Richard Aigner, BSc

Decision Making and Problem Solving in a Group-based Configuration System

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Softwareengineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Alexander Felfernig

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Graz, July 2019

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Acknowledgement

First of all I would like to thank my supervisor Univ.-Prof. Dipl.-Ing. Dr.techn. Alexander Felfernig for his abundant support.
I would also like to gratefully acknowledge Dipl.-Ing Müslüm Atas, Dipl.-Ing. Dr.techn. Martin Stettinger, and MSc Thi Ngoc Trang Tran for their assistance.

Above all, I want to thank my family, especially my parents, my friends, and all other people who have supported me throughout my studies.
Finally, I want to thank my cousin Ingo for proofreading the thesis.

Abstract

Group-based configuration is a new configuration approach in which complex group decisions are interpreted as configuration problems. A group of users configures a product or a service based on predefined parameters and constraints.
Within this master thesis, a prototype of an intelligent user interface to handle group-based configurations was developed. The explanation of this prototype illustrates the challenges, but also the possibilities that come with developing such a system.
Configuration is a very successful application of Artificial Intelligence. There are a variety of related topic areas from which to reuse or compare functionality. The prototype has been developed to be as easy to use as possible while still offering many features and supporting all domains.This master thesis explains these features in detail.
Furthermore, a usability test was performed on the prototype. The results of this test provide further approaches to improve the existing system and continue research on this topic.

Kurzfassung

Gruppenbasierte Konfiguration ist ein neuer Ansatz bei dem komplexe Gruppenentscheidungen als Konfigurationsprobleme interpretiert werden. Die Gruppe konfiguriert dabei ein Produkt oder ein Service mit der Hilfe von vordefinierten Parametern.

Im Rahmen dieser Masterarbeit wurde ein Prototyp einer intelligenten Benutzeroberfläche zur Handhabung gruppenbasierter Konfigurationen entwickelt. Die Erklärung dieses Prototyps zeigt die Herausforderungen, aber auch die Möglichkeiten auf, die das Entwickeln eines solchen Systems mit sich bringt.

Konfiguration ist eine sehr erfolgreiche Anwendung von Artificial Intelligence. In diesem Bereich gibt es eine Vielzahl von verschiedenen verwandten Themen, von denen Funktionsweisen übernommen oder vergleicht werden können. Der Prototyp wurde so entwickelt, dass er möglichst einfach zu bedienen ist und trotzdem viele Funktionen anbietet und alle möglichen Einsatzgebiete unterstützt. Diese Arbeit erklärt diese Funktionen im Detail. Außerdem, wurde ein Usability-Test am Prototyp durchgeführt. Das Ergebnis dieses Tests liefert weitere Ansätze zur Verbesserung des bestehenden Systems und zur Fortsetzung der Forschung an diesem Thema.

Contents

List of Figures

1 Introduction

This chapter gives a short overview of this master thesis. It describes the most important topics, their connections with each other, and their relation to the prototype that was developed in the course of this master thesis. This chapter also reflects the motivation behind this thesis and provides an overview of all other chapters.

This master thesis is about an approach of creating an intelligent user interface prototype to handle *group-based configurations*. This prototype is a web application and it is named *"KonfiGr"*. The name is a mixture of the words *knowledge-based*, *group-based*, and *configuration*, because *KonfiGr* is a *knowledge-based group-based configuration system*. This chapter will offer a detailed explanation of this system.

Group-based configuration (Felfernig, Muesluem Atas, et al., 2016; Felfernig, Stettinger, et al., 2014; Felfernig, Boratto, et al., 2018) is a relatively new and extensive topic. It is an approach to expand *knowledge-based configuration* (Felfernig, Hotz, et al., 2014) which has been evolving since the beginning of the Industrial Age. Hand in hand with the industrial revolution, configuration of products and services had to be constantly improved. Since then, *group-based configuration* is designed to meet new demands of the evolving industry. However, as products and services are very complex and manifold in today's world, there are many challenges in developing such a system. This thesis sheds light on these challenges, but also on other areas related to this topic.

Knowledge-based Configuration

According to Felfernig, Hotz, et al., 2014, in the first part of the last century, *mass production* was introduced and over time, *mass customization* has been established. *Mass customization* (Tiihonen and Felfernig, 2017) forced new technological developments, including *configuration*. *Configuration* (Zhang, 2014; Stumptner, 1997; Felfernig, Hotz, et al., 2014) is the task of assembling products or services of complex systems from parameterizable components. To capture the variety and complexity of configurable products or services, extensive knowledge of them is required, explained by Felfernig, Tiihonen, et al., 2018. Therefore, *configuration* is specified as *knowledge-based*, among others by Sabin and Weigel, 1998.

Felfernig, Hotz, et al., 2014 describe *configuration* as "one of the most successfully applied technologies of *Artificial Intelligence (AI)*". This is especially true, because configuration can be used in almost every branch of industry.

Configuration is an umbrella term that involves multiple operations. However, these operations can be summed up in two main processes. In the first process, *knowledge engineers* define the product by all its possible variants. This can be done in different ways. One of these ways is defining *component types* and *constraints*. The result of this effort is known as *configuration model*. Whereas, the process of developing such a model is known as *knowledge acquisition*. While *component types* describe each component by attributes or a set of alternatives, *constraints* limit the way different components can be combined. The defined model is provided on a *configuration system*, also known as *configurator*. In the second process, users choose one of the variants by configuring the predefined model.

Configuration is required to efficiently make good product or service decisions. Decisions are made not only in everyday life but also in industry. The spectrum of decisions reaches from *"where to spend the next vacation"* to *"financial strategies a company should focus on"*. In *configuration*, such decisions are prepared by *domain experts* and *knowledge engineers*, which enable decision makers to agree on appropriate solutions. These processes are usually supported by a variety of techniques employing *AI*.

Group-based Configuration

Group-based configuration (Felfernig, Muesluem Atas, et al., 2016; Felfernig, Stettinger, et al., 2014; Felfernig, Boratto, et al., 2018) allows configuring products or services by a group of users. *Configurators* are usually designed for single users. However, there are many products and services where this can lead to a suboptimal decision. There are many examples where *configuration* should be rather made by a group to make high-quality decisions. *Software Release Planning* (Felfernig, Zehentner, et al., 2011; Felfernig, Spöcklberger, et al., 2018) is, for example, a task in which a group of stakeholders must decide on the execution of a project. *Holiday Planning* (Jameson, Baldes, and Kleinbauer, 2004) is another scenario in which a group of friends or a family should decide together.

An industry study by Felfernig, Stettinger, et al., 2014 has found out that in addition to *Software Release Planning* and *Holiday Planning*, the following domains are particularly relevant:

- Product line scoping and open innovation
- Bundle configuration for travel groups
- Stakeholder selection for new software projects
- Architectural design in software development
- Financial service configuration
- Building configuration
- Funding decision

Even this single study with N=25 companies indicates that there is a high demand for *group-based configuration systems*. The scope of application is very versatile. To operate in all these different domains, *KonfiGr* was built *domain-independent*.

To assist users and to lead them to a convenient solution, *configuration* is supported by many different features. Most of these features belong to *AI*. *Group-based configuration* not only has to deal with the usual features of single-user configuration, but also with group decision features. It should be noted that most of these features have several variants that function differently depending on the domain and scenario.

Group Recommender Systems

Felfernig, Boratto, et al., 2018 define *recommender systems* (Felfernig, Boratto, et al., 2018; Felfernig, Jeran, et al., 2014; Jannach et al., 2010; Aggarwal et al., 2016; Adomavicius and Tuzhilin, 2005; Ricci, Rokach, and Shapira, 2011) as "decision support systems helping users to identify solutions that fit their wishes and needs". *Recommender systems* are quite similar to *configurators*. However, according to Falkner, Felfernig, and Haag, 2011, "a major difference between *configuration systems* and *recommender systems* in general is the way in which product knowledge is represented. *Configuration systems* are operating on a *configuration knowledge base* (Stumptner, 1997) which describes the properties of all allowed instances. In contrast to *configuration systems*, *recommender systems* are operating on the basis of an assortment of explicitly defined solution alternatives." Nevertheless, applying recommendation technologies to support configuration scenarios becomes increasingly popular, because it improves the quality of decision-making (Falkner, Felfernig, and Haag, 2011; Ardissono et al., 2003; Cöster et al., 2002; Tiihonen and Felfernig, 2010).

There are many processes in *configuration* that can be supported by recommendation. Falkner, Felfernig, and Haag, 2011 list the following examples: *recommendation of features and feature values* (Cöster et al., 2002; Tiihonen and Felfernig, 2010), *recommendation of relaxations* (Felfernig, Friedrich, Schubert, et al., 2009), *reuse of cases for the determination of new configurations* (Tseng, C.-C. Chang, and S.-H. Chang, 2005). The recommendations are typically *knowledge-based*, because they are determined on the basis of constraints (Felfernig and Burke, 2008). However, there are also other recommendation types that have to be considered. In *recommender systems*, a distinction is made between *collaborative filtering*, *content-based filtering*, *constraint-based recommendation*, *critiquing-based recommendation*, and *hybrid recommendation*.

In *collaborative filtering* (Schafer et al., 2007; Felfernig, Boratto, et al., 2018; Mathew, Kuriakose, and Hegde, 2016; Koren and Bell, 2015), the opinion of friends or nearest neighbors are used to provide recommendations.

In *Content-based filtering* (Van Meteren and Van Someren, 2000; Felfernig, Boratto, et al., 2018; Mathew, Kuriakose, and Hegde, 2016), the system identifies similarities between items based on categories or keywords. In other words, items are recommended to users that are similar to the items they have already chosen.

Constraint-based recommendation (Felfernig and Burke, 2008; Burke, 2000; Trewin, 2000; Felfernig, Boratto, et al., 2018; Felfernig, Friedrich, Jannach, et al., 2015) is a *knowledge-based recommendation*, because rules are defined based on deep knowledge about the items. The recommendations are generated according to these rules.

In *critiquing-based recommender systems* (Chen and Pu, 2012; McCarthy et al., 2006; Felfernig, Boratto, et al., 2018), reference items are presented to users and the users accept the items or negotiate them by specifying critiques. These critiques are used as search criteria for the recommendation.

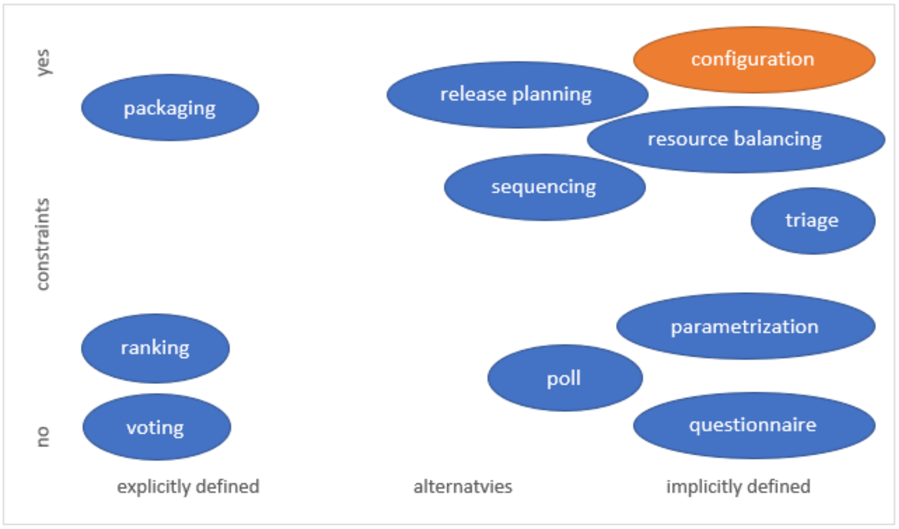
Hybrid recommendation (Burke, 2002; Burke, 2007; Chen and Pu, 2007) combines these different recommendation types to make use of their specific advantages.

Furthermore, *Recommenders* (Felfernig, Boratto, et al., 2018) can be differentiated in the way they support decision-making. They can, for example, act as a supporter to find out suitable candidate items and thus, reduce the consideration set, or they can help users to select among items by presenting them in a specific way. In extreme cases, the recommender system completely takes over the decision-making.

The most common applications provide recommendations for single users. However, the interest in *group recommender systems* is constantly growing. *Group recommender systems* (Felfernig, Boratto, et al., 2018) provide information in order to lead a group to a consensus solution in a personalized way. Therefore, *recommender systems* suggest appropriate items matching the view of the single user as well as the view of the group.

Configuration is just one of many decision scenarios that can be supported by recommendation techniques. As explained by Felfernig, Boratto, et al., 2018, there are many other related scenarios in which decision-making is the main focus. Figure 1.1 shows the relation between these applications. Felfernig, Boratto, et al., 2018 describe *release planning*, *triage*, *resource balancing*, and *sequencing* as subtypes of *configuration*. In contrast to other applications,

Figure 1.1: Decision scenarios categorized with regard to constraint inclusion and the representation of alternatives, designed by Felfernig, Boratto, et al., 2018



configuration includes constraints and the alternatives are defined only implicitly. Therefore, *configuration* is also known as *constraint-based recommender system*.

Figure 1.1 shows that *release planning* (Felfernig, Boratto, et al., 2018) is in terms of knowledge representation and inclusion of constraints a specific type of configuration. *Triage* (Felfernig, Boratto, et al., 2018) is similar to *release planning*, but in *triage*, the overall goal is to determine three partitions of a given set of alternatives. On the other hand, Felfernig, Boratto, et al., 2018 define the goal of *resource balancing* as ”to assign consumers to resources in such a way that a given set of constraints is satisfied.” In *sequencing* (Felfernig, Boratto, et al., 2018), alternatives have to be arranged in a sequence. The items are often represented in terms of parameters and the constraints are related to user preferences and further restrictions.

Parametrization (Felfernig, Boratto, et al., 2018) does not include any restrictions. The decisions are related to detailed aspects of an item. Related alternatives are therefore represented as parameter values. *Polls* (Felfernig, Boratto, et al., 2018) are presented in form of just one question and possible answers. Moreover, polls have no constraints. *Questionnaires* (Felfernig, Boratto, et al., 2018) are similar to polls with the exception that more than one question is usually asked.

In *ranking* (Felfernig, Boratto, et al., 2018), the overall goal is to prioritize a list of items. The alternatives are represented in form of a list of explicitly defined items and the scenarios typically do not include constraints. On the other hand, in *voting* (Felfernig, Boratto, et al., 2018), a group of users decides which alternative should be selected. Voting as well does not support constraints.

In *packaging* (Felfernig, Boratto, et al., 2018), the combinations of items are recommended in consideration of constraints that limit the way in which different items can be combined. In contrast to configuration, items are explicitly specified.

In group recommendations (Felfernig, Boratto, et al., 2018), there are many different algorithms. These different algorithms are more helpful or less helpful depending on the specific scenario. *KonfiGr* supports the most common algorithms to generate recommendations. However, there are many more approaches that should be considered in future work.

Open Configuration

Open configuration (Felfernig, Stettinger, et al., 2014; Zhang et al., 2014) is a new extension to *group-based configuration*. *Open configuration* not only configures the product or service within the group but also the *knowledge engineering* process itself. Moreover, *open configuration* supports *flexible product enhancement*.

Cooperative development of the knowledge base is called *community-based knowledge engineering* (Felfernig, Stettinger, et al., 2014). This new engineering method can tackle the *knowledge acquisition bottleneck*, explained by Felfernig, Reiterer, et al., 2013. *KonfiGr* supports basic *community-based knowledge engineering*, but it needs to be expanded to take full advantage of opportunities within *open configuration*.

Flexible product enhancement (Felfernig, Stettinger, et al., 2014) is the ability to include additional components or constraints in a flexible way. Therefore, the application must supply flexible interfaces that offer an easy integration of new components and constraints. This should be considered in future work.

Motivation and Outline

The number of existing applications in the area of *group-based configuration* is rather low, although the industry is already demanding such systems. Chapter 2 gives an overview of the most important related research. This chapter lists related applications and shows that there is still no existing *domain-independent group-based configuration system*. The lack of such a system was the main motivation in this thesis to develop a prototype that handles *group-based configurations* in all domains. This task offered an innovative approach that offered many new insights but also challenges.

In light of the main concepts mentioned above, it has become clear that the configuration topic is a very extensive one. Depending on the domain and scenario, different approaches should be used. However, *KonfiGr* is *domain-independent* to support the widest possible range of applications. Therefore, the used features have been chosen to support as many domains as possible.

Chapter 3 introduces the requirements for a prototype in order to develop *KonfiGr*. Chapter 4 describes how the prototype was developed and which features were implemented. Chapter 3 and Chapter 4 are based on an example in which a group of users configure a real estate together. Other popular use cases are presented in Chapter 5. This chapter also illustrates the current function of *KonfiGr*. On the basis of these use cases, a usability test was carried out. The evaluation of this test is explained in Chapter 6. Finally, Chapter 7 reflects the status of the prototype and specifies the need for future work.

2 Related Work

This chapter provides an overview of related research on group-based configuration system. Especially, areas that were relevant for the implementation of *KonfiGr* are explained. Furthermore, this chapter discusses related decision-making systems. The comparison of these applications with the developed prototype illustrates the uniqueness of *KonfiGr*.

2.1 Related Literature

Group-based configuration is a very new topic and therefore, the number of related literature is quite manageable. However, this topic is based on many other topics, as mentioned in Chapter 1. Most notably, group-based configuration is adapted from *knowledge-based configuration*. Therefore, it is helpful to understand this broader topic first. Felfernig, Hotz, et al., 2014 give an overview of basic configuration technologies. They explain how configuration has developed and why configuration is extremely important and very useful. Furthermore, they discuss various concepts of configuration knowledge representations. Many of these approaches have influenced the development of *KonfiGr*. Above all, *KonfiGr* can be assigned to the *dynamic constraint satisfaction* approach. This approach defines the configuration problem as *constraint satisfaction problem (CSP)*. Felfernig, Hotz, et al., 2014 define CSP by "a triple (V, D, C) where V is a set of finite domain variables $\{v_1, v_2, ..., v_n\}$, D represents variable domains $\{\text{dom}(v_1), \text{dom}(v_2), ..., \text{dom}(v_n)\}$, and C represents a set of constraints defining restrictions on the possible combinations of variable values $(c_1, c_2, ..., c_m)$". In a configuration task, the set of constraints is additionally defined as $C = C_{KB} * \text{REQ}$, where C_{KB} represents the configuration knowledge base and REQ is a set of user requirements. In contrast to *static constraint satisfaction*, *dynamic constraint satisfaction* ignores irrelevant variables. Moreover, *forward checking* was implemented to eliminate inconsistent values. Felfernig, Hotz, et al., 2014 also explain different approaches of conflict detection and diagnoses. However, since *KonfiGr* is a group-based application, a new concept has been developed with the help of related literature to manage conflicts and find solutions. Furthermore, Felfernig, Hotz, et al., 2014 show numerous case studies and configuration environments to demonstrate the practical use of configuration systems.

Felfernig, Muesluem Atas, et al., 2016 introduced the concept of *group-based configuration* by demonstrating a basic configuration task. This basic configuration task deals with Software Release Planning. Therefore, the paper is mainly focused on this specific domain and does not contain all other different aspects of a domain-independent group-based configuration system. Nonetheless, they show how to deal with inconsistent preferences of group members. Doing so, they distinguish between manual and automatic conflict resolution. *KonfiGr* usually resolves conflicts automatically, but there are two types of situations where users have to manually resolve conflicts (see Section 4.6). Felfernig, Muesluem Atas, et al., 2016 also demonstrate the *least misery* and *average* algorithms which resolve conflicts in user preferences. These two algorithms are implemented in *KonfiGr*. Furthermore, they mention intelligent negotiation mechanisms. Their concept of such an intelligent negotiation mechanism is also implemented in *KonfiGr*. Additionally, fairness in group decision making and predictive search are discussed. These topics require detailed user profiles and are not yet implemented in *KonfiGr*, but are important topics for future improvements.

Group recommender systems help to find consensus in group decisions. Felfernig, Boratto, et al., 2018 provide detailed information about group recommendation techniques. They explain the different recommendation types and algorithms that help users to identify suitable solutions in the area of decision-making. Additionally, different group recommender systems are described, which were also taken into account in the development of

KonfiGr. Furthermore, they provide detailed information on biases in group decisions, which are explained in the following.

Felfernig, Boratto, et al., 2018 interpret *decision biases* as "tendencies to think and act in specific ways that result in a systematic deviation of potentially rational and high-quality decisions." Some biases can be avoided by an intelligent knowledge engineering and others by an intelligent user interface.

The *anchoring effect* (Felfernig, Boratto, et al., 2018) describes a tendency to rely too heavily on first given information. *Emotional contagion* (Felfernig, Boratto, et al., 2018) represents the influence of the affective state of an user on other users' emotions within a group. Like the anchoring effect, emotional contagion can be prevented by blocking information in the early phase.

The *decoy effect* (Felfernig, Boratto, et al., 2018) is about decoy items in an item list that are inferior to all other items. A decoy item can manipulate the selection behavior of an user, because its properties make other items look better. In *KonfiGr*, the knowledge engineers can add such decoy items to influence users.

The *serial position effect* (Felfernig, Boratto, et al., 2018) occurs in item lists. Users tend to prefer the items at the beginning and at the end of a list. In *KonfiGr*, this effect might occur both: by presenting the choices, but also in the sequence of the steps. Knowledge engineers must take this effect into account when they define a project. In the future, an additional option, such as randomizing the order of choices, could prevent this effect.

The *framing effect* (Felfernig, Boratto, et al., 2018) is about the influence of the way of presenting an item. Users tend to prefer gains and avoid losses. Knowledge engineers must consider this effect.

Polarization (Felfernig, Boratto, et al., 2018) is the tendency of a group to shift towards more extreme decisions. The decisions are often riskier. This group polarization can be reduced by including dissent. *KonfiGr* allows users to create custom issues whereby these issues provide discussions. Discussions themselves already reduce this effect. However, in future work, this area should be improved to prevent this effect completely.

GroupThink (Felfernig, Boratto, et al., 2018) occurs in situations in which a group of users avoids conflicts and is not interested in analyzing decisions. This effect can be avoided by hiding the leaders' opinions or by adding experts to the decision process. Both options are supported by *KonfiGr*.

