UNIVERSITY
OF APPLIED SCIENCES
MUNICH

MASTER THESIS

Modeling passengers' seating behavior for simulations of pedestrian dynamics

Modellierung der Sitzplatzwahl von Fahrgästen für Fußgängersimulationen

Author:	Jakob Schöttl
Matriculation number:	05014109
Adviser:	Prof. Dr. Gerta Köster
	Dr. Michael Seitz
Department:	Informatics and Mathematics
Study program:	Master in Informatics
Date of submission:	October 31, 2016

Abstract

In the field of pedestrian dynamics, microscopic models are used to simulate the behavior of crowds. These simulations can be used to estimate evacuation times and crowd densities, but also to optimize efficiency in public transport. When simulating public transport systems, it is necessary to simulate passengers' behavior inside a train. Modeling the situation inside a train can be divided into subproblems, e.g. the inflow process, waiting and seating behavior, leaving, and strategies to cope with congestions at the doors. Many of these subproblems have already been investigated and modeled. However, there is little research on how passengers choose their seats inside trains. Here I present a model to simulate seating behavior in trains with open compartments, each having four seat groups and each seat group having four seats. The model is based on results from a data collection that I conducted in Munich's suburban trains. The collected data consists of logged events, e.g. when a passenger sits down or leaves, or when the train starts and stops. The dataset also includes metadata on baggage, passengers, and groups. The analysis of the dataset yields insights on where passengers prefer to sit down in a compartment and in a seat group. For example, within a compartment, passengers tend to choose the seat group with the smallest number of other passengers. Within a seat group, passengers prefer window seats and forward-facing seats. However, if there is another person, passengers tend to choose the seat diagonally across from that person. These and other aspects are incorporated in a model of passengers' seating behavior. I implemented the simulation model in VADERE, a framework for crowd simulation developed at the Munich University of Applied Sciences. The seating model can be used as one component in larger systems for the simulation of public transport. As such, it helps to distribute simulated passengers throughout the train by assigning them realistic seat positions. The results of this work can help to conduct studies on safety or efficiency in public transportation systems.

Zusammenfassung

Im Forschungsgebiet Fußgängerdynamik werden mikroskopische Modelle eingesetzt, um das Verhalten von Menschenmengen zu simulieren. Diese Simulationen können verwendet werden um Evakuierungszeiten und Menschendichten abzuschätzen, aber auch, um Effizienz im öffentlichen Personenverkehr zu optimieren. Bei der Simulation öffentlicher Verkehrssysteme ist es notwendig, das Verhalten von Fahrgästen im Zug zu simulieren. Die Modellierung der Situation in einem Zug kann in Teilprobleme aufgeteilt werden, z. B. die Abläufe im Eingangsbereich und das Warte- und Sitzverhalten. Viele dieser Teilprobleme wurden bereits untersucht und modelliert. Allerdings ist wenig darüber bekannt, wie Fahrgäste in Zügen ihren Sitzplatz wählen. In meiner Arbeit stelle ich ein Modell vor, mit dem die Sitzplatzwahl in Zügen simuliert werden kann. Es gilt für Züge mit offenen Abteilen und mit je vier Sitzgruppen pro Abteil à vier Sitzplätzen. Das Modell basiert auf den Ergebnissen einer Datenerhebung, die ich in Münchens S-Bahnen durchgeführt habe. Die gesammelten Daten bestehen aus protokollierten Ereignissen, z. B. wenn sich Fahrgäste setzen oder aufstehen oder wenn der Zug anfährt oder hält. Die Analyse des Datensatzes bringt Erkenntnisse darüber, wohin sich Fahrgäste im Abteil und in der Sitzgruppe bevorzugt setzen. Beispielsweise setzen sich Fahrgäste in einem Abteil meistens in die Sitzgruppe, in der am wenigsten andere Personen sitzen. Innerhalb einer Sitzgruppe bevorzugen Fahrgäste Fensterplätze und Plätze in Fahrtrichtung. Wenn bereits eine andere Person in der Sitzgruppe sitzt, wählen Fahrgäste meist den diagonal gegenüberliegenden Platz. Diese und weitere Aspekte sind in einem Modell für die Sitzplatzwahl einbezogen. Ich habe das Modell in VADERE implementiert, einem Framework zur Simulation von Fußgängerdynamik, das an der Hochschule München entwickelt wurde. Das Modell kann als eine Komponente in einem größeren System zur Simulation von öffentlichen Verkehrssystemen verwendet werden. Dabei hilft es, simulierte Personen im Zug zu verteilen, indem es ihnen realistische Sitzpositionen zugeweist. Die Ergebnisse dieser Arbeit können dabei helfen, weitere Studien über Sicherheit oder Effizienz im öffentlichen Personenverkehr durchzuführen.

Acknowledgments

First, I would like to thank Prof. Dr. Gerta Köster's research group at the Munich University of Applied Sciences for discussions and feedback, and for a great atmosphere and the fun collaboration. I would especially like to thank Michael Seitz for all the time he invested in meetings, discussions, and proof-reading, which were all of invaluable help. Then, I would like to thank my adviser, Gerta Köster, for frequent and valuable feedback and for always fast and helpful email responses. I also like to thank my friends from Munich, Vienna (Austria), and San Luis Obispo (CA, USA) for the inspiring discussions and for proof-reading. Finally, I would like to thank my family for the mental support, coffee breaks, and discussions during the time of writing my thesis.

Schöttl, Jakob	München, 2016-10-28
(Familienname, Vorname)	(Ort, Datum)
1990-05-19	IG / WS 2016
(Geburtsdatum)	(Studiengruppe / Semester)

Erklärung / Declaration

Gemäß § 40 Abs. 1 i. V. m. § 31 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

(Unterschrift)

Contents

List of Figures

List of Tables

Listings

1. Introduction

Today, 54 percent of the world's population lives in urban areas.[1] In urban areas, public transportation systems are of great importance for its efficient functioning. One of the most important types of transportation are suburban trains. These trains can be almost empty at midday or overcrowded at peak hours. Depending on the degree of utilization, passengers may have to adapt their behavior. If there are free seats available, passengers often choose to sit down for their journey.

In this work, I investigate passengers' seating behavior in Munich's suburban trains and present a simulation model implemented in the crowd simulation software VADERE. In the following sections, I outline the motivation and the current state of the art, and I present the objectives, the approach, and the structure of this work.

1.1. Motivation

Pedestrian dynamics and crowd simulation is a wide field of research (Daamen et al., 2014). Crowd simulation can be used to predict evacuation time or crowd density in evacuation analyses (Kirchner and Schadschneider, 2002; Pelechano and Badler, 2006; Gao et al., 2014; Alizadeh, 2011). More generally, such simulations are applicable for the study of places where a larger number of people come together, e.g. in public transport systems and especially at train stations. Simulation software can already cope with common scenarios including gates, queues, stairs, and multiple floors (e.g. Köster et al. (2015); Köster and Zönnchen (2014); vad (2016)). Research on people's spatial distribution during waiting times has also brought useful results (e.g. Seitz et al. (2015); Liu et al. (2016b,a)). However, there is little research on people's seating behavior, that can be used to simulate

[1] https://esa.un.org/unpd/wup/

this aspect in the larger system. There are many questions to be answered, for example “where do people sit down in relation to another person?”, “how do groups of people sit together?”, or “how does age or gender influence the choice of seating?”.

1.2. State of the art

For literature research I employed the search terms “choice of seating” and “preferred seating” combined with “public transport” and “trains” in English and German.[2]

There is a number of publications on boarding schemes for airplanes (Steiner and Phillipp, 2009; Jaehn and Neumann, 2015; Qiang et al., 2014). Some of the studies also provide simulation for the boarding process (Steiner and Phillipp, 2009; Qiang et al., 2014). The goal here is to reduce the boarding time.

Another research focus is on public transportation related to passenger’s behavior in train interiors. Some studies contain useful hints related to seating behavior, e.g. studies on the degree of capacity utilization (Cis, 2009), baggage in trains (Plank, 2008), passenger exchange times (Tuna, 2008; Panzera, 2014), and train interior design (Rüger and Ostermann, 2015).

Studies on seating layouts and passengers’ seating behavior in trains are discussed by several authors. Trinkoff (1985) conducted a study on preferred facing directions in the Washington Metro. Rüger and Loibl (2010) surveyed preferred seat choices in inter-city trains. Wardman and Murphy (2015) carried out a survey on passengers’ valuation of seating layouts in public transportation.

Other studies are concerned with the inflow process when people enter a room (Liu et al., 2016b,a; Xiao et al., 2016; Ezaki et al., 2016). While these studies conduct experiments where the subjects cannot sit down, some aspects are still interesting for this work. Furthermore, these studies can serve as a starting point for modeling the behavior of standing passengers in trains, which is not subject of this work.

[2] I searched in libraries of Munich’s major universities, in Google Scholar, and in Google’s standard search engine. Web of Science and Scopus were not available. The main literature research was done in February of 2016.

For pedestrian simulation I mainly build on work done at the Munich University of Applied Sciences, namely the VADERE crowd simulation framework and associated research. Seitz (2016) presents various research results on pedestrian locomotion and gives an introduction to VADERE.

1.3. Objectives

Although there is a number of studies on seating behavior in public transportation, there is no detailed data available for a deeper analysis of various aspects of the choice of seating. To my knowledge, there are also no models or algorithms available to simulate the inflow and the seating process in trains. In this work, I therefore investigate passengers' seating behavior in trains from an empirical point of view and develop a simulation model for the crowd simulation software VADERE.

The objectives are two-fold. First, I want to promote simulations of public transportation systems by adding one part to the collection of simulation models: A model to simulate seating behavior. Second, I want to set up a framework for data collection and provide data from my study for social psychology research. This has a great potential for synergy because findings used for the model can also be interesting for the study of social behavior of individuals or groups in psychology.

1.4. Approach

In this work, I apply empirical and formal methods: I collect data in a field observation, and I develop an algorithmic model that describes seating behavior of passengers in trains.

First, I conduct the data collection in Munich's S-Bahn to explore passengers' seating behavior within compartments. The data collection is an observational field study in which events in the train are logged in real-time. How passengers spread through the train wagon is not covered by the study.

Second, I develop a theoretical model for seating behavior based on the collected data using a combination of cognitive heuristics and statistics. and, I implement the model in the crowd simulation software VADERE and verify it against the collected data.

1.5. Structure of this work

In chapter 2, Data collection, I gather requirements for the data collection and for a software tool to conduct the data collection. I present the resulting app's UI and usage. After a pilot study the main study on seating behavior is conducted and the procedure of data processing is explained. I describe data on passenger counts which was made available from the Münchner Verkehrs- und Tarifverbund (MVV), and I collected and analyzed passenger entrance rates.

In chapter 3, Software, I describe some aspect of the data collection app's architecture, design, and development. I introduce the crowd simulation software VADERE and state contributions to VADERE, which are required for a good integration of the seating model. I present the Traingen tool for generating train scenarios which can be imported in VADERE. Finally, the seating model's software design and implementation are described together with the system for model validation.

In chapter 4, Model, I first specify approach, assumptions, and limitations of the seating model. I analyze the collected data on seating behavior and develop a model based on these results. Then, I explain the model's algorithms and parameters in detail.

In chapter 5, Model evaluation, I first verify the model's implementation by comparing simulated data against real data. Second, I visually validate the model by demonstrating that the train fills up in a realistic manner. Both methods are based on a simulation run with real passenger counts.

The conclusions give a short summary of the results, discussions, and inspiration for future work in this field.

1.6. Terms and definitions

In the following, I define terms used to describe the train interior.

Entrance area The room between two opposing doors.

Compartment The room between two adjacent entrance areas. There are 16 seats per compartment.

Seat group A group of four vis-a-vis seats, that is two facing benches.

Seat number Seats are numbered within a compartment, row by row. A row is four seats facing the same direction. The numbers range from 1 to 16 and increase from left to right and from the front to the rear. Front and rear can be defined arbitrarily but the seat numbering must not change when the train changes its driving direction. Figure 1.1 illustrates the seat numbering schema. The numbering schema is also implemented in the app for data collection as depicted in figure 2.2c.

Compartment plan A top view plan of the compartment including a correct seat numbering e.g. figure 1.1 is such a plan.

1.7. Conventions

1.7.1. Reproducible research

I strive to do reproducible research in this thesis (Peng, 2011). To achieve this goal, I use the following techniques:

- All plots are generated with knitr (Xie, 2012) at the moment this LaTeX document is compiled.
- All numbers are either generated with knitr or have a footnote like this[3], to provide the shell code to reproduce the value.
- Sample data, tables, and sample output of programs are taken from their sources or generated in the process of compiling this document.

[3] `echo -n "this"` (shell command line)

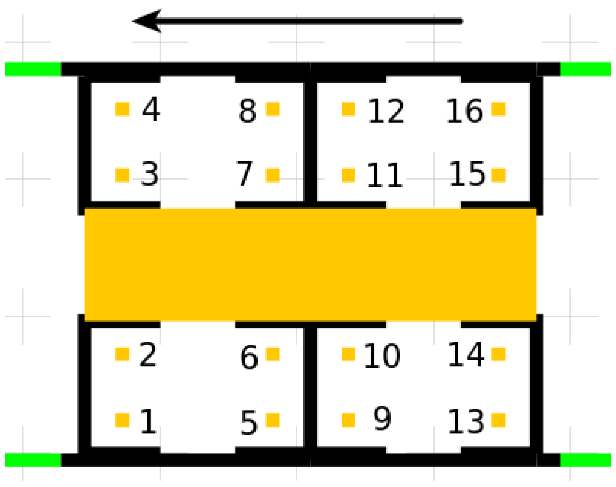


Figure 1.1.: This figure shows a plan of a S-Bahn train compartment with four seat groups, each consisting of four seats. The black arrow on the top denotes the train's driving direction. The yellow dots mark the seats and the yellow area in the middle is the aisle connecting the entrance areas. The green bars are parts of the doors.

- Code samples and listings are taken from their original source files. This is made possible with `Makefiles`, shell scripts, and the `\lstinputlisting` command from the Listings package.
- Listings that show interfaces of Java classes are generated with the `javap` program[4], which can disassemble a compiled Java program and extract the interface of a class.

To reproduce and verify this work, the LaTeX code and the `Makefile` can be examined, R code for plots and statistics can be re-evaluated, and shell command lines in footnotes can be interpreted.

1.7.2. Formatting

Italic font is used when describing user interfaces. It is used for labels of UI elements such as buttons or menu items. It is also used in the text for file

[4]The `javap` tool is distributed with Java; see `javap --help` for more information.

or folder names.

`Constant width` font is used for in-line code snippets and identifiers such as function or class names.

2. Data collection

2.1. Requirements

In this work, I realize a real-time data collection in the form of an observational study. For this, investigators observe train compartments and passengers during train rides over multiple stops. During data collection, the investigator logs all relevant events so that the situation can be reconstructed later.

In the following, I collect requirements for both the data collection and the mobile app used for it. The requirements were collected through brainstorming and interviews with people who use the S-Bahn train daily.

Investigators using the app for data collections should read this section to obtain consistent understandings and interpretations of conditions in the train. Otherwise, collected data from different investigators may not be comparable.

2.1.1. Logged events

The first question is what kind of events are to be logged during the survey. Since the data collection is well supported by the mobile app, there are capacities for collecting more data then necessary for a study only on seating behavior. By capturing more data about the situation in the compartment and the state of the train, the data can also be useful to examine social behavior of groups or individuals, demographic distributions of passengers, or other statistics.

The identified log event types are listed in table 2.1.

2.1.2. Interference factors

This section identifies factors that might interfere with normal seating behavior. Although some of these factors could theoretically be included in a model, the

Event type	Description
INITIALIZATION_END	The initial conditions in the train and in the compartment are captured and the real-time data collection starts.
SIT_DOWN	A person sits down on a seat.
LEAVE	A person leaves its seat and the compartment. See also CHANGE_SEAT.
CHANGE_SEAT	A person changes its seat. It is important to differentiate from SIT_DOWN because that implies a new person sitting down.
PLACE_BAGGAGE	A person places its baggage on another seat. See section Seats in 2.1.3 for a precise definition.
REMOVE_BAGGAGE	A person removes its baggage from another seat.
DISTURBING	A person or seat is marked as disturbing. See section 2.1.2 for a precise definition.
STOPS_DISTURBING	A person or seat stops with disturbing. See section 2.1.2 for a precise definition.
DOOR_RELEASE	The doors are released and can be opened.
TRAIN_STARTS	The train starts moving.
COUNT_STANDING_PERSONS	Currently standing persons are counted for statistics. All persons standing in both adjacent entrance areas and in the compartment inbetween are counted.
DIRECTION_CHANGE	The train's driving direction changes.

Table 2.1.: Description of log event types i.e. events that are captured during the real-time data collection.

situations are rare and out of scope for this thesis. The data collection app will provide functionality to log interfering events. This way, non-reliable data can easily be excluded from the data analysis.

The interference factors are split into two groups: Some are associated with persons and others are not. First, this is a list of interference factors associated with persons:

- Some persons (e.g. children) sharing seats.
- A standing person who occupies a seat with baggage.
- A person using the seat across for his or her legs.
- An obviously drunken or unpleasant person.

- An overly active young child.
- Persons who blocks more than one seat with baggage that they cannot simply take away (e.g. large bags).
- A dog blocking space between the seats.
- An overweight person that takes clearly more than one seat.
- A person that places a bicycle or another large baggage item in the entrance area. This is relevant because it is important for people to have visual contact with their baggage (Rüger and Loibl, 2010).
- A person listening to load music or having a load phone conversation.

Second, this is a list of factors not bound to specific persons:

- A kindergarten group or school class.
- Strong sun light on one window side.
- An obviously dirty seat.
- Clean newspapers or journals on a seat.
- Particularly load conversation between persons.

It is possible that during the phase of data collection new factors arise. In this case, they will be included into the app and the data analysis.

2.1.3. Parameters

This section identifies parameters that have to be observed during the survey.

General

General parameters regarding the survey are:

- Date and time of the survey.
- The train station from that the survey starts.
- The train's line and final destination.
- The train's driving direction. This is only necessary to define the orientation of the compartment plan (see figure 2.2c).

- The investigator's name.
- The investigator's exact position within the train and the compartment.
- The train's type, e.g. ET423 for Munich's current S-Bahn trains.

Most of these parameters are covered by the survey meta data. Only the investigator's position within the compartment, the train's driving direction, and the exact start time are covered by the corresponding log events.

Persons

The following data about persons will be collected in this survey:

- gender,
- age group,
- the group the person is traveling with,
- the seat(s) the person is sitting on,
- the seat(s) the person is placing baggage on, and
- comments.

Some of these parameters are taken from (Plank, 2008; Cis, 2009), which conducted similar surveys in inter-city trains: fellow passengers, carried baggage, seat numbers, and general comments.

The age group of persons has to be estimated during the data collection. In (Tuna, 2008, 4.1), another similar survey, the authors also use age classes that basically break at multiples of ten. In this survey, different age groups are used:

- young children (0–6 years)
- schoolchildren (6–12 years)
- youth (13–18 years)
- young adults (19–30 years)
- adults (30–65 years)
- older people (66 years and older)

People within these age groups might behave more consistently than people put into groups of fixed age intervals. Also, these age groups might be easier to estimate and therefore be more precise. For groups "young children", "schoolchildren", etc. we have role models in our mind. For example, young children usually do not travel alone and schoolchildren have a school bag. This simplifies the classification.

Defining age groups and estimating age are still subjective assessments. Letting different investigators collect data can help to balance the subjective biases and obtain a good mean.

Seats

Parameters regarding the observation of seats are:

- Occupancy by persons. This is covered by the person's parameters above.
- Occupancy by baggage. A seat counts as occupied by baggage if more than one third of its surface is covered. Magazines do not count as baggage unless it is clear that they belong to a person.
- Disturbing factors, e.g. a dirty seat. This is covered by section 2.1.2.
- Seat properties, e.g. facing direction and window vs. aisle side. These properties can be derived from the seat number, the compartment plan, and the train's driving direction.

2.1.4. Non-functional requirements

In the following list, I define a set of non-functional requirements:

- The data collection app and its source code shall be publicly available to enable other researchers to conduct similar surveys.
- The data collection app shall be adaptable to enable use in similar surveys e.g. surveys on other public transportation systems with different seating layouts, waiting areas, or railway platforms.
- The collected data and the code for data processing and analysis must be open source to enable other researchers to reuse the collected data and the data processing algorithms.

2.2. Pilot data

I first conducted a pilot study, to get an idea of how much data on seating behavior can realistically be gathered and to what extent a compartment can be reliably observed.

The data was collected in 11 observations at the S-Bahn line S3 between 2016-03-25 and 2016-04-07 on work days only. Two seat groups (8 seats in total) were observed. The following counts were recorded:

- the initial number of persons (`initial`),
- the number of new persons sitting down (`sitDown`),
- number of persons leaving (`leave`), and
- the number of persons changing place within the observed seats (`switchPlace`).

The person collecting the data was always sitting on one of the observed seats and was counted as well. The counting started as soon as every person, who entered at the start station, has sat down.

This is how the data looks after preprocessing:

```
##    time   start    dest dist initial sitDown leave switchPlace
## 1 16:59   ofing    obhf   10       1       9     4           1
## 2 21:14    obhf   ofing   10       3       0     2           0
## 3 07:39   ofing giesing    8       4       6     2           0
## 4 11:37 giesing   ofing    8       2       0     0           0
```

The above table is only a part of the dataset and some columns are renamed or left out for brevity. The `time` column holds the time at the start station. The `distance` (`dist`) column holds the distance between `start` and `destination` (`dest`) station. The distance values come from a distance matrix. As in graph theory, the distance is the number of "edges" between two stations, not the number of stations inbetween.

The `sitDown` count is a measure for how busy it is during the observation. It also gives an estimate how much useful data I can gather in the later survey because insights on seating behavior can be obtained from these events.

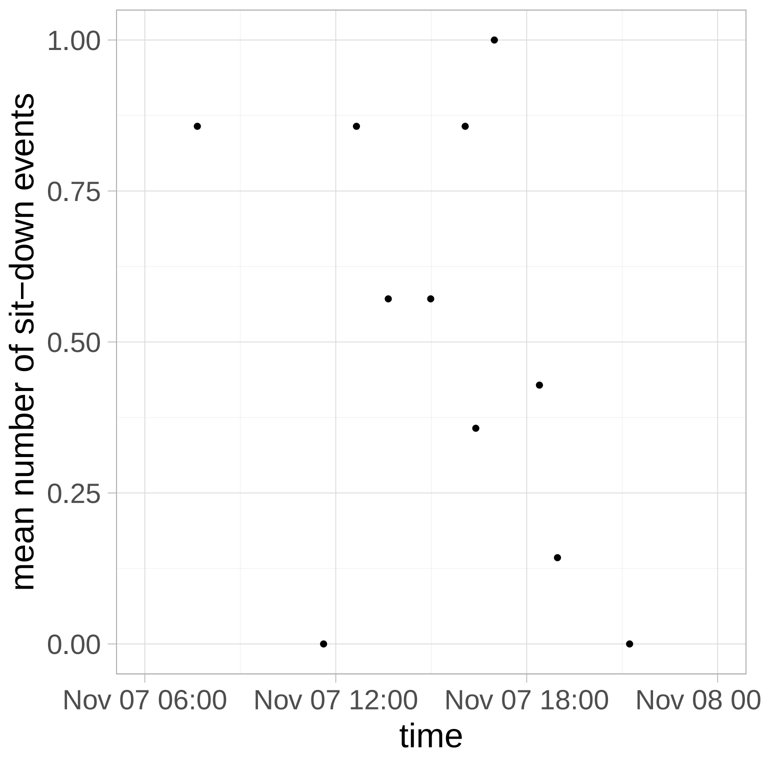


Figure 2.1.: Mean number of sit-down per stop. On the x-axis, only the daytime is valid. The date part is have to be ignored.

```
data$sitDownPerStop = data$sitDown / (data$distance - 1)
sitDownPerStop = mean(data$sitDownPerStop)
```

The mean number of persons sitting down at the observed seats per stop is 0.512987. Although this heavily depends on the time and the track section, it can be estimated that around 16 sit-down events can be observed at 16 seats between Holzkirchen and Hauptbahnhof (Munich's central station). This number can be increased significantly when concentrating the observations on the S-Bahn home line between Ostbahnhof and Hauptbahnhof.

Figure 2.1 shows collected data points. They can help to identify busy times for conducting later data collections.

2.3. Mobile app for data collection

2.3.1. User interface and usage

This section is a short description of the app's UI and how it is used.

The following shows the workflow for conducting a data collection. The paragraphs refer to respective screenshots in figure 2.2.

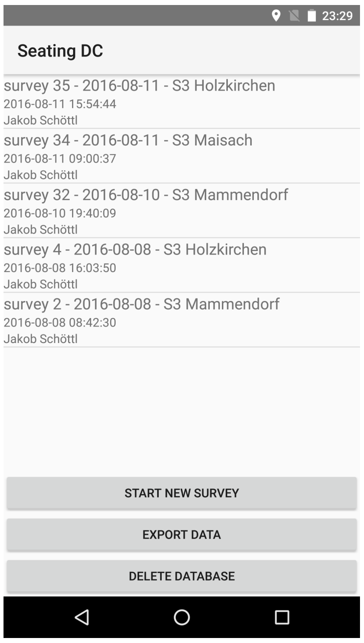
1. On the start screen of the app, there is a list of previous surveys. These previous surveys can be deleted or used as a template for new surveys to save typing (see figure 2.2b). A click on the button *Start new survey* brings the user to the next screen.

