aspen Snowmass App

2.3.2.1 FUNCTIONALITY

Key features of the app:

- "Interactive, on-mountain mapping"
- Tracks user's skiing statistics, for example vertical feel and number of days skied.
- Mountain conditions, for example weather reports and groom reports
- Resort specific information, for example ticket prices
- The app only works for the one resort

2.3.2.2 USABILITY

The groom report is only available as a list of runs. This forces users to navigate back and forth from the map and the mountain conditions to understand how the groom report applies to the mountain.

The "interactive on-mountain mapping" is simply the traditional ski map displayed on the phone, as shown in Fig 2.1. The map doesn't display the user's location. Since phone screens are relatively small, this makes the map difficult to use.

2.3.2.3 PERFORMANCE

The app has compatibility issues with many smartphone models. The tracking feature works for number of days skied but not for vertical feet.

2.3.3 FATMAP

Fatmap[6] is a mapping app which uses satellite imagery to provide a global 3D map of the outdoors for navigating and planning routes, not just for skiing but also hiking and biking.

2.3.3.1 FUNCTIONALITY

Key features of this app:

- Navigating the outdoors.

The user can locate themselves within a topological map of the entire world. Features like roads and peak names are displayed. The user can plan routes using the 3D map to get insight into the terrain.

• Recommended routes.
The app provides many routes that have been created by outdoor professionals.

• Real-time conditions.
The app provides real-time information on conditions, for example snow layers, snow forecasts and resort status updates.

• Works for the whole world.
Any part of the world can be downloaded. Piste overlays are also available. Many ski resorts have been mapped but not all as it is quite an involved process.

• Ski resorts.
For many of the world's popular ski resorts, Fatmap provides overlays showing lifts, pistes and slope gradients, as shown in Fig 2.2.

Fig 2.2. Screenshot from Fatmap.

2.3.3.2 USABILITY

The app provides an interactive way to explore the terrain and makes good use of colors to convey terrain detail. The routes overlay the terrain well which helps the user follow them in the real world. Since the map is 3-dimensional, the relative locations of ski lifts are easily comprehended.

Routes cannot be produced in the app; instead the user has to create them in advance on their desktop.

2.3.3.3 PERFORMANCE

The app is slow and unresponsive and crashes often. Many of the piste maps are inaccurate.

2.3.4 CITYMAPPER

CityMapper[7] has nothing to do with skiing but it is an excellent example of a navigation app for cities. I wanted to review this app because much of the feature set is similar. CityMapper has been designed specifically for cities and therefore makes excellent use of city-specific modes of transport, for example the Tube in London. A more general app like Google Maps[8] is based on walking and driving which often aren't the best options in a city.

2.3.4.1 FUNCTIONALITY

Key features of this app:

• Finds the quickest routes using different combinations of public transport, cycling and walking. The app assumes the starting point is the user's current location but there is the option to change this. The time of departure or arrival can also be set. The user has the option to select from several different route options involving different combinations of modes of transport. The routes are listed in

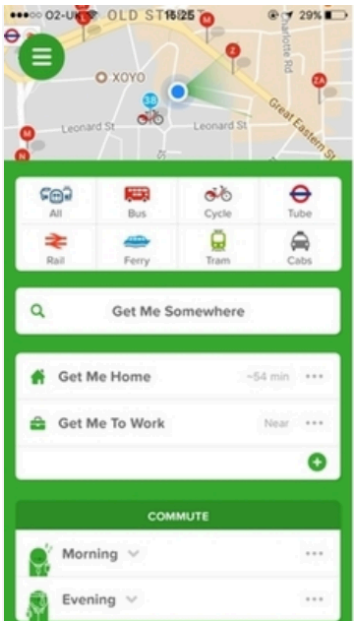

Fig 2.3. Screenshot of CityMapper's homepage

order of time as finding the fastest route is the goal of the app. It is possible to set constraints on the route, for example only taking buses or even finding a route which helps the user avoid rain. All this can be seen in Fig 2.4.

- Step-by-step directions.

Directions are displayed as a list and each component of the list is updated in real-time. For example, if one of the steps is to take a certain bus the live departure times are shown. The directions also update as the user moves through the route, allowing them to focus on the current section of the route.

- Real-time public transport updates

From the home page, the user can select from a list of public transport options to see information and live departure times.

- Offline mode.

- Available for many cities.

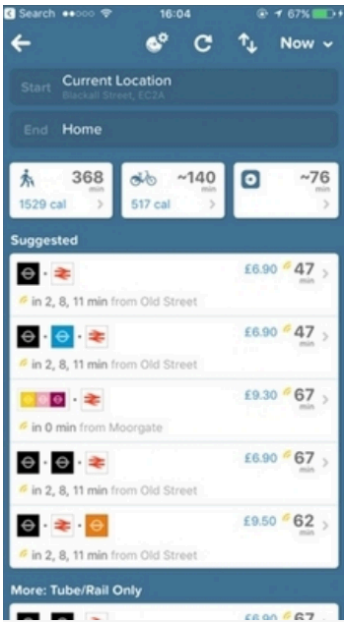

Fig 2.4. Screenshot of CityMapper's suggested routes

2.3.4.2 USABILITY

Throughout the app, the layout strikes an excellent balance between providing lots of information and options but still remaining uncluttered and easy to use.

The 'Get Me Home' and 'Get Me To Work' features (shown in Fig 2.3) are an excellent idea and really useful in practice. The directions are very clear and easy to follow. The combination of text and icons allows the user to comprehend the directions easily.

2.3.4.3 PERFORMANCE

The app is very responsive and loads information fairly quickly. It saves recent searches and favorites which makes it even quicker is practice.

3. Development Methodology

For this project I decided to take a model-driven approach. Model Driven Engineering (MDE) is an approach to software development that emphasizes the use of domain models. These models help simplify the design process by acting as a basis for implementing the system.

My approach also takes inspiration from Eric Evan's *Domain Driven Design* [9] (DDD). DDD has three core principles:

1. Focus on the core domain and domain logic.
2. Base complex designs on models of the domain.
3. Collaborate with domain experts, in order to iteratively refine the model and resolve any domain-related issues.

DDD is based on Object-Oriented Analysis and Design so everything in the domain model is based on an object. This means that components are modular and encapsulated, making the entire system flexible and therefore easily altered and added to. In addition, placing emphasis on the domain ensures that the final product will be tailored to users within that domain. In this project I apply DDD by creating a domain-specific modeling language, as aforementioned.

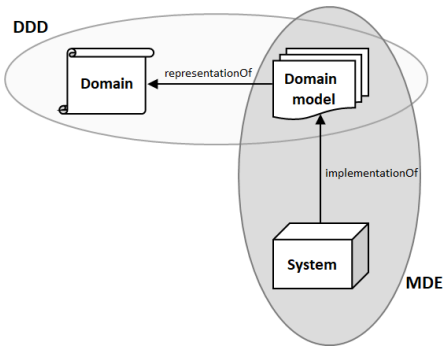

Fig. 3.1. The combination of MDE and DDD (image from [10])

By using a combination of MDE and DDD, as shown in Fig 3.1, I can get the most benefit out of my domain model:

- As a description of the domain
- As a representation of the system to be built
- As the basis of a domain-specific modeling language
- As documentation
- As a specification for testing

Throughout this project I followed these approaches in the following ways:

3.1 REQUIREMENT ENGINEERING

Following Evan's principles, I first established a Ubiquitous Language. I then moved into the Knowledge Crunching phase of eliciting and analyzing requirements to iteratively produce the domain model. This model needs to not only represent the data structures but also knowledge of the ideas, behaviors and rules of the domain. At the same time, the model needs to only contain relevant knowledge. The end result of Knowledge Crunching is a domain model of:

- What the app should do
- How this should be done
- The technical implementation of the app

3.2 IMPLEMENTATION

I followed several of the techniques laid out in DDD [9] while developing the software for this project which I will now discuss.

3.2.1 INTENTION REVEALING INTERFACES

All the names of classes and methods should convey their effect without describing how it is achieved. These names should also conform to the Ubiquitous Language of the project.

3.2.2 SIDE-EFFECT FREE FUNCTIONS

Operations should compute and return results without creating side-effects. A good way of achieving this is by separating methods that read data from those that alter data.

3.2.3 BOUNDED CONTEXTS

DDD divides a large system into bounded contexts. This is because total unification of a domain model is usually unrealistic. Mogosanu best describes this concept as: “a context means a *specific responsibility*. A bounded context means that responsibility is enforced with *explicit boundaries*” [11].

Bounded contexts share some concepts with each other but also have their own unrelated concepts. In my application I have two bounded contexts: one for the ski resort subdomain and one for the graph subdomain. The boundary between can be seen in the way I use language differently in these two contexts. For example, edge has a slightly different meaning in the two contexts. In the graph context it is simply a line connecting two nodes but in the resort context it is the super class of runs and lifts and can even have midpoints. This is shown in the context map in Fig. 3.2.

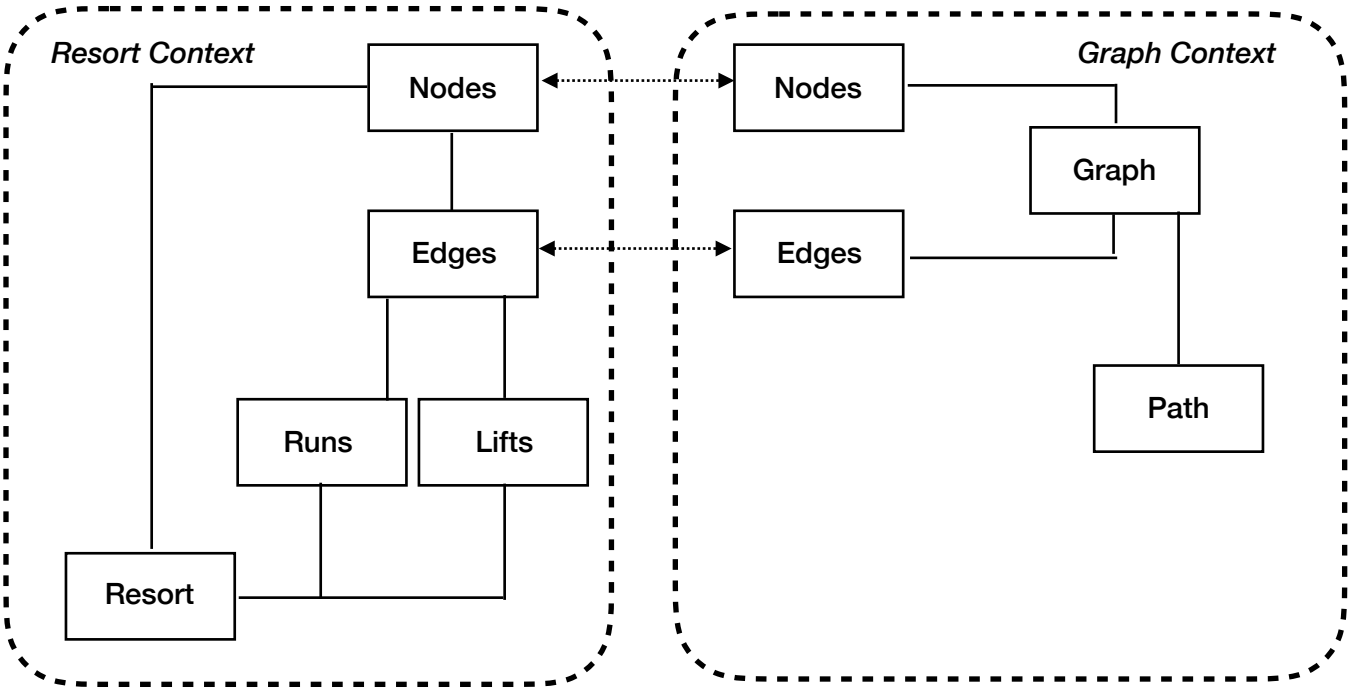

Fig 3.2. Context Map showing relationship between the resort context and the graph context

4. Requirement Engineering

4.1 Stakeholder Identification

A primary voice in requirements engineering is Sharp who identifies the four kinds of baseline stakeholders in her seminal paper *Stakeholder Identification in the Requirements Engineering Process* [12]. The four kinds of stakeholders are:

1. Users
2. Developers
3. Legislators
4. Decision-makers

Within category 1 are the end-users of my app; both the skiers and the ski patrol. I am the sole developer in this project and therefore make up the stakeholders in categories 2 and 4. Category 3 is not applicable for this project. Resort owners are also potential stakeholders in this project, especially for the DSML aspect of it. They could fall under categories 1 and 4.