2.2 Related Applications

This section explains applications related to *KonfiGr*. These applications differ depending on the *choice scenario* to which they are assigned. Moreover, they differ on whether they support groups and whether they are domain-independent. Table 2.2 gives an overview of these comparisons. However, all applications focus on decision-making. Therefore, their implementations

Table 2.1: Related applications and their comparison regarding the support of constraints, groups, and domains

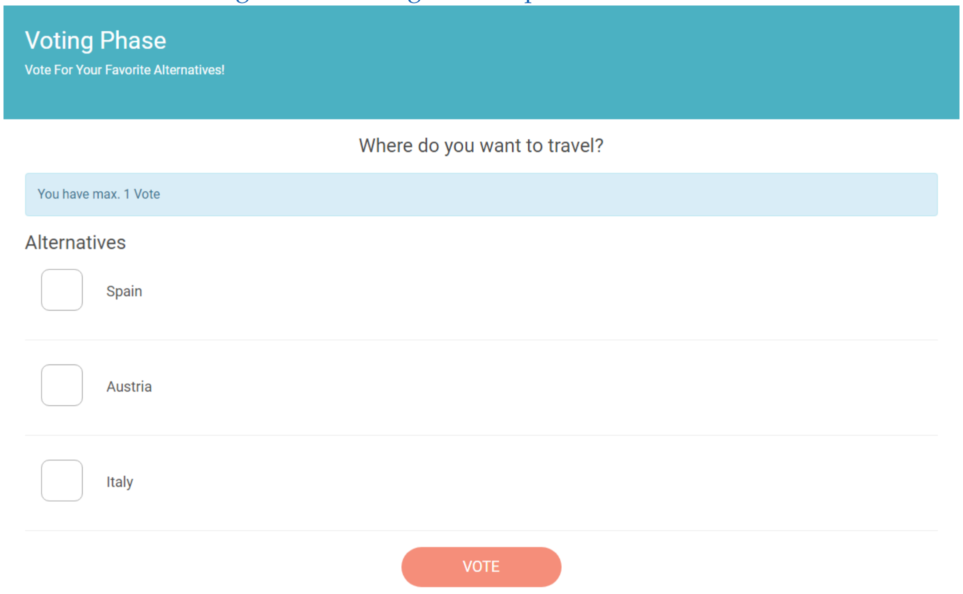
	Constraint-based	Group-based	Domain-independent
ChoiclaWeb		x	x
Doodle		x	x
Microsoft Forms		x	x
CATS		x	
PlanITpoker		x	
IntelliReq	x	x	
Microsoft Surface Configurator	x		

are comparable and partially reusable when developing a *group-based configuration system*.

ChoiclaWeb

ChoiclaWeb (Stettinger and Felfernig, 2014; Stettinger, 2014; Felfernig, Boratto, et al., 2018; Stettinger, Felfernig, Leitner, Reiterer, and Jeran, 2015; Stettinger, Felfernig, Leitner, and Reiterer, 2015; Graz University of Technology, 2018; Tran et al., 2016) is a *domain-independent decision-making tool* developed by the *Applied Software Engineering Group at the Graz University of Technology*.

Figure 2.1: Voting a travel poll in ChoiclaWeb



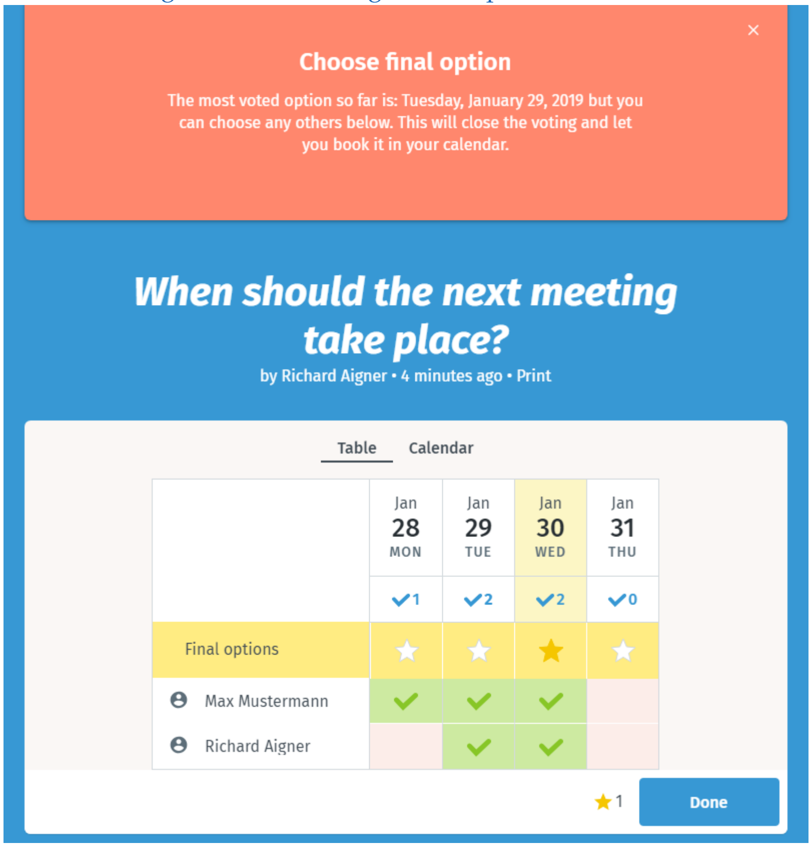
The actual version of *ChoiclaWeb* supports two different areas of application. The first area is called *"poll"*. In a poll, a group of users chooses a favorite item based on a list of alternatives. As the name implies, this area supports poll choice scenarios. Figure 2.1 shows such a poll scenario in which users must select travel alternatives. The second area is called *"challenge"*. In a challenge, users create some sort of test. These tests consist of multiple-choice questions including right and wrong answers. As shown in Figure

1.1, this area can be assigned to the questionnaire choice scenario, because the questions are not connected. Furthermore, *ChoiclaWeb* supports features to avoid biases in group decisions. These biases are explained in detail in Section 2.1.

Doodle

Doodle (Felfernig, Boratto, et al., 2018; Reinecke et al., 2013; Doodle AG, 2019) is probably the most popular system from this list. As *ChoiclaWeb* and *Microsoft Forms*, *Doodle* supports decision-making for groups based on

Figure 2.2: Choosing a final option in Doodle



Choose final option

The most voted option so far is: Tuesday, January 29, 2019 but you can choose any others below. This will close the voting and let you book it in your calendar.

When should the next meeting take place?

by Richard Aigner • 4 minutes ago • Print

	Jan 28 MON	Jan 29 TUE	Jan 30 WED	Jan 31 THU
	✓1	✓2	✓2	✓0
Final options	☆	☆	★	☆
Max Mustermann	✓	✓	✓	
Richard Aigner		✓	✓	

★1

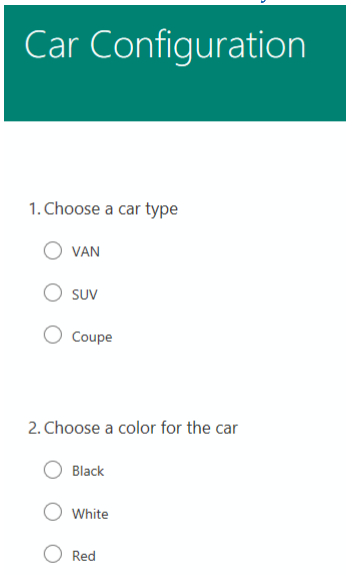
single-choice or multiple-choice lists. However, the main focus of *Doodle* is finding an appointment. *KonfiGr* can also be used for this reason.

Doodle is a *domain-independent* application and it uses the *Majority Voting (MAJ)* to aggregate user preferences. If the decision contains a contradiction, the holder of the vote is asked by *Doodle* to choose a final option (see Figure 2.2). *KonfiGr* supports the *Majority Voting* as well as choosing a final option. However, *Doodle* does not support complex configurations or recommendations and is therefore only suitable for a limited number of applications. *Doodle* can be assigned to the voting choice scenario.

Microsoft Forms

Microsoft Forms (Microsoft, 2019) is a simple app to create *surveys*, *quizzes*, or *polls*. This app is part of the *Office 365 Education Suite*. The main application areas focus on creating tests and evaluating courses for teachers as well as

Figure 2.3: Microsoft Forms survey to configure a car


Car Configuration

1. Choose a car type

- VAN
- SUV
- Coupe

2. Choose a color for the car

- Black
- White
- Red

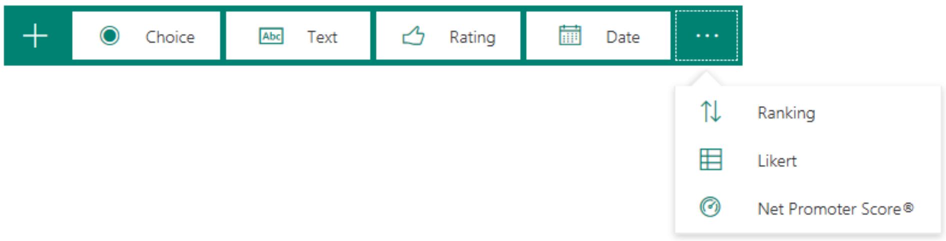
catching up customer feedback and quantifying employee satisfaction for companies.

Microsoft Forms is not a complete *configuration system*, because there are no constraints between the questions. Microsoft Forms supports *voting*, *questionnaire*, and *poll* choice scenarios. Voting scenarios can be performed by surveys, questionnaire scenarios can be performed by quizzes, and poll scenarios can be performed by polls.

In surveys, users can create questions to configure a car, as shown in Figure 2.3. However, a survey can not represent constraints information like *"there exists no red VAN"*. Furthermore, *Microsoft Forms* does not support aggregation functions to merge user inputs into group results. Moreover, discussing decision conflicts is not supported, because the *survey* scenario is not specified for group solutions.

Although *Microsoft Forms* is not a *configuration system*, there are some implementations that are usable for *configuration systems*: parameter types, recommendation for the determination of new alternatives, project handling. Figure 2.4 shows that the system supports many different question (parameter) types that are also suitable for *configuration systems*. Copying projects and sharing projects via a link are other reusable features. Furthermore, *Microsoft Forms* offers recommendations when defining new alternatives.

Figure 2.4: Available question(parameter) types in Microsoft Forms



CATS

CATS (McCarthy et al., 2006) is a *group recommender system* designed to help groups of users planning their skiing vacation. Therefore, *CATS* uses feedback from group members to suggest products that are suitable for the individuals and the entire group. Secondary, group recommendations are additionally proactively generated by shared interactions. The system collects information for creating personal profiles and group profiles. The main interaction component of *CATS* is *criticizing*. The users can criticize the features of vacations. Based on these criticisms, the system provides two recommendations. First, a individual reactive recommendation to single users and second, a proactive group recommendation. The system proposes a strategy that averages the preferences of individuals and the preferences of remaining members of a group. The recommendation is a combination of the individual model and the remaining members model. This quality score is used to rate vacation recommendations.

CATS demonstrates how to handle decisions within a group and how preferences are presented to reach consensus. Displaying the preferences of groups to facilitate the decision-making process is also used in *KonfiGr*.

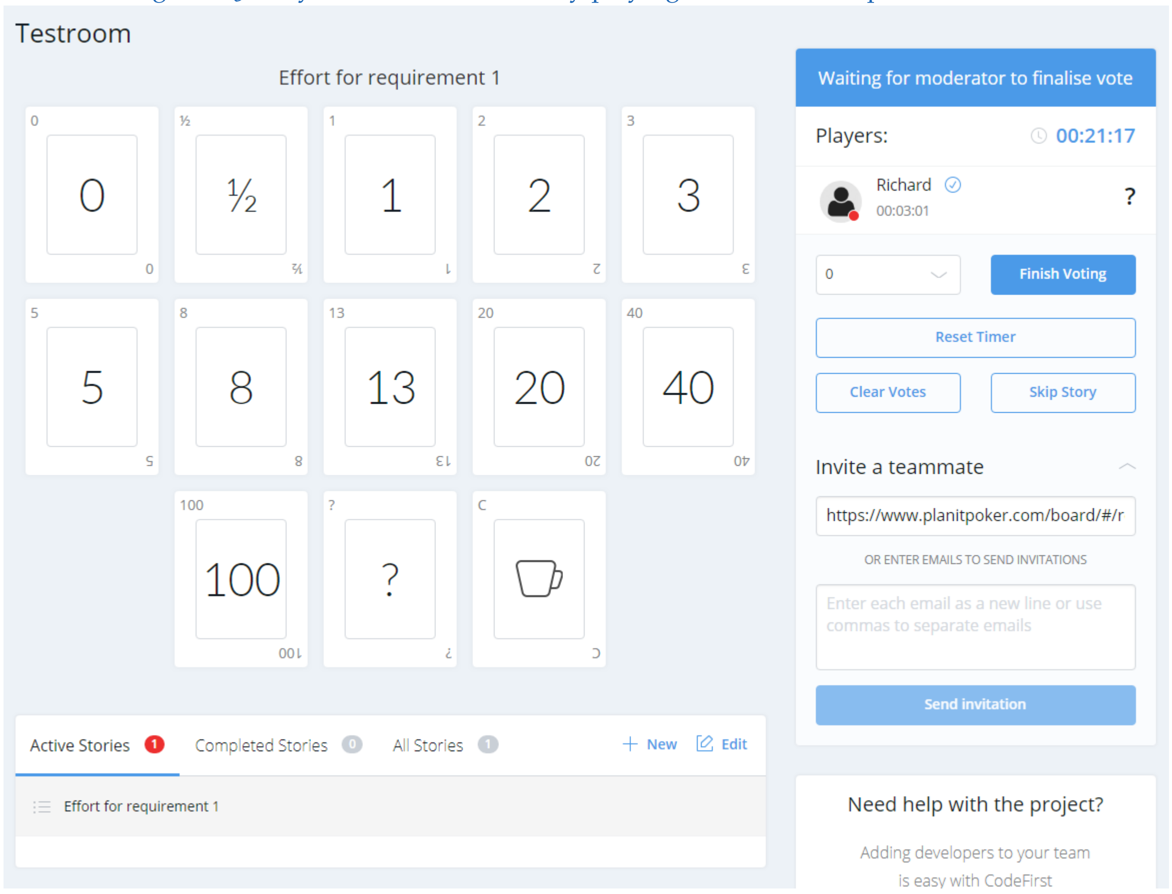
CATS is only suitable for skiing vacations. Therefore, the recommendations focus on problems that only occur in this domain. Additionally, *CATS* is not a real *configuration system*. Users can only choose between predefined destinations without any interactive constraints. Therefore, *CATS* is assigned to the voting choice scenario.

PlanITpoker

PlanITpoker (Felfernig, Boratto, et al., 2018; Code First, 2019) is a tool to estimate the effort of agile projects precisely and playfully. To do so, the users must select cards which number estimated efforts of projects. Users select cards faced down. After everyone has selected a card, the cards are compared. If there is no consensus between the cards, the process is repeated until all users have selected the same card. The effort values on the cards correspond to *Fibonacci* numbers and are especially focused on estimating software requirements.

PlanITpoker shows how to creatively build consensus in groups. This type of decision-making need not only be applied in IT but can be extended to many other areas. Figure 2.6 shows the selection of playing cards and the corresponding voting control menu.

Figure 2.5: Playful effort estimation by playing cards in PlanITpoker


Testroom
Effort for requirement 1
0
½
1
2
3
5
8
13
20
40
100
?
Waiting for moderator to finalise vote
Players:
00:21:17
Richard
00:03:01
?
0
Finish Voting
Reset Timer
Clear Votes
Skip Story
Invite a teammate
https://www.planitpoker.com/board/#/r
OR ENTER EMAILS TO SEND INVITATIONS
Enter each email as a new line or use commas to separate emails
Send invitation
Active Stories 1
Completed Stories 0
All Stories 1
New
Edit
Effort for requirement 1
Need help with the project?
Adding developers to your team is easy with CodeFirst

IntelliReq

IntelliReq (Ninaus et al., 2014) is a *recommender system* based on varieties of recommendation approaches to support stakeholders in their requirements engineering process. In requirements engineering, there are different requirements with different properties and constraints. *IntelliReq* supports hidden relationships between requirements and quality status of requirements. Furthermore, stakeholders prioritize their high-level requirements in an early requirement engineering phase. The result is a consistent set of high-level requirements including a detailed effort estimation and an implementation release plan.

IntelliReq uses *content-based filtering* to exploit similarities between user preferences and item descriptions. Therefore, keywords are extracted and compared. Alternatively, categories can be used to verify the similarity. When defining requirements, stakeholders are supported by showing similar requirements that have already been defined by other stakeholders or by the stakeholder himself. The dependency detection is based on *content-based filtering*. Therefore, stakeholders receive suggestions for potential dependencies.

IntelliReq uses group recommendation techniques to promote group consensus. Requirement evaluation and negotiation are application scenarios for the group recommendation since the stakeholders must make decisions together. *IntelliReq* uses the *Majority Voting* algorithm for group recommendations, because this algorithm has been proven in several studies (Felfernig and Ninaus, 2012). To detect inconsistencies in the requirement models, *IntelliReq* uses *knowledge-based recommendations*.

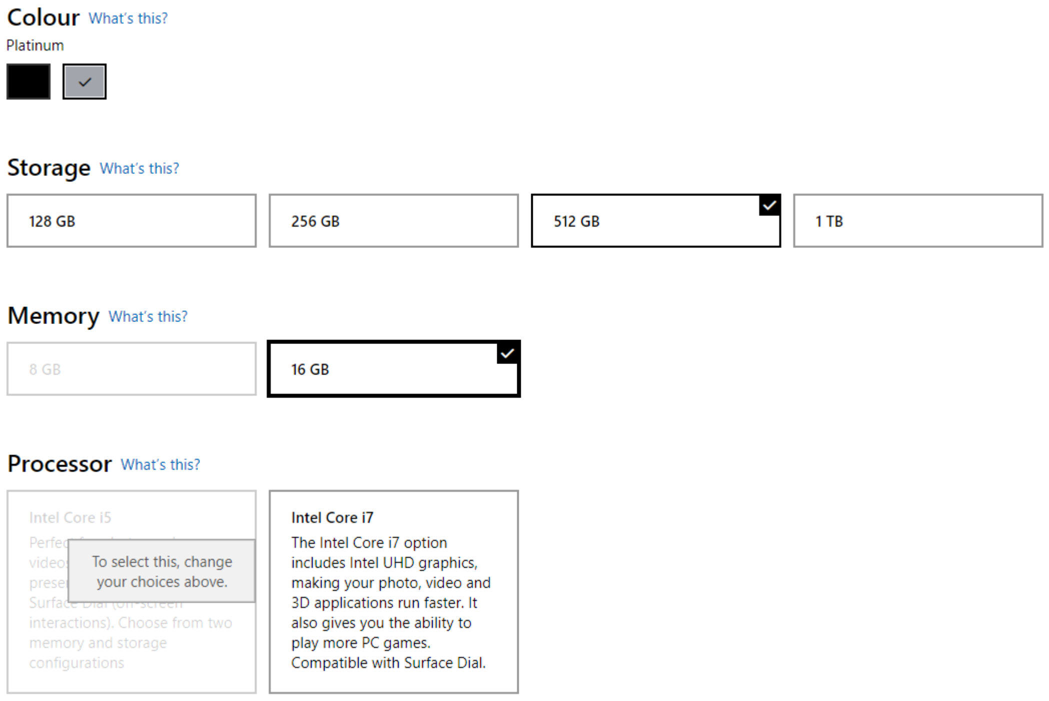
IntelliReq provides an intelligent user interface that allows a group of stakeholders to evaluate their requirements supported by group recommendations. The system automatically detects potential dependencies and displays them in a well-founded sequence. Moreover, open issues are highlighted by traffic lights.

As shown in Figure 1.1, *release planning* is one of the most similar types of *group recommender systems* to *configuration*. Stakeholders must configure requirements and dependencies between them. In contrast to *group-based configuration systems*, *IntelliReq* focuses on software requirements and is therefore only executable in this domain. However, most recommendations and aggregation types are applicable in both recommender system types. For example, the *Majority Voting* algorithm is used by *IntelliReq* and *KonfiGr*.

Microsoft Surface Configurator

The *Microsoft Surface configurator* is a typical product configurator which are used by many companies that produce configurable products. However, all these configurators only apply to a specific product and to single users. Nevertheless, there are many areas that are comparable.

Figure 2.6: Constraint interaction in a common Microsoft Surface configuration



Colour What's this?
Platinum

Storage What's this?
128 GB | 256 GB | [✓ 512 GB] | 1 TB

Memory What's this?
8 GB | [✓ 16 GB]

Processor What's this?

Intel Core i5
Perfec
videos
prese
Surface Dial (on-screen
interactions). Choose from two
memory and storage
configurations

To select this, change your choices above.

Intel Core i7
The Intel Core i7 option includes Intel UHD graphics, making your photo, video and 3D applications run faster. It also gives you the ability to play more PC games. Compatible with Surface Dial.

Figure 2.6 shows an example of a configuration process of *Microsoft Surface Pro 6*. All steps are visible in advance but can only be selected one after the other. Furthermore, steps are interdependent. For example, the choice of storage space determines which memories are available. Such dependencies can also be found in *KonfiGr*.

Motivation

This chapter described applications most similar to *KonfiGr*. Further examples of *configurators* are summarized by Thüm, Krieter, and Schaefer, 2018. As this list of applications indicates, there are many tools in the area of decision-making, but there is no application that is *domain-independent*, *group-based*, and *constraint-based*. Furthermore, many applications have no recommendations or other support mechanisms to reach consensus on decisions. The lack of such an application was the motivation behind this master thesis. Developing an user interface bringing all these different properties together has proven to be a new approach facing various challenges. This development will be explained in detail in the next chapters.

3 Requirements regarding the prototype

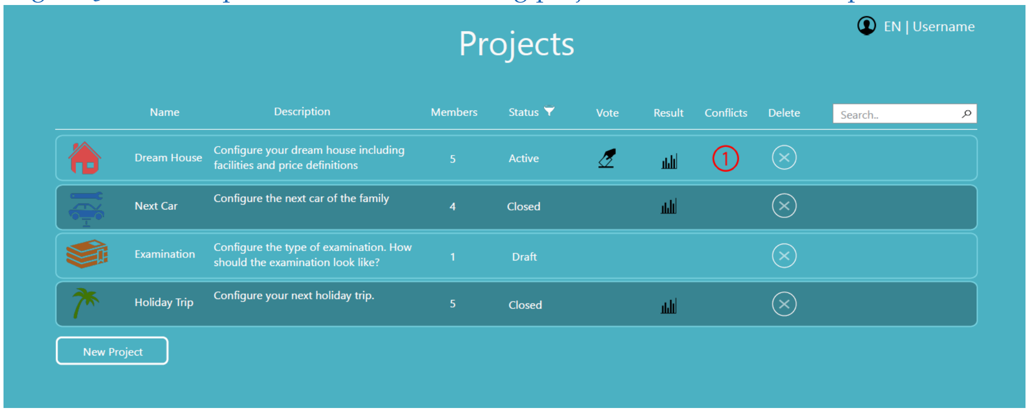
The goal of this master thesis was to develop an application that solves configuration problems that can be interpreted as complex group decision tasks. Beforehand, some scenarios have been defined that should be configurable by *KonfiGr*: "configuring an *apartment*", "configuring a *vacation*", "configuring the *content of an exam*", and "configuring a *car*". In the following two chapters, the configuration of an house or an apartment is used to explain the requirements and implementation. However, the finished prototype can handle all before mentioned scenarios. Chapter 5 demonstrates how these scenarios are operated in *KonfiGr*. Additionally, the same scenarios were used to perform an usability study. The results of this study can be found in Chapter 6.