2. The user must input some general information about the survey. This includes the train's line and destination, the start station, and the user's position in the train. Once finished, the user can proceed to the next screen.

3. This is the initialization phase of the data collection. The user enters the initial occupation, groups of persons, and the driving direction. In addition the user should input the own seating position. To proceed with the real-time data collection, the user can press the button *Start data collection*.

4. The screen for the data collection allows for logging relevant events in real-time. These events include: person sits down, person places or removes baggage on another seat, person changes place, and person leaves.

All this gathered data is saved in a database and can be exported to CSV files for further analysis.


23:29

Seating DC

survey 35 - 2016-08-11 - S3 Holzkirchen
2016-08-11 15:54:44
Jakob Schöttl

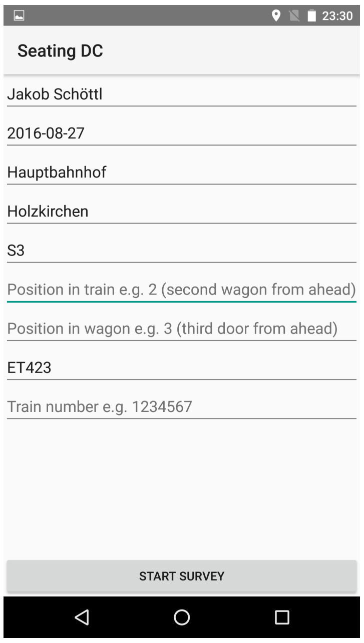
survey 34 - 2016-08-11 - S3 Maisach
2016-08-11 09:00:37
Jakob Schöttl

survey 32 - 2016-08-10 - S3 Mammendorf
2016-08-10 19:40:09
Jakob Schöttl

survey 4 - 2016-08-08 - S3 Holzkirchen
2016-08-08 16:03:50
Jakob Schöttl

survey 2 - 2016-08-08 - S3 Mammendorf
2016-08-08 08:42:30
Jakob Schöttl

(a) Start screen


23:30

Seating DC

Jakob Schöttl

2016-08-27

Hauptbahnhof

Holzkirchen

S3

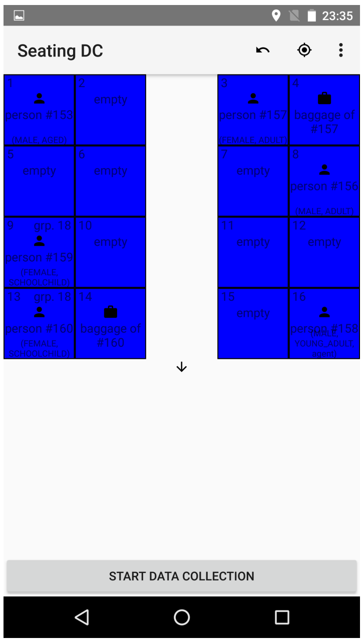
Position in train e.g. 2 (second wagon from ahead)

Position in wagon e.g. 3 (third door from ahead)

ET423

Train number e.g. 1234567

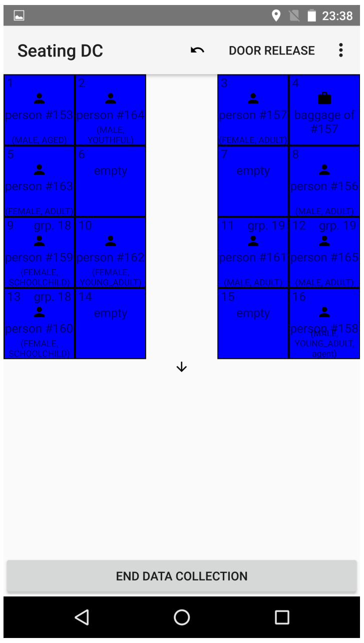
(b) Survey meta data


23:35

Seating DC

1 person #153 (MALE, AGED)	2 empty	3 person #157 (FEMALE, ADULT)	4 baggage of #157
5 empty	6 empty	7 empty	8 person #156 (MALE, ADULT)
9 grp. 18 person #159 (FEMALE, SCHOOLCHILD)	10 empty	11 empty	12 empty
13 grp. 18 person #160 (FEMALE, SCHOOLCHILD)	14 baggage of #160	15 empty	16 person #158 (MALE, YOUNG_ADULT, agent)

(c) Initialization phase

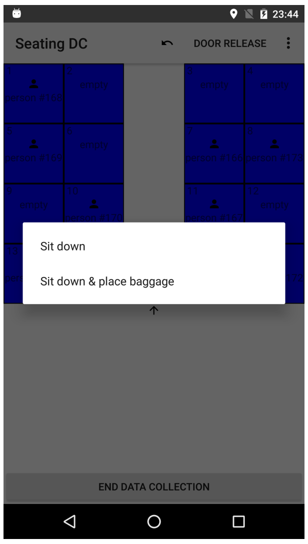

23:38

Seating DC

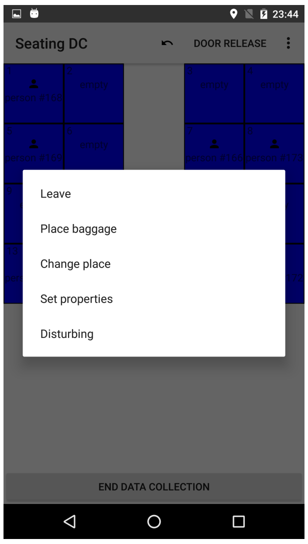
1 person #153 (MALE, AGED)	2 person #164 (MALE, YOUTHFUL)	3 person #157 (FEMALE, ADULT)	4 baggage of #157
5 person #163 (FEMALE, ADULT)	6 empty	7 empty	8 person #156 (MALE, ADULT)
9 grp. 18 person #159 (FEMALE, SCHOOLCHILD)	10 person #162 (FEMALE, YOUNG_ADULT)	11 grp. 19 person #161 (MALE, ADULT)	12 grp. 19 person #165 (MALE, ADULT)
13 grp. 18 person #160 (FEMALE, SCHOOLCHILD)	14 empty	15 empty	16 person #158 (MALE, YOUNG_ADULT, agent)

(d) Real-time data collection

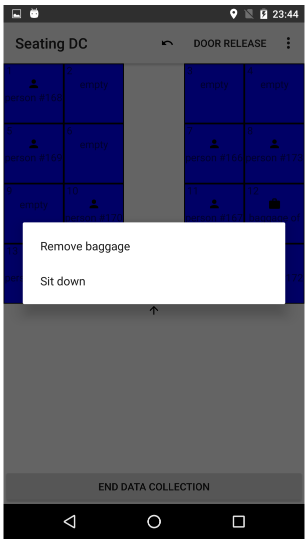
Figure 2.2.: The four screens of the data collection app on a Nexus 6 smartphone. The blue areas in figure 2.2c and 2.2d represent a compartment with 16 seats (4 seat groups). The white area inbetween the seat groups represents the aisle. The small arrow in the middle indicates the driving direction.



(a) Context menu for an empty seat



(b) Context menu for a seat occupied by a person



(c) Context menu for a seat occupied by baggage

Figure 2.3.: Context menus for three states of a seat: empty, occupied by a person, and occupied by baggage. In figure 2.3b, the menu item *Remove from group* is only displayed if the person is member of a group and the menu items *Disturbing* and *Stops disturbing* are displayed depending on the current state.

2.3.2. Usability evaluation

To get a feeling on how average smartphone users cope with the app's UI, I conducted a guided usability study. Three subjects participated in the study, one man and two women. They are all in their mid-twenties, and they are using their smartphones for more than 3 or 5 years, respectively.

The study involves an introduction and a realistic scenario, wherein the subjects have to accomplish a list of tasks. After the trial, a questionnaire with eight questions was given to the subjects. The subjects were observed while performing the tasks. The conductor took notes on any problems and helped the subjects if they were stuck.

Only the most important and most critical features of the app were tested in this study. The study brought some usability problems to light and yielded valuable suggestions for improvements. The complete study and a list of enhancements based on the results is printed in the appendix A.

2.4. Data collection

To obtain real data about the way people choose their seats, a data collection in Munich's public transport was conducted using the app developed in section 2.3. Only S-Bahn trains of type ET423 were considered and mainly the line S3. Table 2.2 lists all surveys.

2.5. Data processing

For data analysis, I use R, a language for statical computing. I use the R unit test frameworks RUnit and especially testthat (Wickham, 2011) to test the R code for data processing and analysis. The testing code amounts to about 600 lines of test cases split into 8 source files.[1]

Both, data and code are open-source and hosted at GitHub: `https://github.com/schoettl/seating-data`. The data is licensed under the Public Domain Dedication and License (PDDL) while the code is licensed under the MIT License.

[1] `wc -l seating-data/scripts/tests/*.R | awk '{print ++n, $0}'` (shell command line)

	DATE	TIME	STARTING_AT	LINE	DESTINATION
1	2016-08-08	08:42:30	Otterfing	S3	Mammendorf
2	2016-08-08	16:04:12	Hauptbahnhof	S3	Holzkirchen
3	2016-08-10	19:40:09	Otterfing	S3	Mammendorf
4	2016-08-11	09:00:37	Otterfing	S3	Maisach
5	2016-08-11	15:54:44	Hauptbahnhof	S3	Deisenhofen
6	2016-08-31	14:54:03	Harthaus	S8	Flughafen
7	2016-09-01	12:59:24	Otterfing	S3	Maisach
8	2016-09-07	09:40:39	Otterfing	S3	Mammendorf
9	2016-09-13	08:59:49	Otterfing	S3	Maisach
10	2016-09-20	14:20:30	Otterfing	S3	Mammendorf
11	2016-09-22	19:19:26	Otterfing	S3	Mammendorf
12	2016-10-12	08:59:23	Otterfing	S3	Maisach
13	2016-10-19	17:00:28	Otterfing	S3	Mammendorf
14	2016-10-21	08:59:38	Otterfing	S3	Maisach

Table 2.2.: List of conducted surveys.

2.5.1. Data format

The data, which was collected using the mobile app, can be exported to CSV files. The exported CSV files reflect the database schema, which is illustrated in figure 2.4. In the following, the structure and fields of each CSV file are described. There are some special features in the data format which are listed here.

- During file export, enum values are converted to strings using their `toString()` method.
- The ORM library used in the mobile app stores the number 0 instead of `NULL` into empty foreign key fields. Thus, the number 0 in the CSV file stands for “no association”. During data preprocessing, the 0 is rewritten to `NA` (R’s representation for N/A).
- The IDs of log event entries in the CSV files might not always be unique or continuous because sometimes, manual fixes have to be made in this file, e.g. adding forgotten events or removing invalid events. This has no negative impacts on data processing.

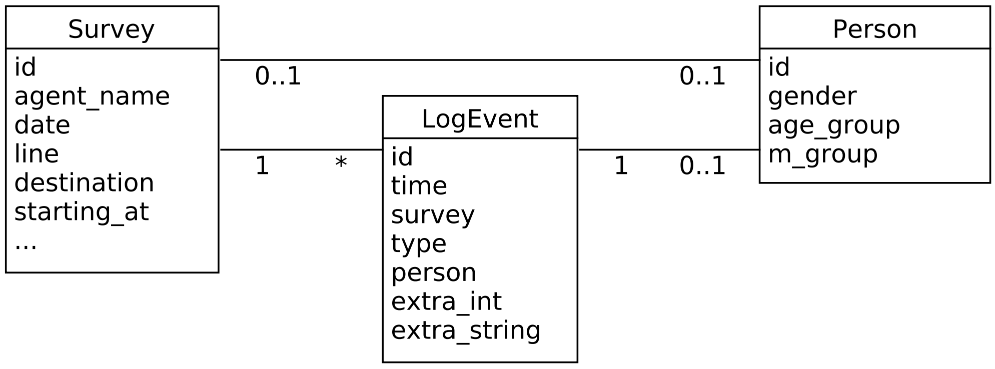


Figure 2.4.: ER diagram of the database schema. The exported CSV files have exactly the same schema.

Survey table

This table contains metadata about each survey.

	Field name	Description
1	`ID`	The ID column.
2	`AGENT`	The investigator's ID from the person table.
3	`AGENT_NAME`	Name of the investigator (because there is no name field in the person table).
4	`DATE`	Date of the survey in the ISO 8601 format (`YYYY-MM-DD`).
5	`DESTINATION`	The train's final destination.
6	`DOOR_NO`	To describe the position of the observed seats within a train wagon: doors are counted from the train's front. Two opposing doors are not counted twice. For example, the number 1 denotes the compartment between door (or entrance area) 1 and 2. Note: This is deprecated; `COMPARTMENT_NO` will replace this column.
7	`LINE`	The train line, e.g. "S3" (in Munich).
8	`STARTING_AT`	The station where the investigator enters and starts the survey.
9	`TRAIN_NUMBER`	The train's ID number.
10	`TRAIN_TYPE`	The train type, e.g. ET423 (Munich's S-Bahn).
11	`WAGON_NO`	To describe the position of the observed seats: The wagon number, counted from the front.

Table 2.3.: Columns of the survey table.

Person table

This table contains anonymous data about the observed persons.

	Field name	Description
1	`ID`	The ID column.
2	`AGE_GROUP`	The person's age group. This is a string field. Possible values are defined in the enum `AgeGroup`, see figure 3.2.
3	`GENDER`	The person's gender. This is a string field. Possible values are defined in the enum `Gender`, see figure 3.2.
4	`M_GROUP`	The ID of the group this person is member of. Persons are member of the same group, if they are family, friends, colleagues, etc.

Table 2.4.: Columns of the person table.

Log-event table

This table contains all logged events. From this data, the situation at any point in time can be restored.

	Field name	Description
1	`ID`	The ID column.
2	`EVENT_TYPE`	The type of the logged event. This is a string field. Possible values are defined in the enum `LogEventType`. See section 2.1 for a full list of possible events. See also figure 3.2 for how it is used in the entity model.
3	`EXTRA_INT`	This is an extra integer field. Depending on the event type it might store extra information e.g. the number of standing persons.
4	`EXTRA_STRING`	This is an extra string field. Depending on the event type it might store extra information e.g. the new driving direction. During preprocessing, the empty string is rewritten to `NA`.
5	`PERSON`	The ID of the corresponding person. For some event types, this field is empty.
6	`SEAT`	The number of the corresponding seat. For some event types, this field is empty.
7	`SURVEY`	The ID of the associated survey.
8	`TIME`	The event's time in this format: `hh:mm:ss`.

Table 2.5.: Columns of the log-event table.

2.5.2. Data preprocessing

First, the three data tables introduced above (see section 2.5.1) are loaded into memory:

```r
surveyData   = readCsvFile('SURVEY')
personData   = readCsvFile('PERSON')
logEventData = readCsvFile('LOG_EVENT')
```

Then, some consistency checks are performed using the R unit test framework testthat (see section 2.5):

```
# Convert TIME from factor to string to enable check for monotonicity
logEventData = mutate(logEventData, TIME = as.character(TIME))

test_that('data is valid', {
    expect_that(surveyData, has_no_duplicate_ids())
    expect_that(personData, has_no_duplicate_ids())
    expect_that(logEventData, has_monotonic_increasing_column('ID'))
    expect_that(logEventData,
                has_monotonic_increasing_column_per_survey('TIME'))
})
```

Before the data is used for further analysis, the following preprocessing steps are applied:

1. Remove duplicate `INITIALIZATION_END` events. For each survey, these events marking the end of the initialization phase and the start of the real-time data collection. Sometimes, there are more than one of these events per survey. For example, investigators might press the *Start survey* button accidentally, before they have captured the initial situation completely, and then they go back to correct the mistake.

2. Convert missing values to `NA`. Some unknown values in the CSV are encoded as 0 or the empty string. Amongst others, missing foreign key values, which should actually be database `NULL`, are encoded as 0. This is due to a limitation of the used ORM library (see section 3.1.1). In R the standard representation for missing values is `NA`.

3. Change data types from `factor`[2] to `character` for some fields. By default, R reads strings as `factor` variables from CSV. This is desired for many fields, e.g. the train line and the final destination. However, it is not desired for times, dates, and text.

4. Add a column for the start time to the survey data. The start time of a survey is the time of its corresponding `INITIALIZATION_END` event.

[2]In R, a `factor` is the data type equivalent of an enum in other languages.

5. Sort the log events in their natural order. For further processing, the log events must be sorted first by ID, then by time (within groups of a survey). The IDs are (weak) monotonic increasing over all events. Therefore sorting first by ID is a save and fast approach. Note that in the original CSV file the events are sorted, but during preprocessing the order can change.[3]

2.5.3. Data processing

Now there are three clean datasets for analysis: `surveyData`, `personData`, and `logEventData`. Together, they contain a huge amount of information and allow to reconstruct the train ride in detail. For this work, I focus on seating data, that is data on people's seating behavior in trains. To obtain the seating data, the original datasets must be transformed, which is described in the following subsections.

Background

The original datasets are not yet usable for deeper analysis of seating behavior. Although the log events contain a lot of sit-down events, there is only little information to be extracted. Each sit-down event, combined with the corresponding person, has about a dozen of features (see section 2.5.1): It captures a specific person that sits down on a specific seat at a specific time.

This cannot answer questions like "Do persons prefer to sit down with others of around the same age?". It cannot even answer simpler questions like "Do persons prefer window seats?", because it is unknown if window seats are available at the moment.

These questions show, that there is information missing: Information on the situation in the compartment before and after the person sits down. In the following, the situation (seat occupation and other conditions) is called *state*.

The data collection app could be programmed to compute the state immediately during the survey, but this has some disadvantages:

- Storing the state from before and after each event bloats the mobile device's database with redundant data.

[3]For example, the join operation in procedure 1 (Remove duplicate `INIT…`) changes the order.

- Computing the state during the data collection is hardcoding part of the data analysis in the app. However, the pure log events are independent of any analysis (seating behavior, social behavior, etc.).
- After a few surveys one might need additional state features for data analysis. Computing the state during the data collection makes it hard to reconstruct additional features for earlier datasets, which were generated with an older version of the app.

A better way to obtain the state is, to play back all events from the beginning and capture relevant state conditions for each sit-down event. This is done with the following command:

```
seatingData = generateMoreData(surveyData, personData, logEventData)
```

Generating more data

The function `generateMoreData` generates extra data for events by playing back all events (separately for each survey). During the playback, a state object is maintained. Listing 2.1 shows this algorithm for one survey.

```
state = newState()
for (j in 1:nrow(logEvents)) {
    event = logEvents[j, ]
    stateBefore = state
    state = updateState(state, event)
    data = collectData(data, stateBefore, state, event)
    # printState(state)
}
```

Listing 2.1: The algorithm for generating extra data for events of a given survey.

The function `collectData` in listing 2.1 is particularly important: It generates the records of the seating data table. It takes four parameters:

1. The data frame to which new records are added.
2. The state before the event was applied.
3. The state after the event was applied.

4. The event, that is, one row of the log event data table.

The function returns an updated version of the data frame that was passed as first parameter.

Feature computation

The function `collectData` needs to store some data across several calls as an internal state. For example, it needs to keep track of whether the `INITIALIZATION_END` event (table 2.5) has already occurred. In other object-orientated languages I would implement a `DataCollector` class with a `collect` method and member variables for the internal state. In R, Object-orientated programming (OOP) is supported in three different ways: the S3, S4, and R5 approach (Bare, 2012). Either way, the definition of classes in R is not very convenient or readable compared to other languages.[4] Therefore, I use a simple closure for the `collectData` function. A closure is, when a function returns a second function and the second function—when called—has access to the environment of the first function (that is, local variables defined there).

The implementation is printed in listing 2.2. The outer function defines local variables, which are used as internal state for the inner function. The inner function is then returned as the result value of the outer function. The client can use the `collectData` function like this:

```
collectData = createCollectDataFunction(logEventData)
data = collectData(data, stateBefore, stateAfter, event)
```

```
createCollectDataFunction = function(logEventData, personData) {
    # closure variables:
    collectionStarted = FALSE

    # collect data function:
    function(data, stateBefore, state, event) {
        isEventType = function(x) isEventOfType(event, x)
```

[4] I use simple S3 classes and methods to handle different event types in the function `updateState` (listing 2.1).

```
        if (collectionStarted && isEventType('SIT_DOWN')) {
            newRow = data.frame( # single-row data frame
                survey               = event$SURVEY,
                person               = event$PERSON,
                seat                 = event$SEAT,
                group                = getGroupOfPerson(event,
                    personData),
                nPersonsCompartment = getNumberOfPersonsInCompartment
                    (stateBefore),
                ...
            )

            data = rbind(data, newRow)

        } else if (isEventType('INITIALIZATION_END')) {
            collectionStarted <<- TRUE
        }

        data
    }
}
```

Listing 2.2: Collector function to collect and compute data for further analysis.

This function is designed to be easily extensible. To add a new feature to the resulting data, only an additional key-value pair has to be added to the `newRow` list. The value can be computed in a separate function. Input for the computation can be any of the function's parameters or variables from the internal state.

Seating data

The seating dataset now contains detailed and easy to analyze data on people's seating behavior. Each row of the seating dataset corresponds to a sit-down event in the log event data. Table 2.6 describes the available columns of the seating dataset. Some columns refer to a number of a seat group. Seat groups are numbered within one compartment from left to right and from the front to the rear.

There are 190 data points (rows) in the seating dataset. 14 surveys have been conducted and the behavior of about 190 individuals was observed.[5]

There are 43 data points associated with 19 distinct groups of persons. Although this is not enough data to develop solid social models, it might be interesting for future research when more data is collected.

[5]It cannot be guaranteed that one person is not observed multiple times across different surveys. However, because of different times, places and positions within the train, this is not very likely and should not be a problem.

	Field name	Description
1	`survey`	The ID of the associated survey.
2	`person`	The new person's ID from the person table.
3	`seat`	The seat's number where the new person sits down.
4	`group`	The person's group ID. `NA`, if the person is not member of a group. See table eftab:person-data-columns for details.
5	`nPersonsCompartment`	Number of other persons sitting in the compartment.
6	`nPersonsSeatGroup`	Number of other persons sitting in the seat group where the new person sits down.
7	`nPersonsSeatGroup1`	Number of other persons sitting in seat group 1.
8	`nPersonsSeatGroup2`	Number of other persons sitting in seat group 2.
9	`nPersonsSeatGroup3`	Number of other persons sitting in seat group 3.
10	`nPersonsSeatGroup4`	Number of other persons sitting in seat group 4.
11	`seatGroup`	Number of the seat group where the new person sits down.
12	`seatSide`	The chosen seat's position—window or aisle side. Possible values: `WINDOW`, `AISLE`.
13	`seatDirection`	The chosen seat's facing direction. Possible values: `FORWARD`, `BACKWARD`.
14	`direction`	The train's current driving direction. Possible values: `FORWARD`, `BACKWARD`.
15	`personNext`	The ID of the person sitting directly next to the new person. `NA`, if there is no person on that seat.
16	`personAcross`	The ID of the person sitting directly across from the new person. `NA`, if there is no person on that seat.
17	`personDiagonal`	The ID of the person sitting diagonally opposite to the new person. `NA`, if there is no person on that seat.

Table 2.6.: Columns of the seating dataset.

2.6. Passenger counts

The Münchner Verkehrs- und Tarifverbund (MVV), the company, which organizes and coordinates Munich's public transport system, regulary conducts passenger surveys. As part of these surveys, the number of entering and leaving passengers is counted at every stop on a given train line by an investigator who sits on the train.

The MVV made data on passenger counts available for this work. This data is used in chapter 5 to provide realistic passenger numbers for simulation when verifying and validating the model.

2.7. Spawn distribution for sources

This thesis focuses on how and where people sit down in a train. However, people must first be spawned before they can enter the train and choose a seat. Spawning is the creation of persons during the simulation and it happens at special spots in the scenario, called sources.

People enter the train at train stops through the doors. I assume that there is a distribution that describes the time deltas between two subsequent persons entering the train. This time delta is known as the intermediate arrival time. In the following, the distribution for the intermediate arrival time is called spawn distribution.

2.7.1. Data collection

To find a suitable spawn distribution and its parameters, I conducted a data collection in Munich's S-Bahn:

- The investigator conducting the study sits on a place from where the entrance area is well observable.
- The investigator has a laptop running a program that allows logging events with timestamps.
- At each stop, the time of the following events are logged:
  1. The door starts to open.
  2. A person enters the train.
  3. The door shuts.

Persons exiting the train are not counted.

The program used to create the event log only writes timestamps into a text file. The event types have to be inferred later.

The data collection took place on multiple days between April 12 and May 31, 2016[6]. Data has been recorded at a total of 46 stops[7].