I decided that in order to elicit more meaningful requirements, I would need to further distinguish between types of skier as this affects what the user would find useful in the app and also how they would use the app. Skiers are often split into three categories based on their ability: beginner, intermediate and experienced. Skiers at these different levels interact with the mountain very differently, therefore it is reasonable to assume they have different requirements from the app.

4.2 Ubiquitous Language

Evans prescribes using a Ubiquitous Language [9]. The idea is to use common terminology with developers, domain experts, etc. and even use the language in code. How we speak influences how we think so talking about the project in terms of the domain helps to keep the model consistent. I use this Ubiquitous Language to give meaningful names in my code. In this way, I am able to translate ideas more fluently between the domain and my model of it.

4.2.1 GLOSSARY

Here I define my Ubiquitous Language.

Run

A path on a mountain created to be skied down.

Lift

Something that takes a skier up the mountain. Examples include chairlift, gondola, button lift, etc

Run Level

The level of a run is typically given based on the angle of the slope. However, grading systems vary from region to region. Maintaining the different grading systems is important as the information on the app needs to reflect the user's region. Here I explain the two most common grading systems.

For North America, the following level definitions laid out in the Colorado Ski Safety Act [13]:

Green
: The easiest runs. These are usually groomed.

Blue
: More difficult runs. Sometimes groomed.

Black
: Most difficult runs.

For Europe, the grading system varies from country to country but generally the levels are:

Green
: Large, open, gently sloping areas at the base of the ski area or traverse paths between the main trails.

Blue
: Almost always groomed, or on so shallow a slope as not to need it.

Red
: An intermediate slope. Steeper, or narrower than a blue slope. Usually groomed, unless the narrowness of the trail prohibits it.

Black
: An expert slope.

Facility
: Refers to a facility that is available on the ski hill. For example restaurant or restrooms.

Groomed
: Groomed runs are those on which the snow has been packed down and smoothed.

4.3 Initial Domain Analysis

According to the core principles of DDD it is necessary to collaborate with domain experts when designing a domain model (see chapter 3). I consider myself a domain expert since I grew up in a ski town and have been skiing for nearly two decades. Referencing a wide selection of ski maps and keeping my Ubiquitous Language in mind, I began the first iteration of designing the domain model.

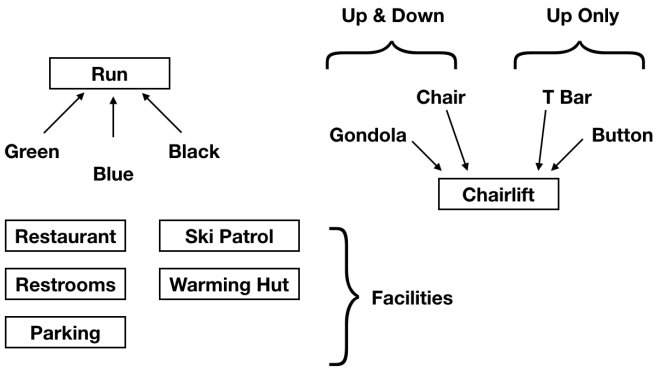

Fig. 4.1. Initial sketch of the elements of a ski resort

I started by sketching out ideas for an abstract syntax: what elements there are in a ski resort. Figure 4.2 shows my initial ideas. From here I had several decisions to make:

- What was an element and what was an attribute of an element.
- Can any of the elements be generalized into types.
- How do the elements relate to each other.

I decided that all the elements of a ski resort fall into one of three categories: runs, lifts and facilities. Each of these categories has its own set of attributes. Figure 4.2 shows my initial idea of what attributes a

chairlift and a run might have. One of the attributes of a facility would be ‘type’ which would be restaurant, restrooms, etc.

I then consulted with Alex Stewart, a peer working on a related project, who was also designing a DSML for a ski resort. We found our initial designs to be similar. We discussed the extent to which runs and lifts were the same. We also discussed whether runs should be represented as an area or a line.

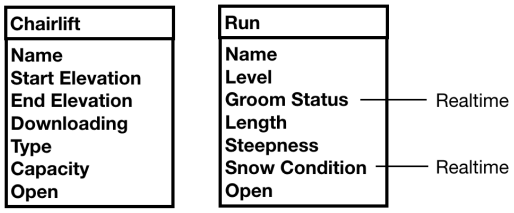

Fig. 4.2. Initial ideas for the attributes of Chairlifts and Runs

4.4 Interviews

4.4.1 PARTICIPANTS

Having identified a wide range of potential users in section 4.1, I wanted to avoid only thinking of requirements from my own limited point of view as an experienced skier. Therefore I decided an appropriate requirements elicitation technique would be to use stratified sampling and do interviews with someone from each subcategory of user. The participants are shown in Fig. 4.3 along with their skiing backgrounds.

Interviewee	Skiing Background
A	Beginner skier
B	Intermediate skier
C	Experienced skier
D	Skier who has been in an accident
E	Ski patrol

Fig. 4.3. Table of the interviewees’ pseudonyms and their skiing backgrounds

4.4.2 METHODOLOGY

The interviews were semi-structured. For the interviews with A, B, C I had them look at a few examples of traditional maps and asked them if they thought there were any shortcomings or unnecessary information. Then I explained my rough idea for the app and asked them what they would find useful in a skiing app. Finally I brought up the subject of emergencies and asked everyone if they knew what do in such a situation. For the interviews with D and C I did the same but also asked what would be useful in an emergency situation.

4.4.3 RESULTS

4.4.3.1 SHORTCOMINGS OF TRADITIONAL SKI MAPS

I asked all of the skiers if they had any issues with traditional ski maps (for example Fig 1.1), in particular if they contained any unnecessary information. A and D have both had difficulty navigating using solely traditional maps. A and B both said that there was extra information, for example topographical detail, that they found irrelevant. B said that although they are unable to ski black runs, they still want to know where they are on the map in order to avoid them. C said they liked the extra topographical information as it was useful for navigating off piste. A and D both have run into trouble using the assumption that the angle of a slope depicted on a traditional map corresponds to the direction it is supposed to be skied. A said they are nervous to go to parts of the mountain they are unfamiliar with for fear of not being to find their way back. By contrast, E said that ski maps aren’t perfect but getting a little lost and exploring the mountain is part of the fun!

4.4.3.2 LEVELS OF THE RUNS

A, B and D all said that the condition of the slope affects its difficulty. For example if a slope has been over-skied rocks may be exposed which increases the difficulty. A is only able to ski groomed runs. A also suggested that the difficulty of a slope is affected by the weather conditions. For example if a slope has become frozen and icy that will increase the difficulty.

4.4.3.3 INFORMATION ON FACILITIES

Warming huts

B and C have both skied to a warming hut only to find that it is closed.

Restrooms

A, B and D all agreed that being able to get to a restroom quickly is often a challenge when skiing, particularly in resorts they aren't familiar with.

Restaurants

B said that often they just want to ski to lunch and then down off the hill right after. B, C and D have all been frustrated with long lines at restaurants. C says they will sometimes just ski to another restaurant if the line is too long.

Lifts

All of the participants have stood in long lift lines and B and C said that they often choose to ski on parts of the mountain they think will be quieter. A said they often need to download (take the lift or gondola down to the bottom of the mountain). This is not possible on all lifts. Sometimes A has run into situations where, even though they are normally allowed to download, the lifts were too busy that day so they weren't allowed to.

4.4.3.4 USING AN APP WHILE SKIING

B said that the last thing they wanted to do when they were skiing, particularly if they were lost, is to take off their gloves and get their phone out. B and D said they would prefer to look at their phone on a chairlift but didn't mind using it while skiing. I asked all the interviewees how they felt about giving feedback on the app while skiing. A, B and C were all unenthusiastic about this. D said they would and thought it would be a good idea to give feedback on run conditions.

4.4.3.5 EMERGENCIES

E emphasized the importance of getting the correct location of the injured skier. Frequently when ski patrol is called in an emergency, the skier doesn't exactly know where they are. This means it takes longer for the ski patrol to find them which can be crucial if the injury is time sensitive. E also suggested that it would be useful to provide the ski patrol with basic information about the nature of the injury. This allows the ski patrol to arrive prepared. D said that when they had the accident, nobody knew what number to call to reach ski patrol. When asked if they know what number to call in an emergency, none of A, B or C knew.

4.5 Refined Domain Model

Reflecting upon the interviews, I realized that the goal of the app should not be to replace traditional ski maps, but rather to complement them. Android apps are limited in the screen size of a typical device so trying to fit an entire ski map onto a phone screen would not be useful. However, Android has many

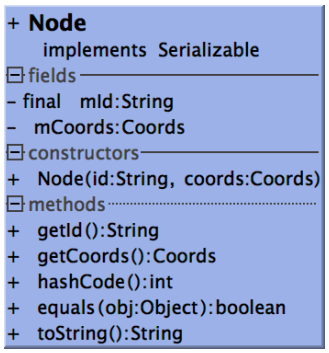

Fig. 4.4. Node Class

features that a traditional map does not, for example location services, so I wanted to take advantage of these.

Another criterion I considered in designing the DSML was ease of use and maintainability. Part of the motivation of designing the DSML is to make it easier for a ski resort to instantiate the app for their resort. Therefore the DSML needs to be easy to work with. Furthermore, the model of the resort defined with the DSML needs to be maintainable.

Taking all this into account I decided to use nodes as the basis of my domain model. Figure 4.4 shows the class diagram of a Node. A node represents a geographical point; it is defined in terms of latitude, longitude and altitude. On a ski hill, nodes are defined for the geographical points that all runs and lifts both start and end. In terms of a skier navigating the mountain, a node is any point at which a skier has to make a decision of which direction to go.

The use cases break into 3 categories:

- Directions (+ constraints)
- Information lookup
- Emergencies

The main functionality is giving a skier directions around the mountain. This can be done in terms of nodes. I decided at this point that runs and lifts should be subclasses of a superclass, Edge. A weighted directed graph can be built using the nodes and edges representing the ski resort. Then algorithms like Dijkstra's can be used to find routes from one node to another. This model is minimalist by design in its representation of a ski resort in order to make it more usable and maintainable for potential ski resorts.