During the planning phase, the functionality of the prototype was defined as follows: A configuration (scenario) is handled in a so-called *"project"*. These projects should be managed via a *dashboard*. In each project, a corresponding configuration model has to be set up. Therefore, a configuration problem has to be defined by a set of *variables* $V = \{v_1, ..., v_n\}$ with *domain definitions* $dom(v_i)$, a set of *constraints* $C = \{c_1, ..., c_n\}$, a *sequence* of these variables, and a set of *users* $U = \{u_1, ..., u_n\}$. The variables (component types) are typically associated with questions to identify the assignments of users. Each component type usually represents a so-called *"step"*. Therefore, the project is divided into steps and the users navigate from step to step to assign the variables. Constraints limit the combination of component types. Therefore, users' choices within a step can have an impact on the sequence of steps and on the possible choices within other steps. The task of the *administrator* is to create the configuration model based on steps and constraints. Furthermore, the administrator adds project members and monitors voting and issue resolving. *Voting* is done by all project members and can be seen as the actual *"configuration"*. Furthermore, project members should be supported during their voting. They should see the preferences of other members or a generated recommendation. After the voting, disagreements between project members should be identified. For each disagreement, a *conflict (issue)* should be created. These conflicts have to be resolved by project members making compromises. The administrator defines how voting is supported and at which point a conflict occurs. However, during the development of *KonfiGr*, some changes had to be made. These changes are described in Chapter 4. Especially, the resolving of conflicts had to be extended.

3.1 Mockup

In the course of planning a mockup was created. This section explains this mockup and all considerations behind it. The mockup starts showing a dashboard with proper information about the projects. This dashboard contains the most important information of all created projects including their status and number of open conflicts. As shown in Figure 3.1, the dashboard additionally gives a clear overview about possible user interactions.

Figure 3.1: Mockup of the dashboard listing projects and their most important details

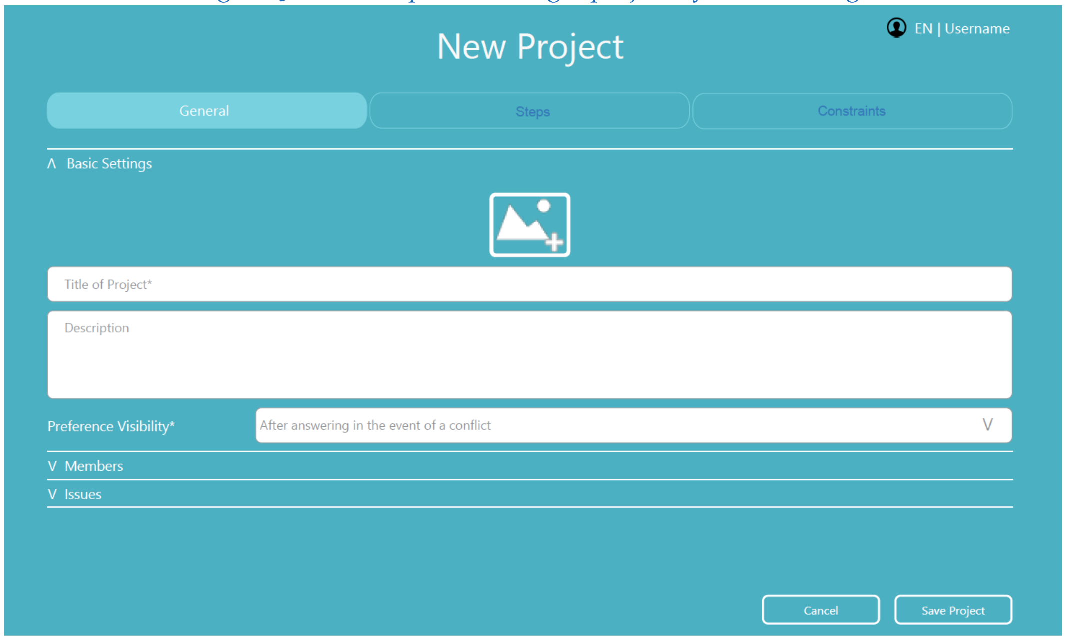


Name	Description	Members	Status	Vote	Result	Conflicts	Delete
Dream House	Configure your dream house including facilities and price definitions	5	Active	✎	📊	1	⊗
Next Car	Configure the next car of the family	4	Closed		📊		⊗
Examination	Configure the type of examination. How should the examination look like?	1	Draft				⊗
Holiday Trip	Configure your next holiday trip.	5	Closed		📊		⊗

A project is defined by name, description, and an image (see Figure 3.2). Furthermore, steps, constraints, and project members can be added. A step (variable) reflects a configuration component type. Therefore, a step typically contains a question to identify component types. A component type in turn can be defined by a set of alternatives or by a specific value. Therefore, the following step types were defined:

- Whole number (value)
- Currency (value)
- Decimal (value)
- Date (value)
- Single choice (alternatives)
- Multiple choice (alternatives)
- Yes or no (alternatives)

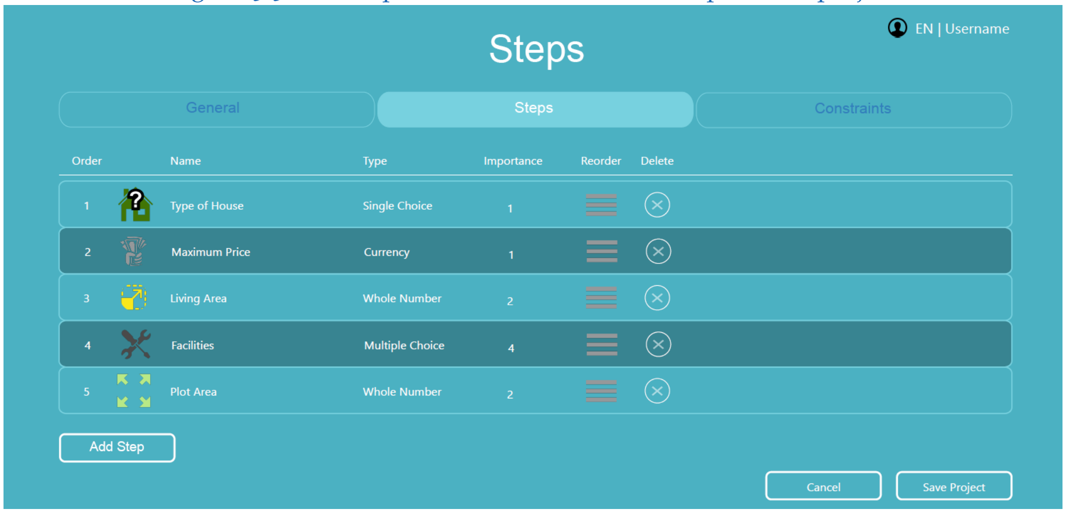
Figure 3.2: Mockup of defining a project by basic settings



During the development of *KonfiGr*, the step types had to be changed. The *yes or no* type has been removed because the *single-choice* type is easily adjustable to this model. Furthermore, a *time* and an *information* step type have been added.

A typical configuration model consists of several steps, which order should be definable. Managing the sequence of steps is illustrated in Figure 3.3.

Figure 3.3: Mockup of the overview of the steps of the project


EN | Username

Steps

General | Steps | Constraints

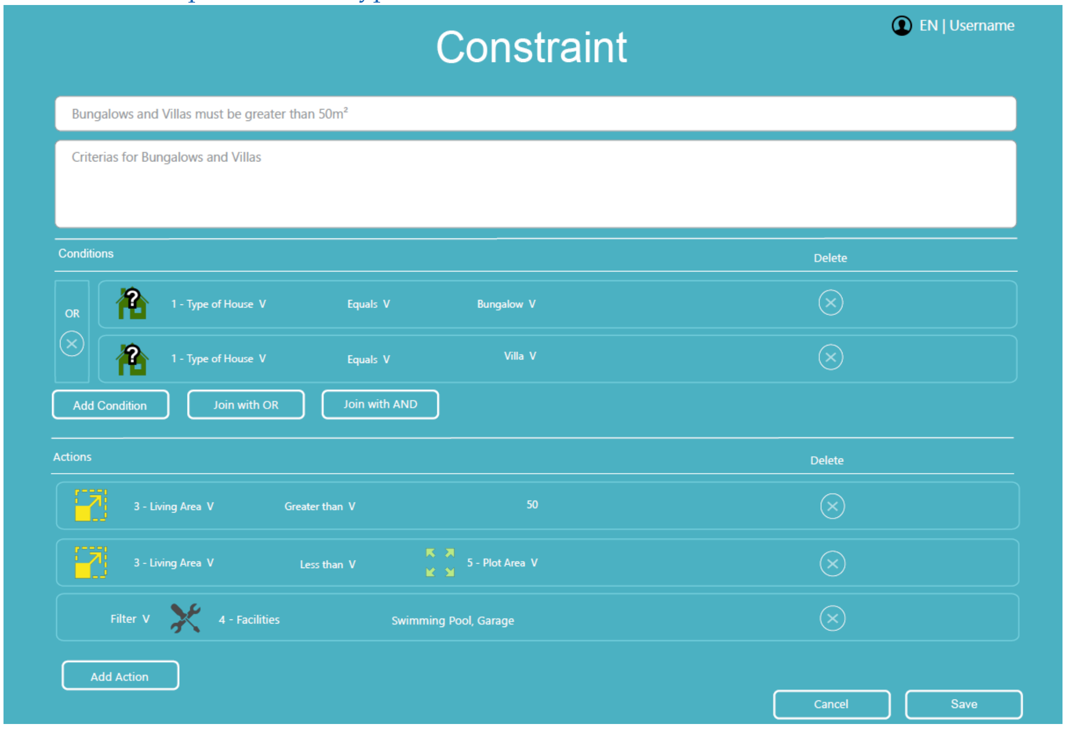
Order		Name	Type	Importance	Reorder	Delete
1		Type of House	Single Choice	1	≡	⊗
2		Maximum Price	Currency	1	≡	⊗
3		Living Area	Whole Number	2	≡	⊗
4		Facilities	Multiple Choice	4	≡	⊗
5		Plot Area	Whole Number	2	≡	⊗

Add Step

Cancel | Save Project

To limit the combinations of steps, constraints can be added. Constraints can be used to define which values or alternatives of steps can be combined with values or alternatives of other steps. These definitions are specified by conditions and actions. Conditions must be fulfilled for the actions to be executed. Additionally, conditions should be able to be linked by logical operators (see Figure 3.4). The example in this figure defines that if the *type of house (t)* is a *bungalow* or a *villa*, the *living area (l)* must be larger than 50, but smaller than the *plot area (p)*. Moreover, only *swimming pool* and *garage* are available as *facilities (f)*. This constraint can be formulated as: $c : \{t = bungalow \lor t = villa \rightarrow l > 50 \land l < p \land f \neq swimming\ pool \land f \neq garage\}$.

Figure 3.4: Mockup of defining a constraint to restrict specific facilities and the living area for specific house types



Constraint

EN | Username

Bungalows and Villas must be greater than 50m²

Criterias for Bungalows and Villas

Conditions — Delete

OR
1 - Type of House V — Equals V — Bungalow V
1 - Type of House V — Equals V — Villa V

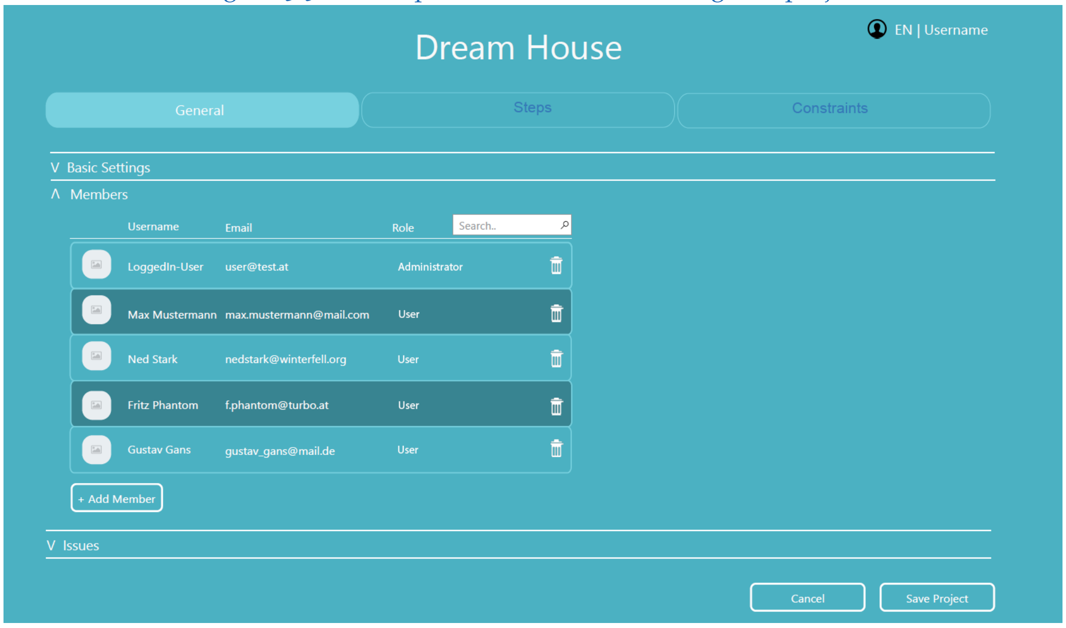
Actions — Delete

3 - Living Area V — Greater than V — 50
3 - Living Area V — Less than V — 5 - Plot Area V
Filter V — 4 - Facilities — Swimming Pool, Garage

Two different user roles were defined. First, the *administrator* who creates the configuration model and who has special rights to monitor the configuration. All other *users* can vote projects and resolve issues (conflicts). Users must be signed in in order to change their voting, discuss their decisions, or to rate choices in the case of a conflict. Project members are specified for each project, as shown in Figure 3.5.

Illustrated in Figure 3.6, the voting is supported by showing the preferences of other users or a generated recommendation. User preferences are the visualization of the choices of all other project members. The recommendation is the result of an aggregation function with the user preferences as input. Whether and which of these two sections are shown should be adjustable. These functions also generate overall result.

Figure 3.5: Mockup of the member handling in a project



Dream House

EN | Username

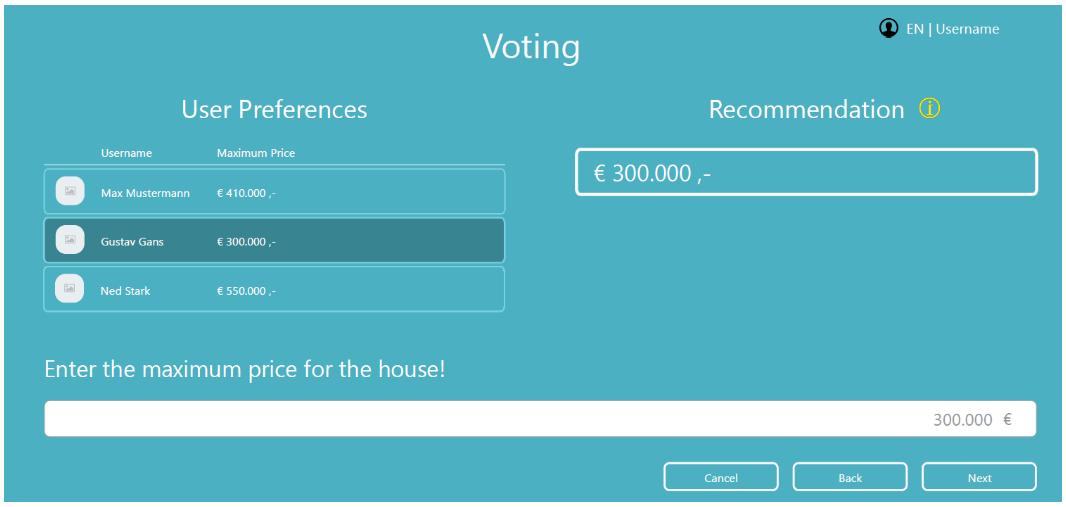
General | Steps | Constraints

V Basic Settings

Λ Members

Username	Email	Role
LoggedIn-User	user@test.at	Administrator
Max Mustermann	max.mustermann@mail.com	User
Ned Stark	nedstark@winterfell.org	User
Fritz Phantom	f.phantom@turbo.at	User
Gustav Gans	gustav_gans@mail.de	User

Figure 3.6: Mockup of voting the price of a house supported by showing user preferences and a recommendation



Voting

EN | Username

User Preferences

Username	Maximum Price
Max Mustermann	€ 410.000 ,-
Gustav Gans	€ 300.000 ,-
Ned Stark	€ 550.000 ,-

Recommendation

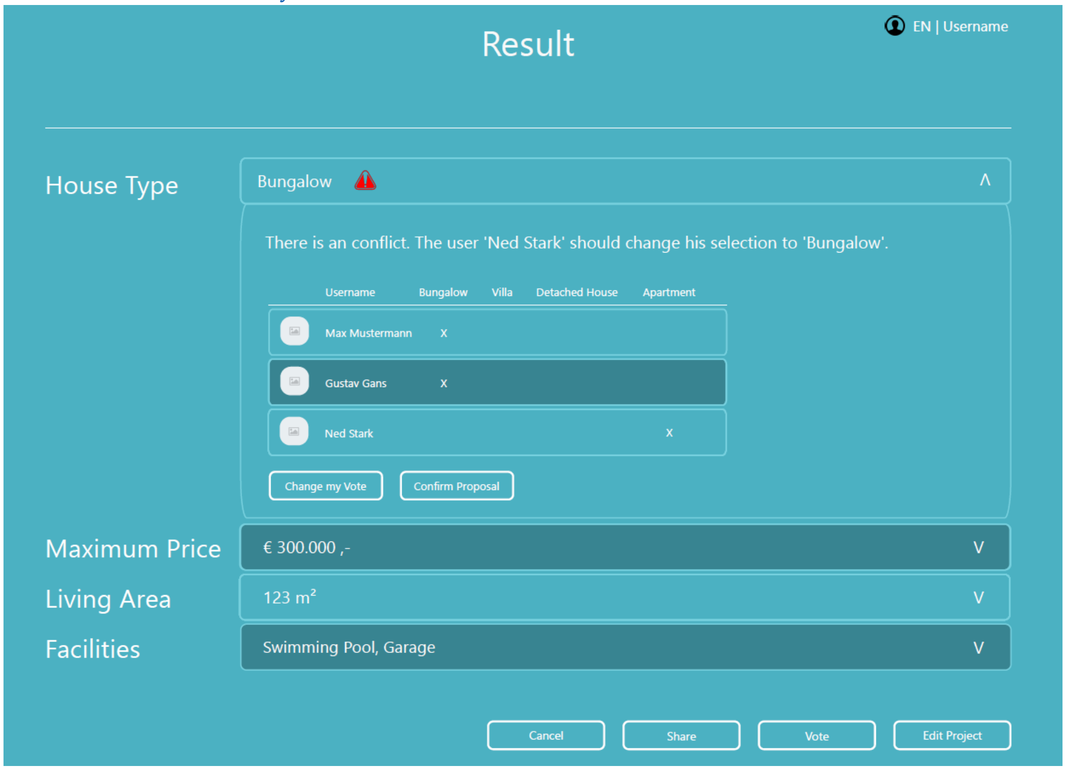
€ 300.000 ,-

Enter the maximum price for the house!

300.000 €

After the voting, a result should be visible, illustrated in Figure 3.7. Inconsistencies between project members should be highlighted and these inconsistencies should be also resolvable. Doing so, the administrators define at which point a conflict occurs. This resolving of conflicts is supported by various techniques.

Figure 3.7: Mockup of the result of the house configuration showing a conflict because of inconsistency between the members



House Type: Bungalow

There is an conflict. The user 'Ned Stark' should change his selection to 'Bungalow'.

Username	Bungalow	Villa	Detached House	Apartment
Max Mustermann	X			
Gustav Gans	X			
Ned Stark				X

Maximum Price: € 300.000 ,-

Living Area: 123 m²

Facilities: Swimming Pool, Garage

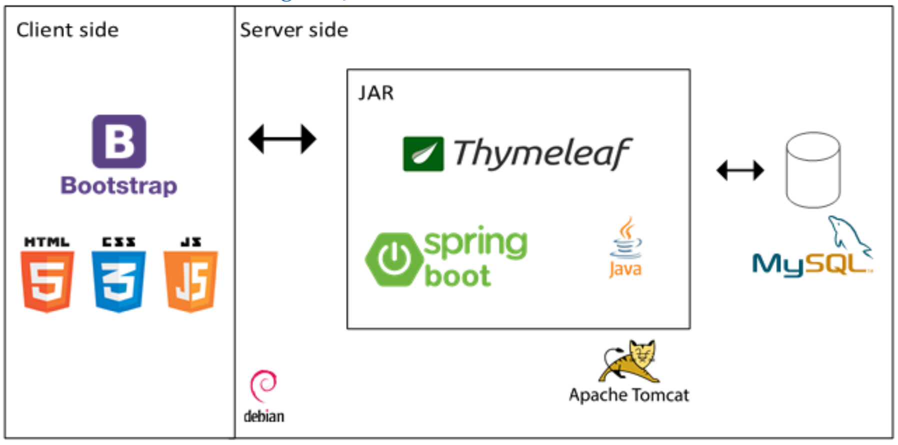
KonfiGr was developed according to this mockup. However, during the development minor changes had to be made. Thus, the design was embellished and adapted to the standards of the institute. Additionally, more importance was given to conflict resolving which features a greater variety of options to resolve a conflict. Furthermore, conflict resolving was integrated into a separate area.

3.2 Development Environment

The task behind the thesis was ordered by the *Institute of Software Technology*. The institute has guidelines and rules for creating such web applications to compare, merge, supplement, and reuse the code in a better way. Additionally, the deployment of multiple web applications is easier if the applications are all of the same type.

Shown in Figure 3.8, a *JAR* file built by *Thymeleaf*, *Spring Boot*, and *Bootstrap* is running on a *Tomcat* web server in a *Debian* system. On the *Debian* server there is a *MySQL* database to store all the data.