2.7.2. Fitting a distribution

Preprocessing

As mentioned above, the raw data is a list of timestamps.

```
head(rawData, n = 5)

##         date     time     nanos
## 1 2016-04-12 16:03:44 420147976
## 2 2016-04-12 16:03:48 418088526
## 3 2016-04-12 16:03:56 941887317
## 4 2016-04-12 16:05:47 624178191
## 5 2016-04-12 16:05:54 338702583
```

The raw data must be prepared for further analysis:

1. For each timestamp, the time differences to the previous timestamp is calculated in fractional seconds.

2. Each row is then labeled with a stop ID and one of the following event types:

```
## [1] "door_closed"             "door_opens"
## [3] "first_passenger_enters" "next_passenger_enters"
```

To discriminate between stops, a threshold of 40 seconds is applied. Consecutive events with time differences below this threshold are labeled with the same stop ID. The threshold is derived from a histogram plot of time differences in figure 2.5.

[6] `grep '^#' survey/rate/rates.txt` (shell command line)

[7] Actually, this is not the number of stops but the number of times the door was open.

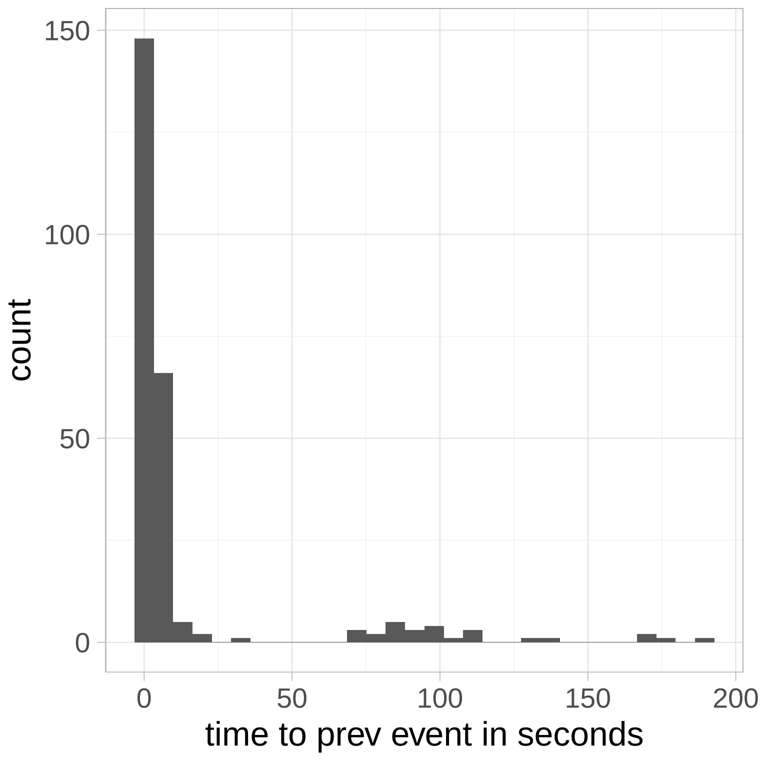


Figure 2.5.: Histogram of the time differences between logged events.

The threshold could indeed also be around 25 seconds. I examined this situation in the pre-processed data and found that the numbers of door open and door close events do not match. Furthermore, 40 seconds seems to be more realistic than only 25 seconds.

After data preprocessing, the dataset looks like this:

```
head(data[, c("time","stopId","eventType","secondsToPrevEvent")])

##       time stopId              eventType secondsToPrevEvent
## 1 16:03:44      1             door_opens                 NA
## 2 16:03:48      1 first_passenger_enters               4.00
## 3 16:03:56      1            door_closed               8.52
## 4 16:05:47      2             door_opens             110.68
## 5 16:05:54      2 first_passenger_enters               6.72
## 6 16:06:02      2            door_closed               8.45
```

Results

Only time differences between "passenger enters" events can be used to examine the intermediate arrival time. Because the time differences are stored as time to the *previous* event, only those events are of interest, where a second, third, fourth, and so forth person enters the train:

```
relevantEvents = subset(data, eventType == "next_passenger_enters")
```

The frequency distribution showed in figure 2.6 can be approximated with an exponential distribution:

$$f_\lambda(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

with $\lambda = 1.52$s, which is the mean of the time differences i.e. the `secondsToPrevEvent` column.

In section 3.3.1 I describe how spawn distributions are implemented in VADERE.

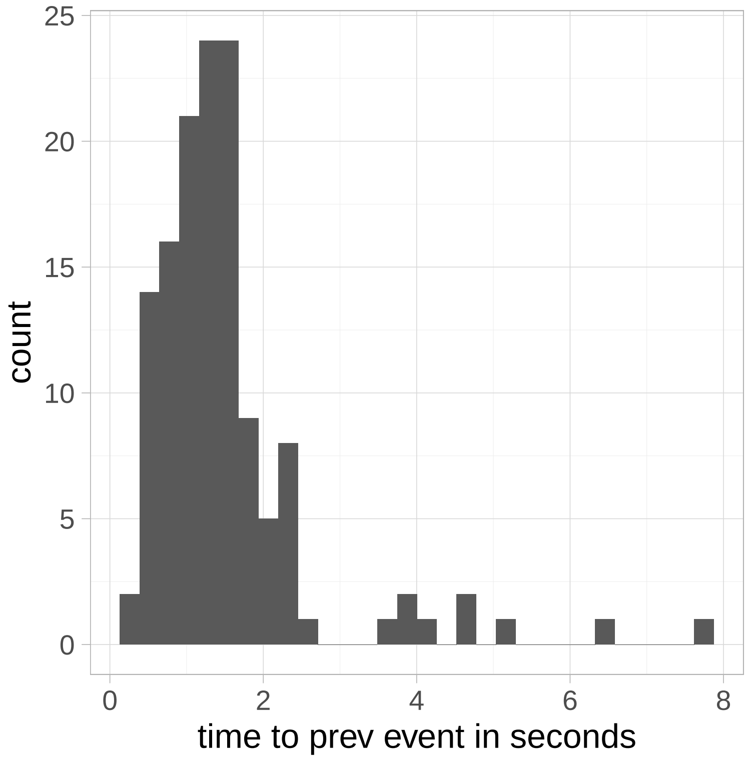


Figure 2.6.: Histogram of the time differences between passengers entering the train.

2.8. Summary

I gathered requirements for a data collection app and presented the resulting Android mobile app. I illustrated the UI and the workflow, and I conducted a usability study to improve the user experience.

I described the outline and format of the collected data and the procedure of data (pre)processing. In the procedure of data processing, the data is transformed from a general form to a form that can specifically be used to analyze seating behavior.

In another data collection, I explored the intermediate enter times of train passengers as a basis for realistic simulations. I fitted a distribution which can be used to parameterize so-called sources where simulated new passengers are created.

3. Software

This chapter starts with a discussion on architecture, design, and implementation aspects of the mobile app for the data collection. Then, new features and changes in the VADERE simulation software are described. In preparation for this work, many changes and enhancements were necessary, which are documented in the following sections. The last three sections present a tool to generate train scenarios, the seating model's implementation, and the subsystem used for model verification.

3.1. Mobile app for data collection

In chapter 2, requirements for an data collection app are gathered and the resulting app is introduced. This section describes design and implementation of this app.

3.1.1. Software design

This section describes some interesting aspects of the app's software design.

Actions

To conduct the survey, the UIs in figure 2.2c and 2.2d (section 2.3) provide a number of actions. Examples for these actions are logging of events ("sit down", "leave", etc.) and assigning properties to persons (age, gender). Requirements related to the actions are:

- The actions must be undoable to allow for correcting mistakes.
- There is a type of actions that requires further user action. For example, for defining a group of persons, the action is started but not finished before the members are selected. These actions are called pending actions and they all have in common that the user have to select seats to finish the action.

- For pending actions, there must be a way to notify the UI when the action has been finished or canceled. For example, the UI might have to change some icon color back to its default.

To implement the actions and requirements, I use a pattern similar to the Java Swing `UndoManager` (which is not available on Android). Each type of action has its own class and all actions are carried out by an `ActionManager` object. Figure 3.1 shows an simplified UML class diagram of this design.

The actions need a reference to the action manager because some of them have to perform subsequent actions and others have to interact with the UI, which is only available through the action manager.

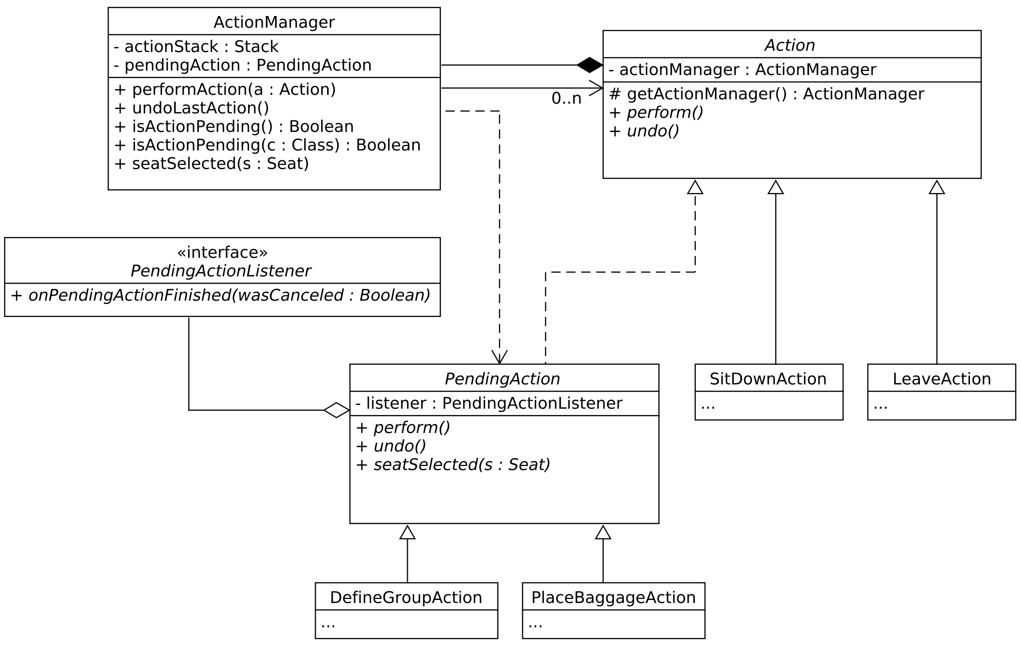

ActionManager
- actionStack : Stack
- pendingAction : PendingAction
+ performAction(a : Action)
+ undoLastAction()
+ isActionPending() : Boolean
+ isActionPending(c : Class) : Boolean
+ seatSelected(s : Seat)
Action
- actionManager : ActionManager
# getActionManager() : ActionManager
+ perform()
+ undo()
0..n
«interface»
PendingActionListener
+ onPendingActionFinished(wasCanceled : Boolean)
PendingAction
- listener : PendingActionListener
+ perform()
+ undo()
+ seatSelected(s : Seat)
SitDownAction
...
LeaveAction
...
DefineGroupAction
...
PlaceBaggageAction
...

Figure 3.1.: UML class diagram of the action manager, actions, and pending actions.

All related classes reside in the Java subpackage `.actions`; see section 3.1.2 for more technical details.

Model and database schema

With the library Sugar ORM, I use Object-relational mapping (ORM) for database access. ORM is a technique to access a database from an object-orientated programming language using objects that correspond to the database entities.

In the case of Sugar ORM, the database scheme is primarily defined through Java classes, which describe the database entities. These classes reside in the Java subpackage `.model` and they are subclasses of `SugarRecord`.

In the following I use the terms *entity class* for the correspondents of database entities and *model class* for classes used as model behind the app's view (in terms of MVC).

Model and entity classes are almost identical. However, model classes have some properties that are only relevant for the running app but should not be persistent. To solve this problem there are a number of possible solutions:

1. The model class can be a subclass of the entity class. In this case, the entity objects cannot directly be reused for the model objects—data must be copied to new model objects.

2. The model objects can store a reference to their corresponding entity objects.

3. Entity and model class can be the same class as long it is guaranteed that model-only properties are not persisted. This is technically possible with Sugar ORM by applying its Java annotation `@Ignore` to these properties.

I implemented the last option: Shared classes for entities and model. This keeps the total number of classes low and does not require copying data or managing extra objects.

Figure 3.2 illustrates the simplified model. The container class for seats, which defines the seating layout, is left out because it is more related to the UI. Classes that are also entity classes are marked with «entity». On the class `Person` (in the center), there are a few things to note:

- The properties `agent` and `disturbing` should not be persisted—they are ignored by the ORM library (`@Ignore` annotation). The property `agent` specifies whether the person is the investigator conducting the survey. This information is already stored in the survey entity object. The property `disturbing` specifies whether the person is currently disturbing. Because this can change over time, it is saved as a log event, instead. Still, both are properties of the person object and both are required as state information for the UI.

- With the property `mGroup` persons can be assigned to a group of friends or colleagues. This might be interesting to observe seating behavior of groups. The "M" in the entity class name `MGroup` can be interpreted as "mates" or "model". The name `Group` could not be used because of a bug in the ORM library.[1]

- The properties `gender` and `ageGroup` are enums. Both enums have a special `NA` value for "not available" (N/A). This way I can directly use the enums as the data source for the Android Spinner components, which allow the user to select a value from a list. Otherwise I would have to hard-code the values or provide an extra wrapper class which adds an N/A value. To assure that the properties cannot be `null`, they have the `@NotNull` annotation.

Android Fragments

Android Fragments are reusable portion in an UI.[2] They gather UI controls and behavior in a self-contained form.

The app's main UI portion is the seating layout which represents the current state of the data collection (seats, persons, baggage, ...). This part is self-contained and it is required twice in the app: for the initialization phase and during the data collection. Therefore it is implemented as a fragment in the class `SeatsFragment`. As a fragment, it takes parameters:

- A survey object that holds the meta data of the survey (date, train line, direction, ...).

[1] A class named `Group` produces an error in the current version of Sugar ORM because `GROUP` (case-insensitive) is a reserved keyword in SQL. This is a bug and there is an open issue on this: https://github.com/satyan/sugar/issues/98

[2] https://developer.android.com/guide/components/fragments.html

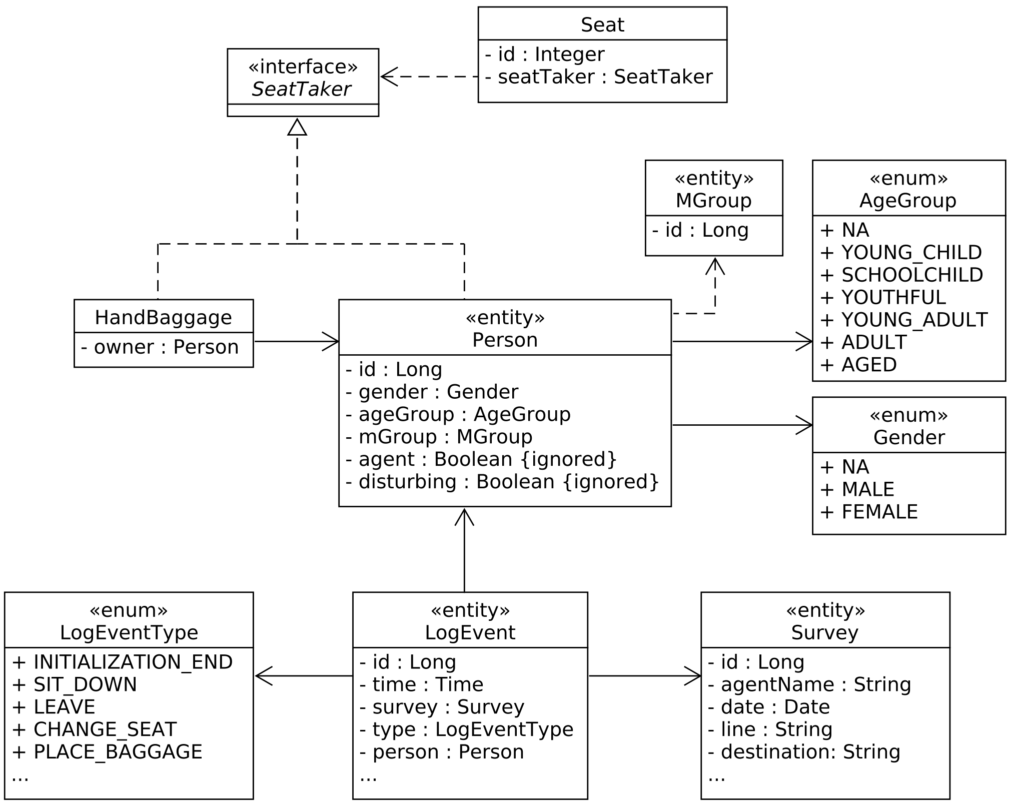


Figure 3.2.: This is an UML class diagram of the app's entity model.

- An optional SeatsState object that holds the state for initializing the fragment. This parameter is used to pass the initial state to the next screen (figure 2.2c and 2.2d). If it is missing, all seats are initialized empty.
- The train direction. It is visualized with an arrow and part of the fragment's internal state.

Fragments are also used to define dialog UIs, e.g. the dialog to set and update person properties.

Database export to CSV

For further data analysis, I need plain-text CSV files on the computer. Therefore, the app must be capable to export its database tables to CSV files and save these files to the phone's storage. The phone can then be connected to the computer and the files can be copied.

Database export and file I/O can be a demanding and time-consuming operation. To not freeze the UI during the export process, I use an Android AsyncTask, which starts a thread in the background. The following listing shows the entry-point to the export implementation. The method's signature is specified by the `AsyncTask` class.

```
protected Boolean doInBackground(Void... params) {
    try {
        exportAllTables();
    } catch (IOException e) {
        Log.e(TAG, "error while exporting: " + e.getLocalizedMessage
            ());
        return false;
    }
    return true;
}
```

Listing 3.1: Entry-point method to the database export.

The following methods are called during the export process:[3]

- `private void exportAllTables()`
- `private List<String> getTableNames()`
- `private void exportTable(SQLiteDatabase db, String tableName, File directory)`
- `private Cursor queryTable(SQLiteDatabase db, String tableName)`
- `private String[] getFieldsAsStringArray(Cursor cursor)`

Two problems arose during implementing the database export:

[3] `grep 'private .*(.*)' seating-datacollection/**/DatabaseExportTask.java` (shell command line)

1. The Android API for saving files does not work consistently. Android can save files to internal and external storage.[4] Files saved to the internal storage are only accessible by the app. Therefore, when exporting files, the external storage must be used. The development device, a Nexus 6 smartphone, does not have an SD card. Thus, the API call Environment.getExternalStorageState() yields a state other than the required Environment.MEDIA_MOUNTED. Despite of this result, the concept of external storage exists on a Nexus 6 and files can be saved into paths returned by Environment.getExternalStorageDirectory() and similar methods. This inconsistency seems to be an Android problem because there is no other `MEDIA_*` state for the case of Nexus 6 and the official guide on data-storage (referenced above) does not mention this problem as of 2016-08-15.

2. Exported files are not visible to the computer until the phone is restarted. The Nexus 6 device can be connected to the computer using the Media Transfer Protocol (MTP). This is the only mode that allows users to access the phone's storage. While the exported files are visible in the device's file manager app, they are initially *not* visible (and subsequently not up-to-date) to the computer. According to (Müller, 2014), this is because no media scan is triggered when files are saved to external storage, despite I am using the standard APIs. I tried different default folders (`Documents`, `Download`, …) as well as the root folder of the external storage—they all have the same problem. The source mentioned above lists a number of ways to initiate a media scan. Some ways do not apply for the development device as it is not rooted and it does not have CyanogenMod[5] installed. Other ways do not work e.g. making a photo to trigger a media scan or start the scan programmatically. This is the only work-around I have found so far: Restart the phone after each database export before copying the files to the computer.

3.1.2. App development

Here are some technical details on the app development.

[4] https://developer.android.com/training/basics/data-storage/files.html

[5] http://www.cyanogenmod.org/

Target Platform	Android
Target SDK version	23
Minimum SDK version	21
IDE	Android Studio 2.1.2
Development platform	Arch Linux
Development device	Motorola Nexus 6, Android 6.0.1
App name	`SeatingDataCollectionApp`
App package	`edu.hm.cs.vadere.seating.datacollection`

Table 3.1.: Table of technical details on the app development.

For the development of the Android app, I used Google's Android Studio 2.1.2. It is "The Official IDE for Android"[6] and it has very good support for the Android SDK.

The app has the following library dependencies, defined in the Gradle build file:[7]

- Sugar ORM, version 1.5 for Object-relational mapping (ORM) between the SQLite database and Java objects.
- opencsv, version 3.7 for export from the SQLite database to CSV.

This app is open-source, available under the GPL v3 license, and hosted at GitHub: `https://github.com/schoettl/seating-datacollection`

3.1.3. Evaluation of a different approach

Before I committed to implement the mobile app for data collection, I evaluated a different approach using plain text files together with a programmable text editor[8] and shell scripts. This idea seemed promising because of two main reasons.

First, there are some required features that an editor like VIM or Emacs already has built in. In contrast, in an app, these features must be implemented again. Some of these features are quick selection of UI elements, support for multiple

[6] `https://developer.android.com/studio/` (accessed: 2016-08-09)

[7] `grep ' compile ' seating-datacollection/SeatingDataCollection/app/build.gradle` (shell command line)

[8] In this section, "editor" refers to a text editor. Notepad++, VIM and GNU Emacs are examples for text editors.

concurrent views, undo and redo capabilities, and facilities to correct the event log.

Second, the collected data are finally required in plain text CSV files for further analysis. An editor is predestined for handling these files. In contrast, an app would use a database and would export the CSV files later which then must be transferred to the computer.

I chose the editor VIM as the platform and prototyped a tool for data collection according to the requirements described in section 2.1. A rough description of the software architecture follows below. Advantages, drawbacks and reasons why I stopped following this approach are listed next.

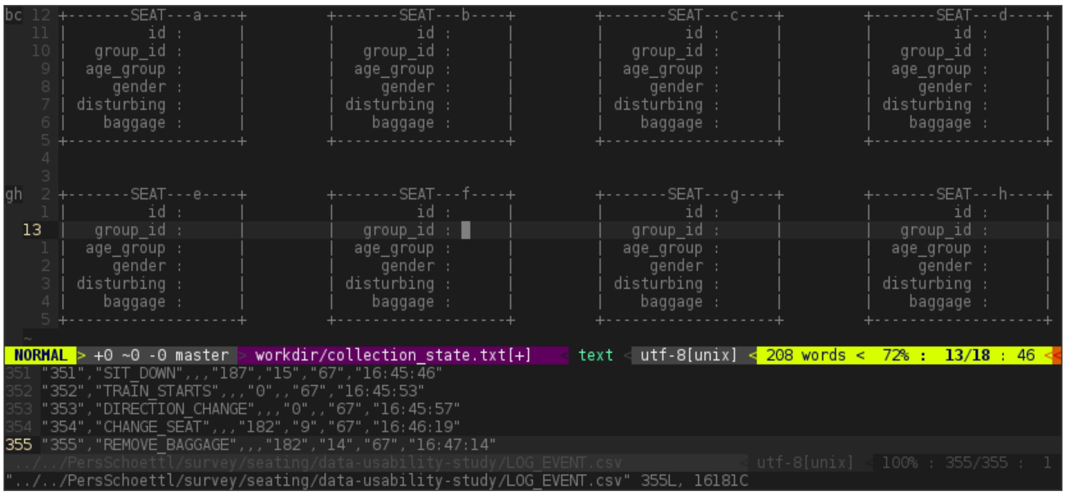
Architecture

The architecture consists of the following components:

- Plain text CSV files for data. These files get appended during data collection and can later directly be used for data analysis and statistics.
- Plain text files representing the UI. These template files are copied at the beginning of the data collection and then displayed in the editor. Together with the editor, they form the User interface.
- Shell scripts as controller component. A few shell scripts act as a controller. To start a data collection, the script `start-survey.sh` is invoked at the command line. This starts the editor with the UI. Another shell script converts the current state of the UI (a plain text file) into one or more lines and appends them to the data CSV files.
- Configuration and shortcuts in the editor to make the UI interactive. The editor must be tweaked to enable or trigger certain actions. For example, a keyboard shortcut has to be configured, to invoke the shell script that writes the current state to the data CSV files.

Figure 3.3 shows a screenshot of the main UI from a prototype.