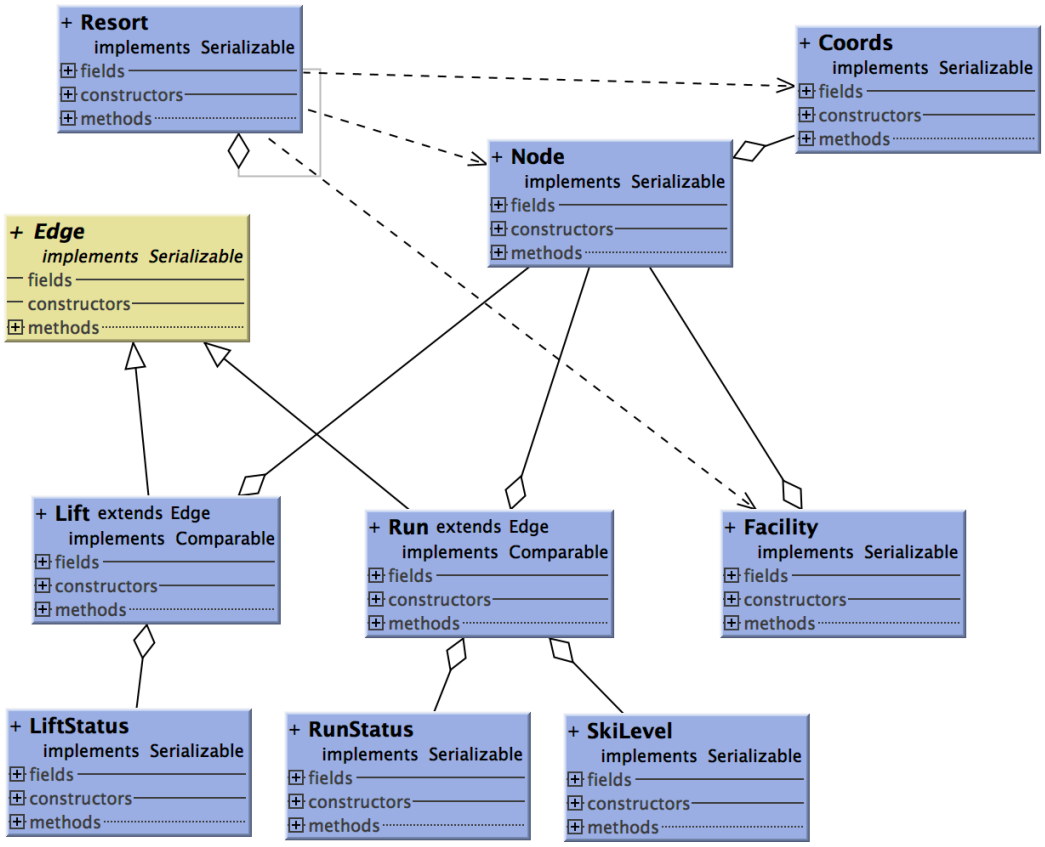

Fig. 4.5. Class diagram of my domain model

5. Implementation

In this chapter I introduce the technologies I used in developing the app in section 5.1 and then explain incremental approach I took to development in section 5.2. In the following sections I describe the implementation of each of my app's features.

5.1 Technologies Used

5.1.1 ANDROID APPLICATION FRAMEWORK

The Android application framework is a set of services that form the environment in which Android applications are run and managed. It includes the following key components:

5.1.1.1 ACTIVITY

An activity is a class responsible for managing how a user interacts with information in the app.

5.1.1.2 FRAGMENT

A fragment is a controller object that an Activity assigns to perform tasks, for example managing the UI.

5.1.1.3 LAYOUT

A layout is an XML file that defines a set of UI objects and their positions on the screen.

5.1.1.4 INTENT

An intent is an object that a component can use to communicate (for example an Activity) with the Android OS.

5.1.2 TARGET PLATFORM VERSION

Android has had a long-standing problem of fragmentation; many devices are still running on out-of-date versions of the Android operating system. So Android developers have to make a compromise between taking advantage of the latest APIs and having their app be backwards-compatible.

In making this decision, it was helpful to look at Android's dashboard for Platform Versions [14] which provides information on the distribution of the versions currently running. Figure 5.1 shows the latest distribution of platform versions; the darker the shade of green, the newer the version.

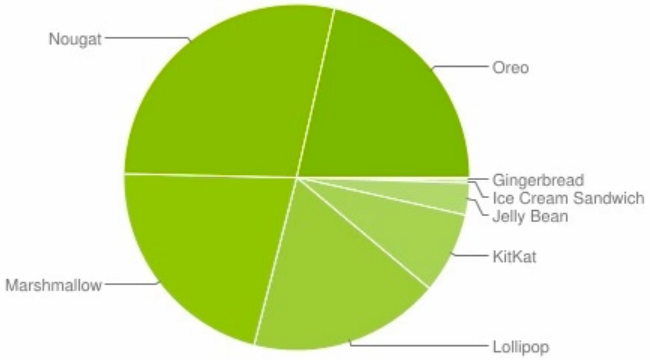

Fig 5.1. The relative number of devices running a given version of the Android platform.

Taking all this into consideration, I set my target version at 8.0 (Oreo) [15] which was the newest when I began developing the Android app and my minimum version at 4.4 (KitKat) [16] so that my app will support over 90% of active devices.

5.1.3 PROGRAMMING LANGUAGE

For a long time Java has been the main language used to develop Android applications, however, Google has recently released a new language for Android development: Kotlin. I decided to work with Java as there are a vast number of libraries and tools available for it, as well as more than a decade's worth of Java-specific Android resources. In addition, since I am taking a model-driven approach to this project I am using the Eclipse Modeling Framework. I factored this into my decision to use Java because EMF is based on the Java platform [17], producing sets of Java classes from models as allowing models to be specified using Java.

5.1.4 FIREBASE REALTIME DATABASE

The status of a run in a ski resort changes: the ski patrol can open or close a run; a run can also be groomed by the mountain crew or become un-groomed if it has been skied out or if there is fresh snow. It is important that my app keeps up-to-date with the status of all the runs in the resort. Typical ski resorts update their groom reports and run closures regularly, so I decided to create a backend where this data could be consolidated and used to keep the app updated in real-time.

I used Google's Firebase - Database [18] for the backend storage for my app. Firebase is a cloud-hosted NoSQL database which synchronizes data across all clients in real-time as the data changes. I chose Firebase because I wanted the Run Statuses to be updated in real-time. If a run is closed, skiers need to know this immediately. In addition, Firebase is a backend-as-a-service which means it provides my app with backend functionality without the overhead of writing APIs and managing servers.

To connect my app to Firebase, I created a new project in the Firebase console and then registered my app. I added the Firebase dependencies to my project as well as the google-services.json configuration file.

5.1.4.1 DATABASE SCHEMA

The Firebase Database is similar to a document-based storage database. Each document contains key-value pairs which are structured as JSON objects. Firebase's SDK has built-in serialization functionality which makes it possible to parse data straight into a POJO as long as the following rules are followed:

- Each variable name must match the name of the keys of the children nodes.
- Each member variable must be a valid JSON type
- An empty constructor must be provided
- Public getter methods must be provided for every member variable

Each entry in my Groom Report database corresponds to an instance of the RunStatus POJO. As you can see in code snippet 5.1, the RunStatus class follows these rules and corresponds to the schema of the Groom Report database.

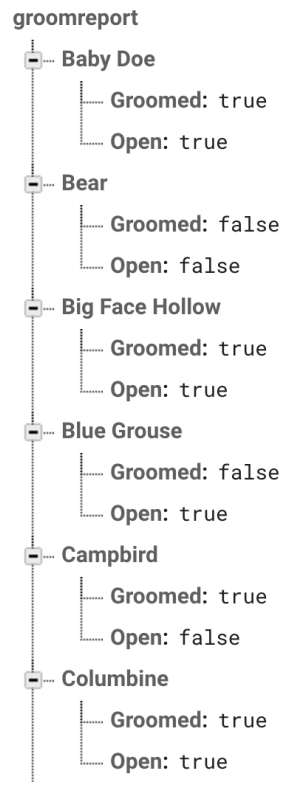

Fig 5.2. Snippet of the Groom Report database

```
public class RunStatus implements Serializable {

    private boolean Open;
    private boolean Groomed;

    public RunStatus() {
        Open = true;
        Groomed = true;
    }

    public boolean isOpen() { return Open; }

    public boolean isGroomed() { return Groomed; }

    public void setGroomed(boolean groomed) { Groomed = groomed; }

    public void setOpen(boolean open) { Open = open; }
}
```

Code Snippet 5.1

5.2 Increments

I wrote the code in many small increments, making sure to have a working app at the end of each increment. I ensured this through a combination of unit tests and manual emulator-based UI tests (as I explain in more detail in chapter 6). At the end of each increment I committed the changes to the GitHub repository for my project. I decided to take this approach in order to minimize risk. Firstly, it means I have working software early in the project. Secondly, it allows for flexibility as I change and refine the requirements. Finally, this approach makes it easier for me to test and debug my code as I go since I only have to focus on a small change with each increment.

5.3 Project Structure, Home & Settings

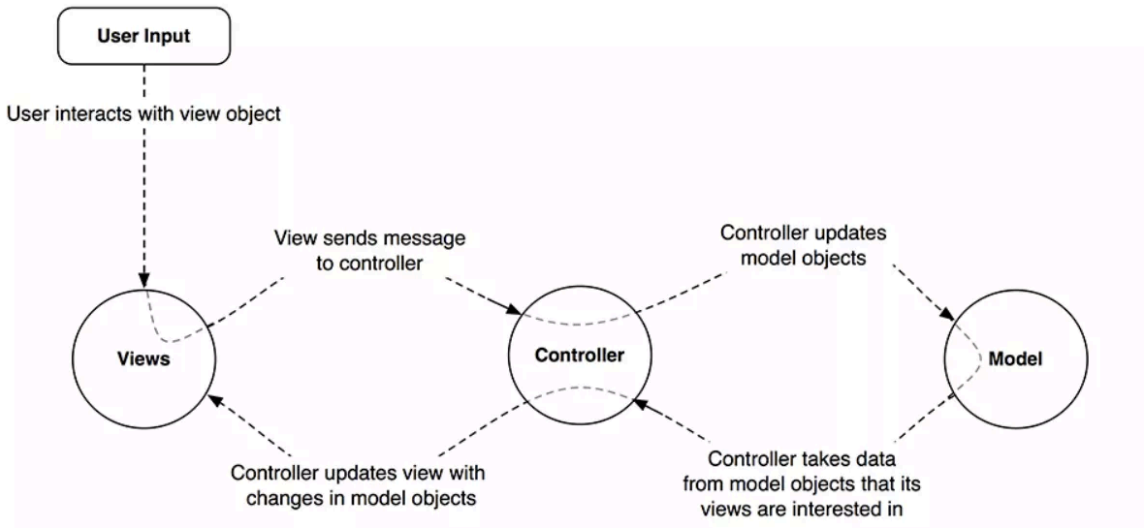

Fig 5.3 Flow of control in the MVC architecture (image from [19])

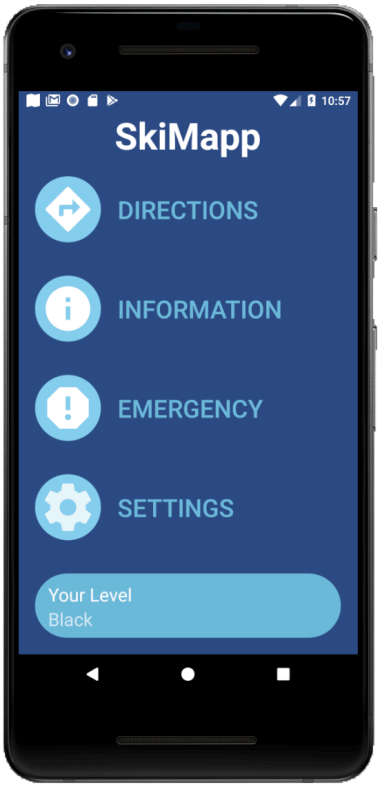

Fig 5.4. Screenshot of Home Page

5.3.1 MVC

I used the Model-View-Controller pattern and organize my application into three interconnected components:

•Model : The data layer which is responsible for handling business logic.