Figure 3.8: Software architecture



Spring Boot

Spring Boot (Pivotal Software, 2019) is an extension of the well-known *Java Spring Framework*. The *Java Spring Framework* is an open-source framework to develop *Java* applications. It is specialized to simplify the programming with *Java*. The *Spring Boot* extension allows building a runnable *Spring Application* file that needs no external *XML* configurations or libraries. This stand-alone *Spring-based Java application* is much easier to handle.

Thymeleaf

Thymeleaf (The Thymeleaf Team, 2019) is a modern server-side *Java* template engine. With *Thymeleaf HTML* templates can be developed in an easy and powerful way. *Thymeleaf* was chosen, among other things, because of its support of *Spring* modules.

Bootstrap

Bootstrap (Mark Otto, 2019) is a very popular *HTML*, *JS*, and *CSS* framework for developing web applications. The focus of Bootstrap is on responsive, mobile-first projects. Nevertheless, the development of *KonfiGr* focused on desktop design. The simple grid system allows the developer to easily design websites for devices of all shapes.

4 Implementation

This chapter contains a detailed description of the implementation of *KonfiGr*. It describes in which parts the application can be divided into, how each of these parts works, which technologies were used, and in which environment these technologies were used. This chapter reflects primarily on technical aspects. Details about the user interface of *KonfiGr* are explained in Chapter 5. Furthermore, this chapter explains the implementation based on an example of configuring an apartment.

4.1 Application Structure

The system can be divided into several parts. These parts are explained in detail in this chapter.

- Knowledge engineering
- Voting
- Aggregation
- Recommendation
- Conflict detection and resolving

4.2 Knowledge Engineering

The application is knowledge-based and domain-independent. Therefore, domain experts and knowledge engineers define the configuration model.

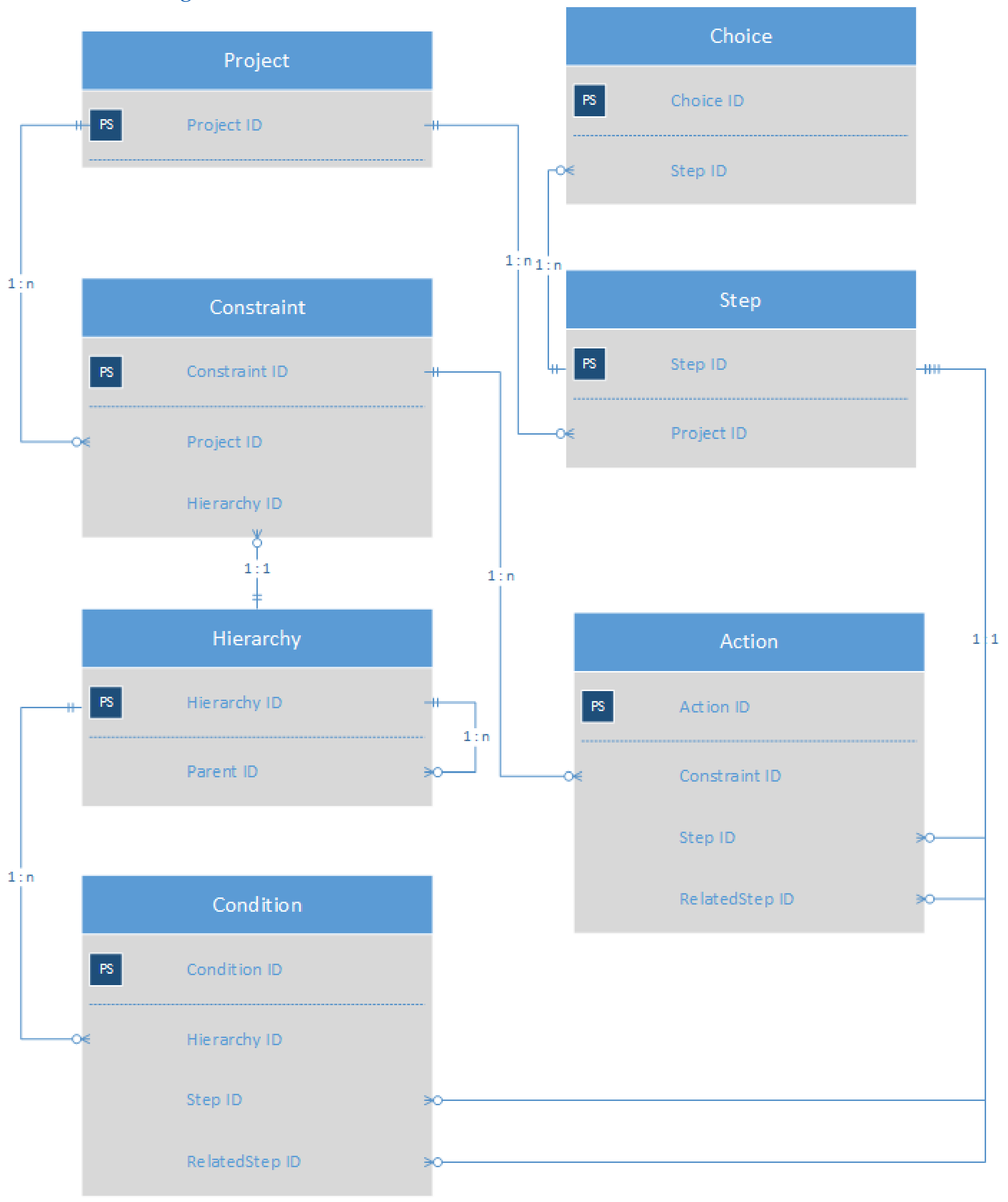
This process is called *knowledge engineering*. Sometimes a knowledge engineer is also a domain expert. The knowledge engineers have specific rights to define the model. Therefore, they are also specified as *administrators* in *KonfiGr*. On the contrary, the domain experts know how the product or service is internally structured. They know all the product's components and which alternatives for each component are available. They also know how these components interact and which solutions are possible. The knowledge engineers are assigned to put this knowledge into a model by creating a *KonfiGr* project. In the case of the apartment configurator, the domain expert might be a real estate agent, because he knows which components make up a property and how these components interact. If this real estate agent is also technically savvy, he could also do the job of the knowledge engineer. Otherwise, an engineer has to build the knowledge base in *KonfiGr* based on the agent's knowledge. Each project represents a product or a service and each *KonfiGr* step typically represents a component of this product. Moreover, constraints can be used to define interactions between components.

Figure 4.1 shows the entity relationship model of all entities involved in the knowledge engineering phase. This model includes only the key fields. Moreover, it is designed in a way that a project consists of various steps and constraints. Additionally, a step has choices (alternatives) or a step is defined by a specific value (attribute). On the other hand, a constraint has actions and conditions. These conditions are grouped by hierarchies. With the help of these entities, products of any domain can be virtually reproduced.

Project

The goal of each project is to configure a specific product or a specific service. To define this product or service, a project has a name, a description, and an image. The example project in this chapter is named *"Apartment Profile"*. Additionally, each project has a status. The status of a project in the knowledge engineering phase is called *draft*. A valid project must have at least one step but the number of constraints are optional.

Figure 4.1: Entity relationship model of the knowledge engineering phase to build up a configuration model



Step

A configuration step is some sort of question to identify a component. *KonfiGr* already offers many step types and different settings for these types. This is necessary to support as many different component types as possible. Therefore, for each step the administrator defines the step type. To represent a list of alternatives (choices), there are the *single-choice* and the *multiple-choice* type. To represent numbers, there are the *whole number*, *decimal*, and *currency* type. Additionally, there are the *date*, the *time*, and the *information* type.

Step Types:

- Single choice (sc)
- Multiple choice (mc)
- Whole number (wn)
- Decimal (dn)
- Currency (cu)
- Date (da)
- Time (ti)
- Information (in)

Configuration problems can be defined as *constraint satisfaction problems (CSP)*. Felfernig, Hotz, et al., 2014 defines CSP by ”a triple (V, D, C) where V is a set of finite domain variables $\{v_1, v_2, ..., v_n\}$, D represents variable domains $\{\text{dom}(v_1), \text{dom}(v_2), ..., \text{dom}(v_n)\}$, and C represents a set of constraints defining restrictions on the possible combinations of variable values $(c_1, c_2, ..., c_m)$”. The set of constraints C is the combination of the union of customer preferences PREF and the configuration knowledge base CKB. The set of variables (steps) V and the corresponding variable domains of the *Apartment Profile* project can be defined as follows:

V = {Object Type (o), Contract Type (c), Budget (b), Space (s), District (d), Rooms (r), Facilities (f)}.

dom(o) = [{Maisonnette, Penthouse, Basement Apartment, Any Type, Studio Apartment}, mc]
dom(c) = [{Purchase, Rent}, sc]
dom(b) = [cu],
dom(s) = [wn, 30-200],
dom(d) = [{Inner City, St. Leonhard, Geidorf, Lend, Gries, Jakomini, Liebenau, St. Peter}, mc]
dom(r) = [wn, 1-10],
dom(f) = [{Balcony, Garden, Parking Lot, Elevator}, mc].

Choice (Alternatives)

The *single-choice* and *multiple-choice* types are representing a set of alternatives. These alternatives are called *choices* and must be defined in the knowledge engineering phase. The difference between the two types is that the *single-choice* type allows only one choice. In the *Apartment Profile* project the steps *Object Type*, *Contract Type*, *District*, and *Facilities* are composed of sets of choices.

Value (Attribute)

Whole numbers, *decimals*, *currencies*, *times*, and *dates* identify a specific value. The user must enter a value that can be limited by a maximum and minimum value. In the *Apartment Profile* project, the steps *Space* and *Rooms* are *whole number* types and the step *Budget* is a *currency* type.

Information

When using the *information* type, the voter just receives information predefined by the knowledge engineer. For example, an *information* step could act as an initial step to display a welcome message.

Constraint

Constraints (Felfernig, Hotz, et al., 2014) can be used to define restrictions on the possible combinations of step values. In *KonfiGr*, a constraint consists of conditions and actions. The constraints apply not only to the configuration of the individual user, but also to the aggregated group result. Simple applications that work with only one component type typically use commutative constraints. The order of assignments is not relevant for commutative constraints. In *KonfiGr*, the sequence of the steps is still relevant, but in future work, the assignment should be possible in both directions. Furthermore, *KonfiGr* uses primitive constraints and basic relations. There are various types of constraints and still room for improvement, as explained by Felfernig, Hotz, et al., 2014. Additionally, *KonfiGr* uses *forward checking* to reduce the size of variables in the domain. Felfernig, Hotz, et al., 2014 describe *forward checking* as: "Depending on the value selected for the current variable, the values of the instantiated variables that can't be a support for the selected value are eliminated from their corresponding domains." The set of constraints in the *Apartment Profile* project are defined as follows:

CKB = $\{c1 : o \neq$ Maissonette $\wedge\ o \neq$ Penthouse $\wedge\ o \neq$ Any Type $\wedge\ o \neq$ Studio Apartment $\rightarrow f \neq$ Balcony $\wedge\ f \neq$ Elevator,
$c2 : c =$ Purchase $\rightarrow b > 10.000 \wedge b < 1.500.000$,
$c3 : c =$ Rent $\rightarrow b > 300 \wedge b < 10.000\}$

Condition

A condition specifies the verification of the value of a specific step. The verification types are different depending on the step type. The knowledge engineer has the choice between *equal*, *unequal*, *greater than*, *less than*, *contains*, *contains not*, *has data*, and *has no data*. The values can be compared with new defined values or values of other steps of the same type. Additionally, conditions can be grouped by logical *AND* and *OR* links into several hierarchical levels to support higher-order constraints.

Action

There are several actions to choose from. Each action refers to a specific step. The action types vary depending on the step type. If the step is a *single-choice* or *multiple-choice* type, then individual choices can be disabled or removed. On the contrary, if the step is a *number*, *date*, or *time* type, then the allowed range can be changed. Furthermore, all step types can be skipped or a default value can be assigned to them. Incidentally, the effect of defining default values is explained by Wang and Mo, 2018.

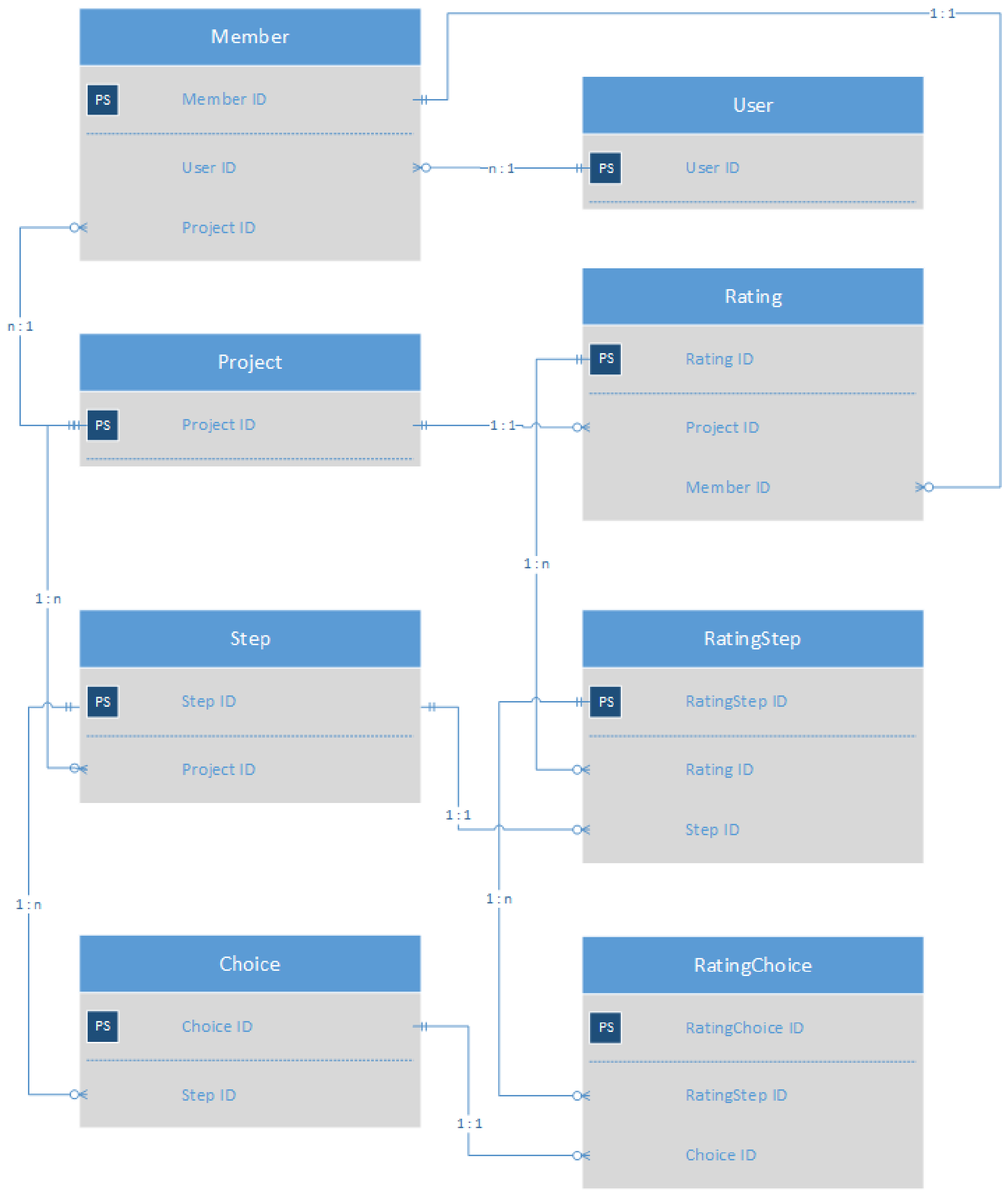
In addition to the steps and constraints, the administrator must specify the project members. All project members are entitled to vote. The administrator ends the *knowledge engineering* phase by activating the project. If the project has the status *active*, no changes to the configuration model are possible. If the knowledge engineer still wants to change the model, the project has to be deactivated, but in that case all existing votings are lost.

4.3 Voting

In *KonfiGr*, *voting* is the process of configuring a product or service. The group of users configures a solution according to a predefined knowledge base. The *voting* can be supported by showing a recommendation and by showing user preferences. These supporting mechanisms are explained in Section 4.5.

Basically, the users configure one individual step after the other. The sequence of these steps is specified in the knowledge base. The constraints are checked forward and no inconsistent values can be chosen, because the constraints are always *"arc consistency"*. Felfernig, Hotz, et al., 2014 explain *arc-consistency* as "property that must be fulfilled by combinations of variables v_1 and v_2 that are connected via a binary constraint c_b^9: each value in the domain of v_1 must have at least one corresponding value in the domain of v_2 such that c_b is fulfilled. Note that arc-consistency is directed; if variable

Figure 4.2: Entity relationship model of the voting process



v_1 is arc-consistent with variable v_2 this does not necessarily mean that v_2 is arc-consistent with v_1.". The constraints are not commutative, because the sequence of steps matters.

4.4 Aggregation

The goal of a group-based configuration is to come to a group solution. Therefore, the individual configurations must be aggregated to one result. While the project is active, a temporary result is visible in which inconsistent data and conflicts may exist. The aggregation is done per step and therefore, the aggregation type is also specified per step. Nevertheless, the constraints apply not only to the individual configurations, but also to the group result. *KonfiGr* primarily uses simple methods and algorithms to aggregate the data. Table 4.1 shows the supported aggregation types per step type.

For *multiple-choice* and *single-choice* steps the system supports mathematical functions of the *Set Theory* and the *Majority Voting* algorithm. Steps identifying a specific value can be aggregated by basic mathematical functions. Furthermore, the *date* type additionally supports the *Majority Voting*.

Table 4.1: Supported aggregation functions per step type

	$\cup$	$\cap$	$\setminus$	MAJ	MIN	MAX	AVG	Median	SUM
Single-choice, multiple-choice	x	x	x	x					
Whole number, decimal, and currency					x	x	x	x	x
Time					x	x	x	x	
Date				x	x	x	x	x	

The following aggregation functions are explained by an example with domain definitions of a given set of facility types dom(f) = {Balcony, Garden, Parking Lot, Elevator} and a set of users U = {A, B, C}.

Table 4.2: Voting example of a given set of facility types and a set of users

	Balcony	Garden	Parking Lot	Elevator
A	X	X		
B		X	X	
C	X	X		

Intersection (∪) The result of this aggregation method is composed of all choices chosen by every user. The result of the example in Table 4.2 would be {Garden}.

Set Union (\) The result of this aggregation method is composed of all choices chosen by at least one user. The result of the example in Table 4.2 would be {Balcony, Garden, Parking Lot}.

Difference (∩) The result of this method is composed of all choices chosen by none of the users. The result of the example in Table 4.2 would be {Elevator}.

Majority Voting (MAJ) MAJ arranges the choices by the number of times they were selected. The resulting sequence of the example in Table 4.2 would be {1: Garden, 2: Balcony, 3: Parking Lot, 4: Elevator}.

Table 4.3: Voting example of a whole number value by users U = {A, B, C}

A	3
B	10
C	5

Minimum Value (MIN) The result is the lowest value. The result of the example in Table 4.3 would be {3}.

Maximum Value (MAX) The result is the highest value. The result of the example in Table 4.3 would be {10}.

Average Value (AVG) The result is the average value. The result of the example in Table 4.3 would be {6}.

Median Value The result is the median value. The result for the example in Table 4.3 would be {5}.

Sum The result is the sum of all entered values. The result of the example in Table 4.3 would be {18}.

In the *Apartment Profile* project, the steps have the following aggregation functions {agg(v_1), agg(v_2), ..., agg(v_n)}:

agg(o) = {MAJ},
agg(c) = {MAJ},
agg(b) = {AVG},
agg(s) = {MAX},
agg(d) = {$\cap$},
agg(r) = {Median},
agg(f) = {MAJ}

These definitions combined with the definitions of the restrictions in Section 4.6 specify that the result will be composed of the most chosen object type, the most chosen contract type, the average of all entered budget values, the maximum entered space value, all non-chosen districts, the median of all entered room numbers, and the two most chosen facility types.

4.5 Recommendation

In group decisions it is necessary to achieve consensus. There are many techniques that support votings to reach consensus, but often these techniques influence individual votings. Depending on the domain or even depending on the step, these influences are intentional or not. *KonfiGr* supports two recommendation techniques. Firstly, it is possible to show a recommendation based on an aggregation function and secondly, the preferences of all other project members can be displayed. The knowledge engineer must configure the type of recommendation per step.

Recommendation based on Aggregation

The first recommendation type is based on aggregation functions and is just called *"recommendation"* in *KonfiGr*. This type uses the same aggregation functions as when generating the group solutions. Therefore, the recommendation can show, among other things, the temporary result of a step. However, the aggregation functions of the recommendation and the result generation can also differ, as shown in the following example:

The question of this example step is: *"Which districts do you want to avoid?"*. User A and user B have voted while the recommendation aggregation type is *set union*, but the result aggregation type is *difference*. The idea behind this definition is to let users pick districts that are not suitable for them. Then the suitable districts remain as solution. In contrast, all districts that are already excluded will be recommended.

$pref_{Ad}$ = {Inner City, Geidorf, Lend, Gries}
$pref_{Bd}$ = {St. Leonhard, Gries, Jakomini, Lend}

Recommendation:

rec_d = {Inner City, Geidorf, Lend, Gries, St. Leonhard, Jakomini}

Solution:

$CONF_d$ = {Liebenau, St. Peter}

User Preferences

Another recommendation technique of the application is *"showing the user preferences"*. These user preferences are simply the selections of all project members. In the example about districts, d_A and d_B would be displayed to the next voter. The knowledge engineer decides whether the preferences are never visible, always visible, or only visible after the early phase. Showing the user preferences only after the early phase could avoid biases in group decisions. In *KonfiGr*, the early phase is over after one third of the project members have submitted their votings.

4.6 Conflict Detection and Resolving

The administrator closes the project after all project members have voted. Then the result is scanned for conflicts. Conflicts arise when voters disagree on one result of a step. The knowledge engineers have to define these disagreements. However, there are only two types of conflict:

First, a conflict may occur in *single-choice* and *multiple-choice* steps if the *number of resulting choices* of the aggregation function does not match the number defined in the step settings. In the *Apartment Profile* example, the defined *number of resulting choices* for the *Facilities* step is 2. However, if the result consists of just one single choice or more than 2 choices, the project members have to fix this.

Second, a conflict may also occur in *number*, *date*, and *time* steps, if the *accepted difference* is exceeded. In the *Apartment Profile* example, the *accepted difference* of the *room* step is 2. If one user chooses *1* and another user chooses *4*, a conflict arises, because the difference exceeds the acceptable limit.