Regarding usage efficiency, performing actions in the editor approach is comparable to a touch UI of a mobile app. However, it cannot be denied that learning how to use an editor like VIM efficiently, is a long way to go. As an example, to position the cursor to enter the age group for the person sitting on seat “a” (see figure 3.3), only four key strokes on a computer keyboard are required[9].



```
bc +-------SEAT---a----+       +-------SEAT---b----+       +-------SEAT---c----+       +-------SEAT---d----+
   |           id :     |       |           id :     |       |           id :     |       |           id :     |
   |     group_id :     |       |     group_id :     |       |     group_id :     |       |     group_id :     |
   |    age_group :     |       |    age_group :     |       |    age_group :     |       |    age_group :     |
   |       gender :     |       |       gender :     |       |       gender :     |       |       gender :     |
   |   disturbing :     |       |   disturbing :     |       |   disturbing :     |       |   disturbing :     |
   |      baggage :     |       |      baggage :     |       |      baggage :     |       |      baggage :     |
   +--------------------+       +--------------------+       +--------------------+       +--------------------+


gh +-------SEAT---e----+       +-------SEAT---f----+       +-------SEAT---g----+       +-------SEAT---h----+
   |           id :     |       |           id :     |       |           id :     |       |           id :     |
   |     group_id :     |       |     group_id : █   |       |     group_id :     |       |     group_id :     |
   |    age_group :     |       |    age_group :     |       |    age_group :     |       |    age_group :     |
   |       gender :     |       |       gender :     |       |       gender :     |       |       gender :     |
   |   disturbing :     |       |   disturbing :     |       |   disturbing :     |       |   disturbing :     |
   |      baggage :     |       |      baggage :     |       |      baggage :     |       |      baggage :     |
   +--------------------+       +--------------------+       +--------------------+       +--------------------+
~
NORMAL > +0 ~0 -0 master > workdir/collection_state.txt[+]  <  text < utf-8[unix] < 208 words <  72% :  13/18 : 46 <
"351","SIT_DOWN",,,"187","15","67","16:45:46"
"352","TRAIN_STARTS",,,"0",,"67","16:45:53"
"353","DIRECTION_CHANGE",,,"0",,"67","16:45:57"
"354","CHANGE_SEAT",,,"182","9","67","16:46:19"
"355","REMOVE_BAGGAGE",,,"182","14","67","16:47:14"
../../PersSchoettl/survey/seating/data-usability-study/LOG_EVENT.csv          < utf-8[unix] < 100% : 355/355 :  1
"../../PersSchoettl/survey/seating/data-usability-study/LOG_EVENT.csv" 355L, 16181C
```

Figure 3.3.: Prototype of a user interface for a data collection using VIM and shell scripts.

Advantages

There are technical and personal reasons speaking for the editor approach.

One important technical aspect is the database. In Android this includes a relational database structure, database connection and version management, entity classes, Object-relational mapping (ORM), User interfaces, and export to CSV files on the external storage. These are many technical challenges, though ultimately I only need the exported CSV files for data analysis. In contrast, the editor approach operates directly on the final data files. This allows for simple appending of new lines and precise corrections. Using an editor, this comes without any of the challenges mentioned above.

Another reasons is my preference for reusing existing functionality instead of reimplementing it in an mobile app. Here is a list of some features that already exist in the editor:

[9] `` `a3j ``, provided that the seats have a cursor mark.

- Split windows to tile the screen into different areas are an integral part of VIM. One window could show the main UI while other windows could show the data files for monitoring and corrections.
- Undo and redo is built-in in editors like VIM. This makes it easy to undo mistaken inputs—in the UI displaying the seats as well as in the data files.
- Automatic reload of data files, when they get appended through the UI, is just a configuration option.

Finally, a personal reason for first looking for another approach is this: I have never designed or developed a mobile app before. Having an app with custom widgets, reusable components, context and option menus, support for undoing actions, database with abstraction layer, and file export is not trivial and maybe not the best way to learn Android programming.

Drawbacks

During the evaluation phase, some problems and difficulties with the editor approach came up.

- The UI text file (see figure 3.3) must be converted to a line of CSV to get appended to the data file. There are simple ways to parse the text file and get the required data, using tools like AWK, but this is not robust against slight mistakes in the UI text file. For example, if the user inadvertently uses a pipe symbol (|) in the text, the conversion might give incorrect or corrupted results.
- For many UI actions, the number of the seat, where the cursor is positioned, is required. To get the seat number from the cursor position, rectangle coordinates have to be defined; either by hard-codeding them or by parsing the UI text file. Parsing has the same problems as described in the previous paragraph.
- Users can only work with the program if they have a high experience with VIM. This means, that very few people interested in this kind of data collection, can actually reuse the program.

Therefore—despite the advantages of using VIM with plain text files—I decided to move on with the mobile app development.

3.2. VADERE crowd simulation

VADERE is an open source framework for the simulation of microscopic pedestrian dynamics (vad, 2016). It has been developed by Prof. Dr. Gerta Köster's research group at the Munich University of Applied Sciences.

VADERE facilitates the development of new simulation models that can be implemented in stand-alone Java classes and then be linked into the simulation framework. This promotes research in the field of pedestrian dynamics and enables the use for educational purpose (Seitz, 2016). Beside pedestrian dynamics, VADERE can also be used for simulation of traffic systems or particle systems.

In August 2016, VADERE was released as an open source project under the terms of the GNU LGPL. Its Git repository is hosted at a GitLab instance and can be found at: `https://gitlab.lrz.de/vadere/vadere`

3.3. Contributions to VADERE

3.3.1. Source controller

During a simulation, pedestrians are created (spawned) at certain spots in the scenario called sources. This process is controlled by source controller instances. I rewrote most parts of the source controller to enable spawn distributions (see section 2.7) and a maximum number of pedestrians to be spawned.

The following Java classes are involved in the implementation:

- `Source`—An object of this class represent a source where persons are spawned. All parameters are stored in the instance variable **`attributes`**, which is of the class `AttributesSource`.
- `AttributesSource`—An object of this class represent static parameters for a source. These parameters include an ID, a shape, and a start and end time amongst others. Objects of this class can be serialized to and deserialized

from JSON. This way, the parameters can be configured via JSON, e.g. in the GUI.

- `SourceController`—An object of this class has a source associated and actually creates and spawns new persons according to the source's parameters.
- `TestSourceController*`—These are unit tests for the `SourceController` class.

Originally, sources were designed to spawn persons only with a configurable constant rate. This is an example JSON configuration for the sources before distributions were introduced:

```
{
  "startTime": 10,
  "endTime": 15,
  "spawnDelay": 1,
  "spawnNumber": 2,
  ...
}
```

Listing 3.2: An example configuration for sources before distributions were introduced.

This source starts with spawning persons at 10s of simulation time. Until 15s (inclusive), every 1s, 2 persons are created.

The `SourceController` class

The `SourceController` class contains all the logic of spawning pedestrians. This class was largely rewritten to support the new features. The algorithms are implemented according to the principles of Clean Code (Martin, 2009) where readability and maintainability are main goals.

During the implementation of spawn distributions and the maximum spawn number, I extended the existing unit test class of the `SourceController` class and added new test methods. This way I can thoroughly test new functionality while making sure that my changes do not break existing tests.

Design and implementation of spawn distributions

In the following, I introduce a new design using spawn distributions to overcome the limitations of `spawnDelay`.

In the class `AttributesSource`, the field `spawnDelay` is deleted in favor of two new fields and one constant:

```
public static final String CONSTANT_DISTRIBUTION =
    ConstantDistribution.class.getName();
private String interSpawnTimeDistribution = CONSTANT_DISTRIBUTION;
private List<Double> distributionParameters = Collections.
    singletonList(1.0);
public void setDistributionParameters(List<Double>
    distributionParameters) {
public void setInterSpawnTimeDistribution(String
    interSpawnTimeDistribution) {
```

Listing 3.3: New fields for spawn distribution in class `AttributesSource`.

The field `interSpawnTimeDistribution` is the fully-qualified class name of a distribution class. Distribution classes must fulfill two requirements:

1. They must implement the interface `org.apache.commons.math3.distribution.RealDistribution` from the Apache Commons Math library[10]. The library also provides a wide range of standard distributions that can be used.

2. They must provide a public constructor with the following parameters: The first parameter must be an `org.apache.commons.math3.random.RandomGenerator`. Further constructor parameters make up the distribution's parameters. They are all of type `java.lang.Double`. The distributions can have any number of parameter. Only the `RandomGenerator` is mandatory.

The field `distributionParameters` defines the list of distribution parameters. Its length must match the number of `Double` parameters in the distribution's constructor.

[10] https://commons.apache.org/proper/commons-math/

The default spawn distribution is the constant distribution, which has the same effect as `spawnDelay` before.

For the train scenario, I can now use the distribution `org.apache.commons-.math3.distribution.ExponentialDistribution` with the parameter λ calculated in section 2.7.

Implementation of a maximum spawn number

There is another minor issue to be addressed: How to spawn a fixed number of pedestrians? This was possible before, e.g. to spawn two pedestrians, the parameters would have been: `startTime = 1, endTime = 2, spawnDelay = 1`. However, with spawn distributions, the time between two spawn events is not fixed anymore but random. Therefore, setting the end time is not applicable.

A solution is to add a another field to the `AttributesSource` class:

```
private int maxSpawnNumberTotal = NO_MAX_SPAWN_NUMBER_TOTAL;
public void setMaxSpawnNumberTotal(int maxSpawnNumberTotal) {
```

Listing 3.4: New field for maximum spawn number in class `AttributesSource`.

This parameter controls the maximum number of pedestrians that can be spawned. The default value zero means, that there is no maximum. Now, to spawn a fixed number of pedestrians,

1. `maxSpawnNumberTotal` is set to the desired number, and
2. `endTime` is set to a large value, e.g. `10e9` (1,000,000,000).

While in theory, the end time can still be the limiting factor, this is a direct and pragmatic solution, which does not require any changes in any existing scenario definitions. Listing 3.5 shows an example of how this attribute can be used.

Example usage

For example, a source can now be configured like this:

```
{
  "startTime": 10,
  "endTime": 100,
  "maxSpawnNumberTotal": 5,
  "interSpawnTimeDistribution": "org.apache.commons.math3.
      distribution.UniformRealDistribution",
  "distributionParameters" : [ 1, 3 ],
  ...
}
```

Listing 3.5: An example configuration for sources with a spawn distribution.

The two distribution parameters `[ 1, 3 ]` are the lower and upper limit of the uniform distribution. It can be looked up in the Apache Commons Math API documentation for the respective distribution.

3.3.2. Target controller

During a simulation, pedestrians are moving towards their individual targets. If they have no target, they do not move. If they have multiple targets, they move from one target to the next one in the order specified in their target list.

Targets are certain spots in the scenario and they have a variety of properties. For example, it can be defined if pedestrians are "absorbed" when they reach the target or if they continue to live in the scenario.

For each target, there is a target controller which handles the case when a pedestrian reaches its target. The following classes are related to the implementation of targets:

- `Target`—An object of this class represent a target. All its static properties are stored in an instance variable of type `AttributesTarget`.
- `AttributesTarget`—An object of this class defines all static parameters of a target. The parameters include an ID, a shape, and the "absorbing" property mentioned above. Attributes objects can be serialized to and deserialized from JSON. This is the way how targets are configured, e.g. in the Vadere GUI.
- `Agent`—An object of this class represents a pedestrian.[11] A pedestrian (as

[11] To be precise, there is a subclass `Pedestrian` but this subclass does not add any properties relevant for this section.

an agent) has a target list which stores the pedestrian's targets in the correct order. A pedestrian also has methods to access the target list and get the current target.

- `TargetController`—For each target, there is one target controller. The target controller detects when a pedestrian reaches its target and handles that event. The process of handling that event includes:
  - The controller removes the pedestrian from the scenario if the target is absorbing.
  - If the target is not absorbing, the controller updates the pedestrian's current target according to the pedestrian's target list.
- `PotentialFieldTarget`—Subclasses of this interface implement floor fields for navigation (Kirik et al., 2007, 2009; Köster and Zönnchen, 2014).

Beside a lot of refactoring in these classes, I corrected the design of the target list and implemented an observer pattern (Gamma et al., 1995) to notify clients about events at targets.

Pedestrian's target list

Initially a pedestrian's target list was a queue. The current target was always the first one in the queue and it was popped when a pedestrian had reached this target. New targets could be added to the end of the queue.

Some simulation models, however, required to query or switch back to previous targets. Some effort was made to change the queue to a list and introduce an index variable which points to the current target. This implementation was incomplete and did not provide a clean interface.

I corrected the design and the implementation of the target list. The affected classes were mainly `Agent`, `TargetController`, `PotentialFieldTarget`, and its subclasses. The updated API propagated into most simulation models, simplifying code and logic therein.

The class `Agent` now provides the following methods to work with the target list:

- `boolean hasNextTarget()`
  Queries if the pedestrian currently has a target.

- `int getNextTargetId()`
  Gets the ID of the pedestrian's current target. This method throws an exception if the pedestrian currently has no target.

- `List<Integer> getTargets()`
  Gets the list of the pedestrian's target IDs. This list can be modified or new targets can be added.

- `void incrementNextTargetListIndex()`
  Increments the index which points to the pedestrian's current target so that it points to the next target in the list.

- `void setNextTargetListIndex(int nextTargetListIndex)`
  Sets the index which points to the pedestrian's current target.

- `int getNextTargetListIndex()`
  Gets the index which points to the pedestrian's current target.

The methods in this list are ordered by importance. The first methods are used most frequently.

Details Beside simulation models, the target controller and the `Potential-FieldTarget` classes use this API.

The target controller checks its target area for pedestrians and processes the "target reached" program for pedestrians that have a matching target. The methods `hasNextTarget` and `getNextTargetId` are used to compare the controller's target with the pedestrian's target. If the target is not absorbing, the controller uses the method `incrementNextTargetListIndex` to update the pedestrian's target.

The `PotentialFieldTarget` classes implement potential floor fields which are used to navigate pedestrians towards their targets. They have two methods to get a potential value and a potential gradient at a specific point related to a specific target. Previously, the parameters for this methods were the position, the pedestrian, and the target list (which was redundant). Now, only the position and the pedestrian are passed as arguments. The implementations use the methods `hasNextTarget` and `getNextTargetId` to get the target.

Backwards compatibility For backward compatibility, the target list can still be used as a queue. Code for compatibility is marked as deprecated and it is well separated from the new implementation so that it can eventually be dropped easily.

Backward compatibility can be enabled by calling `setNextTargetListIndex(-1)` (or specifying this value in the JSON configuration). In this mode, the methods defined above treat the target list as a queue. For example, `getNextTargetId()` always returns the first target in the list.

Target listener

With target listeners, I implemented a way to inform clients about events that occur on targets, e.g. when a pedestrian reaches its target. The implementation's design is based on the Observer pattern.

Before I implemented that feature, when a client wanted to know if a pedestrian reached its target, the client had to compare the pedestrian's position and target at every simulation step. Many clients implemented this polling algorithm which led to redundant and error-prone code and an inefficient simulation loop.

Target listeners provide a more event-driven way. A client can register its listener(s) on a target. The target controller notifies the registered listeners about events.

Listing 3.6 shows the related methods in the `Target` class. The `TargetController` class has the method `notifyListenersTargetReached` to notify registered listeners when a pedestrian has reached its target. Currently, only this event type is supported but other event types can be added easily.

```java
public void addListener(TargetListener listener) {
    targetListeners.add(listener);
}

public boolean removeListener(TargetListener listener) {
    return targetListeners.remove(listener);
}

/** Returns an unmodifiable collection. */
public Collection<TargetListener> getTargetListeners() {
    return Collections.unmodifiableCollection(targetListeners);
}
```

Listing 3.6: The `Target` class' interface for working with target listeners.

Target listeners are used in the seating model (section 3.5) and in the data processor for its verification (section 3.6) as well as in other simulation models and data processors.

3.3.3. Model framework redesign

During the preparations of releasing VADERE as open source, we redesigned the way how simulation models are defined.

Simulation models implement the logic of a model and are often hooked into the simulation loop. Examples for such models are the Optimal Steps Model (Seitz and Köster, 2012; Seitz, 2016), the floor field models, and the seating model.

Requirements are that it must be possible to configure the models in JSON and to combine independent models without having hard dependencies in the code.

Resulting design

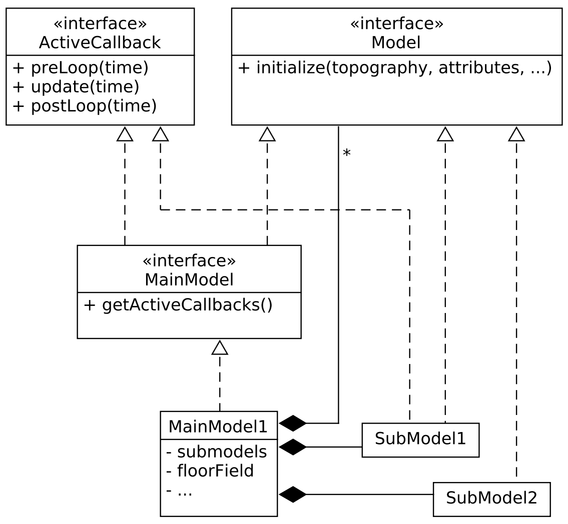


Figure 3.4.: Simplified UML class diagram of the simulation model framework.

Figure 3.4 shows a class diagram of the resulting design. The interface `Model` declares the `initialize` method. All models have to implement this method. The parameters passed to `initialize` are the topography, a list of attributes from which the model can pick its own attributes, and a seeded random generator. By only using the passed random generator, simulations will be reproducible.

The interface `ActiveCallback` declares methods that serve as hooks in the simulation loop. Models that need hooks in the simulation loop implement this interface. There are also models that do not need these hooks, these models do not implement this interface.

The interface `MainModel` combines the two other interfaces and adds the method `getActiveCallbacks`. The simulation loop uses this method to call all `ActiveCallback` hooks. The interface `MainModel` also serves as a marker interface for main models.

A simulation scenario must always have exactly one main model. All other models are managed by this main model. For example, the Optimal Steps Model is a `MainModel`; floor fields, social models, and the seating model are submodels. Listing 3.7 shows how models can be configured in JSON.

```
{
  "mainModel" : "org.vadere.simulator.models.osm.OptimalStepsModel",
  "attributesModel" : {
    "org.vadere.state.attributes.models.AttributesOSM" : {
      ...
      "submodels" : [ "org.vadere.simulator.models.seating.
          SeatingModel" ]
    },
    "org.vadere.state.attributes.models.AttributesSeating" : {
      ...
    },
    ...
  }
}
```

Listing 3.7: This is an example of a model configuration in JSON. It defines the Optimal Steps Model as the main model with the seating model as a submodel.

In the JSON configuration, the fully qualified classnames are used to reference the model. This way, external models from other parties can be included without integrating them in VADERE.

Differences against the previous design

Before we introduce the concept of main model and submodels, the `ModelCreator` class was responsible for setting up a list of `ActiveCallbacks`. It had a large static method in which all dependencies between models were hardcoded. It was hard to maintain and to add new model configurations. We replaced that class in favor of the JSON configuration.

We also removed the enum `ModelType` which listed all available models. This class was redundant because it defined a second name for each model. Models already have their fully qualified class name which is guaranteed to be unique. Removing this class is removing one more source file that have to be maintained. Also, by not hardcoding model names, external models can be used with VADERE.

3.3.4. Other contributions

Project close confirmations I implemented a consistent and solid confirmation system for the situations when projects in the Vadere GUI are closed. A project is closed when users explicitly close the project or the application but also when they open another project or create a new one. In these cases—and if the project was modified—a confirmation dialog comes up asking users if they want to save changes. The users can select “yes” or “no” but they can also cancel the operation. In the latter case, also the secondary operation (e.g. open another project) is canceled. If the selected option was “yes” and the project have not yet been saved, a save-file dialog opens up. In this dialog, users can still cancel the close operation. This logic is now implemented in clean code with good method names and no redundancy.

Introducing distributions from Apache Commons Math The Apache Commons Math library was already a dependency in VADERE. Amongst other things, it provides a class hierarchy for statistical distributions. The distributions are already used as spawn distributions in sources (see section 3.3.1).

I also use some distributions in the seating model. The `EnumeratedDistribution` is used for simulating human seating decisions. The `TruncatedNormalDistribution` is used for selecting a compartment within the train. I implemented the latter distribution by deriving from the `NormalDistribution`. This truncated distribution only returns values in a specified range by drawing from the normal distribution until the result falls into the interval. Another parameter specifies the maximum number of sampling iterations (which is comparable with a timeout). For example, if the specified range is too small and no appropriate value can be drawn in a number of attempts, the class throws an exception. Unit tests cover parameter edge cases and test for plausibility of sampled values.

We located other uses of these types of distributions in existing models. I refactored these models to use the standard distributions instead of separate algorithms. The group size determination for group models was replaced with the `EnumeratedDistribution`. The determination of pedestrian's free-flow velocity was replaced with the `TruncatedNormalDistribution`. This way, duplicate and not unit-tested code was eliminated and well-tested, encapsulated distribution classes from a common library are used.

Refactoring and bugfixes I did refactoring in many classes including `IOUtils`, `ProjectView`, and related GUI classes. I fixed identifier names and source code formatting issues, I extracted methods and constants, eliminated redundant code, removed unused or obsolete code, and fixed compiler warnings. During this work, bugs came to light and could be fixed. Also, inconsistent exception handling around the simulation loop was refactored and fixed.

Infrastructure I introduced and documented guide lines and a coding style guide to streamline contributions to VADERE. The contribution guide lines define basic rules and link to our Java coding style guide. They also provide hints for helpful and consistent commit messages.

The coding style guide is an adapted version of Google's style guide for Java. We only changed the indentation from spaces to tabs, mainly because of an issue with the current Eclipse IDE.[12] There are also XML files with settings for Eclipse's and IntelliJ's code formatter which should be imported in the IDE to keep the code style consistent.

Shortly before VADERE's release I ran a number of scripts on the code base to format the source code according to the style guide, and to generate a report of all source files together with their authors for internal documentation. Also, empty comments (including empty or auto-generated JavaDoc) were deleted.

3.4. Scenario generator

A scenario is a topographic map of the area where the simulation takes place. Besides the model parameters, it is vital input for the simulator. The scenario also defines sources (where agents are created and spawned) and targets (where agents are attracted to). Like all simulation parameters, a scenario is defined in JSON.

For this work, we need train scenarios. A train consists of entrance areas with doors, an aisle connecting the entrance areas, and compartments with seat groups. The scenario has sources and targets for each stop outside of the train and targets inside the train. When it comes to a plan, there are a lot of measures and elements even for only one single compartment. In our scenario description format, an S-Bahn train of type ET423 without any sources and targets for stops has 6566 lines[13] of (pretty-printed) JSON. This is clearly too much to write by hand, especially because I will test different train scenarios.

3.4.1. Design and implementation

It is not practical to define train scenarios by hand. Therefore, I developed Traingen, a train scenario generator tool. It is written in Java to share common code with the simulation framework. Traingen can be used in two ways:

[12] When using spaces for indentation, the Backspace key in the current Eclipse editor only removes one space instead of one indent level.

[13]
```
traingen --number-entrance-areas=12 --block-ends
--train-geometry=org.vadere.state.scenario.Et423Geometry | wc -l
```
(shell command line)

1. As a command line program, it can easily be used in scripting languages to generate scenarios. For example, I generate scenarios according to real passenger counts along the S-Bahn line S3 from Holzkirchen to Mammendorf. For this purpose, the Traingen command line tool is used together with other scripts that generate the definitions of the train stops. The usage help message of the command line version is printed in listing 3.8.

2. As a Java library, it can be used by other Java applications. For example, scenarios could be generated on-the-fly directly before starting the simulation.

```
traingen - generate train scenarios

usage:
  traingen [-n N] [--stop=STOP_PARAMS ...] [options]

options:
  -n, --number-entrance-areas=N
       number of entrance areas [default: 2].
  --stop=STOP_PARAMS
       definition for a stop; format: <time>,<side>,<number_people>. <time> is
       the simulation time point of the stop; <side> must be either "top" or
       "bottom" (top view of scenario map); <number_people> is the total number
       of people entering the train.
  --door-source-distance=METERS
       distance between door and source [default: 0.0].
  --block-exits
       block exits so that agents are forced to walk to their seats inside the
       train.
  --block-ends
       block the ends of the train.
  -c, --compartment-targets
       add a target for each compartment in the aisle.
  -p, --number-sitting-persons=N
       number of already sitting persons, randomly placed [default: 0].
  --random-seed=X
       generate the train using X as seed for the random number generator; if
       this option is not present, a random seed is used.
  --train-geometry=CLASSNAME
       use specified TrainGeometry class. the specified class must extend
       TrainGeometry and must provide a default constructor.
       by default, the geometry for ET423 is used.
  --annotate
       [not working] annotate JSON with comments for post-processing.
```

```
-o, --output-file=FILE
      specify an output file; if not specified, output goes to stdout.
--clipboard
      copy output to the clipboard.
-h, --help
      print this help message.
```

Listing 3.8: Usage help message of the Traingen command line program.

The Java library part provides a Builder interface (Gamma et al., 1995). This is the interface definition:

```
Compiled from "TrainBuilder.java"
public class edu.hm.cs.vadere.seating.traingen.TrainBuilder {
  public edu.hm.cs.vadere.seating.traingen.TrainBuilder(org.vadere.
      state.scenario.TrainGeometry, java.util.Random);
  public void createTrain(int);
  public void setDoorToSourceDistance(double);
  public void blockExits();
  public void blockEnds();
  public void addStop(edu.hm.cs.vadere.seating.traingen.Stop);
  public void addCompartmentTargets();
  public void placePersons(int);
  public org.vadere.state.scenario.Topography getResult();
}
```

Listing 3.9: Builder interface of Traingen.

To generate a train scenario, first, a Builder object has to be created by calling its constructor. Then, the building methods are called, e.g. `createTrain(int)` and `addStop(Stop)`. Finally, the resulting scenario can be retrieved by calling `getResult()`. The result is VADERE's Java representation of the scenario, which can easily be converted to JSON text or directly be used for a simulation.