•View : The UI layer.

• Controller : The logic layer which ties the Model & View together and manages the application logic. It reacts to the user's behavior and updates the Model accordingly.

In Android applications, model classes are the custom classes which model the objects the application is concerned with. Android provides many view classes with view objects which know how to draw themselves on a screen and how to respond to user interaction. The controller objects in Android are usually a subclass of Activity or Fragment. Fig. 5.2 demonstrates the flow of control in the MVC architecture.

Using the MVC pattern produces several benefits due to its separation of concerns. The model classes have no reference to any Android classes and the controller classes don't extend or implement any Android classes which makes both of these easier to unit test. The MVC pattern also makes the code easier to extend.

5.3.2 SPLASH SCREEN

When the user first opens the app, the ski resort is loaded from its xml file. In order to improve performance, I do this in a background thread. However, most of the functionality of the app is based on this data so it needs to be completely parsed before the user can look up information, get directions or even set their level (because the grading system depends on the region which is defined in the xml file describing the resort). To overcome this issue, I added a splash screen to my app. I avoid using timers with the splash screen by using an AsyncTask[20] (see code snippet 5.2). The user sees the splash screen while the resort is being parsed in the doInBackground method and then as soon as that is complete, the onPostExecute method creates and Intent to start the home activity.

5.3.3 HOME PAGE

When the app is opened, the user is first taken to a home page, shown in Fig 5.4. From here the user can easily navigate to the different parts of the app.

5.3.4 SETTINGS

The user needs to be able to set a value for their ski ability. I decided ski ability should be treated as a user configurable setting since it affects the behavior of the application. The user is able to update this value at any time and the app will remember the value that was set when it is reopened. Using the AndroidX Preference Library [21] I defined a Preference hierarchy in xml.

The value of the skier's ability corresponds to the most difficult grade of run they are able to ski. However, as explained in 4.2.1, grading systems vary. To accommodate for this, I have defined a different

Fig 5.5. Screenshot snippet from Settings

Fig 5.6. Screenshot snippet from Settings

Preference hierarchy for each grading system. When the resort is loaded from the DSML, the region is set and the corresponding grading system is used. I had to define multiple Preference hierarchies because the different grading systems are not directly convertible.

In the Settings, the levels are all given brief explanations to help the use chose the correct level. These are shown for the American levels in Fig. 5.5 and the European levels in Fig 5.6.

5.4 Resort Model

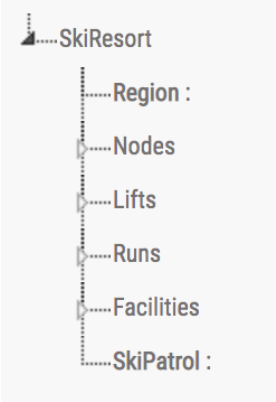

Fig 5.7. XML Schema of the Resort meta-metamodel.

5.4.1 XML AS THE META-METAMODEL

I wanted to use a meta-metamodel that was derived from an existing language to define my DSML so I decided on an XML schema. XML is extensible which allowed me to create custom tags in order to describe all the components of a resort. Furthermore, there are many libraries available for parsing XML. Any ski resort can be represented in this format, each resort as an XML document with the schema shown in Fig 5.7. I store these documents in the assets directory (as opposed to the resources directory). The set of these resort XML documents changes, so I wanted to avoid having to declare a list of all their resource IDs. Assets are read-only at runtime and can be navigated like a typical Java filesystem.

5.4.2 RESORT

My Resort class is a singleton as I wanted my app to only ever have a single instance of a Resort. It is responsible for its own creation and initialization which occurs only when the app is opened. It stores a list of all the geographical nodes of the actual resort and provides a global point of access to all the Resort's lifts, runs and facilities which are defined in terms of these nodes. Making Resort a singleton also allows me to easily pass its data between controller classes.

5.4.3 PARSING

When the instance of Resort is created, the XML document describing the resort is parsed into my resort model. To do this I create an input stream from the file and then use the JavaX XML parser libraries[22] to create a DOM document[23]. First I parse all the nodes into the Resort as these are the fundamental structure of which the other elements of the resort are defined in terms of. I can then parse the lifts, runs and facilities using these nodes. I sort these alphabetically, combine the runs and lifts into a superset of edges, and then add them to the Resort along with the emergency information.

5.4.4 LIVE STATUS UPDATES

As I explained in section 5.1.2, I have created a backend which the ski patrol and mountain crew can use to update the status of runs.

In my app, every instance of Run has a ValueEventListener which listens for data changes on that particular run. I only want to update an individual run when that particular run changes as opposed to updating all of the runs when any of them change. To do this, each instance of Run has a reference to its particular child in the database and the ValueEventListener is added to that reference. When the data changes for a run, the RunStatus of that Run is immediately updated accordingly.

5.5 Information

One of the key use cases of this app is looking up information about a resort. This can be split into information about lifts and information about runs. For the layout of the information fragment, I display two scrollable lists, one of runs and one of lifts, as shown in Fig 5.8. Each item in the list can be selected to reveal its specific information as seen in Fig 5.9.

I achieve this using two RecyclerViews[24], one for runs and one for lifts. Each item in the list is an instance of a ViewHolder object, so I created two classes which extend RecyclerView.ViewHolder; RunHolder and LiftHolder as well as a layout XML for each which define how the items are displayed. For each run, an icon indicating its level is displayed next to the name. Each of these ViewHolders needs to be managed by an adapter, so I created two classes, RunAdapter and LiftAdapter, which both extend RecyclerView.ViewAdapter. The adapters assign each ViewHolder to a position and then bind it to its data based on this position. The instance of Resort stores all its runs and lifts in alphabetical order, so the lists are also displayed in alphabetical order, making it easier for a user to find a particular one. Only the visible view holders are created. If the user scrolls down the list more are created as necessary.

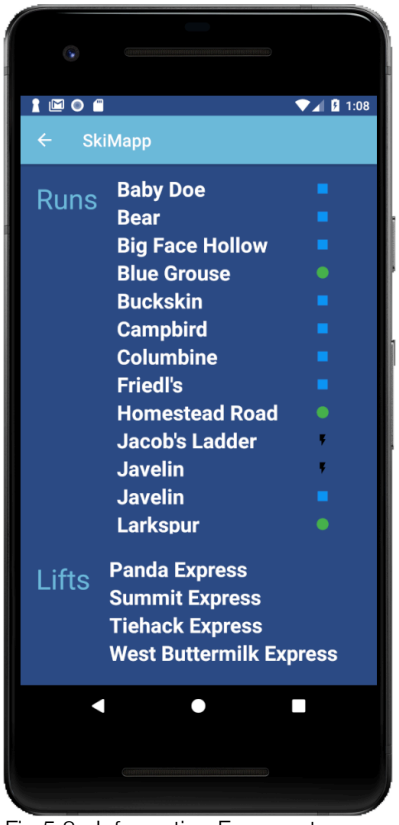

Fig 5.8. Information Fragment

The holders implement OnClickListeners which open a DialogFragment when selected. This dialog opens a floating screen in front of the Information fragment which displays information specific to the selected run/lift, without having to change activity. The run that was clicked is passed to the DialogFragment which then extracts information about it to display.

I chose to use RecyclerView because it is much more customizable than ListView, not only for creating layouts but also OnClickListeners.

5.6 Route Finding

5.6.1 SKIER'S LOCATION

I created a wrapper class, SkiersLocation, to encapsulate the logic for getting the user's location both in terms of the real world of the ski resort model. I use a LocationListener to keep the instance of SkiersLocation updated as the user's location changes.

In order to access the user's location, I need their permission to do so. I have publicized this in the application's manifest. After opening the app for the first time, the first time the user performs an action that requires access to their location I request permission to do so.

5.6.1.1 LOCATING SKIER WITHIN MY MODEL

To see whether the skier is at a given node, I check if the latitude and longitude of that node is within a 5 meter radius of the skier's location. To see where a skier is on the mountain, I run through all the nodes checking if they are at that node until I either find the node they are at, or determine that they aren't at a node.

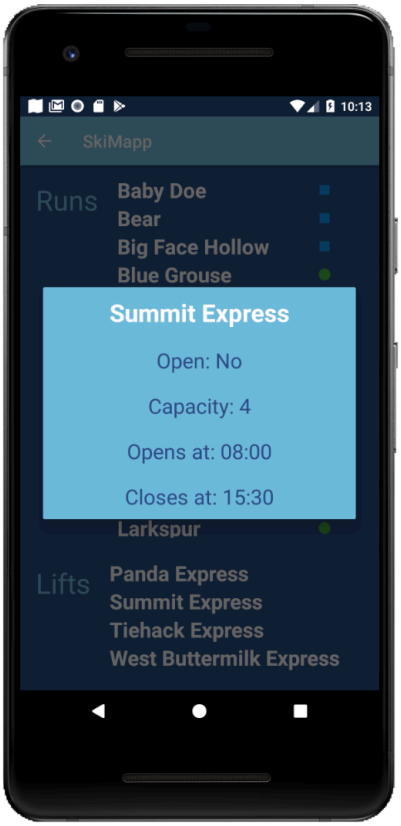

Fig 5.9. Screenshot of Lift Dialog

5.6.2 GRAPH

One of the goals for the functionality of this app was to calculate routes around the resort that are customized to the user's ski ability. I decided to do this by building a directed graph from the nodes and edges of my resort model but omit any runs (an instance of edge) that are of a level the user isn't able to ski. So if the user has set their level as 'Blue' I build a graph using all the lift edges, green edges and blue edges, but no black edges. In most cases, the digraph will be strongly connected[1], however, in a few cases removing runs could create a weakly connected[2] digraph. This won't result in the skier getting stranded unless they specifically request to be directed to the end of a weakly connected edge.

By using a graph, not only am I able to create subsets of a resort based on ski ability, but I am also able to produce directions using graph-based route finding algorithms like Dijkstra's.

5.6.2.1 RUN SEGMENTS

Runs often have midpoints. A midpoint is any node that the run crosses besides its start and end nodes. In terms of a ski resort, a node is a decision point: a point at which a skier can chose to go down one run or another, or even take a lift. Often a skier may only complete a segment of a run and then join onto another run. So for the purposes of route finding, each edge between nodes is added to the graph separately. This means that if a run has any midpoints, it will be split into segments when the graph is generated.