The restrictions $\{\text{res}(v_1), \text{res}(v_2), ..., \text{res}(v_n))$ of the *Apartment Profile* example are defined as follows:

$\text{res}(o) = \{1\}$,
$\text{res}(c) = \{1\}$,
$\text{res}(b) = \{\infty\}$,
$\text{res}(s) = \{10\}$,
$\text{res}(d) = \{\infty\}$,

res(r) = {2},
res(f) = {2}

Conflicts can be resolved in several ways:

- The users change their selections
- The administrator selects the final choices
- The users weight the choices and these weightings in turn are rated by an algorithm
- The values are approximated

Of course, the administrator can only select the final choices or weight choices, if it is a *multiple-choice* or *single-choice* step. On the contrast, the approximation of values only works for *number*, *time*, and *date* steps.

When *weighting* choices, the voters assign importance to each choice, varying from 1 to 10. A progress bulk shows how many voters have already weighted. After that, the administrator resolves the conflict by selecting one of the following algorithms:

LEAST MISERY (LMS) This algorithm ranks the choices by the lowest evaluations. The highest of all lowest ratings is the highest ranked choice. This algorithm leads to choices that no one really dislikes.

AVERAGE (AVG) This algorithm calculates the average rating of each choice.

AVERAGE WITHOUT MISERY (AVM) This algorithm also calculates the average rating of each choice, but removes those whose individual rating is below a defined threshold. In *KonfiGr* this threshold is defined as 3.

Table 4.4 shows that the three algorithms can result to three different solutions (highlighted in bold).

Table 4.4: Weighting example presenting the algorithms least misery, average, and average without misery

	User A	User B	User C	LMS	AVG	AVM
Balcony	3	7	8	3	**6**	6
Garden	8	4	9	**4**	**7**	**7**
Parking Lot	5	5	5	**5**	5	5
Elevator	8	2	2	2	4	**8**

Regardless of the step type, each voter can comment on conflicts to start a discussion. Discussions are very important for decision-making. Discussions can prevent biases and assist in finding a solution.

In group-based configurations, the biggest challenge is finding a solution that will satisfy everyone. The way to achieve consensus depends on domain, group size, and many other factors. *KonfiGr* already offers some different types of resolving conflicts to achieve consensus. However, a main purpose of this work was to find out, if the usability of the used approaches is satisfying the end user. The next chapter shows the working with *KonfiGr* based on some use cases.

5 Use Cases

This chapter presents the user interface of *KonfiGr* based on practical examples. The first use case is called *"Apartment Profile"*. This example has already been used in Chapter 4 to explain the details of the implementation. This use case will introduce the typical process of a project in *KonfiGr*. Additionally, this chapter explains three more projects. All these use cases were used for the usability test, the results of which are presented in Chapter 6.

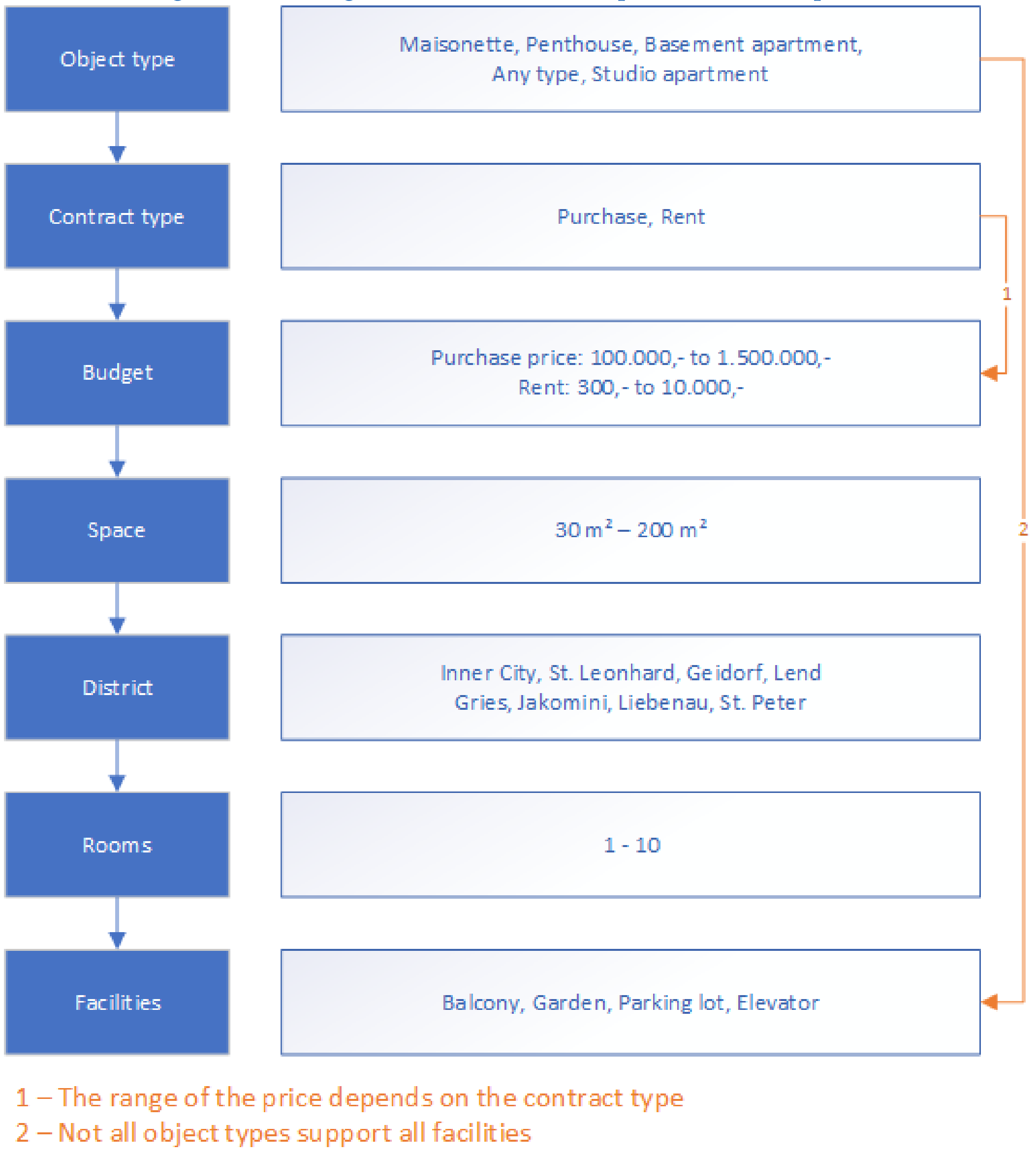
5.1 Apartment Profile

In this use case, a group of users wants to create a profile for searching apartments. Therefore, one user must be familiar with this topic area. This user is called *domain expert*. *Knowledge engineers* in turn are those users who insert the domain information into a *KonfiGr* project. This whole process is called *knowledge engineering*. In practice, the domain expert could be a real estate agent, enabling his clients to create search profiles via *KonfiGr* to find suitable apartments for them.

Knowledge Engineering

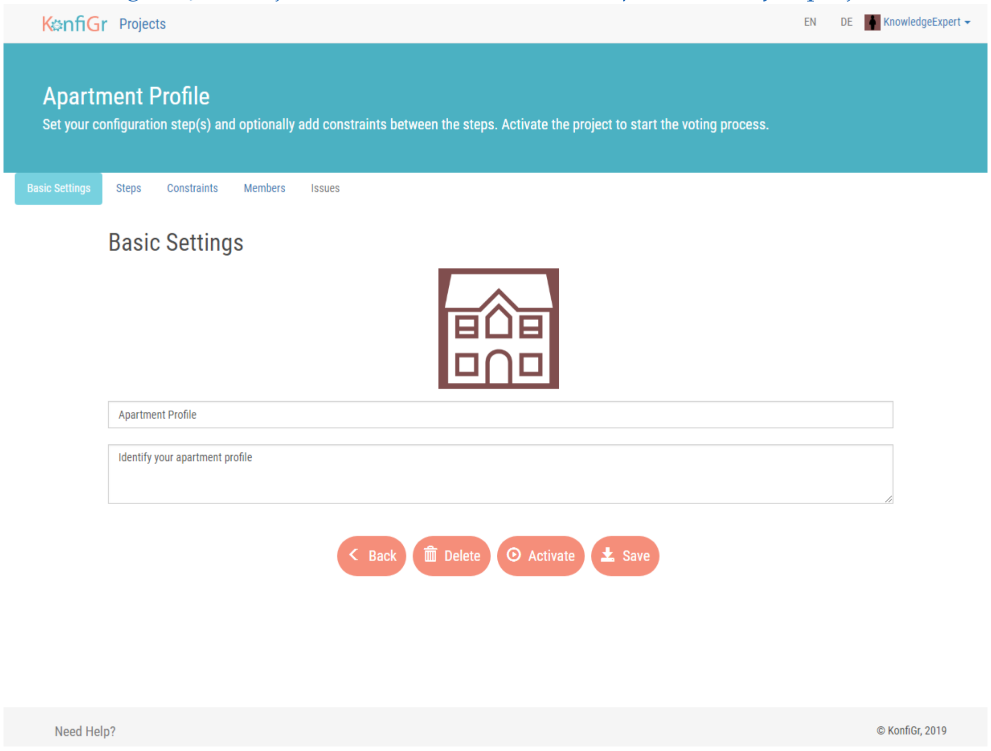
Typically, the domain expert defines a model that represents all information about the domain. An overview of the configuration model of this use case is shown in Figure 5.1. More information about the knowledge model in this example is explained in Chapter 4. The model consists of seven components and two constraints. The group has to make a decision regarding *object type*, *contract type*, *budget*, *space*, *district*, *rooms*, and *facilities* to create their search profile.

Figure 5.1: Configuration model of the apartment search profile


Object type
Maisonette, Penthouse, Basement apartment, Any type, Studio apartment
Contract type
Purchase, Rent
Budget
Purchase price: 100.000,- to 1.500.000,-
Rent: 300,- to 10.000,-
Space
30 m² – 200 m²
District
Inner City, St. Leonhard, Geidorf, Lend Gries, Jakomini, Liebenau, St. Peter
Rooms
1 - 10
Facilities
Balcony, Garden, Parking lot, Elevator
1
2
1 – The range of the price depends on the contract type
2 – Not all object types support all facilities

Additionally, there are two constraints between the components in this model. The first constraint implies that the available *budget* range depends on the *contract type*. This is the case, because a *purchase price* is much higher than a *rent*. In the second constraint, the available *facilities* depend on the *object type*, as a ground floor apartment do not have a *balcony* or *elevator*. The task of the knowledge engineer is to integrate this whole model into *KonfiGr*. The following section explains this process in detail. When the knowledge engineer creates a project, he defines a *name*, a *description*, and an *image*, as shown in Figure 5.2.

Figure 5.2: Project creation formula in the *Apartment Profile* project



Apartment Profile

Set your configuration step(s) and optionally add constraints between the steps. Activate the project to start the voting process.

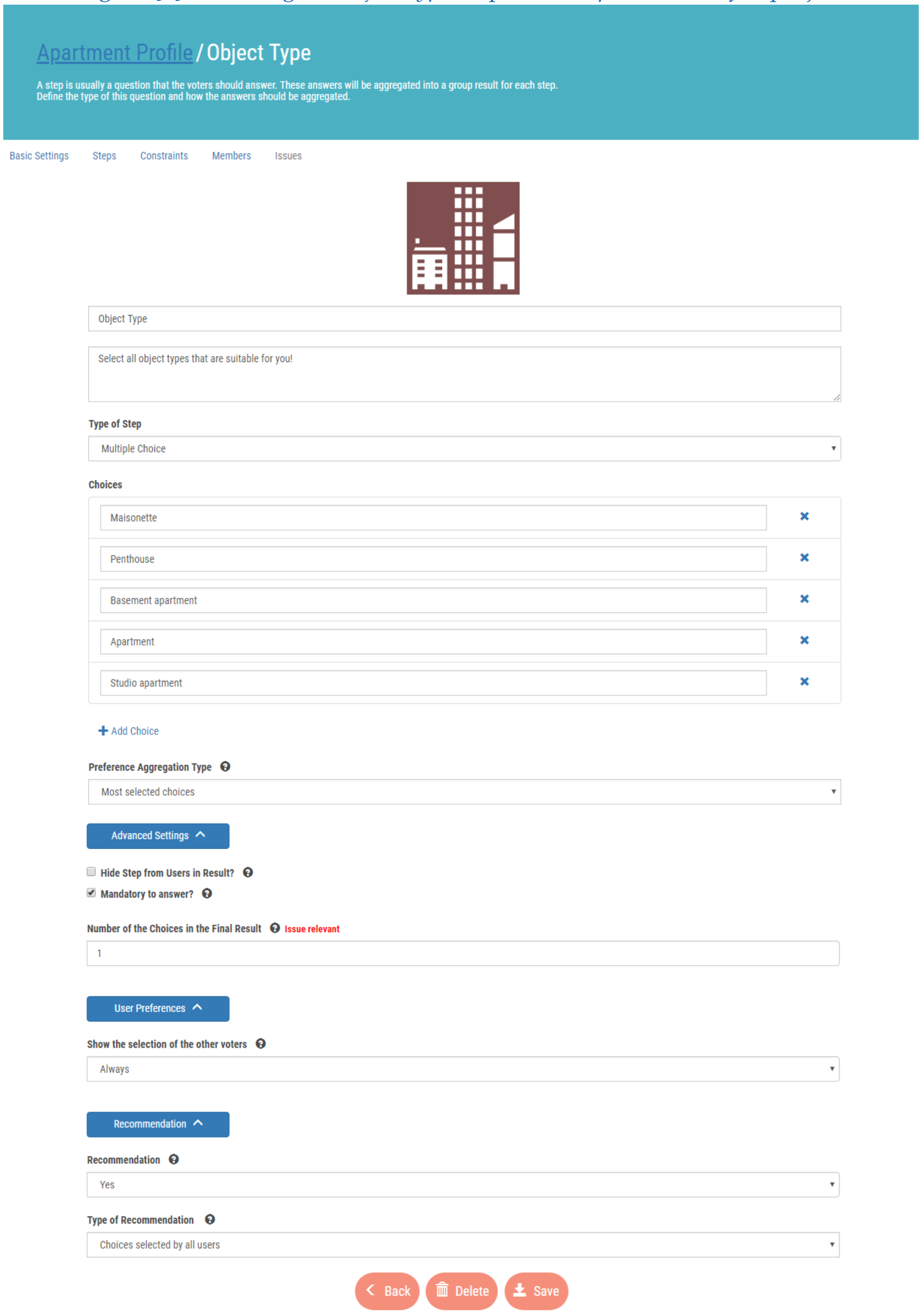
Basic Settings

Apartment Profile

Identify your apartment profile

After defining these basic project descriptions, the knowledge engineer creates the individual steps. A step typically corresponds to a component type of the configuration model. Figure 5.3 shows the full definition of the step that identifies the *object type*. Each step must have a *name* and a *step type*. Furthermore, except for information steps, each step must have a *preference aggregation type*. This type indicates how the individual votings are merged. Since the knowledge engineer wants users to be able to select

Figure 5.3: Defining the *object type* step in the *Apartment Profile* project



Apartment Profile / Object Type

A step is usually a question that the voters should answer. These answers will be aggregated into a group result for each step.
Define the type of this question and how the answers should be aggregated.

Object Type

Select all object types that are suitable for you!

Type of Step: Multiple Choice

Choices:
- Maisonette
- Penthouse
- Basement apartment
- Apartment
- Studio apartment

Preference Aggregation Type: Most selected choices

☐ Hide Step from Users in Result?
☑ Mandatory to answer?

Number of the Choices in the Final Result — Issue relevant: 1

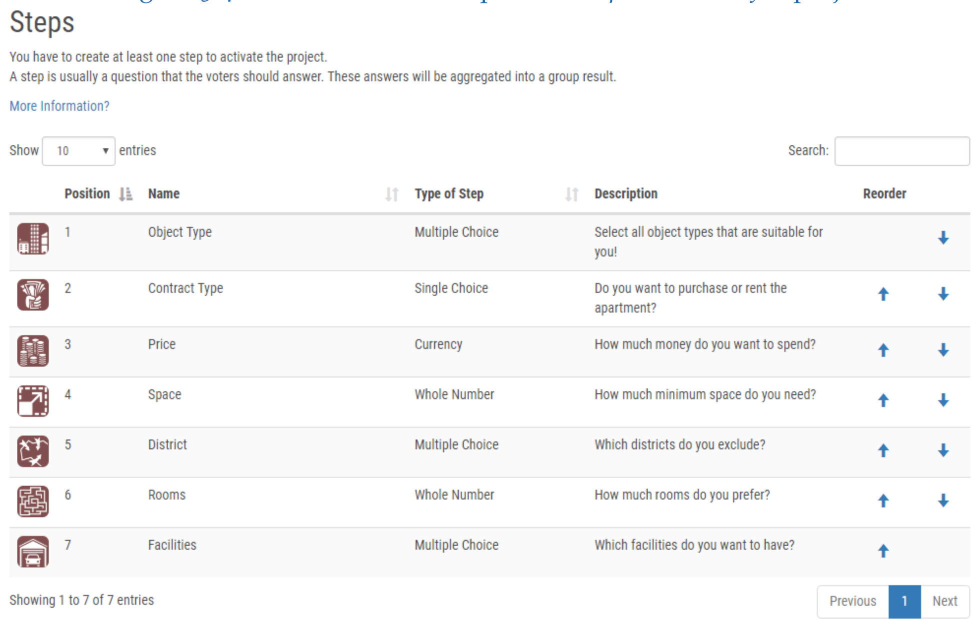
Show the selection of the other voters: Always

Recommendation: Yes

Type of Recommendation: Choices selected by all users more than one *object type*, he assigns the *multiple-choice* step type for this example step. He inserts all available *object types* as choices. Additionally, the knowledge engineer selects *"most selected choices"* as *preference aggregation type* and defines 1 as *number of choices in the final result*. This configuration should result in a single *object type* that has been chosen most frequently. If the result does not contain exactly the defined number of choices, an issue is created. Therefore, the *number of choices in the final result* is marked as *issue relevant* (see Figure 5.3). Furthermore, the visibility of *user preferences* is activated as *always*. Therefore, users can decide based on the decisions of other voters. The *recommendation* is also activated and it shows the *"choices selected by all voters"*. Displaying the user preferences or the recommendation is an additional influence for users to make their decisions.

The following sections do not explain all step configurations in detail, but the most important parts of them. Figure 5.4 provides an overview of all steps of this example project.

Figure 5.4: Overview of the steps in the *Apartment Profile* project



Steps

You have to create at least one step to activate the project.
A step is usually a question that the voters should answer. These answers will be aggregated into a group result.

More Information?

Show 10 entries

Search:

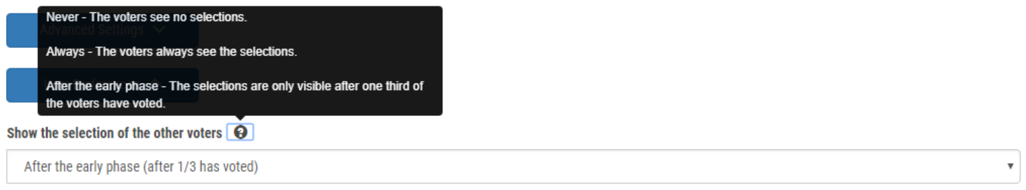
	Position	Name	Type of Step	Description	Reorder
	1	Object Type	Multiple Choice	Select all object types that are suitable for you!	↓
	2	Contract Type	Single Choice	Do you want to purchase or rent the apartment?	↑ ↓
	3	Price	Currency	How much money do you want to spend?	↑ ↓
	4	Space	Whole Number	How much minimum space do you need?	↑ ↓
	5	District	Multiple Choice	Which districts do you exclude?	↑ ↓
	6	Rooms	Whole Number	How much rooms do you prefer?	↑ ↓
	7	Facilities	Multiple Choice	Which facilities do you want to have?	↑

Showing 1 to 7 of 7 entries

Previous 1 Next

The *contract type* step has to find out, if users want to buy or rent an apartment. Therefore, the step is a *single-choice* type containing two choices. The aggregation function of this step is the *"most selected choice"*. In order to avoid biases in this decision and to support reaching consensus, the user preferences are only displayed after the early phase. In *KonfiGr*, the *early phase* is over after a third of users have voted. Figure 5.5 not only shows how to choose the visibility of user preferences, but also an helpful information box as it can be found throughout the whole application.

Figure 5.5: Selection of the visibility of the user preferences



Never - The voters see no selections.

Always - The voters always see the selections.

After the early phase - The selections are only visible after one third of the voters have voted.

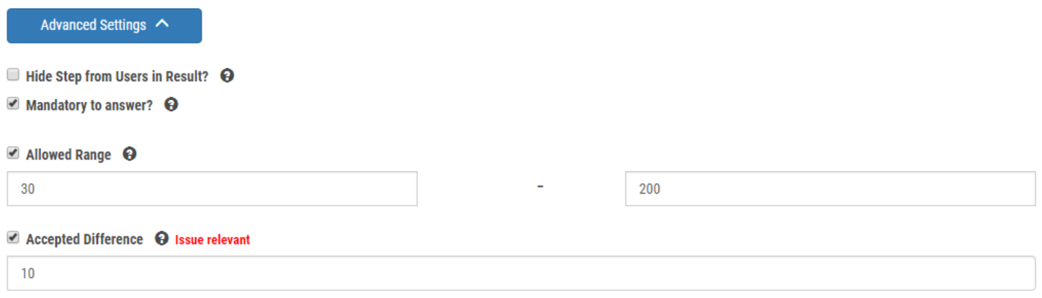
Show the selection of the other voters

After the early phase (after 1/3 has voted)

The *budget* step is defined as the *currency* type. The individual values are summarized to the average value.

The *space* step is defined as a *whole number*, and the system only allows entries between 30 and 200. Additionally, the entries can only differ by 10 or less, as shown in Figure 5.6. This *accepted difference* is *issue relevant*. Therefore,

Figure 5.6: Advanced settings of the *space* step



☐ Hide Step from Users in Result?

☑ Mandatory to answer?