The Builder interface is a great way to decouple the client program (which uses the interface) from the scenario implementation. Multiple applications can use the Builder interface without knowing any details about the scenario implementation. The scenario implementation can change without affecting the interface and the clients using it.

The plan of the train (to generate e.g. the S-Bahn) is too complex to reasonably pass by command line options or Builder methods. Instead, these details are stored in subclasses of the `TrainGeometry` interface. There is currently one implementation, the `Et423Geometry` for Munich's S-Bahn trains.

The geometry classes do not reside in the Traingen project but in the simulator project because the seating model depends on them.[14] Note that the simulator project cannot use Traingen as a library because Traingen already depends on the simulator project for its scenario and JSON classes.

I could not get a detailed interior plan of ET423. Therefore, to implement the geometry class, I measured one compartment and two entrance areas of an ET423 train by hand. Figure 3.7 shows the measured part of the train with its dimensions.

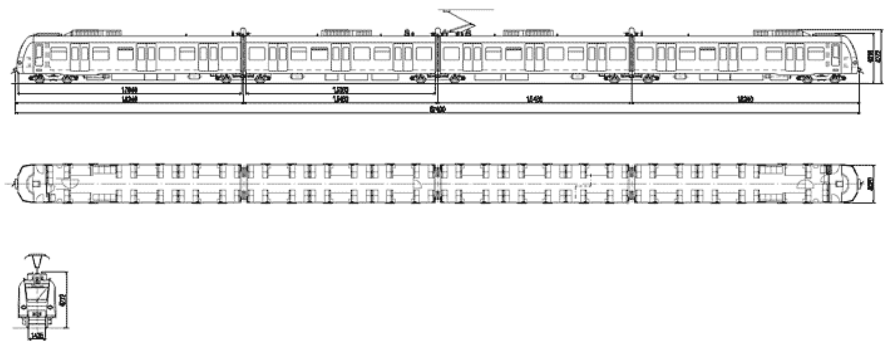


Figure 3.5.: Technical drawing of ET423. Source: `http://de.bombardier.com/`, filename: "et-423-electric multiple unit-techdraw.gif", license: , accessed: 2016-07-22

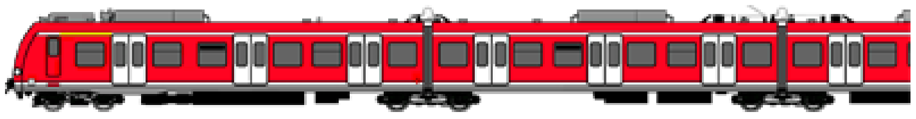


Figure 3.6.: Drawing of ET423. The measured part is between the 5th and 6th door. Source: `https://commons.wikimedia.org/`, filename: "", license: CC BY-SA 3.0, accessed: 2016-07-22

[14]Although the train geometry is encoded in the scenario, it is very hard to extract the needed information. The JSON representation only stores obstacles (walls), sources, and targets. It has no information on how these elements are related.

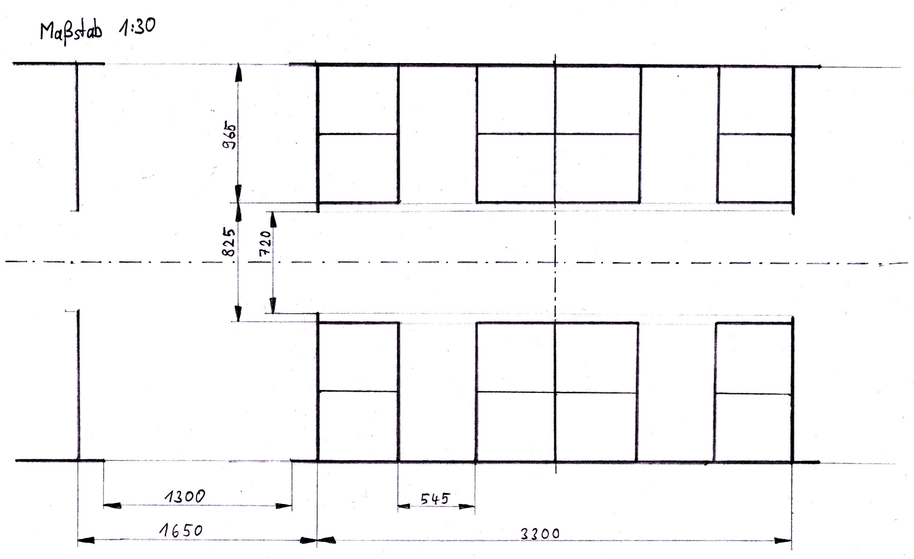


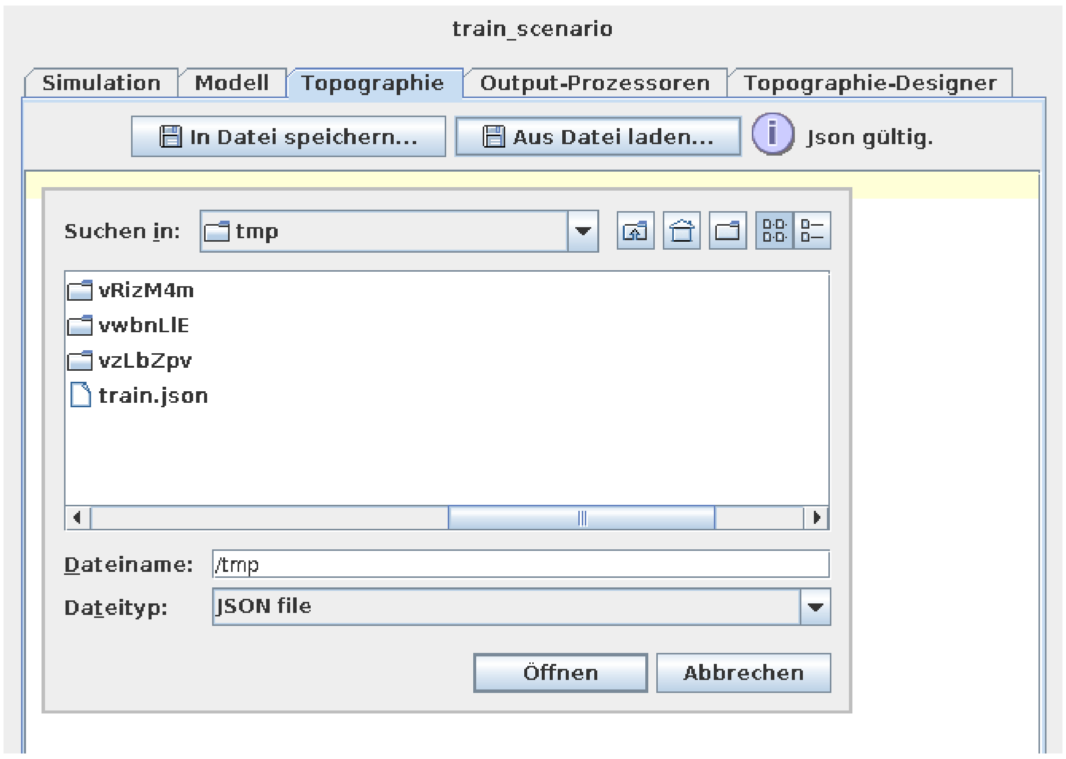
Figure 3.7.: Technical drawing of the interior of ET423. Measures were taken at the 5th and 6th entrance area and at the compartment inbetween.

3.4.2. Usage

Traingen can be started from the IDE or from the command line. When started from the IDE, the command line arguments have to be set there. In Eclipse, this is done via *Run* → *Run Configurations* in the *Arguments* tab.

When started from the command line, the wrapper script `traingen.sh` can be used. It lets Java run the Traingen JAR file and passes all given command line arguments to it.

The scenario file can then be imported into the Vadere GUI and viewed or edited in the *Topography Designer* tab (see figure 3.8 and 3.9).

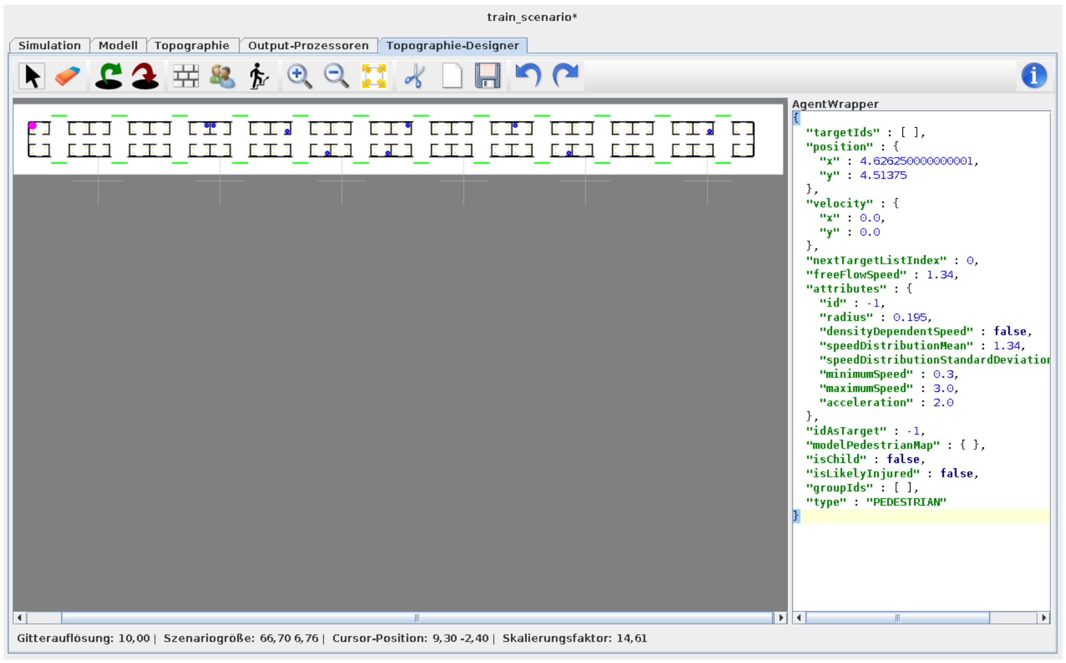


Suchen in: tmp

vRizM4m
vwbnLlE
vzLbZpv
train.json

Dateiname: /tmp
Dateityp: JSON file

Figure 3.8.: Import of a scenario file into the Vadere GUI. It is done in the *Topography* tab by clicking at the *Load from file* button and then selecting a scenario file in JSON format.



```
{
  "targetIds" : [ ],
  "position" : {
    "x" : 4.626250000000001,
    "y" : 4.51375
  },
  "velocity" : {
    "x" : 0.0,
    "y" : 0.0
  },
  "nextTargetListIndex" : 0,
  "freeFlowSpeed" : 1.34,
  "attributes" : {
    "id" : -1,
    "radius" : 0.195,
    "densityDependentSpeed" : false,
    "speedDistributionMean" : 1.34,
    "speedDistributionStandardDeviatio
    "minimumSpeed" : 0.3,
    "maximumSpeed" : 3.0,
    "acceleration" : 2.0
  },
  "idAsTarget" : -1,
  "modelPedestrianMap" : { },
  "isChild" : false,
  "isLikelyInjured" : false,
  "groupIds" : [ ],
  "type" : "PEDESTRIAN"
}
```

Gitterauflösung: 10,00 | Szenariogröße: 66,70 6,76 | Cursor-Position: 9,30 -2,40 | Skalierungsfaktor: 14,61

Figure 3.9.: A train scenario opened in the *Topography Designer* tab of the Vadere GUI.

3.5. Model implementation

This section describes the implementation of the seating model in VADERE. The seating model itself is described in chapter 4.

3.5.1. Overview

Simulation models in VADERE have to implement the `Model` interface which defines a common interface for initialization. Models also have to implement the `ActiveCallback` interface. This interface declares an `update` method (amongst others) which is repeatedly called from VADERE's main simulation loop. More details on this are provided in section 3.3.3.

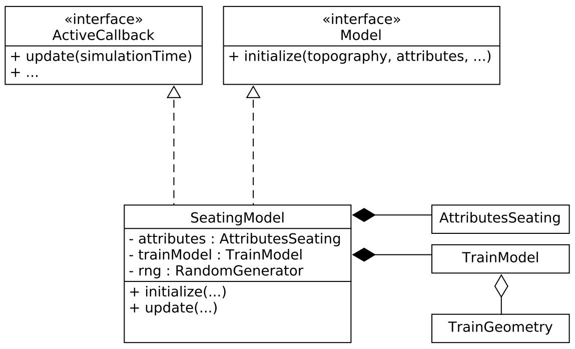


Figure 3.10.: Simplified UML class diagram of the seating model implementation.

Figure 3.10 shows a high-level overview of the seating model implementation. The class `SeatingModel` implements the interfaces `Model` and `ActiveCallback` and has the following instance variables:

- An attributes object which contains the model parameters. These parameters can be configured via JSON in the scenario file or from the Vadere GUI.
- A `TrainModel` object which is a model of the train. It is built from the topography and can be viewed as a higher-level wrapper class. It provides access to compartments, seat groups, seats, and the persons sitting on the seats.
- A seeded random number generator used for statistical modeling of seating decisions.

The attributes, the topography, and the random number generator are passed from VADERE to the `initialize` method.

3.5.2. Implementation

The model's algorithm is explained in chapter 4. The following is a short outline of what is done for a single person:

1. For a new person entering the train, choose a compartment and assign it as target to the person.
2. For a person arriving in its target compartment, choose a seat group and a seat and assign it as target to the person.
3. For a person arriving at its seat, update the seat object in the train model accordingly.

The `update` method (listing 3.10) is only concerned about step 1. For those persons that have no target assigned, it assigns a compartment as target. This is a polling approach which slows down the simulation. A cleaner and more efficient way would be to introduce source listeners in VADERE, similar to the target listeners introduced in section 3.3.2. For now, the existing facilities are used.

```
public void update(double simTimeInSec) {
    // choose compartment for those peds without a target
    trainModel.getPedestrians().stream()
            .filter(this::hasNoTargetAssigned)
            .forEach(this::assignCompartmentTarget);

    // the next steps are done by target listeners registered in
        initialize()
}
```

Listing 3.10: The seating model's `update` method which is hooked into the simulation loop.

The next two steps of the algorithm are implemented using target listeners. Each compartment and each seat is a target and gets a listener registered in the `initialize` method.

For step 2, a target listener is registered on every compartment target. Once a person arrives in its compartment, the listener implementation chooses and assigns a seat as the new target.

For step 3, a target listener is registered on every seat target. Once a person arrives at its seat, the train model's seat object is updated to reference the now sitting person.

Special cases are not described in this section. Instead, they are identified and described in section 4.3.5 and implemented accordingly.

3.5.3. Tests

The seating model is tested with JUnit 4 unit tests. There are currently 72 test methods which cover 52% of the `SeatingModel` class and 98% of the related train model classes on which the seating model builds upon (branch coverage, (Myers et al., 2004)).[15] There is a JUnit test suite to combine all tests related to the seating model in a hierarchical way which allows for running all related tests together.

3.6. Model verification

The basic idea for a higher-level verification of the seating model is letting the simulation generate the same kind of data as that from the data collection. The real and the simulated data can then be compared which is described in chapter 5. This section describes how the data is generated in the simulation.

In VADERE, the data processor subsystem is used to generate data during a simulation. Data processors aggregate, process, and store data about the simulation. They are hooked into the simulation loop—similar to `ActiveCallbacks` (section 3.3.3). When the simulation is finished, the aggregated data is written to output files in a table format, e.g. CSV. A more detailed documentation of this subsystem can be found in the VADERE repository in the *Documentation* folder.

The data processor for verification is called `LogEventProcessor` because it produces the same data format as the log-event data from the survey (see section 2.5.1

[15]The Eclipse plugin EclEmma 2.3.3 was used to measure code coverage.

and table 2.5). However, only sit-down events are recorded in the simulation; other events are not needed for verification.

Involved classes are `LogEventProcessor`, `AttributesLogEventProcessor`, `LogEventOutputFile`, `IdDataKey`, and `LogEventEntry`. The latter two classes are used in key-value pairs to store the data in the processor. `LogEventEntry` represents one logged event in the output file.

The logic is implemented in the `LogEventProcessor` class:

- During initialization, the class reads its attributes (the compartment to observe, the survey ID to use, and the initial value for the person ID counter). It also gets the `TrainModel` instance from the seating model to obtain access to compartments and seats. It then registers a target listener[16] on all seats in the assigned compartment.
- Before the simulation starts, the processor sets an instance variable to the current time. During the simulation, this time variable is updated and used as timestamp for the logged events. For persons already sitting in the train, the processor stores initial sit-down events and then the "initialization end" event. These events are not considered in the data analysis but capture the initial situation in the train.
- During the simulation, the time variable is periodically updated in the `doUpdate` method. Asynchronously, the target listener is called when a person sits down on a seat. This sit-down event is then stored in the processor's key-value store.

After the simulation, the `LogEventOutputFile` writes the aggregated data to disk. Because the processor is parameterized with its `AttributesLogEventProcessor`, multiple compartments can be observed during one single simulation. The output files from multiple simulations can also be combined to one single dataset. The resulting dataset is used in chapter 5 to verify the seating model.

[16]See section 3.3.2 for details.

3.7. Summary

I presented some aspects of design, implementation, and development of the mobile app for data collection and discussed a different architectural approach.

I introduced VADERE, an existing simulation framework for pedestrian dynamics, and I presented enhancements and new features. New modules have been carefully designed around the simulator without invading or bloating existing core modules. Shortcomings in the software design have been located and fixed. I also introduced coding standards and contribution guidelines for the VADERE open source project which help to ensure code quality.

I implemented a new simulation model in VADERE to simulate people's seating behavior in S-Bahn trains, and I integrated a system to verify the simulation by comparing its output to real data from a data collection. I developed a tool to auto-generate train scenarios with a variety of options. These scenarios are not restricted to seating model applications but can be used in other areas of research on pedestrian dynamics as well.

4. Model

In this chapter I use research results and the collected data (see chapter 2) to build a model for seating behavior.

4.1. The seating model

The seating model assigns a seat within a train to new passengers who wish to sit. It takes over when a person without a target enters the train scenario. It relinquishes control when a person has reached the assigned seat target.

The inflow process—when passengers enter the train—is predefined by the train scenario. The scenario is generated by Traingen (section 3.4) which creates sources with realistic spawn distributions investigated in section 2.7.

For pedestrian movement and navigation, the Optimal Steps Model (OSM) (Seitz, 2016) with target floor fields is used. Alternatives would be the Behavior Heuristic Model (BHM) (Seitz, 2016) or the OSM without floor fields. These models work without floor fields so that the CPU-intensive computation of a target floor fields is dropped. This would make the simulation start faster. Models without floor fields, however, require that pedestrians always have visual contact with their current target. This would increase the scenario's and the model's complexity because at least 4 new interim targets would have to be added to each compartment.

4.1.1. Approach

For the seating model, I use a combination of cognitive heuristics (Seitz et al., 2016) and statistical decisions.

A model purely based on cognitive heuristics would require every pedestrian to have a set of predefined preferences, e.g. a preference for window seats. With the seating model, a pedestrian only sits down one time. Therefore, there is no need for consistent preferences and no need to predefine these preferences. Furthermore, it would add extra code and complexity to the simulation software.

The combination of heuristical and statistical decision making is in this case equivalent to a pure cognitive heuristic approach but simplifies the model implementation.

4.1.2. Model assumptions

Choice of compartment

The choice of the compartment is not investigated in this study, and no conclusions can be drawn from the collected data.

Except for the first and last two compartments, all compartments in a S-Bahn train are equal in their seating layouts. Yet there are some differences, e.g. the train noise level or the bellows at the train joints. Passengers might have preferences regarding these properties.

Previous studies investigated the distribution of passengers at the platform (Panzera, 2014), the popularity of different seating and compartment layouts (Wardman and Murphy, 2015), and the popularity of open and closed compartments in ICE trains (Cis, 2009). While the latter two are less relevant for this model, the results of the first study seems to be important. Furthermore, according to experts at the MVV[1], passenger's often choose their compartment (and hence the position within the train) based on local circumstances at their destination train station. This theory is also supported by Panzera (2014, 50ff.). My own qualitative observations yield three more statements:

- Passengers often choose one of the compartments next to their entrance area.
- Passengers tend to proceed to the next compartment instead of turning around and going back.
- Passengers often choose a compartment in a different train section and walk directly to this section.

From these factors I derive a model to choose a compartment based on a normal

[1]Meeting with representatives from the MVV. MVV Geschäftsstelle. München, 2016-05-11.

distribution along the train's length. This part of the seating model is described in more detail in section 4.3.

Choice of seat group and seat

In reality, passengers cannot see all seats of a compartment at a glance. Especially small persons or baggage behind the seat backrest may not be visible. Also, passengers would not only examine one compartment but also parts of the next compartments as they walk through the train.

For this model, I assume that a person decides for a seat group and a seat as soon as he or she reaches the compartment. This involves the assumption that the person can see all seats in the compartment immediately when he or she arrives there—but not earlier.

A compartment as a target is defined as the aisle between two adjacent entrance areas. Therefore, a compartment counts as reached as soon as passengers enter the aisle—no matter from which direction they come.

4.1.3. Limitations

This section lists known limitations of the seating model and issues that are out of scope.

- No baggage. There is no support for any form of baggage.
- No groups. There is no support for any form of group behavior.
- No standing. It is assumed that all passengers want to sit.
- Boundary effects. There can be boundary effects at the first and last half-compartment.
- No changing place. In particular, persons do not make place for a new person. This is not investigated in this study and not covered by the seating model.
- No leaving. The seating model is not responsible to make passengers leave the train.

- Compartment choice is not investigated. As described in section 4.1.2, the choice of the compartment is not based on statistical evidence.

In addition, the interference factors listed in section 2.1.2 are not covered by the seating model.

4.2. Data analysis

In the following, I test for statistical significance using the exact binomial test `binom.test` from R's `stats` package (Clopper and Pearson, 1934). The significance level is set to $\alpha = 0.05$.

For the seating model, I use only data about persons traveling alone. Figure B.5 shows that these persons are the majority (188 out of 232).

There is data about 97 women and 91 men (figure B.3). The difference is not not statistically significant (exact binomial test, p-value = 0.715, n = 188). Therefore, the gender is not included as a factor in the seating model. Data from all persons is used for analysis.

The majority of people is adult i.e. older than 18 years (159 out of 188). A bar chart is printed in figure B.4. Therefore, the age group is not included as a factor in the seating model. Data from persons of all age groups is used for analysis.

4.2.1. Choice of the seat group

This section analyzes the choice of the seat group within a compartment.

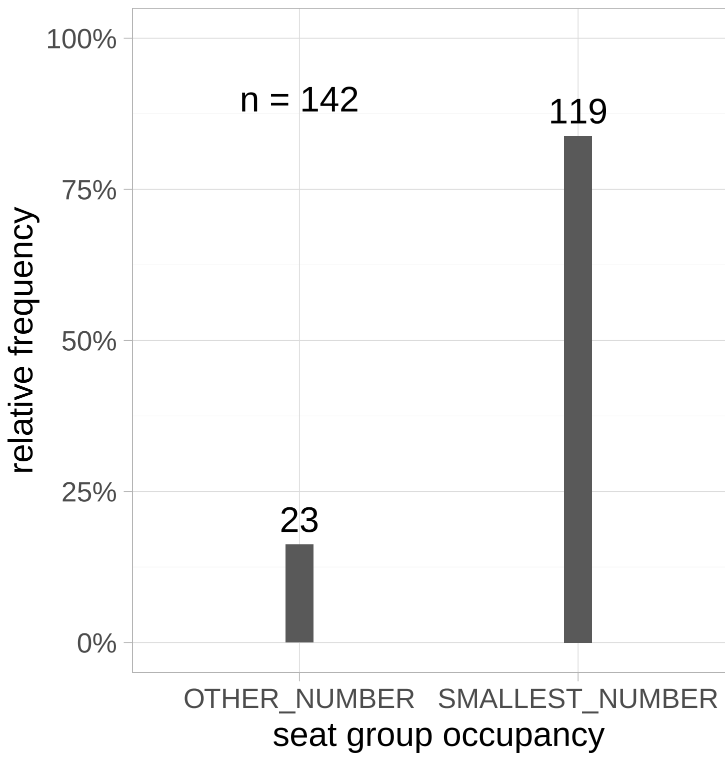


Figure 4.1.: People's preference for a seat group within a compartment: choosing one with the smallest number of other persons or any other.

The differences in figure 4.1 are statistically significant (exact binomial test, p-value < 0.001, $n = 142$).

4.2.2. Choice of a seat

This section analyzes the choice of the seat within a seat group under different conditions.