The generated graph consists of two linked lists:

- `LinkedList<Node> mNodes`
- `LinkedList<Edge> mEdges`

Node and Edge here refer to the classes I defined in my resort model (as opposed to the Nodes and Edges of some graphing library). The edge list is composed only of minimal edge segments. I use linked lists because they allow for constant-time insertions or removals. I use these operations frequently when adjusting the graph to the user's ski ability and also in my implementation of Yen's algorithm (see section 5.6.3.1) so I chose a data structure that allows me to do so efficiently.

5.6.3 ROUTE FINDING

For basic route finding, I use Dijkstra's algorithm. Dijkstra's algorithm has time complexity $O(N^2)$ but my implementation uses a Fibonacci heap which reduces this to $O(M + N \log N)$ where M is the number of edges in the graph.

```
private class ExampleAsyncTask extends AsyncTask</* Params, Progress, Result */> {

    @Override
    protected /* Result */ doInBackground(Params...) {
       // Perform background computation
       ...
    }

    @Override
    protected void onPostExecute(/* Result */ result) {
        // invoked on the UI thread after the background computation finishes
       ...
   }
}
```

Code Snippet 5.2

[1] A digraph is *strongly connected* if every node is reachable from every other by following the directions of the edges.

[2] A digraph is *weakly connected* if it is not strongly connected but would be connected if it were considered as an undirected graph.

The route calculating algorithms, especially for longer routes, are computationally expensive and therefore shouldn't run on the main thread as they might block it making the app unresponsive. To overcome this issue, I initially decided to run the route calculating algorithms in a background thread using an AsyncTask [20]. An AsyncTask has the structure shown in code snippet 5.2. The method onPostExecute doesn't execute until the route calculation, which is done inside the doInBackground method, is complete. It takes a long time to calculate the route, the app will just do nothing and wait when the user expects to be taken to NavMode.

To overcome this I added a timeout. So instead of using Android Framework's AsyncTask, I extend Scott Tomaszewski's AsyncTaskWithTimeout [25].

5.6.3.1 LONGER ROUTE

Most of the time, skiing isn't about getting from A to B as quickly as possible. Rather it's about maximizing the skiing. So I wanted to create an option for finding a longer route with more skiing.

My initial idea was to find the longest path. However, the graph that represents a ski resort is not acyclic which makes the longest path problem is NP-hard [26]. Furthermore, the longest path could potentially span over the entire mountain which isn't typically a desirable route for a skier. So instead I wanted to come up with a way to calculate a route that is longer than the shortest path returned by Dijkstra's, but not excessively long. I also wanted the paths to be loopless. To do this I implemented Yen's K-shortest paths algorithm [27] using Dijkstra's algorithm as the basis. My implementation is shown in code snippet 5.3.

The time complexity of Yen's is $O (KN (M + N \log N))$ [28] and the space complexity is $O (N^2 + KN)$ [29]. As there is a trade-off between obtaining a longer path and computational complexity, I experimented with different values of K and decided K = 4 was a good compromise.

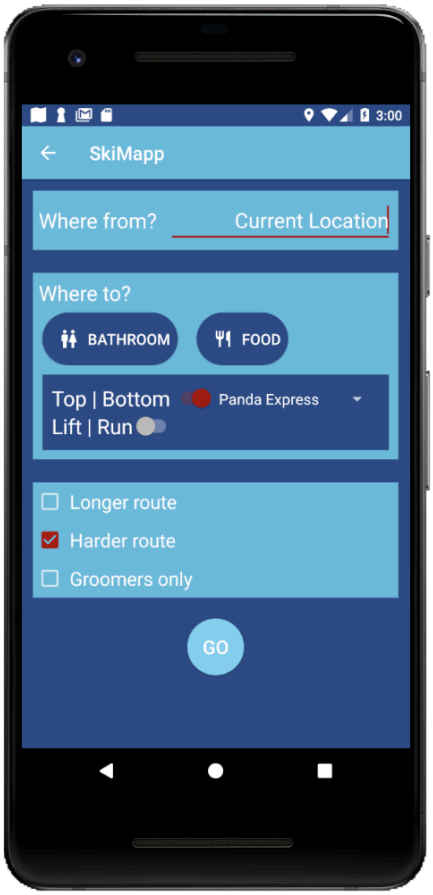

Fig. 5.10 : Screenshot showing route selection view

5.6.4 ROUTE OPTIONS

When selecting a route, the user is given several options to customize this route to their preferences. Besides having the route tailored to their ski ability and having the option of being shown a longer route, the user can also choose to have a harder route or a route with only groomed runs. If either of these options are selected, the graph that is passed to the route finding algorithms first needs to be modified accordingly. I use an AsyncTask[20] to first create a modified graph and then once this has been completed I pass this graph into the second AsyncTask (see section 5.6.2.1) which calculates a route from this now modified graph.

5.6.4.1 HARDER ROUTE

If the user selects the “Harder route” checkbox as shown in Fig. 5.10, then when their route is being calculated, preference will be given to runs of a higher level. I achieve this by modifying the graph of the resort in the following way:

- increase the weight of green runs eightfold
- increase the weight of blue runs fourfold
- leave the weight of black runs as is

```
1.  public List<Path> YenKSP(Graph graph, Node source, Node sink, int K) {
2.      List<Path> shortestPaths = new ArrayList<>();
3.
4.      Dijkstra dijkstra = new Dijkstra(graph);
5.      dijkstra.execute(source);
6.      shortestPaths.add(dijkstra.getPath(sink));
7.
8.      if (shortestPaths.get(0) == null) {
9.          return shortestPaths;
10.     }
11.     PriorityQueue<Path> potentialPaths = new PriorityQueue<>();
12.
13.     for (int k = 1; k <= K; k++) {
14.         int d = shortestPaths.get(k - 1).getDistance() - 2;
15.         for (int i = 0; i <= d; i++) {
16.             Node spurNode = shortestPaths.get(k - 1).getNode(i);
17.             Path rootPath = shortestPaths.get(k - 1).getNodes(0, i);
18.
19.             for (Path p : shortestPaths) {
20.                 if (rootPath.equals(p.getNodes(0, i))) {
21.                     graph.temporarilyRemoveEdge(p.getEdgeFromNodes(
22.                        mResort, p.getNode(i), p.getNode(i + 1)));
23.                 }
24.             }
25.             for (Node rootPathNode : rootPath.getNodePath()) {
26.                 if (rootPathNode != spurNode) {
27.                     graph.temporarilyRemoveNode(rootPathNode);
28.                 }
29.             }
30.             Dijkstra dijkstraSpur = new Dijkstra(graph);
31.             dijkstraSpur.execute(spurNode);
32.             Path spurPath = dijkstraSpur.getPath(sink);
33.             Path totalPath = new Path(rootPath.getNodePath());
34.             if (spurPath == null) {
35.                 break;
36.             }
37.             totalPath.joinPath(spurPath);
38.             potentialPaths.add(totalPath);
39.             graph.restoreEdges();
40.             graph.restoreNodes();
41.         }
42.
43.         if (potentialPaths.isEmpty()) {
44.             break;
45.         }
46.         shortestPaths.add(potentialPaths.poll());
47.     }
48.     graph.restoreNodes();
49.     graph.restoreEdges();
50.     return shortestPaths;
51. }
```

Code Snippet 5.3

Dijkstra's algorithm finds the shortest route by comparing the weight of edges in a graph and closing those with less weight. By increasing the weight for easier runs, this gives preference to harder runs when calculating a route.

5.6.4.2 GROOMERS ONLY

If the user selects the "Groomers only" checkbox as shown in Fig. 5.10, then a route will be calculated that only contains groomed runs. I achieve this by removing any run edges from the resort graph which are not currently groomed. In order to prevent concurrent modification, I use an Iterator to remove these edges safely. The resulting graph only contains lifts and groomed runs. This graph is then passed to the route finding algorithms.

5.6.5 REJOINING A PATH CONTAINING FRAGMENTS

```
Node startNode = path.getNode(0);
Node penultimateNodeInPath = path.getNode(path.getNodePath().size()-2);
Node finalNodeInPath = path.getNode(path.getNodePath().size()-1);

while (startNode != finalNodeInPath) {
    int currStartIndex = path.getNodePath().indexOf(startNode);

    String edgeFromStart = resortGraph.getEdgeBetweenNodes(startNode,
path.getNode(currStartIndex +1 )).getName();
    // This is the name of the edge between startNode and its adjacent node

    // Compare the name of this first edge to the name of its adjacent edges
    for (int i = currStartIndex+1; i < path.getNodePath().size()-1; i++) {
        int j = i+1;
        String nextEdge = resortGraph.getEdgeBetweenNodes(path.getNode(i),
path.getNode(j)).getName();

        if (!nextEdge.equals(edgeFromStart)) {
            mEdgeNamePath.add(edgeFromStart);  // set end of current edge as i
            mEndNodes.add(path.getNode(i));    // add edge to SkiRoute
            startNode = path.getNode(i);       // set startNode to node at i
            break;                             // exit the for loop
        }

        // check if at final node
        if (j == path.getNodePath().size()-1) {
            mEdgeNamePath.add(edgeFromStart);
            mEndNodes.add(path.getNode(j));
            startNode = path.getNode(j);
        }
    }

    if (startNode == penultimateNodeInPath) {
        String finalEdge = resortGraph.getEdgeBetweenNodes(penultimateNodeInPath,
finalNodeInPath).getName();
        mEdgeNamePath.add(finalEdge);
        mEndNodes.add(finalNodeInPath);
        startNode = finalNodeInPath;    // exit while loop
    }
    mCompleted = new boolean[mEdgeNamePath.size()];
}
```

Code Snippet 5.4

The Paths returned by the route-finding algorithms contain edges which are run fragments (see 5.6.1.1). If the Path contains several consecutive run segments which belong to the same run, these need to be rejoined into a single edge, otherwise the directions would be in terms of edge segments as opposed to runs which isn't as understandable for the end-user. For example, a path like "Summit Express -> Ridge Trail -> Ridge Trail" should be displayed to the user as "Summit Express -> Ridge Trail". This was complicated as often the rejoined edges are still only a subset of the entire run. I overcame this by defining a new class, SkiRoute, which allows a route to be defined in terms of rejoined run segments. I created the algorithm shown in code snippet 5.4 to convert a Path into a SkiRoute.

5.6.6 FACILITY FINDER

The idea behind FacilityFinder is, given a type of facility and a start node, to find the shortest path to the nearest facility of the specified type. In the "Where to" section of my Directions fragment I have two buttons, Restaurant and Restroom, which when pressed give the user the quickest route to the selected facility.
I achieve this with the following algorithm:

- For every facility in the resort:
  - If that facility is of the correct type or contains the correct type:
    - First I check if that facility is at the start node. If it is, I return a Path consisting just of the start node
    - If not, I calculate the shortest Path from the start node to the facility. If it is shorter than the shortest route, I update the shortest route
- Finally, I return the Path in the shortest route

5.6.7 UI FOR SELECTING A DESTINATION

Creating the user interface for the Directions fragment took several iterations.