☑ Allowed Range

30 - 200

☑ Accepted Difference Issue relevant

10

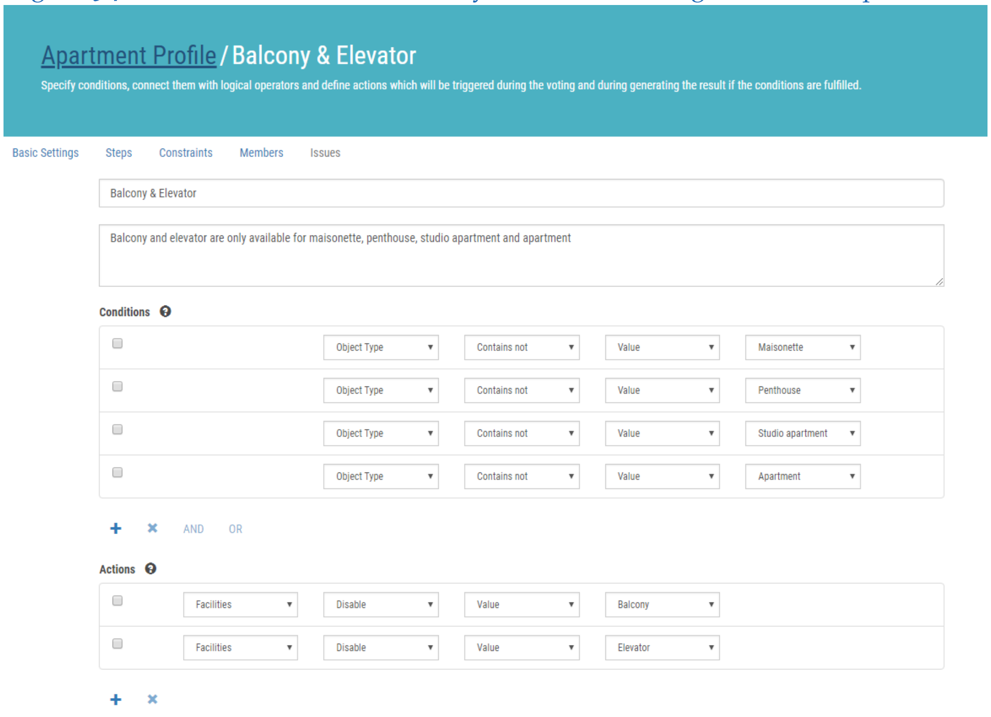
if the difference is too high, an issue is created. Moreover, the result of this step is formed by the *minimum* value of all individual values.

A different aggregation variant was used for the *district* step. *"Choices selected by no user"* was assigned, because users should choose which districts they want to exclude, rather than their preferred districts.

The *room* step is defined as a *whole number* from *1* to *10* and an *accepted difference* of *1*. The *aggregation type* of this step is the *median*.

After all steps have been defined, the constraints can be created. One of these constraints limits the use of the *balcony* and the *elevator*. If the user has selected *apartment types* that are only located on the ground floor, the *balcony*

Figure 5.7: Constraint to disable balcony and elevator for ground floor apartments


Apartment Profile / Balcony & Elevator

Specify conditions, connect them with logical operators and define actions which will be triggered during the voting and during generating the result if the conditions are fulfilled.

Balcony & Elevator

Balcony and elevator are only available for maisonette, penthouse, studio apartment and apartment

Conditions

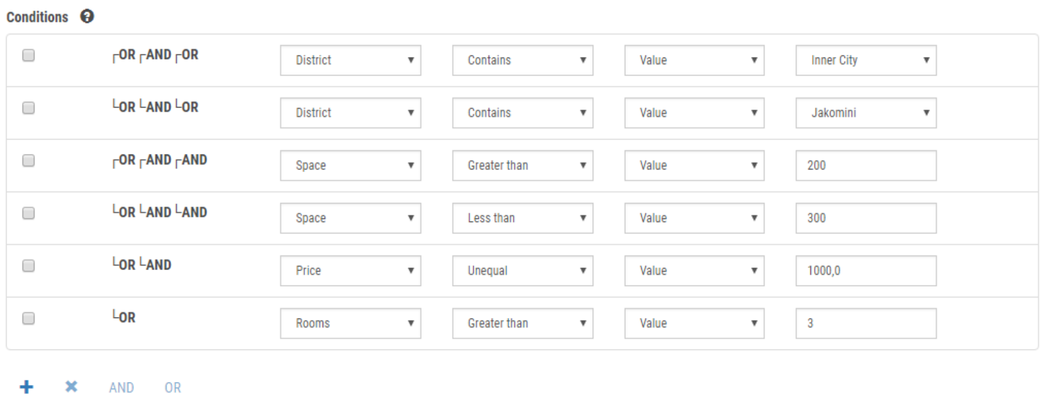
☐	Object Type	Contains not	Value	Maisonette
☐	Object Type	Contains not	Value	Penthouse
☐	Object Type	Contains not	Value	Studio apartment
☐	Object Type	Contains not	Value	Apartment

Actions

☐	Facilities	Disable	Value	Balcony
☐	Facilities	Disable	Value	Elevator

and *elevator* should not be available. Figure 5.7 shows the configuration of this constraint. The conditions are linked by default by logical *AND* links. However, complex logical groupings can be formed between conditions. Figure 5.8 shows such a complex example with several hierarchical levels.

Figure 5.8: Complex example of connecting conditions

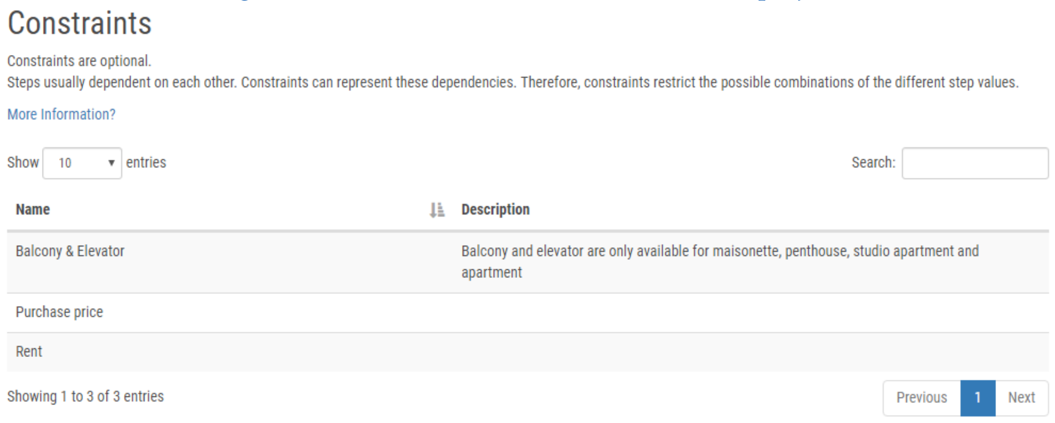


Conditions

☐	┌OR ┌AND ┌OR	District	Contains	Value	Inner City
☐	└OR └AND └OR	District	Contains	Value	Jakomini
☐	┌OR ┌AND ┌AND	Space	Greater than	Value	200
☐	└OR └AND └AND	Space	Less than	Value	300
☐	└OR └AND	Price	Unequal	Value	1000,0
☐	└OR	Rooms	Greater than	Value	3

In this use case, the knowledge engineer has created three *KonfiGr* constraints to map the two dependencies in the data model. The list of these constraints is shown in Figure 5.9. One of these constraints sets the *budget range* from *100,000* to *1,500,000*, if the *contract type* is *purchase*. Another

Figure 5.9: Overview of the constraints in the project



Constraints

Constraints are optional.
Steps usually dependent on each other. Constraints can represent these dependencies. Therefore, constraints restrict the possible combinations of the different step values.

More Information?

Show 10 entries   Search:

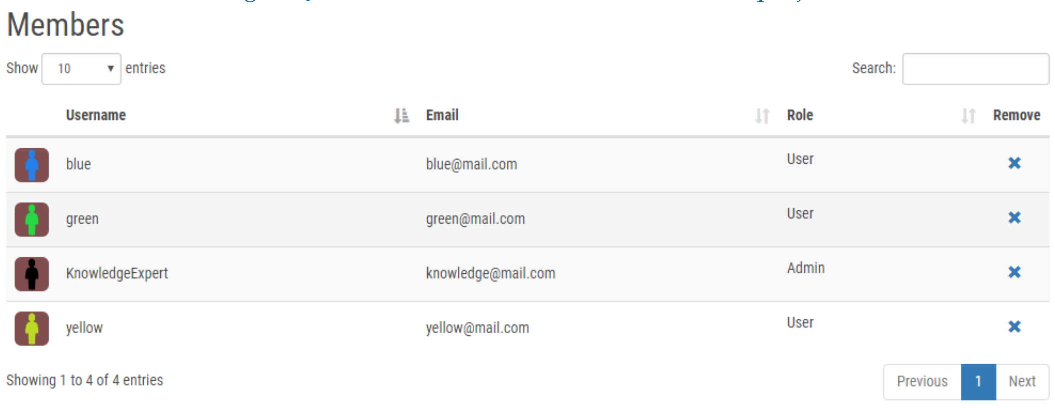
Name	Description
Balcony & Elevator	Balcony and elevator are only available for maisonette, penthouse, studio apartment and apartment
Purchase price	
Rent	

Showing 1 to 3 of 3 entries

constraint sets the *budget range* from *300* to *3,000*, if the *contract type* is *rent*. The last constraint disables the *balcony* and the *elevator*, if the *object type* does not contain a type that is located above the ground floor (see Figure 5.7). The constraints are checked during the voting of individual users and the generation of the group result.

Finally, the administrator adds the members who should be able to vote. Incidentally, in this example, all users have colors as names, as shown in Figure 5.10.

Figure 5.10: Overview of the members in the project



Members

Show 10 entries Search:

	Username	Email	Role	Remove
	blue	blue@mail.com	User	✖
	green	green@mail.com	User	✖
	KnowledgeExpert	knowledge@mail.com	Admin	✖
	yellow	yellow@mail.com	User	✖

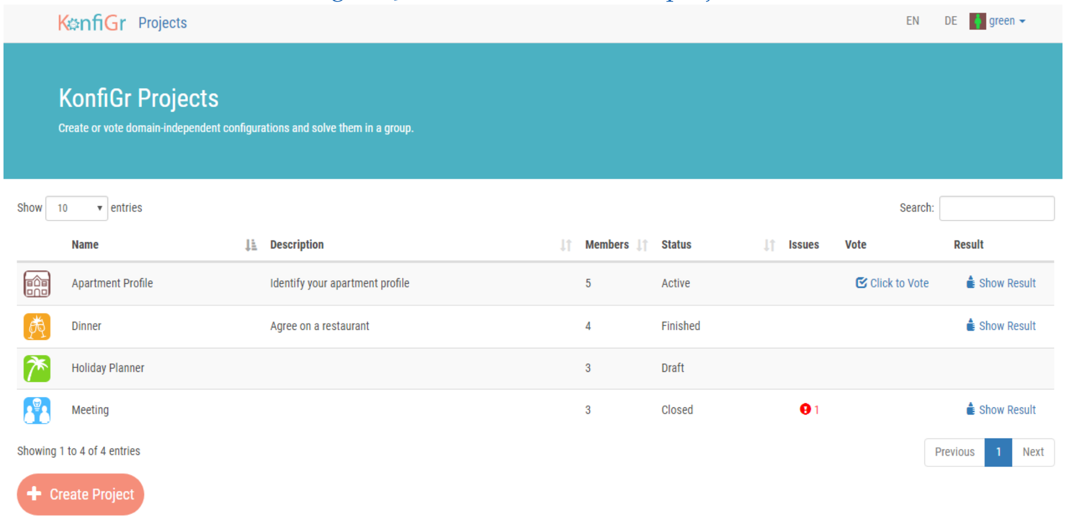
Showing 1 to 4 of 4 entries

Once the *knowledge engineering* process is completed, the knowledge engineer activates the project to enable *voting*.

Voting

The following section shows the *voting* process and how this process can be supported. Users usually start from the project overview (see Figure 5.11). The *apartment* project is *active*. Therefore, users can vote or view the temporary result. Projects with status *draft* are still in the *knowledge engineering* phase. *Closed* projects can no longer be voted. In contrast to *finished* projects, closed projects have no final result, because they have still issues that must be resolved.

Figure 5.11: Dashboard of the projects


KonfiGr Projects EN DE green

KonfiGr Projects

Create or vote domain-independent configurations and solve them in a group.

Show 10 entries Search:

Name	Description	Members	Status	Issues	Vote	Result
Apartment Profile	Identify your apartment profile	5	Active		Click to Vote	Show Result
Dinner	Agree on a restaurant	4	Finished			Show Result
Holiday Planner		3	Draft			
Meeting		3	Closed	1		Show Result

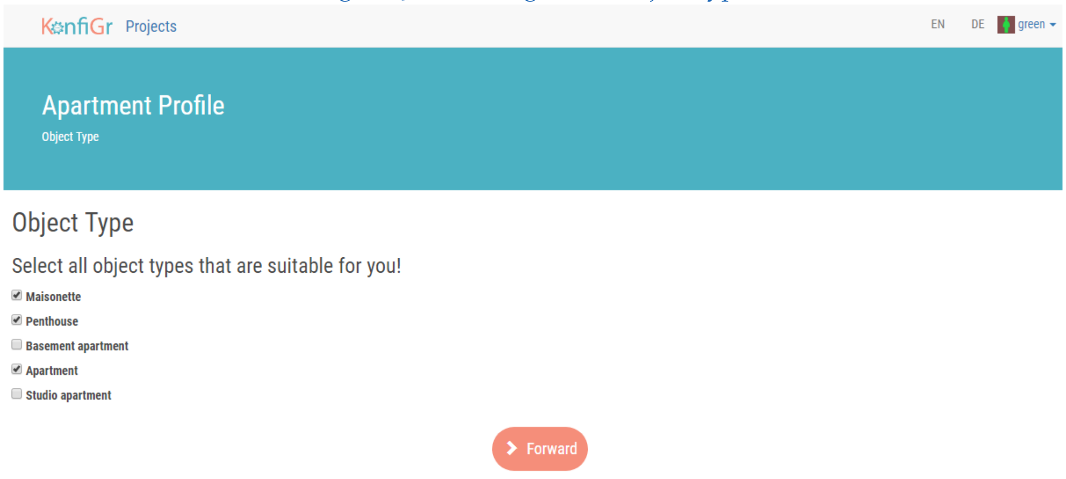
Showing 1 to 4 of 4 entries

Previous 1 Next

Create Project

The *voting* will be explained based on a few cutouts from different users. The explanation begins with the first step. The user *green* votes first, shown in Figure 5.12. The visibility of recommendation and user preferences is enabled but not visible, because there is no data yet.

Figure 5.12: Voting of the object type


KonfiGr Projects EN DE green

Apartment Profile

Object Type

Object Type

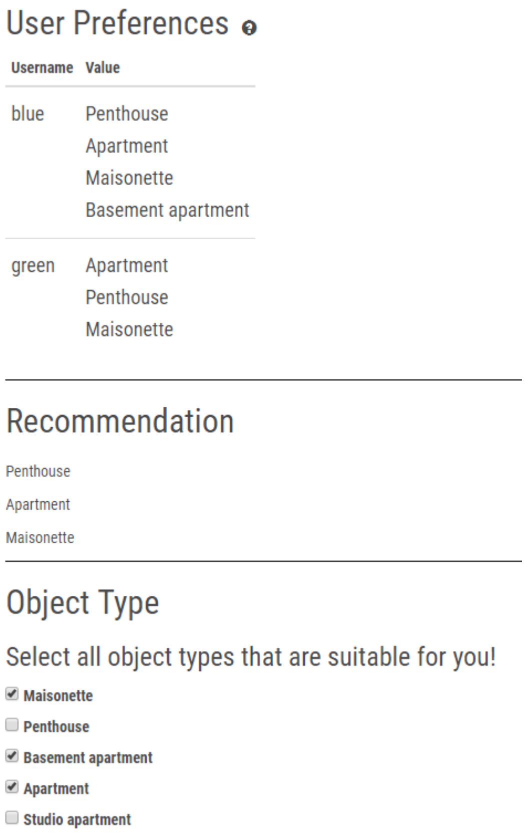
Select all object types that are suitable for you!

- [x] Maisonette
- [x] Penthouse
- [ ] Basement apartment
- [x] Apartment
- [ ] Studio apartment

Forward

After the users *green* and *blue* have submitted their votings, the user *yellow* votes. In Figure 5.13, the user *yellow* receives help by seeing the user preferences of other users and a recommendation generated from these preferences. These two aids can be activated independently per step.

Figure 5.13: Voting of the object type showing a recommendation and user preferences



User Preferences

Username	Value
blue	Penthouse Apartment Maisonette Basement apartment
green	Apartment Penthouse Maisonette

Recommendation

Penthouse
Apartment
Maisonette

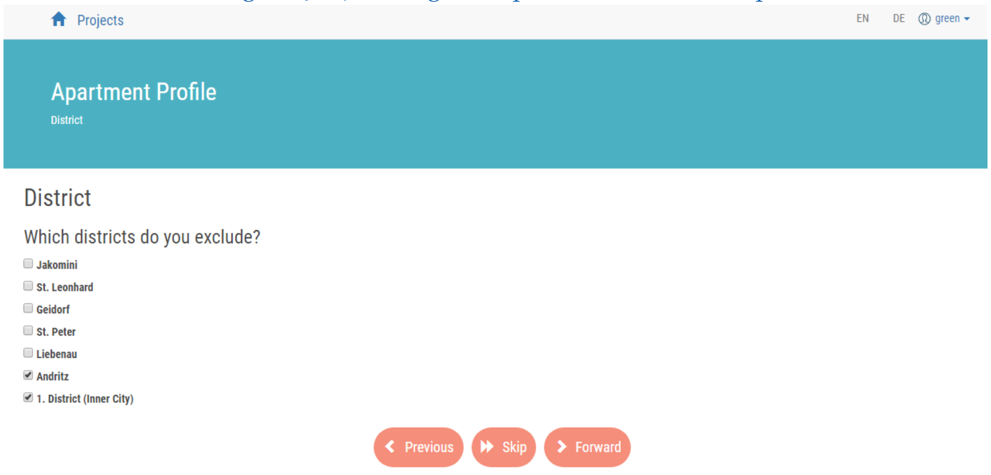
Object Type

Select all object types that are suitable for you!

- [✓] Maisonette
- [ ] Penthouse
- [✓] Basement apartment
- [✓] Apartment
- [ ] Studio apartment

In Figure 5.14, the user *green* votes the *district* step. The user can skip the step, because this step is not mandatory.

Figure 5.14: Voting example of the district step



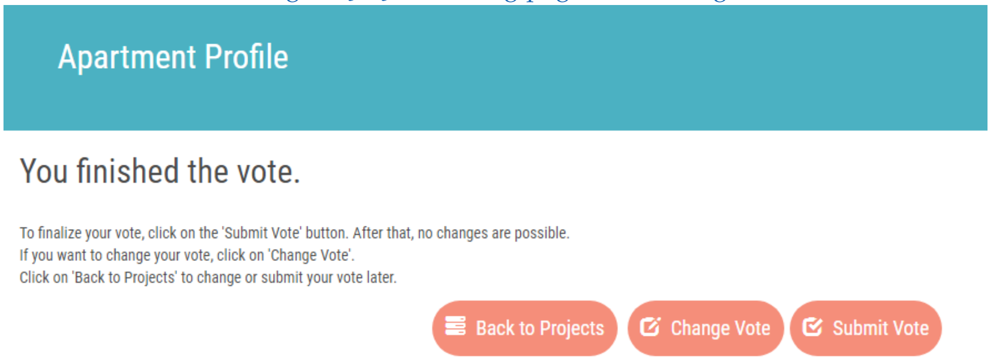
District

Which districts do you exclude?

- [ ] Jakomini
- [ ] St. Leonhard
- [ ] Geidorf
- [ ] St. Peter
- [ ] Liebenau
- [x] Andritz
- [x] 1. District (Inner City)

Completing the voting, the users may either change their answers or finally submit their voting (see Figure 5.15). After submitting the *voting*, users can only change their voting, if an issue occurs.

Figure 5.15: Finishing page after voting

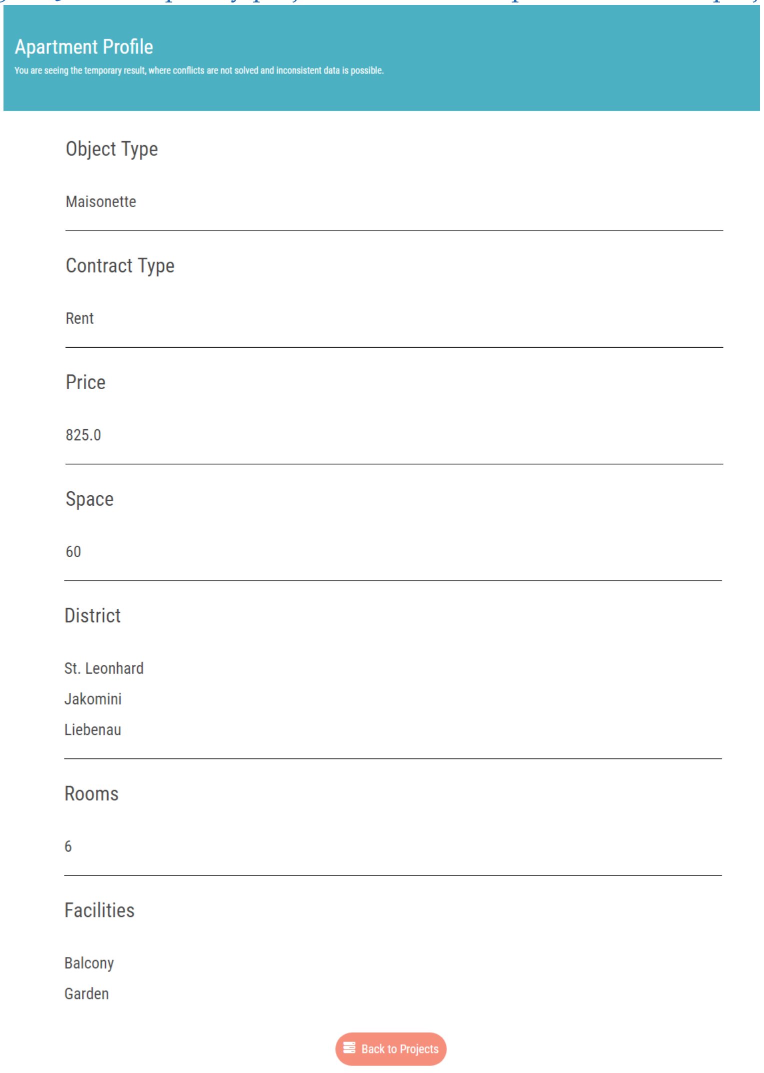


You finished the vote.

To finalize your vote, click on the 'Submit Vote' button. After that, no changes are possible.
If you want to change your vote, click on 'Change Vote'.
Click on 'Back to Projects' to change or submit your vote later.

Once a user has voted, a temporary result is displayed, shown in Figure 5.16. This result changes after each voting. Therefore, the temporary result does not indicate issues. Once a user has voted, the administrator can close the project. After that, the group result is checked for conflicts. An issue is created for each conflict. These issues have to be resolved. If no conflict is found, the project is set directly to *finished*.