Empty seat group

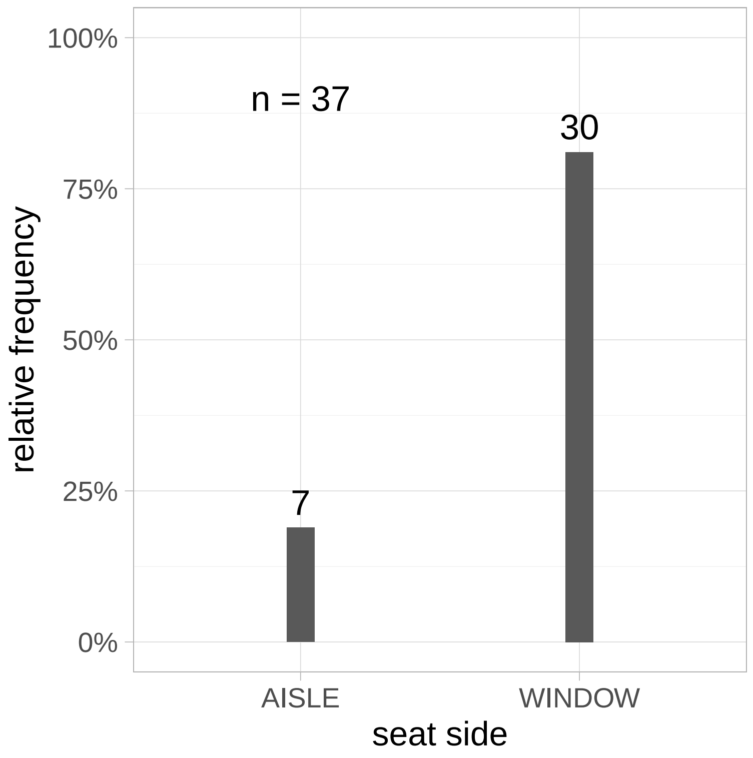


Figure 4.2.: People's preference for window vs. aisle seats in an empty seat group.

The differences in figure 4.2 are statistically significant (exact binomial test, p-value < 0.001, $n = 37$).

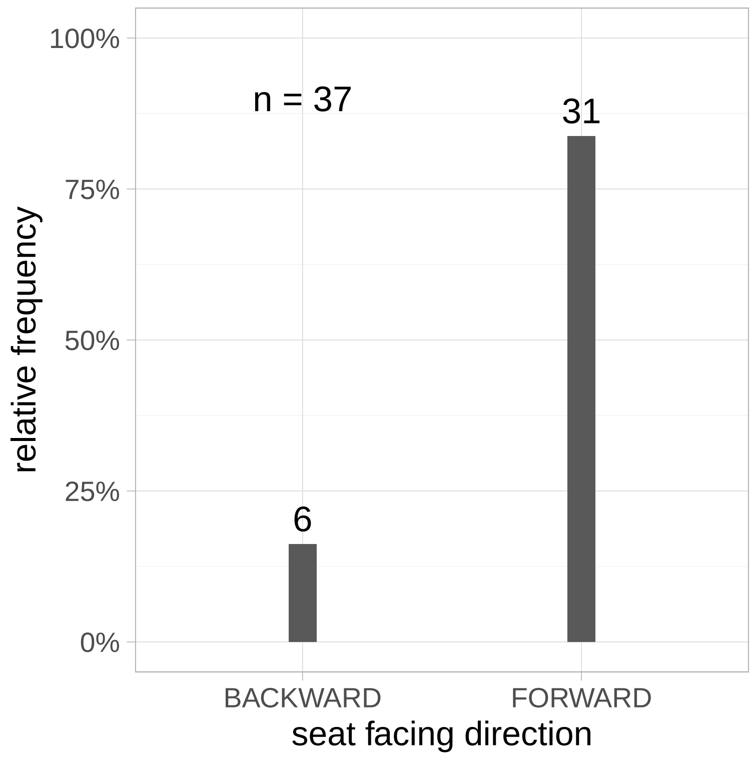


Figure 4.3.: People's preference for forward vs. backward-facing seats in an empty seat group.

The differences in figure 4.3 are statistically significant (exact binomial test, p-value < 0.001, $n = 37$).

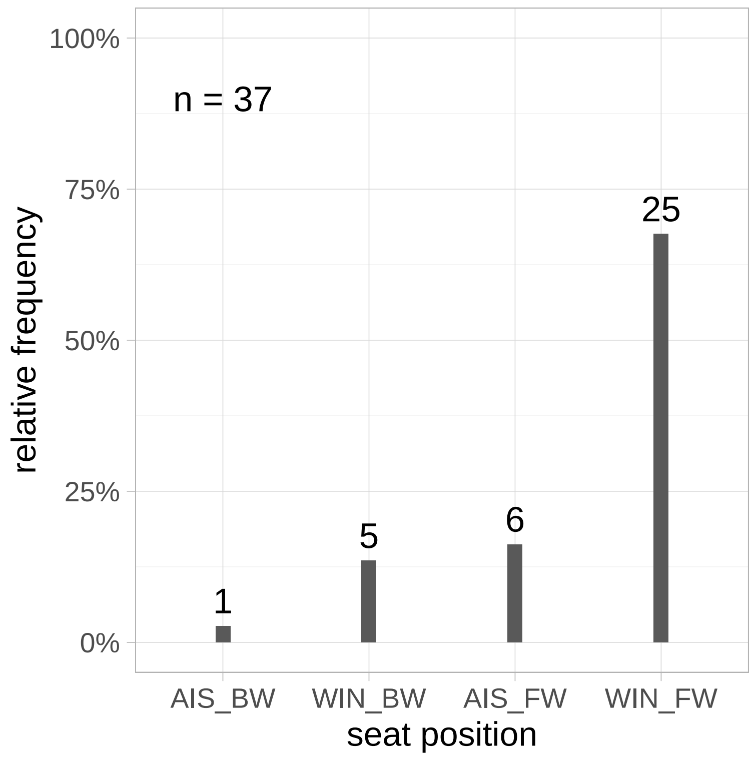


Figure 4.4.: People's preference for a specific seat in an empty seat group. The abbreviations are combinations of the seat's side and facing direction. AIS and WIN stand for the aisle and window side, respectively. BF and FW stand for backward and forward-facing direction, respectively.

In figure 4.4, the differences between the forward-facing window seat and each of the other seats are statistically significant (exact binomial test, p-value ≤ 0.001, $n \leq 31$). The differences between the lower three bars are not statistically significant (exact binomial test, p-value ≥ 0.125, $n \leq 7$).

Seat group with one other person

In the case that one other person sits in the seat group, there are $4 \cdot 3 = 12$ possible configurations. When looking at all configurations together, sitting diogonally across from the other person is the most popular seating choice. Dividing the 12 configurations into groups may yield more detailed insights but there is to few data.

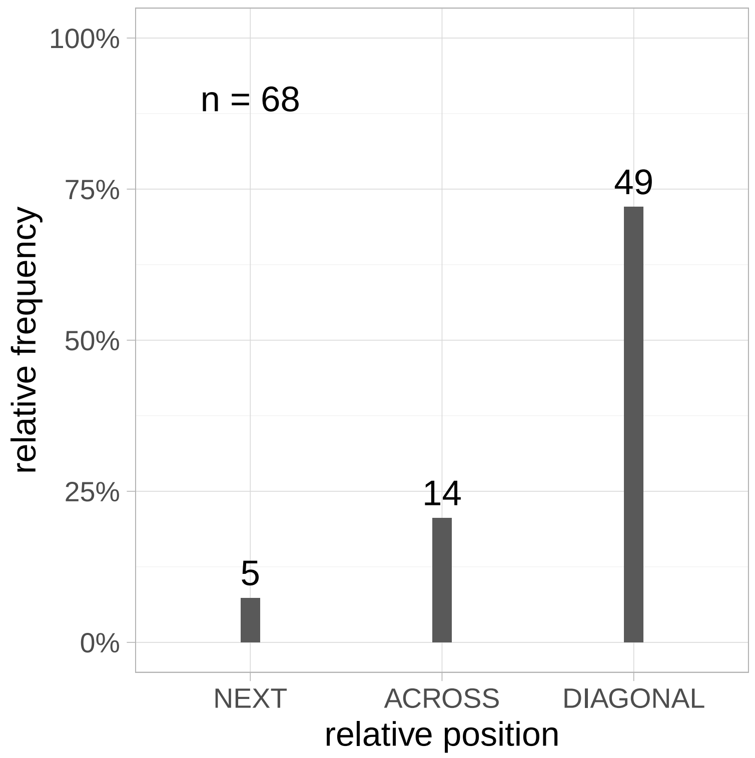


Figure 4.5.: People's preference for seat positions in a seat group with one other person. Possible positions are next to, across from, and diagonally across from the other person.

In figure 4.5, the differences between the diagonal seat position and each of the other seat positions are statistically significant (exact binomial test, p-value < 0.001, $n \leq 63$). The difference between the lower two bars is not statistically significant although there is a tendency (exact binomial test, p-value $= 0.064$, $n = 19$).

Seat group with two other persons

In this case there are $\binom{4}{2} = \frac{4!}{2! \cdot 2!} = 6$ possible configurations. When the third person comes, he or she always have the choice between 2 available seats. So there are $6 \cdot 2 = 12$ possible scenarios. These possible scenarios can be splitted into 3 groups:

1. Both available seats have the same facing direction.
2. Both available seats are at the same side (window/aisle).

3. The available seats are diagonally arranged. That is they have different facing direction and they are at different sides.

For cases 1 and 2, passengers are indifferent in their choice as evident in figures B.9 and B.10, respectively. For case 3 there is very few data. In the model it is therefore decided randomly following an uniform distribution in these three caseses.

4.3. Algorithm

An outline of the seating model's algorithm is depicted in figure 4.6. The algorithm handles all passengers individually.

The algorithm's first decision is whether the passenger wants to sit at all. Only in the "yes" case the algorithm continues. Otherwise a different model would have to handle the situation, e.g. by choosing a position to wait in the entrance area. For this work, all passengers are handled as if they want to sit down.

The rest of the algorithm consists of three steps: Choosing a compartment, choosing a seat group, and choosing a seat therein. The probabilities used for the decisions depend on the model parameters described in the next section (4.4).

In the rest of this chapter, the word "randomly" means, following an uniform distribution.

4.3.1. Choosing the compartment

For a new person entering the train a compartment is chosen and assigned as target to the person. Since the choice of the compartment is not investigated, the compartment is selected with a truncated normal distribution.

The probability density function's x axis goes along the row of compartments. The scale is defined by the (zero-based) compartment indexes, not by the physical train length. The function is truncated at 0 and $n_c - 1$ where n_c is the number of train compartments. The distribution's mean is exactly between the two compartment indexes where the person enters. For example, if the person enters at the entrance area between compartment 0 and 1, the mean would be 0.5. The standard deviaton $\sigma = n_{ea}/2$ where $n_{ea} = n_c - 1$ is the number of entrance areas in the

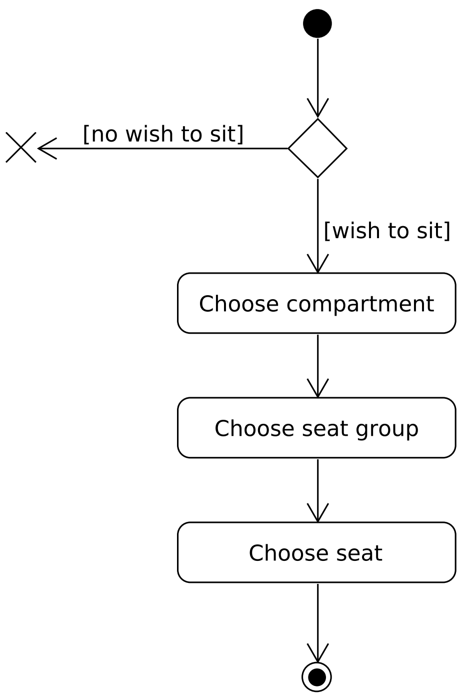


Figure 4.6.: Overview of the seating model's algorithm. Special cases are not covered in this diagram.

train. That is, the standard deviation σ corresponds to a half of the train length. The value drawn from the distribution is rounded to the next integer and indexes the chosen compartment. The first and the last half-compartment are underrepresented in the distribution because they also serve as a gathering place when the train is full.

4.3.2. Choosing the seat group

For a person arriving in its target compartment, a seat group is chosen. The seat itself is chosen in the next step.

The seat groups that have seats available are devided into two sets: those with the lowest number of persons and those with a higher number of persons. A binomial distribution decides between the two sets.

- If it is decided for the set of seat groups with the lowest number of persons, one of these seat groups is randomly chosen. If it is only one seat group, this one is chosen.

- If it is decided for the set of seat groups with a higher number of persons, the same algorithm is applied again on this set.

If all seat groups have the same number of persons, a seat group is randomly chosen. If it is only one seat group, this one is chosen.

4.3.3. Choosing the seat

After a seat group is chosen for a person, a seat within this seat group is chosen next.

Empty seat group For an empty seat group, the seat is selected using a discrete distribution. This is illustrated in figure 4.4; e.g. the forward-facing window seat is most popular.

Seat group with one other person For a seat group with one other person, a seat position relative to that other person's position is selected using a discrete distribution. The possible relative positions are "next to", "across from", and "diagonally across from". This is illustrated in figure 4.5; e.g. the seat diagonally across from the other person is most popular.

Seat group with two other persons For a seat group with two other persons, one of the two available seats is randomly chosen.

Seat group with three other persons For seat groups where only one seat is available, this seat is selected.

4.3.4. Reaching the seat

When a person reaches its seat, i.e. entering the area defined by the seat target, it is immediately placed at the seats' center.

4.3.5. Special cases

The algorithm introduced above does not cover all possible situations. The following special cases are handled differently.

Compartment already full Persons get a compartment assigned when they enter the train. Especially if the assigned compartment is not next to the door where a person entered, it happens that the compartment is already full when the person arrives there. In this case the person cannot choose a seat group and a seat and will proceed to the compartment directly next to the current compartment. The Person will not change its direction and turn because according to my observations, persons usually keep their direction.

Seat already occupied When persons arrive their assigned compartment, they get a seat group and a seat assigned. They proceed to their assigned seat. If another person sat down while the first person still approaches the same seat, this is detected when the person arrives the occupied seat. In this case, the person randomly chooses another available seat in the same seat group. If there is no available seat, the person's next target is updated to be the current compartment again. When arriving at the compartment target, the usual algorithm for choosing a seat group and a seat is triggered automatically. This simple rule makes the implementation of this special case easy.

4.4. Model parameters

The seating model has the train geometry and discrete probability distributions as parameters. In addition, it depends on parameters of the context it is used in. For example, the seating model is implemented to work with the OSM (the OSM is the so-called main model). The OSM has its own parameters, e.g. the floor field and the pedestrian parameters. The OSM's parameters are left to their defaults (described in (Seitz and Köster, 2012; Seitz, 2016)), except for the parameters defined in the next subsections.

All parameters of the model implementation in VADERE can be configured in

JSON, e.g. using the Vadere GUI.

4.4.1. Seating parameters

These parameters are preconfigured with values from real data found in section 4.2. The default values are defined in the Java class `AttributesSeating`. The parameters for the choice of seating are defined in the following list. All these parameters are lists of pairs. A pair consists of

- a value which is a possible outcome of a random choice,
- and a probability fraction which is the count of the associated outcome drawn from the data.

These are the parameters for choice of seating:

- `seatGroupChoice`—This is the choice for a seat group in a compartment. There are only two possible outcomes: `true` for the seat group with the lowest number of other persons (or one of the seat groups, if there are multiple such seat groups), and `false` otherwise. If the outcome is `true`, and there are multiple seat groups with the same lowest number of persons, one of this seat groups is randomly drawn. If the outcome is `false` the procedure is started again with all remaining seat groups.

- `seatChoice0`—This is the choice for a seat within a seat group when the seat group is empty (0 other persons). Possible outcomes are the seat indexes from 0 to 3.

- `seatChoice1`—This is the choice for a seat within a seat group when there is 1 other person sitting there. Possible outcomes are the relative positions to the sitting person ("next to", "accross from", "diagonal accross from").

The train geometry is another parameter which defaults to the S-Bahn train ET423.

4.4.2. Seat target parameters

Each seat is a target in the simulation. Important parameters for targets are position, size, and the so-called `deletionDistance`[2] around the target. These three parameters define the area at which the target counts as reached when a pedestrian enters the area. The targets are part of the topography which is created by Traingen. Therefore, target parameters must be changed in Traingen's source code in the method which creates the targets.

The seats are parameterized to have the smallest possible size. The `deletion-Distance` is set to 0.5m to make it easier for persons to reach their targets. This reduces the possibility of congestions within a seat group. However, it is important that the area of the seat target is fully inside of the seat group and does not protude any walls.

4.4.3. Floor field parameters

The floor field used by the Optimal Steps Model is a potential field. For each target there is one potential field allowing the pedestrians to navigate toward their targets. Other pedestrians and obstacles modify these potential fields with their own three-dimensional bell curve.

The curves produced by pedestrians and obstacles respectively have a width and a height parameter. The curve width of pedestrians is changed from 0.5m to 0.2m. The curve width of obstacles is changed from 0.25m to 0.05m. The new values are more appropriate in a train context because people have to pass each other in the narrow aisle, they lean against the wall, and they sit on seats surrounded by walls and other people. With the original default values, the simulation would not work because people would not be able to pass each other or to enter certain seat groups.

[2] A better name would be `reachedDistance` because pedestrians do not necessarily get removed from the scenario once they reach their targets. This parameter will probably be renamed in the next release of VADERE.

4.5. Summary

In this chapter, I presented the seating model. I showed how the model's algorithm is based on cognitive heuristics combined with statistical decisions. I defined the model's capabilities, scope, and limitations, and outlined how it can be integrated in a larger simulation system. I analyzed and visualized aspects of the collected data and developed the model based on the results. I presented the seating model, described the algorithm's individual steps and possible special cases. Finally, I listed the most important model parameters and pointed out their effects on the model.

5. Model evaluation

In chapter 4, I presented a theoretical model for seating behavior of train passengers. The model is implemented in the crowd simulation software VADERE. More information on the implementation is provided in chapter 3, more specifically in the sections 3.4, 3.5, and 3.6.

In this chapter, I evaluate the seating model. First, I verify that the model's algorithm works as expected by comparing simulation output against empirical data collected in section 2.4. Second, I visually validate the model by analyzing a series of screenshots of the simulated scenario.

5.1. Model verification

In this section, I verify the seating model's algorithm by comparing simulation output against real data on seating behavior. This is a higher-level test of the model implementation that complements the unit tests described in section 3.5.

I let the simulation generate data in exactly the same format as the collected data. Section 2.4 defines the data format and section 3.6 describes how the data is generated. I then apply the same procedure of data processing that I used for the collected data to the simulated data. The data processing is explained in section 2.5. In section 4.2, I analyzed the seating data and used bar diagrams to guide the algorithm design. In the following, I show these diagrams side by side with diagrams generated from the simulated data. I compare the bar diagrams visually for equivalence to verify that the algorithm works as expected. Another method would be to perform binomial tests on each pair of corresponding relative frequencies.

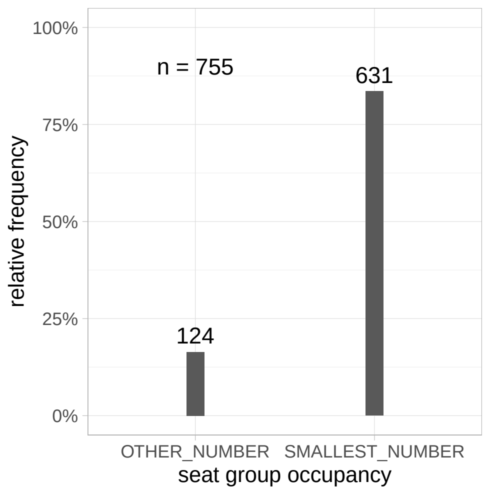
5.1.1. Simulation run

For the model verification, I simulate a realistic scenario using passenger counts from statistics provided by the MVV, one of Munich's organizations for public transport (see section 2.6). The simulation reflects the situation on Friday, 2013-07-26, starting at 08:40 a.m. in Deisenhofen on the S-Bahn line S3. However, only passengers entering the train are simulated. The simulation of passengers leaving is not covered by the seating model.

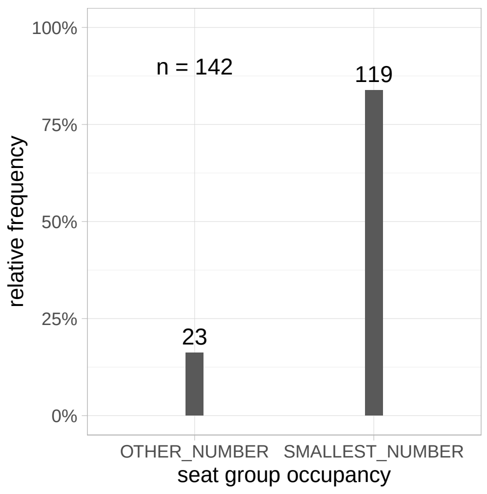
Data is generated by observing 11 compartments of the simulated train wagon simultaneously. The simulation was run 5 times, each time using a random seed for the internal random number generator. Each run stopped after 37 minutes[1] (simulation time) when the train's seat capacity is exceeded. The data of all observed compartments is then combined in one log event dataset. This dataset has the same form as the dataset from the real data collections (see section 2.5.1). The datasets for surveys and persons are artificially created from the log event data.

[1] `grep finishTime scenarios/train_scenario_with_mvv_counts.scenario` (shell command line)

5.1.2. Choice of the seat group



(a) Simulated data.

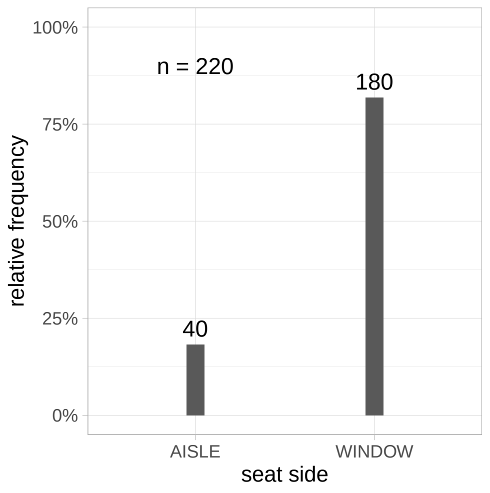


(b) Data from data collection.

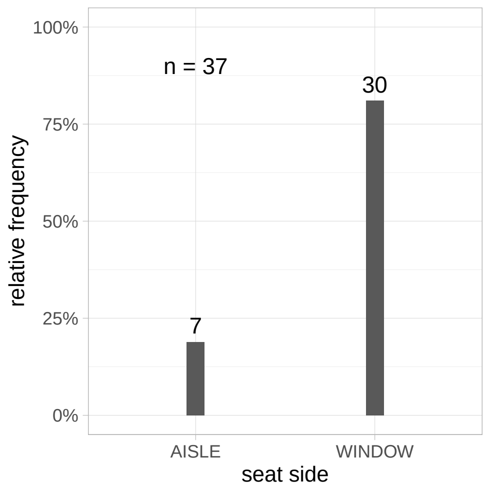
Figure 5.1.: People's preference for a seat group within a compartment: choosing one with the smallest number of other persons or any other.

5.1.3. Choice of a seat

Empty seat group

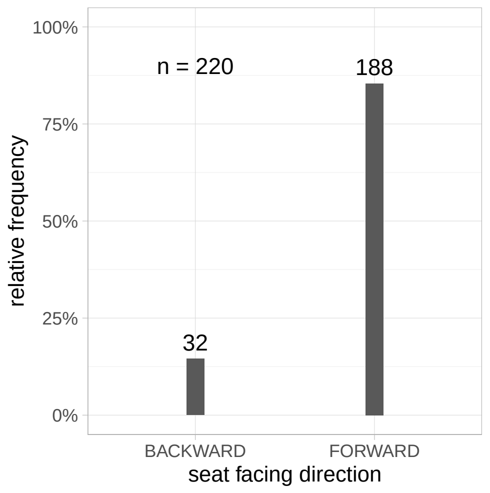


(a) Simulated data.

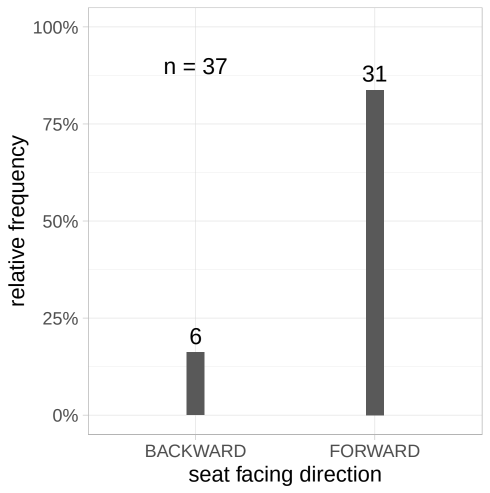


(b) Data from data collection.

Figure 5.2.: People's preference for window vs. aisle seats in an empty seat group.