5.6.7.1

In the first iteration, I wanted to create something simple just so I could see if the route finding logic was working correctly. So I simply had two text inputs which took the IDs of the start and end nodes, a 'Go' button and then once the route was calculated it was displayed as a list underneath the other features. Of course I can't expect a user to know the ID of the node where they want to go so I needed to come up with a better user interface for choosing a destination.

5.6.7.2

For navigation apps like CityMapper[7] that are designed for cities, the user usually just types the address of the place where they want to go. However, in a ski resort, a spot on the mountain doesn't have an address. To gain more insight, I did some informal user evaluation, asking a selection of people the following questions:

- *How do you choose a destination to ski to?*
- *What do you describe this destination in terms of?*
- *What is a typical destination on a ski mountain?*

The majority of people select a destination by pointing to it on a map. When they are asked to describe it, they typically did so in terms of runs and lifts, for example "At the top of Summit Express". The answers to the third question showed me that the destinations chosen are usually at the top or bottom of a lift, occasionally the top or bottom of a run, and never at an arbitrary point on the mountain.

5.6.7.3

My next idea was based on the responses from the second and third questions in 5.4.1.2: a drop-down menu of all the lifts and runs and a toggle to select either top or bottom. This means the selected destination will correspond to a node. I created an initial implementation of this new user interface and then did some more informal user evaluation, asking a few people to test it out and give me feedback on the usability. I then adjusted the layout according to their suggestions.

The final version can be seen in Fig. 5.10 and an example of the drop down menu is shown in Fig. 5.11.

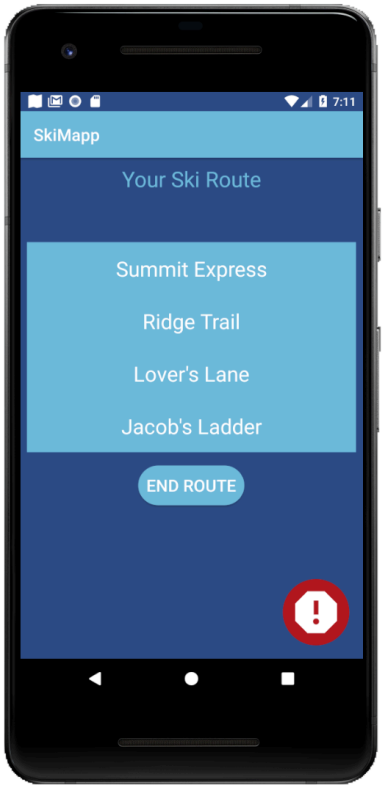

Fig 5.12. Navigation mode.

5.6.8 NAV MODE

After the user presses go and their directions are calculated, they are taken to a new fragment which displays the route they need to follow as a list, as shown in Fig 5.12.

To help make following the directions displayed in NavMode easier for the user, I wanted to visually indicate what has been completed in real time. When a user has skied to the bottom of a run or arrived at the top of a chairlift, that section of the directions is faded to grey to indicate it is complete, as shown in Fig. 5.13. This makes it easy for the user to see where they need to go next.

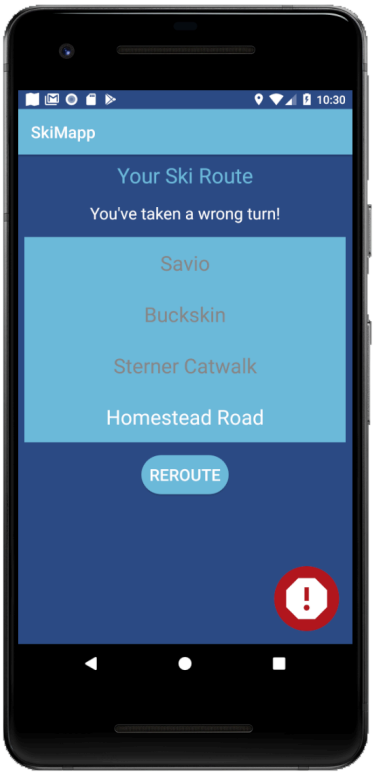

Fig 5.13. Navigation Mode

Every time the skier arrives at a new node on the map, I check to see if it is the end node of the section they should be on. If it is, this indicates they have completed this section and I fade that part of the directions to grey and update which is the current section.

I discovered from my interviews in section 4.4.3.4 that most people don't want to use their phone while skiing. To address this, I use Android's Text-To-Speech API [30] to announce directions. This allows users to keep their gloves on and their phones in their pockets while they ski, but still use the app to navigate. When the skier arrives at a node, the next run or lift they need to take is announced. I also take into account the region the user is skiing in and configure the Text-To-Speech language accordingly. I make the assumption that the language will always be English as the app itself is in English but I configure the Text-To-Speech accent to match the region.

I also check if the skier has taken a wrong turn. If the skier is following the directions correctly, once they have left a correct node, the next node they should arrive at should be either the start node of the next section in the route, or a midpoint of the current run (as explained in section 5.6.2.1). If they arrive at another node it means they have taken a wrong turn at the last correct node. When this happens, the app alerts them that they have taken a wrong turn and offers the option to reroute (see Fig 5.13).

The most likely time for a skier to have an accident or come across an accident is while they are skiing. So I wanted to make it as easy and quick

as possible for a user to get to the emergency page from the navigation fragment. I put a big red button in the bottom right corner of the screen (see Fig. 5.12) which will take the user to the emergency fragment.

5.7 Emergency

An important use case of my app is to help a skier in the case of an emergency. The two things the app needs to provide in such a situation is the correct phone number of the local ski patrol and the exact location of the accident so the ski patrol can get to the right place as quickly as possible.

I implement this straightforwardly in a fragment which retrieves and displays the user's exact location and the emergency phone number.

6. Testing

Testing played a vital role in the development of this app. It was important for me to verify my app's functional behavior, usability and performance.

I based my approach to testing on Mike Cohn's Testing Pyramid [31] which is depicted in Fig. 6.1. However in practice much of my app required testing that involved movement and location and as a result I still had to do a lot of manual testing.

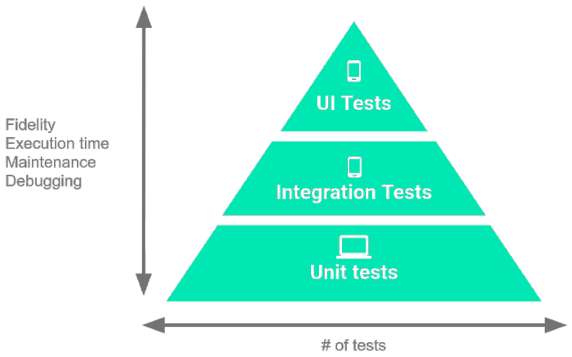

Fig 6.1 . The Testing Pyramid (image from [32])

6.1 Unit Tests

As I added features while developing the app, I ensured that these behaved correctly using unit tests. There were some methods in my app I wasn't able to test in this isolated manner due to the nature of mobile applications but I made sure to test the business logic at a unit level.

For each unit test, I aimed to cover all possible interactions with it; standard interactions as well as invalid inputs and cases where resources aren't available. Using unit tests allowed me to test functionality much quicker than if I were to perform emulator-based tests.

Several of these tests require interaction with the Android Framework, so I used Robolectric[33] which allowed me to run the tests on my local JVM while still using the Android API.

The code snippet below shows an example of a test that uses a Context object retrieved from Robolectric.

```
@RunWith(RobolectricTestRunner.class)
public class ResortTest {

    private Context context = ApplicationProvider.getApplicationContext();
    private Resort resort = Resort.get(context);

    @Test
    public void emergency_test() {
        assertEquals(resort.getEmergencyNumber(), "970-920-0969");
        ...
    }
    ...
}
```

6.2 Integration Tests & System Tests

After testing each unit of my app, I needed to verify that components work well together. At this stage I am not testing the full app but rather if the app is able to coordinate certain units. My app is dependent on the GPS hardware component of an Android device, so it was important for me to test the components that depend on this.

Fig. 6.2. Traditional map of the Buttermilk ski area. [34]

6.2.1 TESTING THE DSML WITH THE APP

In order to do this well, I wrote up an actual ski resort in my DSML. This is a time-consuming process (the XML file for this resort was 704 lines long) so I chose one that wasn't too big but still had sections in the traditional map that could cause confusion in the following ways:

- Sections of run where the directionality is unclear.
- Runs where it is unclear if it is possible to ski to a particular lift.
- Areas where the easiest route down isn't the most obvious route.
- Parts of the mountain where it isn't clear where the nearest facilities are.

Based on these criteria I chose Buttermilk mountain, whose traditional ski map is shown in Fig 6.2. In addition to the above criteria, I also wanted to choose a mountain which I am very familiar with so that I can verify that my app is working correctly.

Writing the entire ski resort in terms of my DSML gave me the opportunity to evaluate my model. I was mostly able to describe the ski resort well in terms of my model. The only issue I ran into was the midway loading station on West Buttermilk (see Fig. 6.2). This meant I needed to update my model to accommodate midpoints, not just in runs but also in lifts. However the nature of midpoints for lifts is slightly different since the midpoint could either be an option to get off the lift or to get on the lift or even both in the case of a gondola. Upon further investigation I decided that if a lift has a midpoint it falls into one of three categories:

- A loading station
- An unloading station
- Both loading and unloading

The category of the lift's midpoint affects the calculation of directions. If midpoint allows both loading and unloading it can be treated in the same way as a midpoint in a run (see section 5.6.2.1). If the midpoint only allows loading, the when the lift is decomposed into its fundamental edge segments, these will consist of:

- An edge from the bottom of the lift to the top of the lift
- A second edge from the midpoint to the top

If the midpoint only allows unloading, the edge segments will be:

- An edge from the bottom to the midpoint
- An edge from the bottom to the top

Writing the resort in my DSML also confirmed the confusing nature of traditional ski maps. I chose a resort I am very familiar with so I knew from experience how the mountain is actually laid out, but there were many occasions where it was unclear on the map where runs started and ended and a few runs where the directionality wasn't clear.

I checked that XML file describing the resort was parsed correctly into my app's resort model by checking the following:

- The number of runs, lifts & facilities is correct
- I chose a random selection of 10 runs and lifts and checked for each one that the information displayed by my app matched the information in the DSML.

6.2.1.1 TESTING ROUTES CALCULATED WITH VARIABLES

I tested all combinations of variables that a user could set for a route. For each combination I tested three different start and end combinations and checked each of the routes produced against the following criteria:

1. Route is valid
2. Route is within skier's ability
3. Route conforms to the additional criteria.

To judge the first criterion I used my knowledge of Buttermilk ski area and the traditional map to ensure the routes were indeed valid. Judging the second criterion I simply checked that the level of each run in the route was within the set ability. To judge the third criterion for longer and harder route options, I compared them to the route produced with no options selected. To judge the third criterion for the groomers only option, I checked the runs against the current groom status. My results are displayed in Table 6.3.

It is not possible to produce a harder route if the user's ski ability is set to green. This is indicated in Table 6.3 by N/A.