Figure 5.16: Temporary project result of the Apartment Profile project



Apartment Profile

You are seeing the temporary result, where conflicts are not solved and inconsistent data is possible.

Object Type

Maisonette

Contract Type

Rent

Price

825.0

Space

60

District

St. Leonhard
Jakomini
Liebenau

Rooms

6

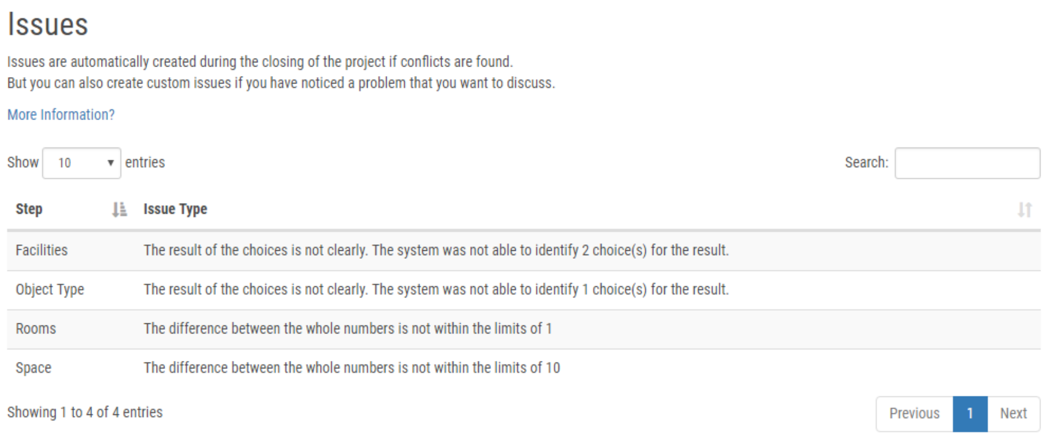
Facilities

Balcony
Garden

Issue Resolving

During the closing of the project, the system checks whether the selection of the choices aggregates to a clear group result and whether the values are not too far apart from each other. Details about the aggregation are explained in Section 4.6. Figure 5.17 shows the issues found in this example.

Figure 5.17: Overview of all founded issues of the Apartment Profile project



Issues

Issues are automatically created during the closing of the project if conflicts are found.
But you can also create custom issues if you have noticed a problem that you want to discuss.

More Information?

Show 10 entries Search:

Step	Issue Type
Facilities	The result of the choices is not clearly. The system was not able to identify 2 choice(s) for the result.
Object Type	The result of the choices is not clearly. The system was not able to identify 1 choice(s) for the result.
Rooms	The difference between the whole numbers is not within the limits of 1
Space	The difference between the whole numbers is not within the limits of 10

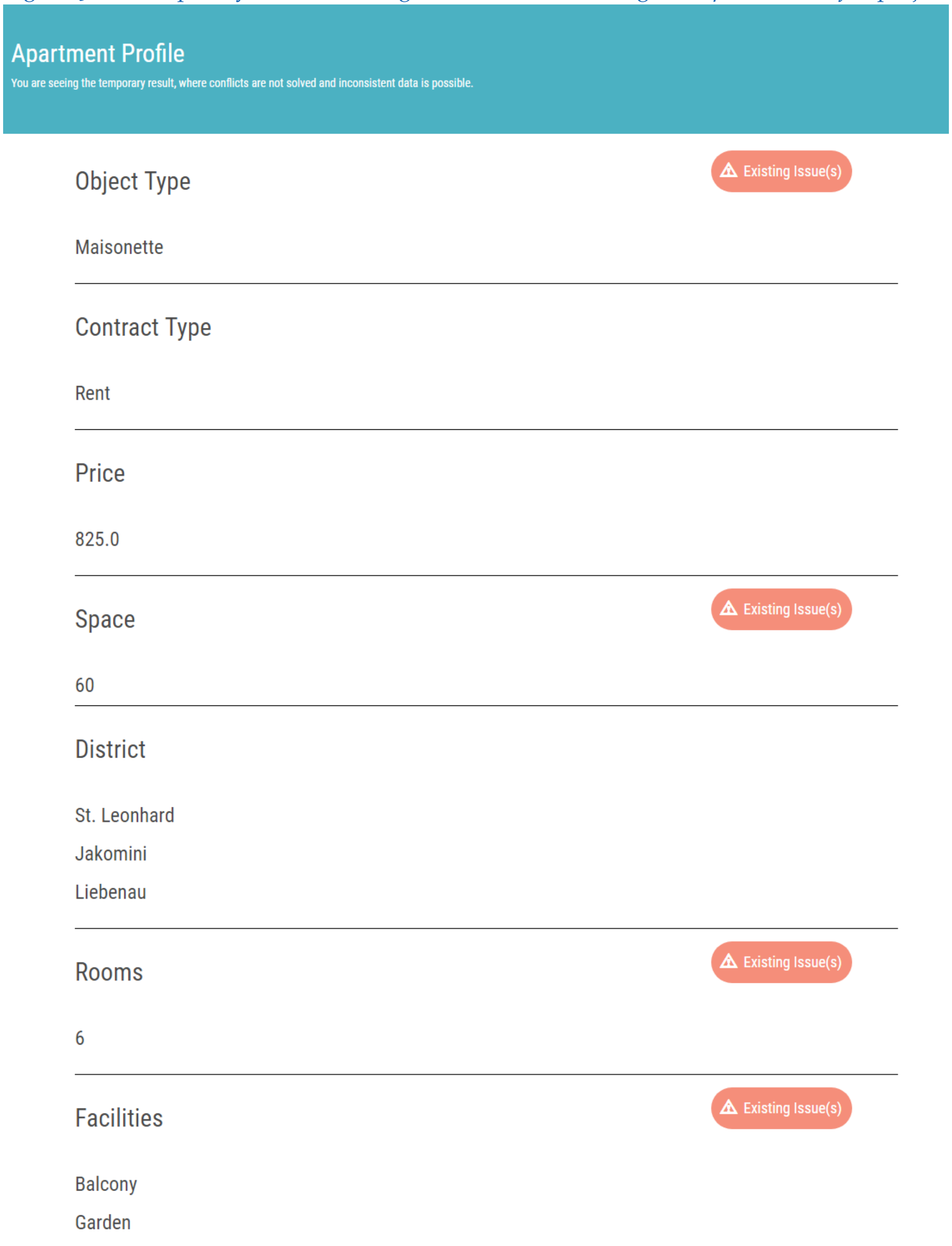
Showing 1 to 4 of 4 entries

The issues are visible to all members of the project. Open issues are displayed on the dashboard and on the result page (see Figure 5.18).

Depending on the step type, users have various options to resolve issues. The *single-choice* and *multiple-choice* steps can be resolved by *weighting* or by the administrator by selecting the final choices. *Value* (attribute) steps can be resolved by an *approximation*. Beside that, all steps can be resolved by the users changing their minds. In order to demonstrate the different variants of resolving issues, the *object type* step is resolved by *weighting*, the *space* step is resolved by *changing one's own mind*, and the *room* step is resolved by an *approximation*.

The three voters have chosen *apartment* and *maisonette* the same number of times. Therefore, the result is not clear. The user *green* starts the weighting,

Figure 5.18: Temporary result showing all issues after closing the *Apartment Profile* project


Apartment Profile

You are seeing the temporary result, where conflicts are not solved and inconsistent data is possible.

Object Type

Existing Issue(s)

Maisonette

Contract Type

Rent

Price

825.0

Space

Existing Issue(s)

60

District

St. Leonhard

Jakomini

Liebenau

Rooms

Existing Issue(s)

6

Facilities

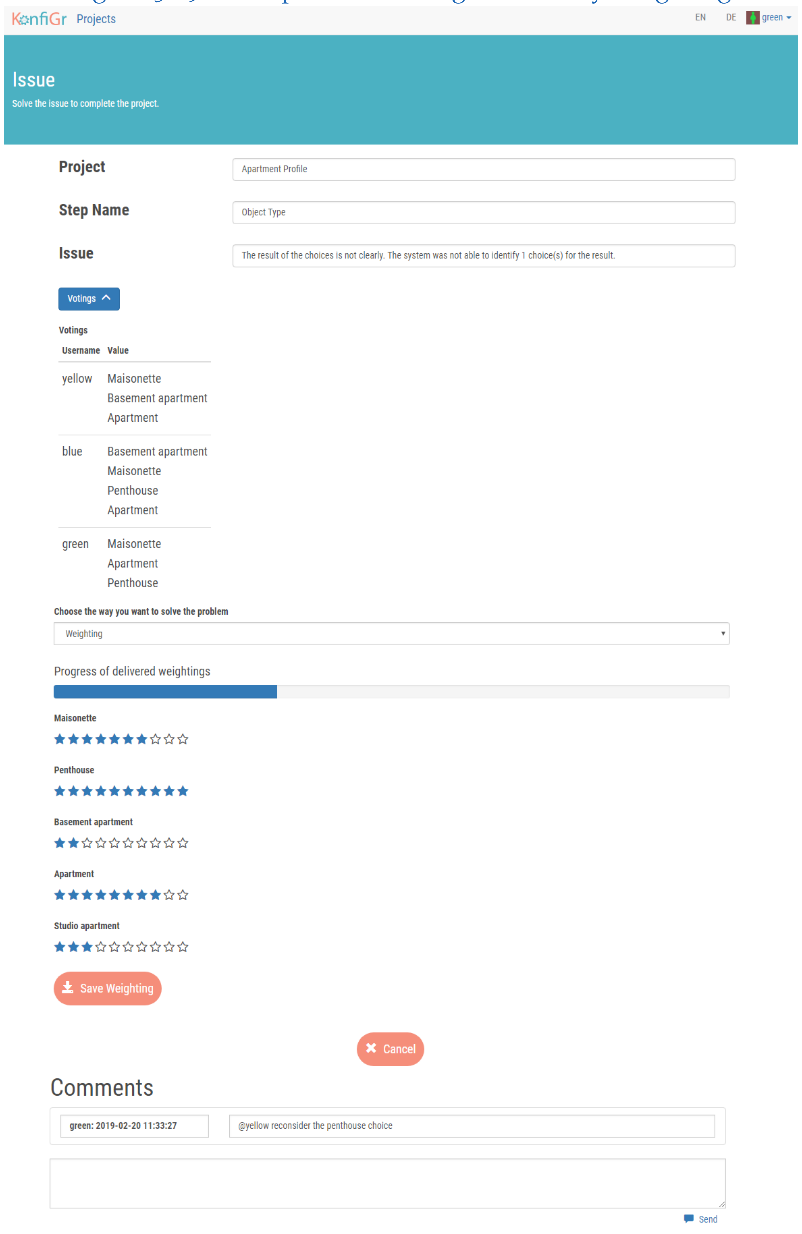
Existing Issue(s)

Balcony

Garden

shown in Figure 5.19. The user can see which step is affected, why an issue was created, and how the other users have voted. While weighting,

Figure 5.19: Example of resolving an issue by weighting



Issue

Solve the issue to complete the project.

Project	Apartment Profile
Step Name	Object Type
Issue	The result of the choices is not clearly. The system was not able to identify 1 choice(s) for the result.

Votings

Username	Value
yellow	Maisonette Basement apartment Apartment
blue	Basement apartment Maisonette Penthouse Apartment
green	Maisonette Apartment Penthouse

Choose the way you want to solve the problem

Weighting

Progress of delivered weightings

Maisonette

★★★★★★★☆☆☆

Penthouse

★★★★★★★★★★

Basement apartment

★★☆☆☆☆☆☆☆☆

Apartment

★★★★★★★★☆☆

Studio apartment

★★★☆☆☆☆☆☆☆

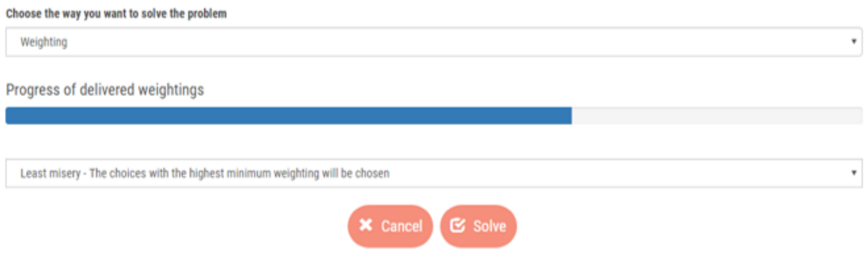
Comments

green: 2019-02-20 11:33:27	@yellow reconsider the penthouse choice

users rank the importance of each choice from 1 to 10. The *progress of delivered weightings* indicates how many users have already completed their weighting. Furthermore, users can add comments to issues to discuss their decisions.

In Figure 5.20, the administrator checks the progress of the delivered weightings. If the administrator considers the progress to be high enough, he can choose an algorithm to generate a solution. Therefore, the administrator can choose between the algorithms *least misery*, *average*, and *average without misery*.

Figure 5.20: Resolving an issue by choosing an algorithm to rate the weightings



Choose the way you want to solve the problem

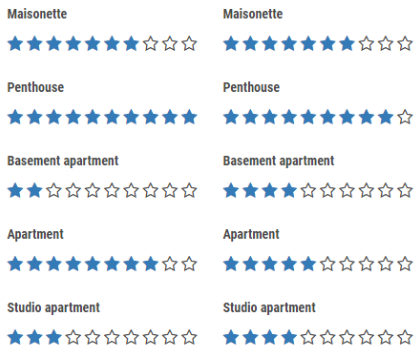
Weighting

Progress of delivered weightings

Least misery - The choices with the highest minimum weighting will be chosen

The system used the *weightings* in Figure 5.21 to identify *penthouse* as the final result for this step. Resolving this first issue, resolved a second issue

Figure 5.21: Weighting example of the *object type* step in the *Apartment Profile* project

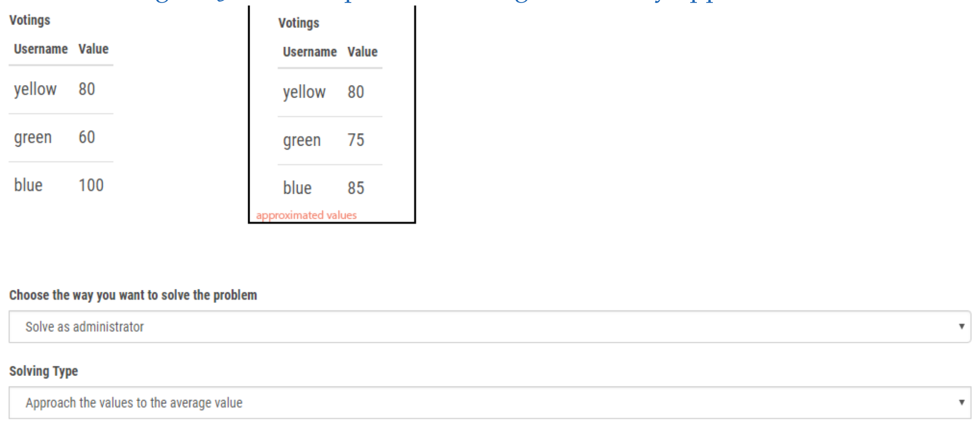


Maisonette	Maisonette
★★★★★★★☆☆☆	★★★★★★★☆☆☆
Penthouse	Penthouse
★★★★★★★★★★	★★★★★★★★★☆
Basement apartment	Basement apartment
★★☆☆☆☆☆☆☆☆	★★★★☆☆☆☆☆☆
Apartment	Apartment
★★★★★★★★☆☆	★★★★★☆☆☆☆☆
Studio apartment	Studio apartment
★★★☆☆☆☆☆☆☆	★★★★☆☆☆☆☆☆

too. The system could not select two *facilities*, because *elevator*, *balcony*, and *garden* were chosen the same number of times. However, as the *object type* has changed to *penthouse* and a constraint indicates that *penthouses* may not have a *garden*, there are only two possible facilities left. Therefore, this issue is also resolved.

The issue of the *space* step is resolved by the administrator by approximating the values to the *average value* (see Figure 5.22).

Figure 5.22: Example of resolving an issue by approximation



Votings

Username	Value
yellow	80
green	60
blue	100

Votings

Username	Value
yellow	80
green	75
blue	85

approximated values

Choose the way you want to solve the problem

Solve as administrator

Solving Type

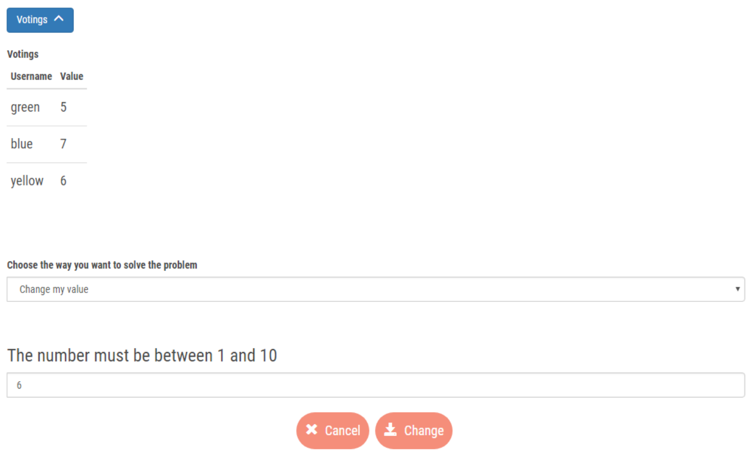
Approach the values to the average value

The *room* step is resolved by user *green* by changing his value for the number of *rooms* from 5 to 6, as shown in Figure 5.23.

If all issues are resolved, the administrator can finish the project. The final result is visible for all members of the project. Furthermore, administrators have the right to view the votings of all members.

The results of the remaining steps contain also interesting properties and are explained in the following sections. Two constraints define the influence of the *contract type* on the *price*. Figure 5.24 shows that the user *green* chose

Figure 5.23: Example of resolving an issue by changing the own value



Votings

Username	Value
green	5
blue	7
yellow	6

Choose the way you want to solve the problem

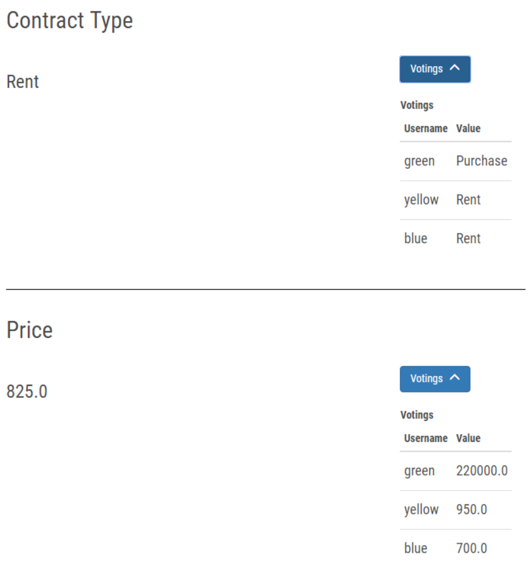
Change my value

The number must be between 1 and 10

6

purchase and therefore had to specify a *purchase price*. However, since *rent* is the final result, the voting of the user *green* has become irrelevant.

Figure 5.24: Step results of the *contract type* and *price* in the Apartment Profile project


Contract Type

Rent

Votings

Username	Value
green	Purchase
yellow	Rent
blue	Rent

Price

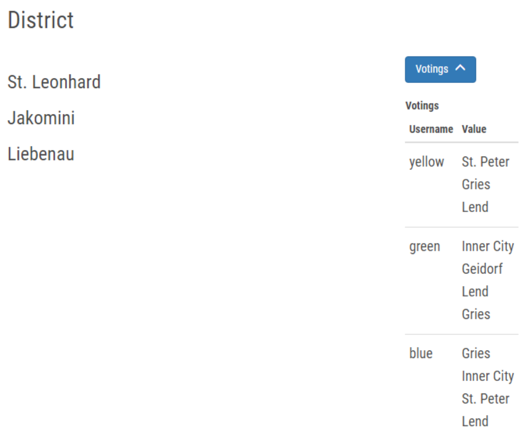
825.0

Votings

Username	Value
green	220000.0
yellow	950.0
blue	700.0

Furthermore, users had to choose *districts* they did not want to live in. The result is composed of the remaining *districts* (see Figure 5.25).

Figure 5.25: Result of the *district* step



District

St. Leonhard

Jakomini

Liebenau

Votings

Username	Value
yellow	St. Peter
	Gries
	Lend
green	Inner City
	Geidorf
	Lend
	Gries
blue	Gries
	Inner City
	St. Peter
	Lend

To sum up, this example has shown all important parts of the *KonfiGr* user interface in detail. The other use cases are only briefly described in the following sections.

5.2 Exam Definer

Exam Definer is a project where a group of tutors defines the content of an exam. Therefore, they have to make decisions on topics, exam questions, and administrative details. The project is defined by the following knowledge base:

- V = {topics (t), question type (q), free text questions (f), multiple-choice questions (m), multiple-choice type (mt), permission of documents (pd), duration (d), date of exam (da), number of rooms (r)}

- dom(t) = [{recommenders, configurators, decision psychology & AI, software engineering & AI, self-organizing systems}, mc],
  dom(q) = [{free text questions, free text questions & multiple-choice questions, multiple choice questions}, sc],
  dom(f) = [5 - 20, wn],
  dom(m) = [5 - 20, wn],
  dom(mt) = [{answers must be completely correct, there are partial points}, sc],
  dom(do) = [{yes, no}, sc],
  dom(d) = [30 - 120, wn],
  dom(da) = [27.05.2019 - 31.05.2019, da],
  dom(r) = [{HS i13, HS i8, HS i9, HS FSI 1, HS i11}, mc]

- C = {c1 : q $\neq$ free text questions $\wedge$ q $\neq$ free text questions & multiple-choice questions $\rightarrow$ skip f,
  c2: q $\neq$ free text questions & multiple-choice questions $\wedge$ q $\neq$ multiple-choice questions $\rightarrow$ skip m $\wedge$ skip mt}

- agg(t) = {$\cap$},
  agg(q) = {MAJ},
  agg(f) = {AVG},
  agg(m) = {AVG},
  agg(mt) = {MAJ},
  agg(do) = {MAJ},
  agg(d) = {Median},

$\text{agg}(\text{da}) = \{\text{MAJ}\}$,
$\text{agg}(\text{r}) = \{\cup\}$

- $\text{res}(\text{t}) = \{\infty\}$,
  $\text{res}(\text{q}) = \{1\}$,
  $\text{res}(\text{f}) = \{3\}$,
  $\text{res}(\text{m}) = \{3\}$,
  $\text{res}(\text{mt}) = \{1\}$,
  $\text{res}(\text{do}) = \{1\}$,
  $\text{res}(\text{d}) = \{\infty\}$,
  $\text{res}(\text{da}) = \{1\}$,
  $\text{res}(\text{r}) = \{\infty\}$

All topics selected by all members are part of the test. The number of topics is not limited. Moreover, the users need to decide whether to ask only free text questions, if they ask only multiple-choice questions, or if they want to ask a combination of this two question types. The most commonly chosen variant is the result. The constraints define whether the *free text question* step, the *multiple-choice question* step, and the *multiple-choice type* step are displayed, depending on the selected question type. The *free text question* step and the *multiple-choice question* step determine the number of questions. The individual values must not differ by more than 3. The *multiple-choice type* step, on the other hand, clarifies whether multiple-choice questions also contain partial points. Additionally, users must decide on the *permission of documents* step. This is a yes or no question where the most selected answer wins. The *duration* of the exam is clarified in minutes by a *whole number* step. The result is the median of all entered values. Users can choose between 5 days to set the date of the exam. The most frequently selected date is the final date of the exam. Finally, suitable rooms are identified by summarizing all rooms selected by all group members.