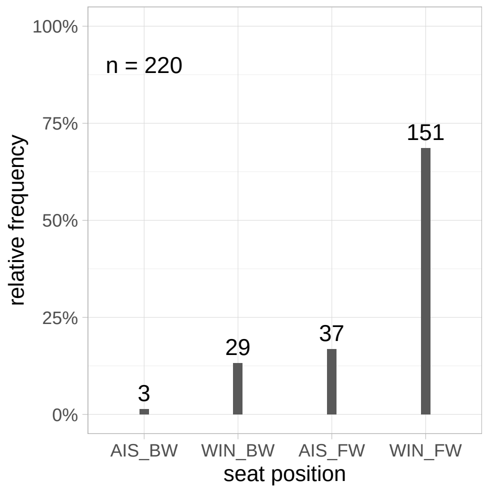


(a) Simulated data.

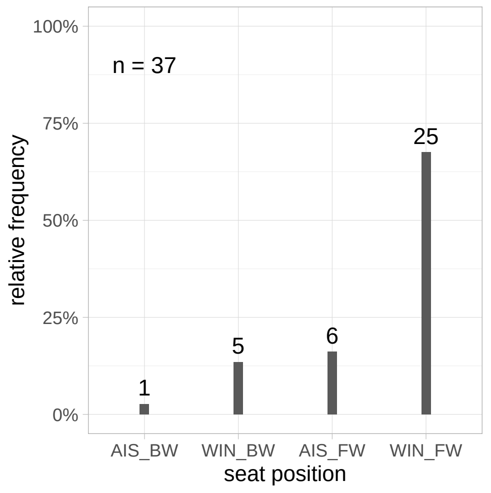


(b) Data from data collection.

Figure 5.3.: People's preference for forward vs. backward-facing seats in an empty seat group.



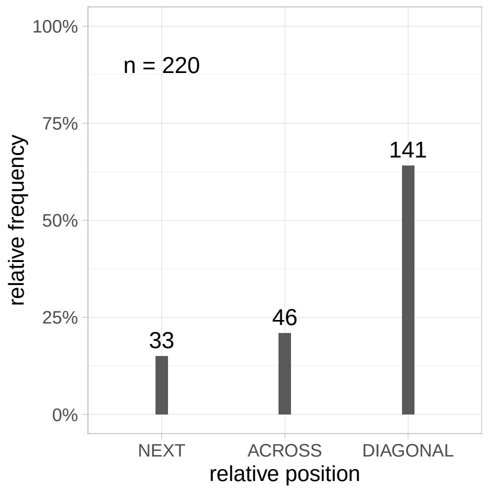
(a) Simulated data.



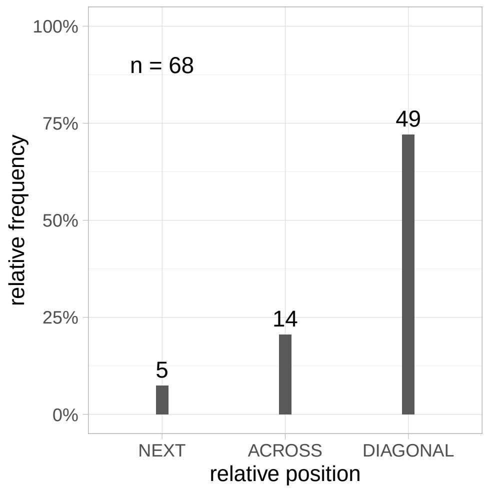
(b) Data from data collection.

Figure 5.4.: People's preference for a specific seat in an empty seat group. The abbreviations are combinations of the seat's side and facing direction. AIS and WIN stand for the aisle and window side, respectively. BF and FW stand for backward and forward-facing direction, respectively.

Seat group with one other person



(a) Simulated data.



(b) Data from data collection.

Figure 5.5.: People's preference for seat positions in a seat group with one other person. Possible positions are next to, across from, and diagonally across from the other person.

5.1.4. Assessment

The preceding diagrams illustrate the most important rules in the model's algorithm. The clear matches within the pairs of plots show that the simulation results reflect the model design. It therefore can be concluded that the model's implementation works as expected regarding the tested algorithm elements.

However, the model verification does not prove that the model actually reflects passengers' seating behavior correctly. In the next section, I undertake a visual validation.

5.2. Visual validation

In this section a visual validation is conducted. This validation is based on the same simulation described in section 5.1. Screenshots of the simulated train scenario are printed in figure 5.6. Each screenshot is taken after a train stop as soon as all new passengers have sat down. If the situations depicted in the pictures look realistic and if the simulated passengers seem to choose their seats in a natural way, this is an indication for a valid model.

Screenshots (1) and (2) show passenger's trajectories.[2] These trajectories show that many passengers choose a compartment next to the door they entered. Other passengers walk through the train passing multiple compartments. These results are expected as argued in section 4.1.2.

Within a compartment, the simulated passengers mostly choose the seat group with the smallest number of other passengers. This can be observed by comparing the same compartment in adjacent screenshots or also by viewing all seat groups: the numbers of passengers sitting in seat groups of a compartment are mostly similar. This is behavior we expect in the German culture.

Within an empty seat group, the simulated passengers mostly choose forward-facing or window-side seats, especially the forward-facing window seat. This is expected according to experts at the MVV[3] and also supported by this and other

[2]There are no trajectories in later screenshots because there are too much passengers. Trajectories would be ambiguous and would clutter the pictures.

[3]Meeting with representatives from the MVV. MVV Geschäftsstelle. München, 2016-05-11.

studies (e.g. Rüger and Loibl (2010)).

Within a seat group with one passenger, the seat diagonal across from that passenger is most frequently chosen. This is behavior we expect in the German culture because strangers tend to keep distance to each other.

For seat groups with two passengers, the choices a new passenger has are too complex to examine with this validation schema. About these seat groups and those with only one available seat, it can be told that they also fill up during the simulation, which is expected.

There are some problems evident in the screenshots of the simulation: Some pedestrians stand in seat groups or in the aisle and are apparently unable to sit down. This is most probably a problem with the underlying pedestrian navigation model. I examined debug logs and found that these pedestrians have a valid seat target but never reach this target even if the seat is available. Possible reasons might be problems with the floor field parameters (section 4.4.3) or the discrete stepping algorithms of the Optimal Steps Model (OSM).

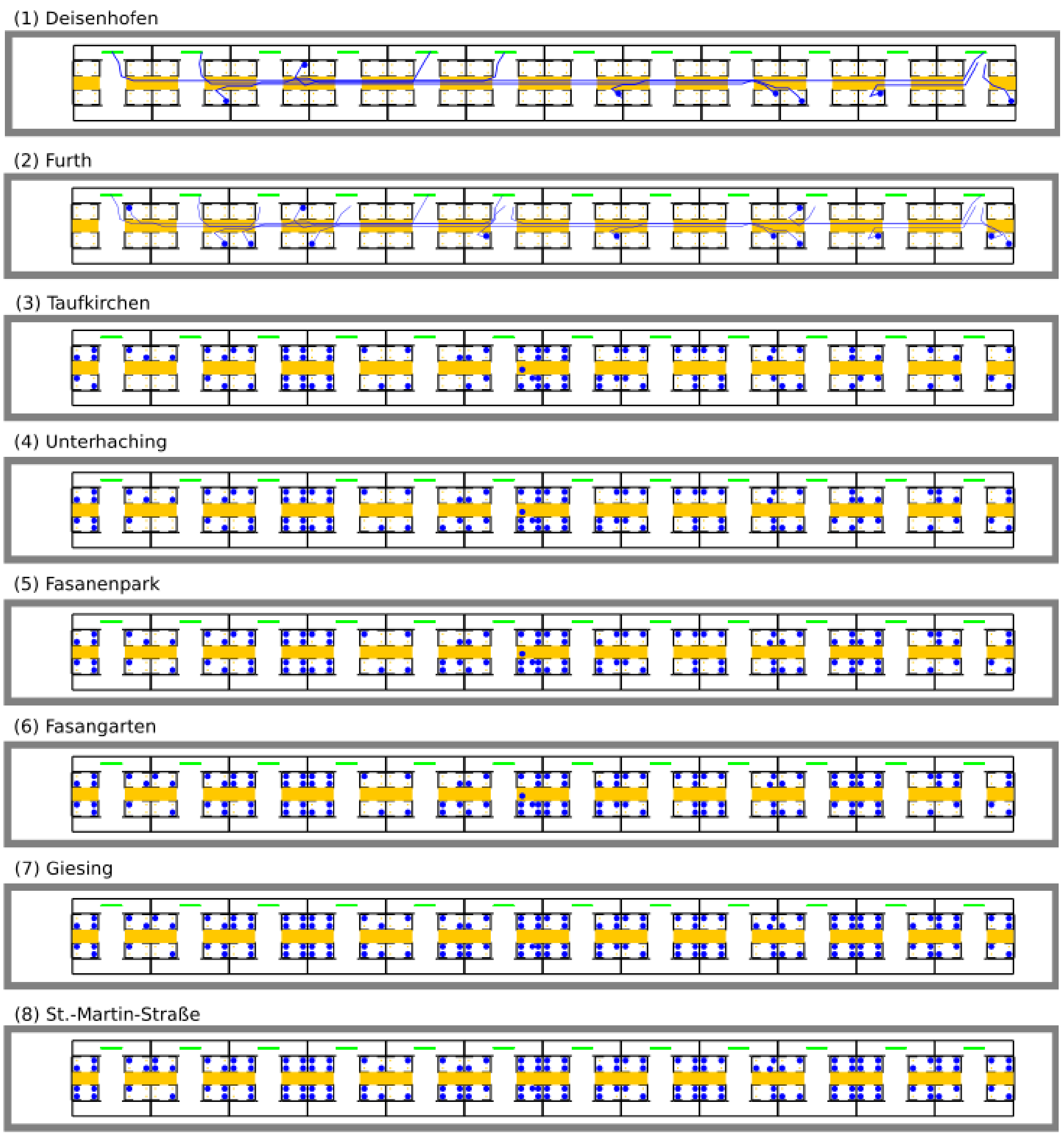


Figure 5.6.: This figure shows screenshots of a simulated train scenario. The train's moving direction is from right to left. For the simulation, passenger counts from the S-Bahn line S3 to Mammendorf are used, starting at Deisenhofen. Above each screenshot, the train station is denoted after which the picture was taken. Picture (1) and (2) additionally show passengers' trajectories to visualize their ways. The green bars above the entrance areas are the sources where passengers come from. The extra, black border lines around the train are there to prevent simulated passengers from walking outside of the train towards their seats. Unfortunately, there are a few cases where passengers cannot reach their target. This is probably a flaw in the navigation model.

5.3. Summary

In this chapter, I verified and validated the seating model based on a simulation with real passenger counts.

For the model verification, train compartments in the simulation were observed to collect the same kind of data as collected in the field study. I then compared plots from both datasets pairwise, and found clear matches between them. This indicates that the seating model has a solid implementation.

For the model validation, I examined screenshots of the simulated scenario. By comparing pairs of subsequent pictures and by checking single pictures for consistency, I demonstrated that the seating model yields a natural looking seating behavior.

6. Conclusions and outlook

In this work, I designed a study on passengers' seating behavior in trains and developed a mobile app to support the data collection. I conducted the data collection, and processed and analyzed the data with respect to the choice of seating. The data analysis yielded insights on where passengers prefer to sit down in a compartment and in a seat group. For example, within a compartment, passengers tend to choose the seat group with the smallest number of other passengers. Within a seat group, passengers prefer window seats and forward-facing seats. However, if there is another person, passengers tend to choose the seat diagonally across from that person. Based on the results of the data analysis, I designed a model for seating behavior and implemented it in the crowd simulation software VADERE. Finally, I verified and validated the model by evaluating output data generated in a simulation run.

The seating model I developed has shown to be capable of choosing seats for simulated train passengers in a realistic way. It can be included in larger simulation systems for public transport.

Especially relevant would be the combination with other simulation models for the train interior. These can be models

- for the inflow process, i.e. the process when passengers enter the train and possibly choose a standing position to wait.
- for the act of leaving the train including queuing and waiting until the doors open (Zönnchen, 2013; Köster and Zönnchen, 2015; Seitz et al., 2016).
- for organizing the passenger exchange process; for example passengers first exit the train before new passengers enter.
- for a specific stepping behavior in crowded trains, which has been implemented in a recent study (von Sivers and Köster, 2015).

The question how passengers choose a compartment remains open in this work.

According to experts at the Münchner Verkehrs- und Tarifverbund (MVV)[1], this depends on multiple factors, including the local circumstances at the passenger's start and destination station. Additional studies would be necessary to investigate these aspects.

From the perspective of social psychology, there are interesting topics that could be further investigated. Some research ideas are described in the following list. The dataset collected in my work can constitute a basis for these.

- The collected data indicates that most passengers apparently try to maximize the distance to other passengers when choosing a seat. Hall (1966) describes a related phenomena—known as personal space—in his book. A follow-up study could compare these aspects of seating behavior between different cultures by conducting the same data collection in other countries.
- Another question is how gender and age affect seating behavior. For example, based on the dataset, it can be examined, whether women prefer to sit with other women or passengers of a specific age group prefer to sit with others of a similar age.
- Passengers arriving in groups might show some specific behavior. For groups, typical seating constellations could be investigated, e.g. the relative seating position of two friends.
- Placing hand baggage on free seats is often seen as a way to protect personal space (Rüger and Loibl, 2010; Cis, 2009; Plank, 2008). How effective this measure is and at which crowding level passengers freely remove their hand baggage to make space for others could also be examined using the data.

The results of this work can especially be interesting for public transport organizations when they plan passenger surveys. One of the goals of passenger surveys is to obtain statistics on the usage of different transportation modes. These statistics can be used for a fair division of revenues or costs in public transport systems operated by multiple companies, especially if they have a shared ticket system. If the surveys are conducted in form of interviews, the question arises, which passengers to select as interviewees. The group of interviewees should be representative

[1]Meeting with representatives from the MVV. MVV Geschäftsstelle. München, 2016-05-11.

for all passengers. However, this might not be easily achievable. For example, passengers working for some big company are overrepresented in certain parts of a train because they prefer compartments near the best exit of their destination train station. To prevent this issue, it could help to simulate passengers' distribution inside the train in advance, to decide for compartments or seat groups in which the interviews should be conducted. The seating model alone is not sufficient for such simulations because it is not concerned with choosing compartments for passengers. Therefore, the development of an additional model would be required. This model could incorporate the data on passenger counts described in section 2.6.

Acronyms

API	Application programming interface
BHM	Behavior Heuristic Model
CPU	Central Processing Unit
CSV	Comma-separated values
ER	Entity—relationship
GUI	Graphical user interface
I/O	Input/Output
ICE	Intercity Express
IDE	Integrated Development Environment
ID	Identifier
ISO	International Organization for Standardization
JAR	Java Archive
JSON	JavaScript Object Notation
MTP	Media Transfer Protocol
MUAS	Munich University of Applied Sciences
MVC	Model View Controller
MVV	Münchner Verkehrs- und Tarifverbund
N/A	“not available”

OOP Object-orientated programming

ORM Object-relational mapping

OSM Optimal Steps Model

PDDL Public Domain Dedication and License

SDK Software Development Kit

SQL Structured Query Language

UI User interface

UML Unified Modeling Language

XML Extensible Markup Language

Bibliography

VADERE at GitLab. https://gitlab.lrz.de/vadere/vadere, August 2016. (Accessed on 10/12/2016). 1.1, 3.2

R. Alizadeh. A dynamic cellular automaton model for evacuation process with obstacles. *Safety Science*, 49(2):315–323, 2011. ISSN 0925-7535. doi: http://dx.doi.org/10.1016/j.ssci.2010.09.006. URL http://www.sciencedirect.com/science/article/pii/S0925753510002262. 1.1

Christopher Bare. OO in R | R-bloggers. https://www.r-bloggers.com/oo-in-r/, September 2012. (Accessed on 08/25/2016). 2.5.3

Paul Cis. Auslastungsgrad von Eisenbahnwagen in Abhängigkeit von individuellem Fahrgastverhalten. Diplomarbeit, Technische Universität Wien, 2009. URL http://katalog.ub.tuwien.ac.at/AC07806180. 1.2, 2.1.3, 4.1.2, 6

Charles J. Clopper and Egon S. Pearson. The use of confidence or fiducial limits illustrated in the case of the binomial. *Biometrika*, 26(4):404–413, 1934. 4.2

Winnie Daamen, Dorine C. Duives, and Serge P. Hoogendoorn, editors. *The Conference in Pedestrian and Evacuation Dynamics 2014 (PED 2014)*, volume 2 of *Transportation Research Procedia, Pages 1–818*, Delft, The Netherlands, 2014. Elsevier. URL www.sciencedirect.com/science/journal/23521465/2/. 1.1

Takahiro Ezaki, Kazumichi Ohtsuka, Mohcine Chraibi, Maik Boltes, Daichi Yanagisawa, Armin Seyfried, Andreas Schadschneider, and Katsuhiro Nishinari. Inflow process of pedestrians to a confined space. *arXiv preprint arXiv:1609.07884*, 2016. 1.2

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing, Boston, MA, USA, 1995. ISBN 0-201-63361-2. 3.3.2, 3.4.1

Ziyou Gao, Yunchao Qu, Xingang Li, Jiancheng Long, and Hai-Jun Huang. Simulating the dynamic escape process in large public places. *Operations Research*, 62(6):1344–1357, 2014. doi: 10.1287/opre.2014.1312. 1.1

Edward Twitchell Hall. *The Hidden Dimension*. Doubleday, New York, 1966. 6

Florian Jaehn and Simone Neumann. Airplane boarding. *European Journal of Operational Research*, 244(2):339–359, 2015. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/j.ejor.2014.12.008. 1.2

Ansgar Kirchner and Andreas Schadschneider. Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics. *Physica A: Statistical Mechanics and its Applications*, 312(1):260–276, 2002. 1.1

Ekaterina Kirik, Tat'yana Yurgel'yan, and Dmitriy Krouglov. An intelligent floor field cellular automation model for pedestrian dynamics. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, pages 21:1–21:6, 2007. ISBN 1-56555-316-0. URL `http://dl.acm.org/citation.cfm?id=1357910.1358127`. 3.3.2

Ekaterina S. Kirik, Tatyana B. Yurgelyan, and Dmitriy V. Krouglov. The shortest time and/or the shortest path strategies in a CA FF pedestrian dynamics model. *Mathematics and Physics*, 2(3):271–278, 2009. 3.3.2

Gerta Köster and Benedikt Zönnchen. Queuing at bottlenecks using a dynamic floor field for navigation. In *The Conference in Pedestrian and Evacuation Dynamics 2014*, Transportation Research Procedia, pages 344–352, Delft, The Netherlands, 2014. doi: 10.1016/j.trpro.2014.09.029. 1.1, 3.3.2

Gerta Köster and Benedikt Zönnchen. A queuing model based on social attitudes. In *Traffic and Granular Flow '15*, Nootdorp, the Netherlands, 2015. 27–30 October 2015. 6

Gerta Köster, Daniel Lehmberg, and Felix Dietrich. Is slowing down enough to model movement on stairs? In *Traffic and Granular Flow '15*, Nootdorp, the Netherlands, 2015. 27–30 October 2015. 1.1

Xiaodong Liu, Weiguo Song, Libi Fu, and Zhiming Fang. Experimental study of pedestrian inflow in a room with a separate entrance and exit. *Physica A: Statistical Mechanics and its Applications*, 442:224–238, 2016a. ISSN 0378–4371. doi: http://dx.doi.org/10.1016/j.physa.2015.09.026. 1.1, 1.2

Xiaodong Liu, Weiguo Song, Libi Fu, Wei Lv, and Zhiming Fang. Typical features of pedestrian spatial distribution in the inflow process. *Physics Letters A*, 380(17):1526 – 1534, 2016b. ISSN 0375-9601. doi: http://dx.doi.org/10.1016/j.physleta.2016.02.028. URL http://www.sciencedirect.com/science/article/pii/S0375960116001651. 1.1, 1.2

Robert C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009. 3.3.1

Zsolt Müller. Manually running a media scan on android | monline. http://muzso.hu/node/5001, May 2014. (Accessed on 08/15/2016). 2

Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. The art of software testing. 2004. 3.5.3

Nikolaus Panzera. Die Haltezeit bei hochrangigen, innerstädtischen Verkehren – Einflussfaktoren und Optimierungspotenziale. Diplomarbeit, Fachhochschule St. Pölten GmbH, 2014. 1.2, 4.1.2

Nuria Pelechano and Norman I Badler. Modeling crowd and trained leader behavior during building evacuation. *Departmental Papers (CIS)*, page 272, 2006. 1.1

Roger D Peng. Reproducible research in computational science. *Science*, 334 (6060):1226–1227, 2011. 1.7.1

Viktor Plank. Dimensionierung von Gepäckablagen in Reisezügen. Diplomarbeit, Technische Universität Wien, 2008. URL http://katalog.ub.tuwien.ac.at/AC05039323. 1.2, 2.1.3, 6

Sheng-Jie Qiang, Bin Jia, Dong-Fan Xie, and Zi-You Gao. Reducing airplane boarding time by accounting for passengers' individual properties: A simulation based on cellular automaton. *Journal of Air Transport Management*, 40:42–47, 2014. ISSN 0969-6997. doi: http://dx.doi.org/10.1016/j.jairtraman.2014.05.007. 1.2

Bernhard Rüger and Carina Loibl. Präferenzen bei der sitzplatzwahl in fernreisezügen. *Eisenbahntechnische Rundschau (ETR)*, 59(11):774–777, November 2010. URL http://www.eurailpress.de/etr. 1.2, 2.1.2, 5.2, 6

Bernhard Rüger and Norbert Ostermann. Der Innenraum von Reisezugwagen – gratwanderung zwischen sinn und effizienz. *Eisenbahntechnische Rundschau (ETR)*, (3):38–44, March 2015. URL http://www.eurailpress.de/etr. 1.2

Michael J. Seitz. *Simulating pedestrian dynamics: Towards natural locomotion and psychological decision making.* PhD thesis, Technische Universität München, Munich, Germany, 2016. URL https://mediatum.ub.tum.de/?id=1293050. 1.2, 3.2, 3.3.3, 4.1, 4.4

Michael J. Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical Review E*, 86(4):046108, 2012. doi: 10.1103/PhysRevE.86.046108. 3.3.3, 4.4

Michael J. Seitz, Stefan Seer, Silvia Klettner, Gerta Köster, and Oliver Handel. How do we wait? Fundamentals, characteristics, and modeling implications. In *Traffic and Granular Flow '15*, Nootdorp, the Netherlands, 2015. 27–30 October 2015. 1.1

Michael J. Seitz, Nikolai W. F. Bode, and Gerta Köster. How cognitive heuristics can explain social interactions in spatial movement. *Journal of the Royal Society Interface*, 13(121):20160439, 2016. doi: 10.1098/rsif.2016.0439. 4.1.1, 6

Albert Steiner and Michael Phillipp. Speeding up the airplane boarding process by using pre-boarding areas. In *Swiss Transport Research Conference*, 9 2009. 1.2

A. M. Trinkoff. Seating patterns on the washington, DC Metro Rail System. *Am J Public Health*, 75(6):657–658, June 1985. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1646210/. 1.2

Doris Tuna. Fahrgastwechselzeit im Personenfernverkehr. Master's thesis, Technische Universität Wien, 2008. URL http://katalog.ub.tuwien.ac.at/AC05036489. 1.2, 2.1.3

Isabella von Sivers and Gerta Köster. Dynamic stride length adaptation according to utility and personal space. *Transportation Research Part B: Methodological*, 74:104 – 117, 2015. doi: 10.1016/j.trb.2015.01.009. 6

Mark Wardman and Paul Murphy. Passengers' valuations of train seating layout, position and occupancy. *Transportation Research Part A: Policy and Practice*, 74:222–238, April 2015. URL http://www.sciencedirect.com/science/article/pii/S0965856415000154. 1.2, 4.1.2

Hadley Wickham. testthat: Get started with testing. *The R Journal*, 3(1):5–10, 2011. 2.5

Yao Xiao, Ziyou Gao, Yunchao Qu, and Xingang Li. A pedestrian flow model considering the impact of local density: Voronoi diagram based heuristics approach. *Transportation Research Part C: Emerging Technologies*, 68:566–580, 2016. ISSN 0968-090X. doi: dx.doi.org/10.1016/j.trc.2016.05.012. 1.2

Yihui Xie. knitr: Elegant, flexible and fast dynamic report generation with R | knitr. http://yihui.name/knitr/, 2012. (Accessed on 07/19/2016). 1.7.1

Benedikt Zönnchen. Navigation around pedestrian groups and queueing using a dynamic adaption of traveling. Bachelor's thesis, Hochschule München, September 2013. 6

Appendices

A. Usability study for the mobile app

The following list describes improvements that have been introduced after evaluating the results of the usability study.

1. Making the form for general survey data more clear by changing defaults and hints.
2. Enhancing usage efficiency in the dialog for person-related properties.
3. Allowing “sit down & place baggage” in one step.
4. Making a shortcut for the “leave” action available (without extra confirmation) to be consistent with other actions.
5. Implementing undo support for all actions to correct mistakes.
6. Adding black borders around seats.
7. Using icons for persons or baggage on seats, in addition to text labels.
8. Shortening the app’s title in the menu bar to make more place for menu items.
9. Fixing severe bug on screen rotation (all inputs were lost).
10. Indicating the current train direction with an arrow icon.
11. Remembering the current state of the train and hiding “door release” or “train starts” actions, respectively.
12. Showing a hint when actions are pending, e.g. when marking the investigator or defining a group.
13. Highlighting the menu icon and title during a pending action, to signal the user that the action is not yet finished.