Table 6.3

	Green			Blue			Black		
	1	2	3	1	2	3	1	2	3
Nothing selected	☑	☑	☑	☑	☑	☑	☑	☑	☑
Longer route*	☑	☑	☑	☑	☑	☑	☑	☑	☑
Harder route	☑	☑	N/A	☑	☑	☑	☑	☑	☑
Groomers only	☑	☑	☑	☑	☑	☑	☑	☑	☑
Longer & Harder	☑	☑	N/A	☑	☑	☑	☑	☑	☑
Longer & Groomers	☑	☑	☑	☑	☑	☑	☑	☑	☑
Harder & Groomers	☑	☑	☑	☑	☑	☑	☑	☑	☑
Longer & Harder & Groomers	☑	☑	N/A	☑	☑	☑	☑	☑	☑

*For the longer route option, there were several routes where the algorithm didn't return a route that was longer than the standard route. This can happen on routes where the start and end nodes lie along "dead-ends", in other words there is only a single edge out of the start node or only a single edge into the end node. This results in no spur paths being able to be created which can be seen in lines 43-45 of code snippet 5.3. This is more common in a graph when the skier's ability is more limited. When this happens, the route returned is just the standard shortest route which I considered valid.

6.2.3 TESTING LIVE RUN STATUS

For testing the live run status updates, I ensured the following behaviors were working correctly:

- Only the run whose status was changes updates.
- When a field is updated in the cloud, it is immediately updated in the app.
    - This should work when the app is in any activity.
    - The fields are updated correctly.
    - If an invalid entry is put into the database, the app won't crash and won't update the RunStatus to the incorrect value.

6.2.4 ROBO TESTS

I wanted to test how well my app worked on different of screen sizes, hardware configurations and locales. I did this by running a series of Robo tests using the Firebase Test Lab[35]. Robo tests analyze my app's interface and then explore it automatically by simulating a series of user activities. I ran tests for API levels 19 - 28, a wide range of screen sizes, and also two different locales: United States and Europe.

6.2.4.1 RESULTS

In the first set of Robo tests that I ran, I had a failure due to a NumberFormatException error. This showed me that I needed to check the input in the 'Where to' section of the directions

fragment more carefully. After fixing this, I had no test failures which means my app works on a wide range of devices.

The averages for the graphics stats and the time to initial display are shown in the table below.

Graphics Metric	***Explanation** (from [36])*	**Before Optimizations**	**After Optimizations**
Missed VSync	The number of missed Vsync events, divided by the number of frames that took longer than 16 ms to render.	10%	7%
High input latency	The number of input events that took longer than 24 ms, divided by the number of frames that took longer than 16 ms to render.	1%	0%
Slow UI thread	The number of times the UI thread took more than 8 ms to complete, divided by the number of frames that took longer than 16 ms to render.	13%	6%
Slow draw commands	The number of times that sending draw commands to the GPU took more than 12 ms, divided by the number of frames that took longer than 16 ms to render.	3%	3%
Slow bitmap uploads	The number of times that the bitmap took longer than 3.2 ms to upload to the GPU divided by the number of frames that took longer than 16 ms to render.	0%	0%

To achieve this performance improvement, I moved some computationally expensive methods off the UI thread and into background threads. In order to do this I had to create a splash screen (as explain in 5.3.3). These Robo tests also showed that the average time the splash screen was displayed for was 0.02 seconds. This performance improvement can also be seen in the decrease of the average time to initial display from 442ms to 261ms.

6.3 User Acceptance Tests

It is also important to test common workflows that involve the entire application stack. I wanted to make sure that everything from the user interface down to the data layer was working together correctly.

6.3.2 MANUAL TESTS WITH A LOCATION SIMULATOR

A large part of my app's functionality is based on using the user's location on the mountain. I tested these features using a location simulation app, Lockito [37]. I chose this app as it allows me to create a fake ski itinerary with control over the speed and altitude.

Using this location simulator, I tested the following location-dependent features:

6.3.2.1 LOCATING THE SKIER WITHIN MY MODEL

I used Lockito to place the skier at various different points around the mountain and then used my app to locate them in terms of my DSML model. It worked well when the skier was directly on a node and also when the skier was near a node, always displaying the correct node. This confirms that the nodes of ski resort were defined correctly in my DSML and also that the logic that locates the skier within the model using Location Services works correctly.

Currently the app is only able to locate the skier in the model if they are at a node. Runs aren't straight lines but rather two dimensional areas that often curve. My model doesn't capture this meaning I am unable to locate which run a skier is on while they are skiing it. I don't think this is a flaw since the edges of runs in a ski resort are clearly marked, so while a skier is on the run, all they have to do is ski down. As soon as they arrive at a decision point, that is where there is potential for them to get lost, but my app is able to locate them well at these decision points. I tested that my app behaves correctly in this way by checking that when the skier was not at a node, the app didn't place them at a node incorrectly.

6.3.2.2 UPDATING DIRECTIONS AS SKIER COMPLETES ROUTE

In order to test this feature, I generated five different sets of directions using my app and then created a ski itinerary in Lockito that followed each of these directions. I wanted to cover a range of different routes including routes of different length, routes that involved multiple lifts and routes that included run sections that were very short. The behavior I was expecting was that once a section of the route is completed, it is faded out in the directions list. For all of the routes I tested, the app behaved correctly, only fading the section out after it was completed.

6.3.2.3 FACILITY FINDER

To test the facility finder, I used Lockito to simulate the user being at several locations around the mountain. These included several random locations as well already being at a facility, being at the wrong type of facility and being extremely far from any facilities. I tried these with all three settings of the skier's ability. For each test I checked that the directions produced were a valid route and that they did indeed lead to the nearest facility and that that facility was of the correct type. When I placed the user already at a facility I made sure that the app told them they were already there and didn't compute any unnecessary directions. In all of these situations the app behaved correctly.

7. Evaluation

I have already discussed the various methods of testing I used for this project in the previous chapter. In this chapter I discuss the usability testing I did and then I evaluate my DSML.

7.1 Usability

7.1.1 OBJECTIVES OF THE TEST

When assessing the usability of my app, I had the following objectives:

- Intuition
  Does the user understand how to use the app?
- Presentation
  Is the user able to understand the information?
- Clarity
  Is the presentation of menu, information and directions clear?

7.1.2 PARTICIPANTS

The participants were from a range of skiing backgrounds, with the majority of participants having no experience of skiing. This enabled me to gather insight into how a range of potential users would interact with the app. The participants with no skiing background in particular provided a lot of insight into how intuitive and clear the app was.

7.1.2 TASKS TO BE PERFORMED

I provided participants with a mixture of general tasks and specific scenarios which are likely use cases. Having participants complete tasks was an efficient way for me to assess usability; specific scenarios “provide more context for why the participant is doing the task and hence look more like natural interactions that a typical user will perform with your application.” [38]

In creating my tasks I followed the criteria specified by Mifsud [38]:
- Realistic, actionable and without any clues on how to perform the steps
- Ordered in a sequence that ensures a smooth flow of the test session
- Tied to one or more objectives

These were the Tasks and Scenarios I created for Feedback Day:
1. Set your level
2. When does Tiehack Express close?
3. Is Blue Grouse groomed?
4. Get directions to here (show on map)
5. You need to get to a bathroom quickly!
6. Your friend just fell and broke their leg!

These tasks target all four sections of my app: task 1 demonstrates the usability of the home menu and settings section; tasks 2 and 3 gave me insight into the usability of the information section; tasks 4 and 5 targeted the directions section; task 6 represents the emergency section.

7.1.3 METHODOLOGY

I initially used Google Forms [39] to create the evaluation questionnaires. This tool lends itself well to evaluation as it graphs the user responses making it easier for me to analyze.

However, after a trial run of this questionnaire I realized it was inappropriate for Feedback day as it was too lengthy and required a level of detail that wasn't necessary for me to obtain valuable data.

In contrast, the Think Aloud Protocol [40] didn't ask too much of the participants while giving me very good insights into the usability of my app. This entails asking test participants to vocalize their thoughts as they use the app. I wanted to see how someone would intuitively use the app and what things they would try to do, as well as seeing how they would use the app when trying to accomplish a specific task.

I collected data by taking notes on what participants were saying, focusing mostly on negative feedback and critiques for improvement.

7.1.4 RESULTS

7.1.4.1 BUTTONS

Almost all participants tried to click the icons on the homepage rather than the text next to them. Many participants tried to click on the description of the levels in an attempt to select a level. A few participants tried to click the text displaying their current level in order to choose a different level. Another common problem that participants ran into was that I didn't have a home button.

7.1.4.2 INFORMATION LOOKUP

Several users didn't notice that they could scroll the list of runs. A few users suggested that a specific run would be easier to find if the runs were listed alphabetically. One user suggested that instead of clicking on a run to reveal information about it, all this information could be distilled into symbols next to the run name to indicate if the run was groomed and/or open, similar to how I display the level of the run.

7.1.4.3 SETTING ABILITY LEVEL

Almost all participants had trouble with this in one way or another. A few didn't think to look in Settings. Most couldn't find where they needed to click to set their level. A common complaint was the lack of feedback once the level was selected; several participants tried to set their level a second time thinking it hadn't worked the first time.

7.1.4.4 DIRECTIONS

Most participants understood the directions well, however there were a few points for improvement. One suggested that I could indicate the level of the run in the directions as a symbol similar to how runs are displayed in the information section. Another suggestion was to send a push notification when the skier arrives at a decision point.

Reviewing these results in terms of the objectives of the test (see section 7.1.1), the intuitiveness of my app was lacking with users often clicking the wrong thing. Besides the alphabetization issue, the presentation and clarity of my app were both good. Participants were able to understand the information well and they found the directions and menu to be clear. This was particularly encouraging as most of the participants were from a non-skiing background.

7.1.5 UI IMPROVEMENTS

Based on the feedback in section 7.1.4, I made the following improvements:

- The icons in the homepage are now also buttons
- Everything to do with setting the user's ski ability is now clickable
- There are now buttons in the action bar at the top of the app so users can navigate home
- Runs and Lifts are now alphabetized
- A user can now set their ski ability from the home page where it is displayed, as well as in the settings.
- The user now gets feedback after selecting a level.

After making all these adjustments I did further usability testing to ensure the changes I made were indeed improvements. I again used the Think Aloud Protocol and had three people who had not used the app before run through the same tasks described in section 7.1.2. All participants completed the tasks successfully with none of the previous complaints reoccurring.

7.2 DSML

Frank is well known for his work on multi-perspective enterprise modeling and has laid out criteria for deciding if a term should be incorporated into a DSML in his paper *Some Guidelines for the Conception of Domain-Specific Modelling Languages* [41]. He also proposes criteria for assessing whether a term should be considered a type or a meta-type. I evaluated my DSML against these criteria.

7.2.1 CRITERIA

A. NOTEWORTHY LEVEL OF INVARIANT SEMANTICS

The term can be expected to have the same meaning across all application areas a DSML is supposed to cover. It should be possible to define the essential meaning of the concept.

B. RELEVANCE

If a concept is expected to be used on a regular basis, making it part of the language would increase the language's value.

C. NOTEWORTHY SEMANTIC DIFFERENCES BETWEEN TYPES

A meta type should allow for a range of types with clear semantic differences. If the types that can be instantiated from the potential meta type are all too similar, the effort it takes to define the types is questionable.

D. INSTANCE AS TYPE INTUITIVE

Would instances be regarded as types intuitively by prospective language users?

E. "INSTANCE" AS ABSTRACTION

Whether a term that does not allow for further instantiation can still be regarded as an abstraction.