Usability study

Jakob Schöttl

November 7, 2016

Contents

1 Usability evaluation

This is an usability test for the seating data collection app. It involves a realistic scenario, wherein the subject has to accomplish a list of tasks. After the trial, a short questionnaire is given to the subject.

The subjects are observed while performing the tasks. The conductor takes notes on any problems.

The subjects know the S-Bahn train very well.

Only the most important and most critical features of the app are tested in this study.

1.1 Procedure and questionnaire

1.1.1 Introduction

This app helps to conduct surveys on seating behavior in S-Bahn trains. It allows to start a new survey and then capture all related events (during the ride).

For now, please put yourself in the situation of a person who collects data for this survey in the S-Bahn.

1.1.2 Scenario and tasks

1. Start the app
2. Start a new survey and fill in the required information given the following situation:
    - You are starting at Rosenheimer Platz and you are entering the S3 to Holzkirchen.
    - You are entering the front wagon of the train at door number 4 (counted from ahead).
    - You don't know the train identification number.
3. Capture the initial situation within the train. Don't forget to identify groups sitting there and also yourself. This is the situation:
    - You sit down on the right window side, looking in driving direction, and you're sitting in the seat group that is closer to the front of the train.
    - At the seat group to your left, there are two friends both sitting at the window side and having hand baggage on the other seats.
    - Right behind you (back to back) there is another person without any baggage.
4. You have finished the initial phase. Now, start the data collection and capture the following events.
5. The train reached the next station, the doors open.
6. One person enters and sits down on seat number 13 placing hand baggage on seat number 14.
7. The train starts again.
8. Another person comes along and sits down on seat number 6 which was occupied by hand baggage.
9. The new person obviously belongs to the group of friends.
10. The train stops at Ostbahnhof, the doors open.
11. The group of three persons to your left exit.
12. Four persons enter and sit down at seats number 1, 3, 10, and 15.

13. The train starts but it's driving direction has changed!
14. The person on seat number 13 changes its place to seat number 9 and removes the hand baggage from the other seat.
15. OK, that's enough for this trial. Please end the survey and you're almost done.

Thank you very much for contributing to this usability study! We are almost done. Please take a moment and help me to finish a short questionnaire.

1.1.3 Questionnaire

1. Gender
2. Age
3. Years using a smartphone (<=2, >2, >3, >5)
4. Time needed to finish the trial
5. How straight-forward was it for you to find out how the app is used?
6. How is it to work with the app once you are familiar with it?
7. Do you have any suggestions on how to make the app easier to use or to understand?
8. What features did you like about the app?

B. Additional data analysis

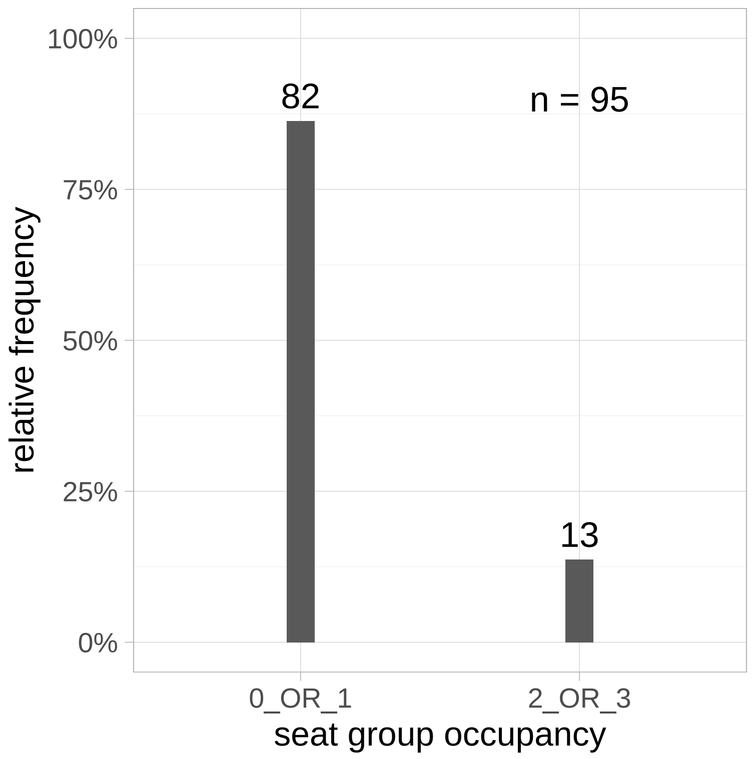


Figure B.1.: People's preference for a seat group within a compartment: choosing one with 0 or 1 other persons or one with 2 or 3 other persons.

The differences in figure B.1 are statistically significant (exact binomial test, p-value < 0.001, $n = 95$).

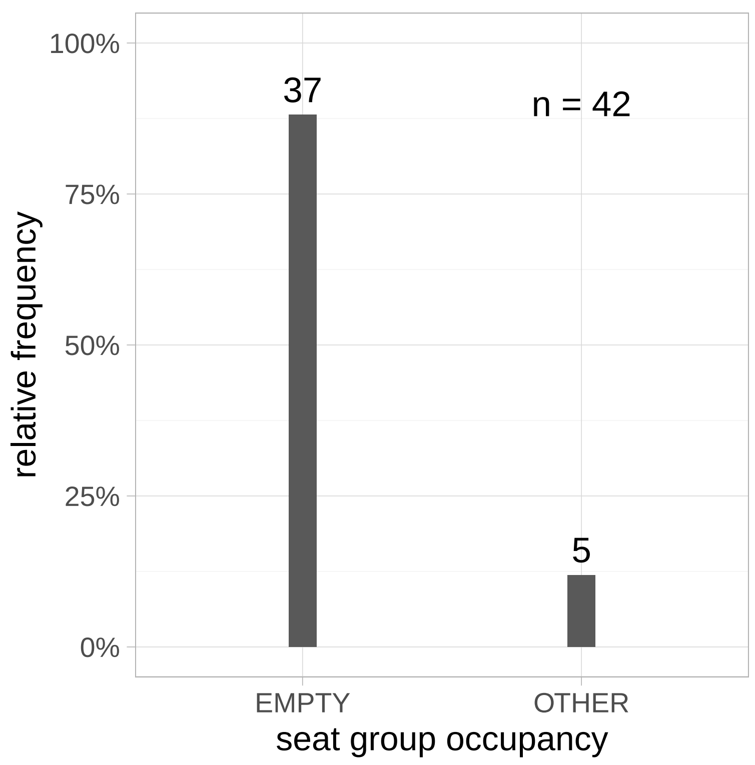


Figure B.2.: People's preference for a seat group within a compartment: choosing an empty seat group or any other.

The differences in figure B.2 are statistically significant (exact binomial test, p-value < 0.001, $n = 42$).

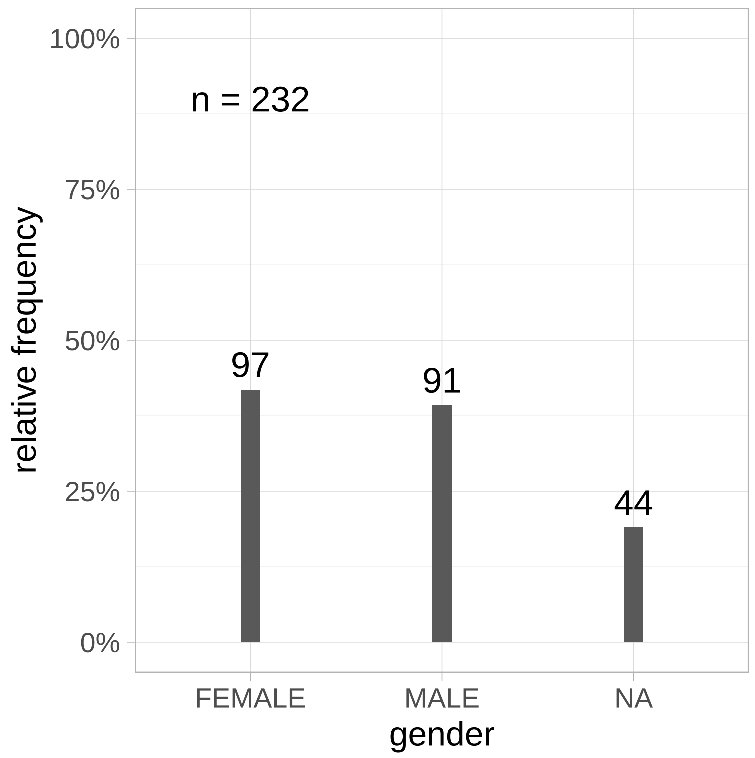


Figure B.3.: Number of passengers grouped by gender. This feature was not recorded in earlier surveys (`NA` is “not available”).

The difference between the counts of women and men in figure B.3 are not statistically significant (exact binomial test, p-value = 0.715, n = 188).

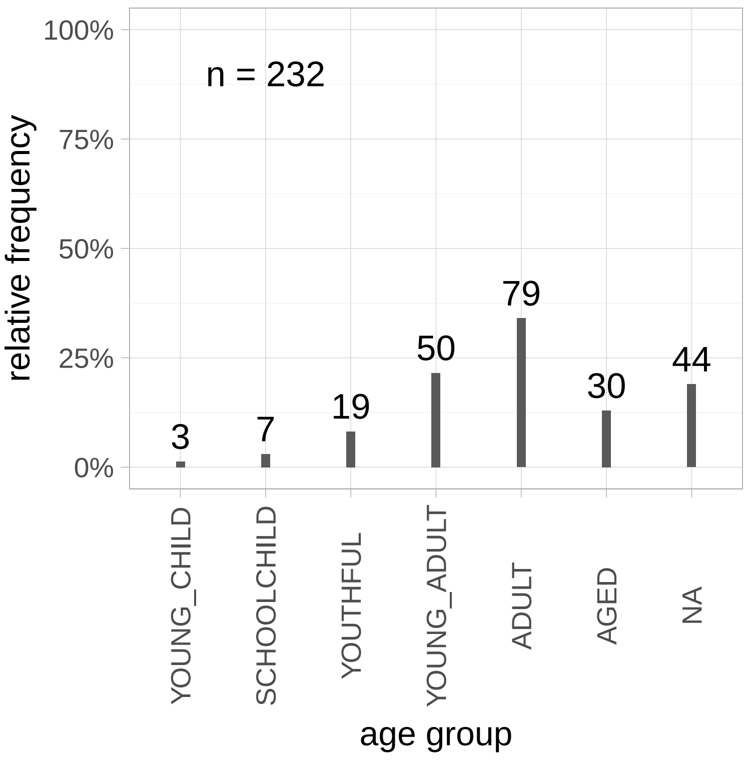


Figure B.4.: Number of passengers grouped by age group. This feature was not recorded in earlier surveys (`NA` is "not available").

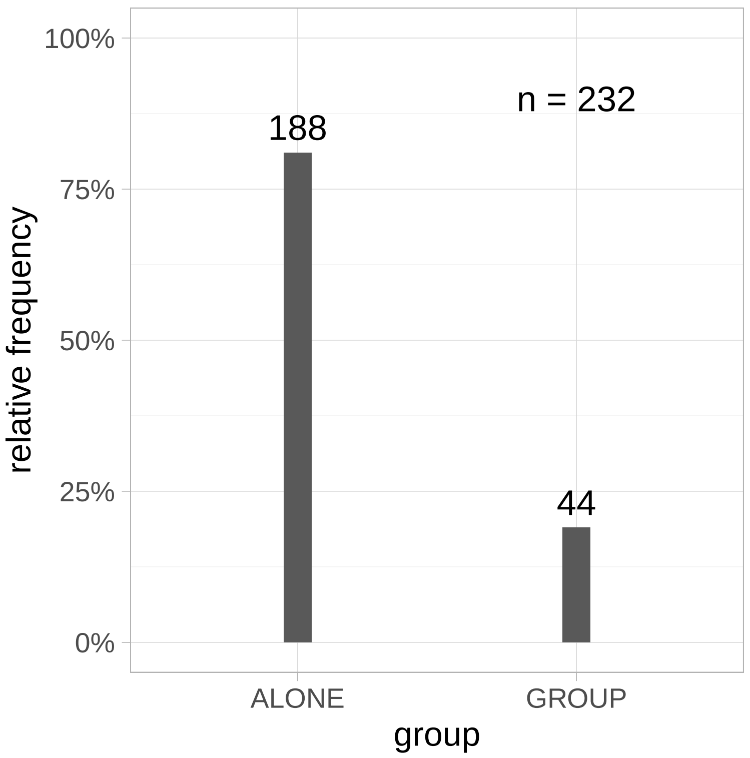


Figure B.5.: Number of passengers traveling alone vs. traveling in groups.

The differences in figure B.5 are statistically significant (exact binomial test, p-value < 0.001, $n = 232$).

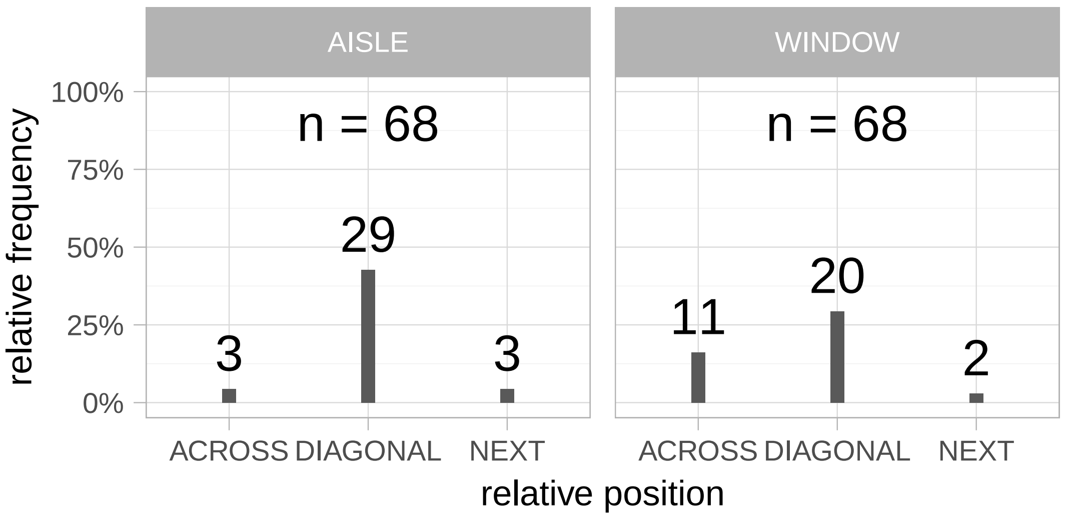


Figure B.6.: People's preference for seat positions in a seat group with one other person when deciding for an aisle or window seat respectively.

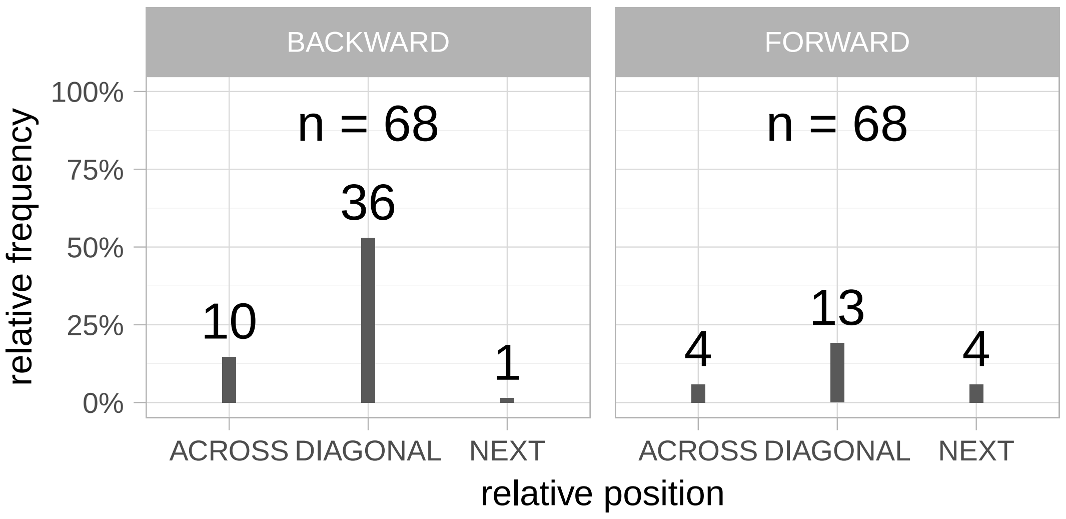


Figure B.7.: People's preference for seat positions in a seat group with one other person when deciding for an forward or backward-facing seat respectively.

The previous plot show that people most often choose the seat diagonally across from another person. However, this preference seems to be less pronounced when they choose a window seat or a forward-facing seat (which are the most popular seat categories). Possible hypothesis are:

- If they decide for a window seat they accept a closer position to the other passenger more often. That is, they give up on distance in favor of a window seat. See figure B.6.
- If they decide for a forward-facing seat they accept a closer position to the other passenger more often. That is, they give up on distance in favor of a forward-facing seat. See figure B.7.

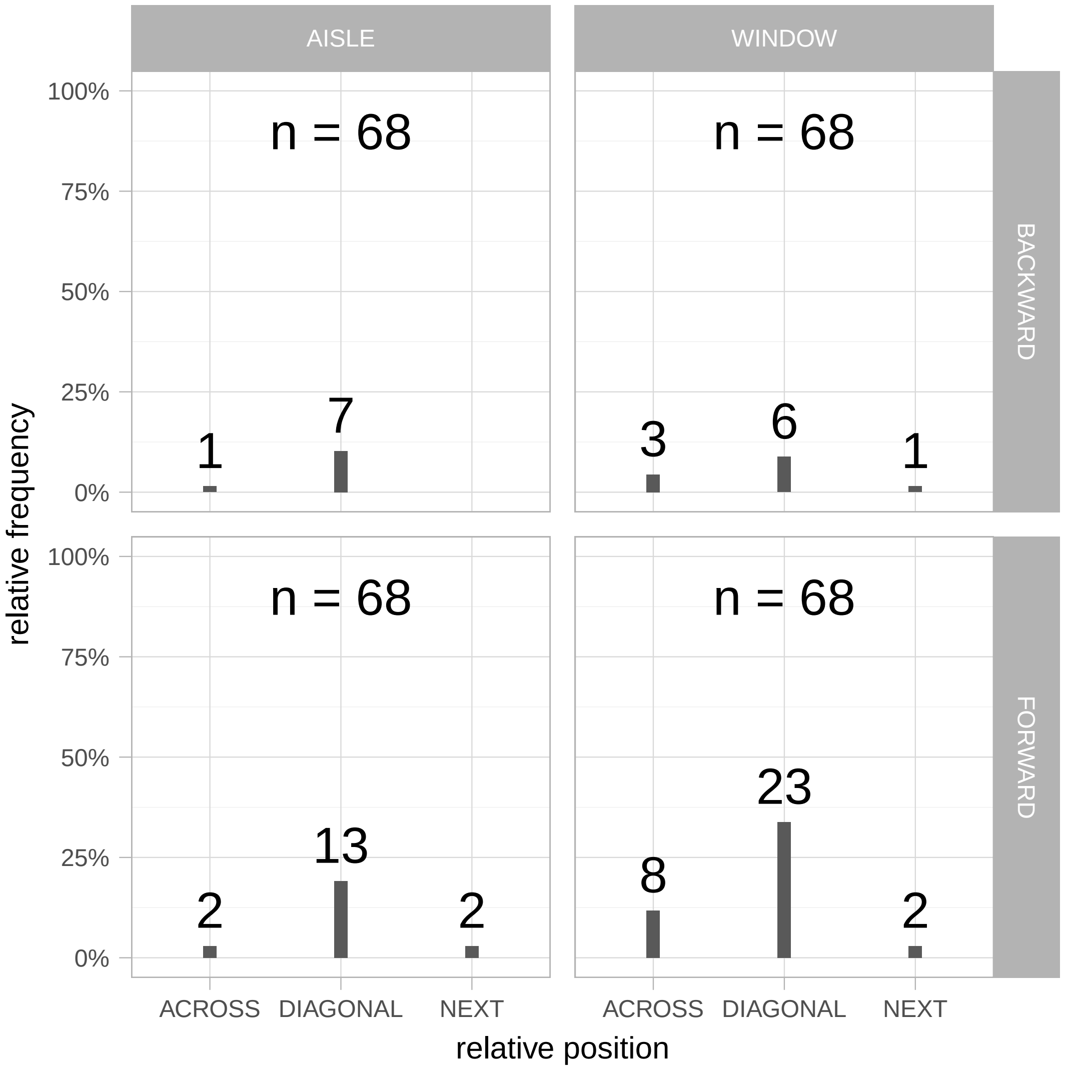


Figure B.8.: People's preference for seat positions in a seat group with one other person depending on the other person's seat position.

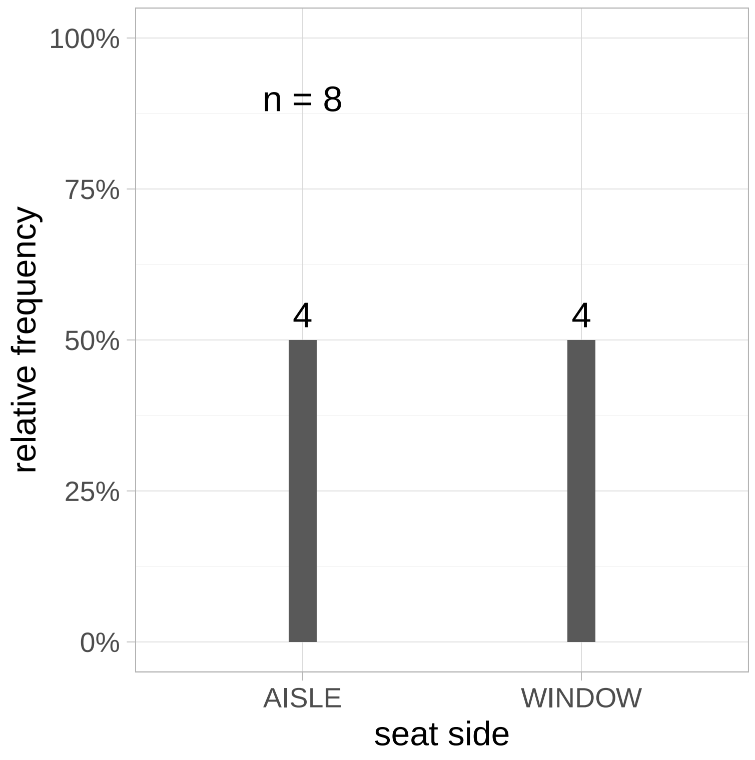


Figure B.9.: People's preference for window vs. aisle seats in a seat group with two other persons. Only those decisions are considered where there was no choice between forward vs. backward-facing seats.

The differences in figure B.9 are not statistically significant (exact binomial test, p-value = 0.289, n = 8).

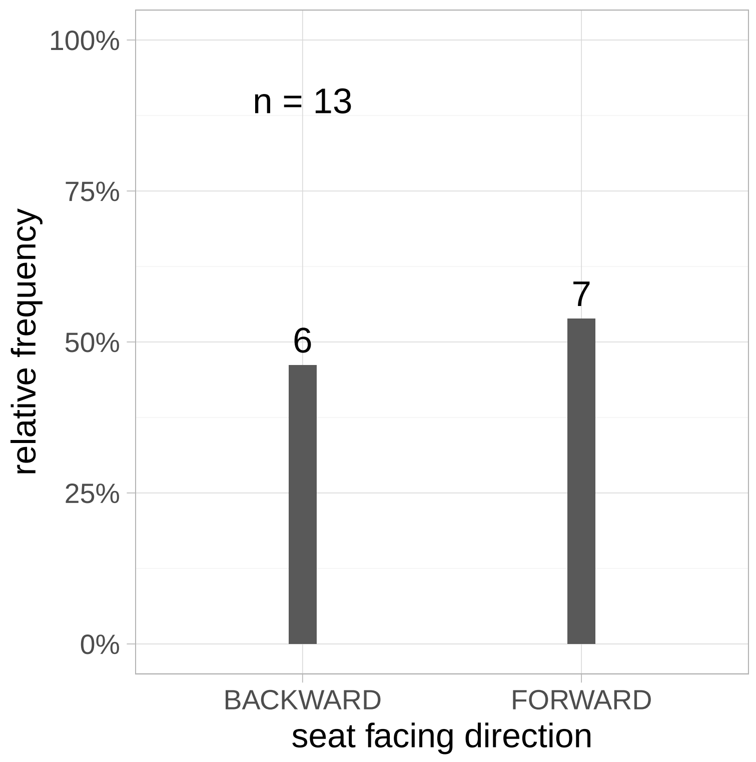


Figure B.10.: People's preference for forward vs. backward-facing seats in a seat group with two other persons. Only those decisions are considered where there was no choice between window vs. aisle seats.

The differences in figure B.10 are not statistically significant (exact binomial test, p-value = 1, n = 13).