F. INVARIANT AND UNIQUE INSTANCE IDENTITY

The first two criteria, A and B, determine whether a term is suited to be incorporated in a language. The following four criteria determine whether a concept should be reconstructed as a type or a meta type. If a concept fulfills C and D it should be represented by a meta type. If it fulfills E and F if should be represented by a type.

7.2.2 RESULTS

As I explained in section 4.3, one of the critical decisions in designing my DSML was deciding the extent to which runs and lifts are similar and whether or not to include the concept of an edge in the model. The following tables show the results of evaluating these terms against the above criteria. I felt that for other terms I included in my DSML, although important, the decisions to include them were more obvious and therefore unnecessary to further evaluate.

The results show that Run, Lift and Edge are all well suited as language concepts. Edge is suited to be represented as a meta-type while Run and Lift are better suited to be represented by a type.

Edge

An edge has a start node, an end node and a direction.

A	The term is used on a high level of abstraction. Its essential meaning doesn't vary.	✔
B	This term is used frequently when calculating routes.	✔
C	Its types, Run and Lift, are semantically very different.	✔
D	A language user would intuitively understand that a Run or a Lift is a type of Edge.	✔

Run

A run is a marked path on a mountain, created to be skied down.

A	Its directionality, down, is specific to a run. Its essential meaning doesn't vary.	✔
B	This term is frequently used.	✔
C	The semantics of its instances don't vary.	✘
D	A instance of run wouldn't be regarded as a type.	✘
E	There is a clear distinction between an instance of a run and the type Run.	✔
F	Any instance of a run is invariant and unique within the resort.	✔

Lift

A lift is any mechanism that takes a skier up the mountain.

A	Its directionality, up, is specific to lifts. Its essential meaning doesn't vary.	✔
B	This term is frequently used.	✔
C	The semantics of its instances don't vary.	✘
D	A instance of a lift wouldn't be regarded as a type.	✘
E	There is a clear distinction between an instance of a lift and the type Lift.	✔
F	Any instance of a lift is invariant and unique within the resort.	✔

8. Conclusion

8.1 Review of Achievements

I successfully achieved the goal of this project which was to develop a domain-specific modeling language to describe a ski resort and then use this as a basis for an Android app that provides skiers with information and directions based on their preferences and location. Taking a Domain Driven Design approach, I was able to design the DSML well and use the resulting domain model not just to describe ski resorts but also as the basis for developing the Android app. Both the DSML and the Android app were tested and evaluated.

Here I review the functional and non-functional requirements I achieved.

8.1.1 FUNCTIONALITY

Most ski resorts make small changes each year and my DSML makes it is easy to keep a model up-to-date. This is contrasted with an app like Fatmap which quickly becomes inaccurate since making changes to their maps is both time-consuming and expensive.

The ability to easily instantiate my app for any ski resort is useful to both skiers and resort owners. A skier used to using the app in a particular resort won't have to download another app and learn how to use it when they travel to a new resort. A resort owner won't have to commission an entirely new app for their resort; instead all they have to do is describe their resort using the DSML. In this way they are able to save both time and money.

The app successfully guides users around the mountain within the constraints they set combined with real-time run status updates. This is a particularly valuable feature for less experienced skiers who want to enjoy skiing whilst being confident they won't get lost or be confronted with anything beyond their ability. This feature is also useful for more experienced skiers hoping to explore more of the mountain and push themselves to ski harder runs.

Finally, my app provides skiers with easy access to the information they need in an emergency. This means allows the ski patrol to respond more quickly and more accurately.

8.1.2 COMPATIBILITY

Not only is Android the preferred smartphone platform, by my app is compatible with over 90% of active Android devices. The results of my Robo tests show that the app works well on all compatible API levels as well as a wide range of screen sizes.

8.1.3 PERFORMANCE

I was able to improve the performance of my app throughout this project. Mobile devices are limited in their computational resources so I was mindful in my choice of data structures. I also was able to improve performance by arranging the components of my app to run on different threads.

8.1.4 USABILITY

Although it took several iterations, I was ultimately able to create a user-friendly app. The end product is intuitive to use and the information is clear and easily understandable, even for non-skiers. The directions are easy to follow and accommodate for the fact that most skiers don't wish to take out their phone while skiing by providing spoken directions.

8.1.5 SCALABILITY

Because I took a DDD approach, everything in the domain model is based on an object which means that components are modular and encapsulated. Furthermore, I followed the SOLID design principles [42] to ensure my app is flexible and maintainable. I did this by having boundaries and clear responsibilities for classes. I so doing, I have made my app scalable.

8.2 Future Work

Throughout the usability tests I did for this app, I received a common response: the user would like to be able to select their destination by pointing at a map. I still maintain that small-scale maps are challenging to work with on small phone screens but I wanted to investigate how this might be achieved. Initially I looked into using Google's ski maps but there are only a very limited number of resorts available and even those aren't available for free. Next I looked into what open-source ski maps were available and found OpenSnowMap[43]. Despite a complete lack of documentation, I was able to display the ski resorts in the app and even locate a skier on the map. However, OpenSnowMap crowd-sources its piste data which can lead to questionable data quality. Furthermore, the overlay tiles are only openly available if you download the entire world at once which is 5GB worth of data. At this point I had already spent a significant amount of time investigating this feature and decided to focus on other aspects of the app. If I were to return to this in the future, I could define a concrete syntax for my DSML so that I could visually display a resort that a user could interact with.

I have really enjoyed working on this project and I keep coming up with ideas for more features I could add. As mentioned in section 8.1.5, I ensured my app is extensible so that it is possible for me to continue to add features.

One example of such an idea is using the app to crowd-source data about lift queues. Every time a skier arrives at a node that is the bottom of a lift, I would measure how long they stayed at that node. If when they then left the node they were moving in an uphill direction, I would send this length of time to the cloud. For each lift, all of these times could be aggregated within some small time window to produce a live lift queue length estimation. However, this would require the app to be used by many people for the data to be useful.

9. Bibliography

[1] News article from PlanetSki, 2012. https://www.planetski.eu/news/3980.

[2] News article from the Telegraph, 2017. https://www.telegraph.co.uk/news/2017/01/17/australian-family-rescued-getting-lost-japanese-ski-resort-spending/.

[3] Frank, Ulrich. Outline of a Method for Designing Domain-Specific Modelling Languages. Econstor. 2010. https://www.econstor.eu/obitstream/10419/58163/1/716089785.pdf

[4] Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018. Statista. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/. Accessed March 2019.

[5] Aspen Snowmass App. https://play.google.com/store/apps/details?id=com.aspensnowmass.app&hl=en. Accessed April 2019.

[6] Fatmap Android Application. https://play.google.com/store/apps/details?id=com.fatmaprn&hl=en_GB .

[7] CityMapper Android Application. https://play.google.com/store/apps/details?id=com.citymapper.app.release .

[8] Google Maps Android App. https://play.google.com/store/apps/details?id=com.google.android.apps.maps&hl=en_GB. Accessed April 2019.

[9] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2006.

[10] Cabot, Jordi. Modeling Languages. Image: 'Domain-driven design plus model-driven engineering'. https://modeling-languages.com/comparing-domain-driven-design-model-driven-engineering/ .

[11] Mogosanu, Mike. Sapiens Works. The Bounded Context Explained. http://blog.sapiensworks.com/post/2012/04/17/DDD-The-Bounded-Context-Explained.aspx . 2012.

[12] H Sharp, A. Finkelstein, G Galal. Stakeholder Identification in the Requirements Engineering Process. http://discovery.ucl.ac.uk/744/1/1.7_stake.pdf .1999.

[13] The Ski Safety Act. Colorado Revised Statuses. https://www.coloradoski.com/sites/default/files/uploads/Colorado-Ski-Safety-Act.pdf . 2006.

[14] Android Developers. Distribution Dashboard. https://developer.android.com/about/dashboards. Accessed March 2019.

[15] Android Oreo. https://www.android.com/versions/oreo-8-0/ . Accessed March 2019.

[16] Android KitKat. https://www.android.com/versions/kit-kat-4-4/ . Accessed March 2019.

[17] Eclipse Modeling Framework. https://www.eclipse.org/modeling/emf/. Accessed March 2019.

[18] Google Firebase's Realtime Database. https://firebase.google.com/products/realtime-database/ . Accessed March 2019.

[19] B. Phillips, C. Stewart, K.Marsicano. Image. *Android Programming: The Big Nerd Ranch Guide*. Big Nerd Ranch Guides; 3 edition, February 9, 2017.

[20] AsyncTask. https://developer.android.com/reference/android/os/AsyncTask. Accessed March 2019.

[21] AndroidX Preference Library. https://developer.android.com/reference/androidx/preference/package-summary. Accessed March 2019.

[22] JavaX XML Parser Library. https://docs.oracle.com/javase/8/docs/api/index.html?javax/xml/parsers/package-summary.html. Accessed March 2019.

[23] Document Object Model (DOM) from the Java API for XML Processing. https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html. Accessed March 2019.

[24] RecyclerView. https://developer.android.com/guide/topics/ui/layout/recyclerview . Accessed March 2019.

[25] S. Tomaszewski's AsyncTaskWithTimeout. https://gist.github.com/scottTomaszewski/3c9af91295e8871953739bb456de937b . Accessed March 2019.

[26] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction To Algorithms.* MIT Press. 2001

[27] J. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics* 27 (4): 526–530*,*1970.

[28] E. Bouillet. Path routing in mesh optical networks. Chichester, England: John Wiley & Sons. 2007.

[29] J. Yen. Finding the *k* Shortest Loopless Paths in a Network. *Management Science*. 1971.

[30] Android Text-To-Speech API. https://developer.android.com/reference/android/speech/tts/TextToSpeech. Accessed April 2019.

[31] M. Cohn. The Forgotten Layer of the Test Automation Pyramid. https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid. Mountain Goat Software (2009)

[32] Fundamentals of Testing. Image. https://developer.android.com/training/testing/fundamentals.html . Accessed March 2019.f

[33] Robolectric. http://robolectric.org/ . Accessed March 2019.

[34] Image. Buttermilk trail map from Snow-Online. https://www.snow-online.com/ski-resort/aspen-buttermilk_trailmap.html. Accessed March 2019.

[35] Firebase Test Lab. https://firebase.google.com/docs/test-lab/ .

[36] Firebase. Analyze Firebase Test Lab Results. https://firebase.google.com/docs/test-lab/android/analyzing-results. Accessed March 2019.

[37] Lockito Android app. https://play.google.com/store/apps/details?id=fr.dvilleneuve.lockito&hl=en.

[38] J. Mifsud. Usability Geek. https://usabilitygeek.com/usability-testing-mobile-applications/ ,(2016). Accessed February 2019.

[39] Google Forms. https://www.google.co.uk/forms/about/. Accessed April 2019.

[40] C. Lewis. Using the "Thinking Aloud" Method In Cognitive Interface Design (Technical report). (1982).

[41] U. Frank. Some Guidelines for the Conception of Domain-Specific Modelling Languages. 93-106. EMISA, 2011.

[42] R. Martin. *Design Principles and Design Patterns*. 2000.

[43] OpenSnowMap. http://www.opensnowmap.org/. Accessed March 2019.

[44] Image. Map of Passo Tonale. https://www.bergfex.com/passo-tonale-val-di-sole/. Accessed March 2019.