After the configuration, the tutors know which topics to use, how to ask questions, when the exam should take place, how long the exam will be, whether documents will be allowed, and which rooms are suitable.

5.3 Car Configurator

In this use case, a family configures their new car. The aim is to find suitable car models and to identify important buyer details. In practice, a car dealer can sort his available models by their characteristics and create a configuration model. An example of such a model is defined by the following knowledge base:

- V = {Manufacturer (m), Vehicle Type (v), Actuator Type (a), Car State (cs), Registration Date (r), Mileage (m), Budget (b), Color (c), Extras (e), Suitable Cars (sc)}

- dom(m) = [{BMW, Audi, VW}, mc],
  dom(v) = [{SUV, Coupe, Limousine}, sc],
  dom(a) = [{Gas, Hybrid, Diesel}, mc],
  dom(cs) = [{New Car, Used Car}, sc],
  dom(r) = [da]
  dom(m) = [wn],
  dom(b) = [cu],
  dom(c) = [{White, Black, Red, Blue, Silver}, sc],
  dom(e) = [{NAVI, All-wheel, Air-conditioning System, Multi-function Steering wheel, Xenon Lights, Trailer Hitch, Parking Assistant}, mc],
  dom(sc) = [{Audi Q2, Audi Q3, Audi Q5, Audi Q7, Audi A5 Coupe, Audi TT Coupe, Audi A3, Audi A4, Audi A6, Audi A8, BMW X1, BMW X3, BMW X5, BMW X7, BMW 6er, BMW 8er, BMW i8 Coupe, BMW 3er, BMW 5er, BMW 7er, VW T-Cross, VW Toureg, VW T-Roc, VW Tiguan, VW Scirocco, VW Arteon, VW Passat B8}, mc]

- C = {c1 : m $\neq$ Audi $\rightarrow$ sc $\neq$ Audi Q2 $\wedge$ sc $\neq$ Audi Q3 $\wedge$ sc $\neq$ Audi Q5 $\wedge$ sc $\neq$ Audi Q7 $\wedge$ sc $\neq$ Audi A5 Coupe $\wedge$ sc $\neq$ Audi TT Coupe $\wedge$ sc $\neq$ Audi A3 $\wedge$ sc $\neq$ Audi A4 $\wedge$ sc $\neq$ Audi A6 $\wedge$ sc $\neq$ Audi A8,
  c2: m $\neq$ BMW $\rightarrow$ sc $\neq$ BMW X1 $\wedge$ sc $\neq$ BMW X3 $\wedge$ sc $\neq$ BMW X5 $\wedge$ sc $\neq$ BMW X7 $\wedge$ sc $\neq$ BMW 6er $\wedge$ sc $\neq$ BMW 8er $\wedge$ sc $\neq$ BMW i8 Coupe $\wedge$ sc $\neq$ BMW 3er $\wedge$ sc $\neq$ BMW 5er $\wedge$ sc $\neq$ BMW 7er,
  c3: m $\neq$ VW $\rightarrow$ sc $\neq$ VW T-Cross $\wedge$ sc $\neq$ VW Touareg $\wedge$ sc $\neq$ VW T-Roc $\wedge$ sc $\neq$ VW Scirocco $\wedge$ sc $\neq$ VW Arteon $\wedge$ sc $\neq$ VW Passat B8,
  c4: v $\neq$ Coupe $\rightarrow$ sc $\neq$ Audi A5 Coupe $\wedge$ sc $\neq$ Audi TT Coupe $\wedge$ sc

$\neq$ BMW 6er $\land$ sc $\neq$ BMW 8er $\land$ sc $\neq$ BMW i8 Coupe $\land$ sc $\neq$ VW Scirocco,
c5: v $\neq$ Limousine $\rightarrow$ sc $\neq$ Audi A3 $\land$ sc $\neq$ Audi A4 $\land$ sc $\neq$ Audi A6 $\land$ sc $\neq$ Audi A8 $\land$ sc $\neq$ BMW 3er $\land$ sc $\neq$ BMW 5er $\land$ sc $\neq$ BMW 7er $\land$ sc $\neq$ VW Arteon $\land$ sc $\neq$ VW Passat B8,
c6: v $\neq$ SUV $\rightarrow$ sc $\neq$ Audi Q2 $\land$ sc $\neq$ Audi Q3 $\land$ sc $\neq$ Audi Q5 $\land$ sc $\neq$ Audi Q7 $\land$ sc $\neq$ BMW X1 $\land$ sc $\neq$ BMW X3 $\land$ sc $\neq$ BMW X7 $\land$ sc $\neq$ VW T-Cross $\land$ sc $\neq$ VW Touareg $\land$ sc $\neq$ VW T-Roc $\land$ sc $\neq$ VW Tiguan,
c7: a $\neq$ Gas $\rightarrow$ sc $\neq$ Audi TT Coupe,
c8: a $\neq$ Gas $\land$ a $\neq$ Diesel $\rightarrow$ sc $\neq$ Audi Q2 $\land$ sc $\neq$ Audi Q3 $\land$ sc $\neq$ Audi A3 $\land$ sc $\neq$ BMW X3 $\land$ sc $\neq$ BMW X7 $\land$ sc $\neq$ BMW 6er $\land$ sc $\neq$ BMW 8er $\land$ sc $\neq$ VW T-Cross $\land$ sc $\neq$ VW T-Roc $\land$ sc $\neq$ VW Scirocco $\land$ sc $\neq$ VW Arteon,
c9 : a $\neq$ Hybrid $\rightarrow$ cs $\neq$ Audi Q7 $\land$ sc $\neq$ Audi A8 $\land$ sc $\neq$ BMW i8 Coupe,
c10: a $\neq$ Hybrid $\land$ a $\neq$ Diesel $\rightarrow$ sc $\neq$ Audi Q5 $\land$ sc $\neq$ Audi A4 $\land$ sc $\neq$ Audi A6,
c11: cs $\neq$ Used Car $\rightarrow$ skip r $\land$ skip m,
c12: skip sc}

- agg(m) = {MAJ},
  agg(v) = {MAJ},
  agg(a) = {$\cap$},
  agg(cs) = {MAJ},
  agg(r) = {AVG},
  agg(m) = {MAX},
  agg(b) = {SUM},
  agg(c) = {MAJ},
  agg(e) = {$\cap$},
  agg(sc) = {\}

- res(m) = {2},
  res(v) = {1},
  res(a) = {1},
  res(cs) = {1},
  res(r) = {∞},

res(m) = {∞},
res(b) = {3.000},
res(c) = {1},
res(e) = {∞},
res(sc) = {∞}

In contrast to the *Exam Definer* example in Section 5.2, this project contains a *solution* step. This solution step *Suitable Cars* is not evaluated directly by users, but indirectly by the choices of voters in other steps. Therefore, the solution step is skipped by the constraint *c12*. Most other constraints set the result of the *Suitable Cars* step by the results of the *Manufacturer*, *Vehicle Type*, and *Actuator Type* steps. When selecting *used cars*, users must also vote on the *Registration Date* and *Mileage* steps. Moreover, users decide on budget, color, and extras.

5.4 Vacation Planning

In this example, a group of friends or a family wants to set a vacation destination. Therefore, a travel agency creates a configurator. This configurator is set up based on the travel destination information specified in the following knowledge base. The group decides not only on the destination, but also on important travel information such as budget, number of passengers, duration, and activities.

- V = {Category (ca), Continent (co), Budget (b), Fellow Passenger (fp), Vacation Period (vp), Accommodation Type (at), Activities (a), Destination (d)}

- dom(ca) = [{City, Beach, Nature, Skiing}, mc],
  dom(co) = [{Europe, North America, South America, Asia}, mc],
  dom(b) = [cu],
  dom(fp) = [0 - 5, wn],
  dom(vp) = [1 - 30, wn],
  dom(at) = [{Hotel, Apartment, Hostel, Guesthouse}, sc],
  dom(a) = [{Bicycle rental, Swimming, Museum, Concert, Theater, Sport Event, Wine and Dine, Bus Tour, Sauna}, mc],
  dom(d) = [{New York, Miami, Chicago, Buenos Aires, Sao Paulo, Rio de Janeiro, London, Vienna, Lissabon, Bangkok, Dubai, Tokio, Ko Samui, Saint Tropez, Bahamas, Grand Canyon, Galapagos Island, Dolomites, Plitvicer Lakes, Li River, Aspen, Beaver Creek, Las Lenas, Kitzbühel, St. Moritz, Muju}, mc]

- C = {c1: ca $\neq$ Beach $\rightarrow$ d $\neq$ Bahamas $\wedge$ d $\neq$ Ko Samui $\wedge$ d $\neq$ Saint Tropez,
  c2: ca $\neq$ Beach $\wedge$ ca $\neq$ City $\rightarrow$ d $\neq$ Miami $\wedge$ d $\neq$ Buenos Aires $\wedge$ d *neq* Sao Paulo $\wedge$ d $\neq$ Rio de Janeiro $\wedge$ d $\neq$ Lissabon,
  c3: ca $\neq$ Beach $\wedge$ ca $\neq$ City $\wedge$ ca $\neq$ Skiing $\rightarrow$ d $\neq$ Dubai,
  c4: ca $\neq$ City $\rightarrow$ d $\neq$ New York $\wedge$ d $\neq$ Chicago $\wedge$ d $\neq$ London $\wedge$ d $\neq$ Vienna $\wedge$ d $\neq$ Bangkok $\wedge$ d $\neq$ Tokio,
  c5: ca $\neq$ Nature $\wedge$ ca $\neq$ Skiing $\rightarrow$ d $\neq$ Aspen $\wedge$ d $\neq$ Beaver Creek $\wedge$ d $\neq$ Las Lenas $\wedge$ d $\neq$ Kitzbühel $\wedge$ d $\neq$ St. Moritz $\wedge$ d $\neq$ Muju,
  c6: co $\neq$ Asia $\rightarrow$ d $\neq$ Bangkok $\wedge$ d $\neq$ Muju $\wedge$ d $\neq$ Li River $\wedge$ d $\neq$ Ko

Samui $\wedge$ d $\neq$ Tokio $\wedge$ d $\neq$ Dubai,
c7: co $\neq$ North America $\rightarrow$ d $\neq$ New York $\wedge$ d $\neq$ Beaver Creek $\wedge$ d $\neq$ Aspen $\wedge$ d $\neq$ Grand Canyon $\wedge$ d $\neq$ Bahamas $\wedge$ d $\neq$ Chicago $\wedge$ d $\neq$ Miami,
c8: co $\neq$ South America $\rightarrow$ d $\neq$ Buenos Aires $\wedge$ d $\neq$ Las Lenas $\wedge$ d $\neq$ Galapagos Island $\wedge$ d $\neq$ Macho Picchu $\wedge$ d $\neq$ Rio de Janeiro $\wedge$ d $\neq$ Sao Paulo,
c9: co $\neq$ Europe $\rightarrow$ d $\neq$ London $\wedge$ d $\neq$ St. Moritz $\wedge$ d $\neq$ Kitzbühel $\wedge$ d $\neq$ Plitvicer Lakes $\wedge$ d $\neq$ Dolomites $\wedge$ d $\neq$ Saint Tropez $\wedge$ d $\neq$ Lissabon $\wedge$ d $\neq$ Vienna,
c10: skip d}

- agg(ca) = {MAJ},
  agg(co) = {$\cap$},
  agg(b) = {MIN},
  agg(fp) = {SUM},
  agg(vp) = {AVG},
  agg(at) = {MAJ},
  agg(a) = {$\cap$},
  agg(d) = {$\setminus$}

- res(ca) = {2]},
  res(co) = {1},
  res(b) = {∞},
  res(fp) = {∞},
  res(vp) = {3},
  res(at) = {1},
  res(a) = {∞},
  res(d) = {∞}

As in the *Car Configurator* project, there is also a solution step in this project. This *Destination* step depends on the results of the *Category* and *Continent* steps. Additionally, the group decides on important travel information. They agree on a budget, the number of passengers is summed up, the vacation period is arranged, the type of accommodation is planned, and the suitable activities are listed.

This chapter illustrated practical use cases in *KonfiGr*. Doing so, the user interface has been explained in detail. These examples have also shown that *KonfiGr* is fully functional and already useful, although it is just a first prototype of such an application. The next chapter summarizes the evaluation of an usability test including these use cases. Finally, Chapter 7 explains how to improve *KonfiGr*.

6 User Study

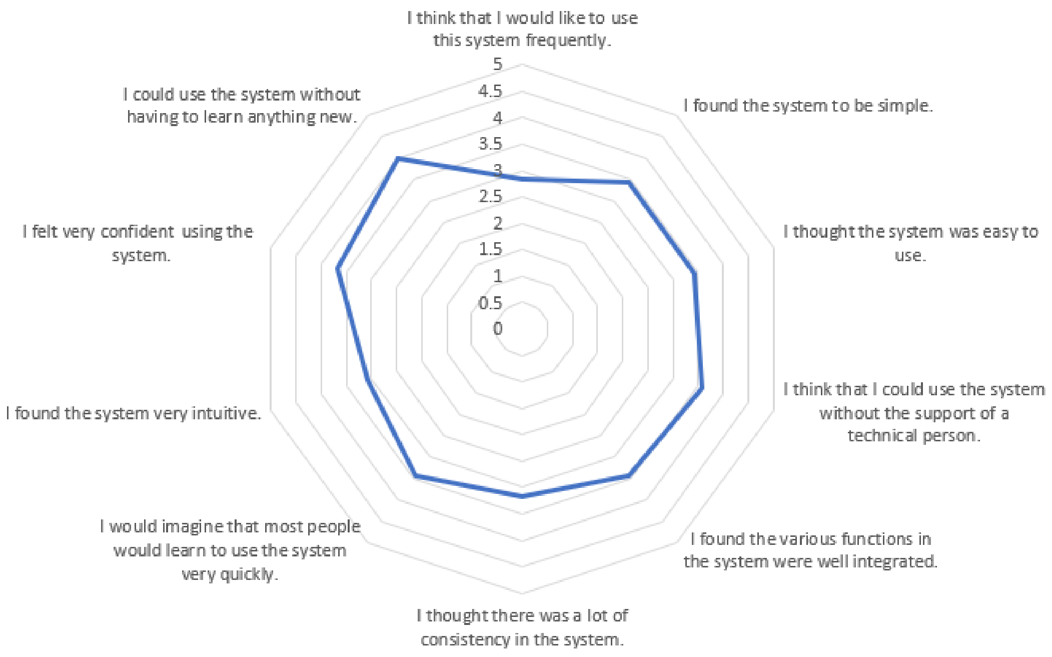
A usability test was performed with N=13 participants to check the current status of *KonfiGr* and to analyze potential improvements. The result of this study will be discussed in this chapter. The usability test was based on following projects, already mentioned in Chapter 5: *Apartment Profile*, *Exam Definer*, *Car Configurator*, and *Vacation Planning*. Each project was assigned to a team of 3 to 4 members. One member of each team was appointed administrator. The participants were all technically skilled, but lacked detailed knowledge of group-based configuration. Since the knowledge engineering process is very extensive, only voting and resolving issues has been tested. Therefore, each team received access to a predefined project. Furthermore, a questionnaire was created, which at the end of the usability test each user had to fill out. This questionnaire consisted of *10 System Usability Scale (SUS)* questions that assess the usability of a system and 6 general questions. The usability test procedure was defined as follows:

- The administrator activates the project
- All members of the group vote
- The administrator closes the project
- All issues found are resolved by the team
- The administrator finishes the project
- All members of the group fill out the questionnaire

System Usability Scale Questions

Figure 6.1 shows the evaluation of the SUS questions. The result of this evaluation is on average positive. This outcome suggests that the system is already usable, but there is room for improvements.

Figure 6.1: SUS evaluation: average ratings, N=13 (1 = I do not agree, ..., 5 = I totally agree)



General Questions

Potential improvements were identified on the basis of general questions. The outcome of these 6 general questions will be presented in this section. Therefore, the overall experience of the participants is summarized per question.

How would you describe your overall experience with the application?

The reviews show that the design is appealing and that the voting is fairly easy and understandable for most participants. However, there are some ambiguities in the user interface that need to be solved. Some participants were overall very satisfied with the system, but others had difficulties with it. Especially, generating group results by merging all the individual votings was incomprehensible to most of the users. Graphical representations could make the merging process more understandable. This will be an important but very extensive task in future work.

What did you like the most about using this application?

The fact that the system has been kept as simple as possible despite its extensive functionality is well received by users. The overall idea behind the system was also perceived as reasonable. Additionally, the e-mail notifications were mentioned as very helpful.

What did you like the least?

One user was surprised to see no group recommendation. When someone votes first, there is still no data to generate a recommendation. This should be communicated to the user to avoid ambiguity.

Some users criticized the knowledge base of the project. In this way, the impression of the system depends strongly on the respective project. Of course, the system itself can only have a limited impact on how knowledge engineers create their projects. Therefore, users should be able to provide feedback on the configuration model to the administrator.

Resolving issues is still a bit confusing and should be more intuitive. For example, to suggest solutions would be helpful in making this process quick and easy.

What, if anything, surprised you about the experience?

The participants were surprised by the simplicity of the system. They expected more complex options, but not all users had to resolve all types of issues and therefore did not see the full functions of the system. For example, one user missed weightings, though a weighting mechanism is present in resolving issues. However, the concept of weightings should already be part of the voting process. This in turn would probably prevent many issues in advance. Allover, this improvement is one of the main tasks for future work.

What, if anything, caused you frustration?

Some users were not satisfied with their result. This criticism is in turn mainly due to the specific content of the knowledge base.

Unfortunately, working in teams asynchronously can lead to long waiting times, which are very frustrating. Adding timers or reminders can shorten these waiting times.

How would you improve this application?

The participants want the system to be more interactive. They also want to understand how the results and the recommendations are generated in a user-optimized way. Furthermore, the result page should show details about resolved issues. Overall, the usability and understandability of the entire system should be further improved.

The whole process of resolving issues has to be simplified. There may already be too many options. Limiting these possibilities could vastly simplify the process.

Another great idea for improving the system is adding choices during the voting by the users themselves. However, it should be noted that members who voted earlier did not have this choice.

This chapter discussed the reviews of the participants who took part in the usability test. Areas that need to be improved have been identified by this test. Therefore, the next chapter summarizes the current status of the application and provides suggestions for improvements. By and large, the participants were already very satisfied with *KonfiGr*, even though it is a first prototype.

7 Conclusion

This chapter gives an overview of the current version of *KonfiGr*. It describes which areas have already been developed, but also which areas should be improved in future work. Some potential improvements have already been mentioned in the previous chapters. These and other improvements will be summarized in this chapter.

7.1 Actual Status of KonfiGr

KonfiGr is a prototype of an intelligent user interface for managing *group-based knowledge-based domain-independent configurations*. The topic of *configuration* is a very far-reaching one. The implementation of all possible techniques and variants of this topic provides therefore enough work for several projects. The task of this master thesis was to develop a first prototype, which contains just the most important functions.

As mentioned in Chapter 4 and Chapter 5, *KonfiGr* is, among other things, a *domain-independent* application. The prototype is therefore not limited to specific products or services and can be used in any domain. In order to be able to define all products and services, there are *steps* and *constraints* in *KonfiGr*. To represent the different component types, there are many step types. *Constraints* between *steps* allow to define complex products or services. Moreover, *KonfiGr* supports the most common aggregation functions for generating group results. The same aggregation functions are used when generating group recommendations. In addition to recommendations, user preferences can be displayed to influence the votings of project members.

Furthermore, at each step it is possible to decide at which point a conflict occurs. Thus, opinions that are extremely different are recognized and have to be resolved. *KonfiGr* already offers several variants to do so. *KonfiGr* proves that the efficient configuration of complex products or services within a group of users is possible.

7.2 Future Works

KonfiGr is a complete application that is already fully functional. Moreover, the usability test has proven that the system is already useful. However, there are still many areas that could be improved. Suggestions for improvements are covered in this section.

Project

The way projects are managed in *KonfiGr* could be further improved. For example, there should be options to *copy* or *share* a project. Additionally, a *history* of project changes would be very helpful.

Member

In practice, users make their group decisions with the same set of certain people. Therefore, there should be a possibility to form *teams*. In these teams there should be a *point system* to give higher weights to the responses of group members, who have made compromises in the past. This idea is already mentioned by Felfernig, Boratto, et al., 2018. Furthermore, adding *notifications* in the navigation bar area can increase the interactivity and usability of *KonfiGr*. Another useful feature would be a *chat* to better discuss decisions. Typically, there is much to discuss in order to reach consensus in a group decision. In any case, more attention needs to be paid to this area in the future.

Step

KonfiGr already supports many different step types. The knowledge engineer can choose between *single-choice* and *multiple-choice* types to form a list of alternatives. Issues of such steps can be resolved by weighting. This *weighting* should also be a separate step type. Furthermore, the *date* type should provide a *calendar* for picking dates. Additionally, the administrator should be able to limit the selection not only to one period, but to multiple periods or several days. Users should then be able to agree on a single day or a period of days like in Doodle (see Chapter 2).

Constraints

Currently, it is only possible to create constraints between two steps. In the future, it should be possible to create conditions including choices which affects other choices of the same step. Moreover, there are many different possibilities to built up knowledge bases. In future work, other variants should be considered.

Usability

The usability of *KonfiGr* is a main factor for success and must be further improved. In particular, graphic elements such as diagrams could communicate information to users in a better way. The usability test has shown that it is important for users to understand the results. The users should also know how their choices affect the result. In the best case, this should be communicated to the users in a simply way and without much effort for the users.

Avoiding Biases

In Chapter 2, different types of biases in group decisions have been listed. Some of them can already be prevented, while others still need additional functions to avoid them. For example, the *serial position effect* could be avoided by randomizing the order of choices. The *polarization effect* can be avoided by adding a possibility to lead a discussion during the voting process. In short, these biases should be taken into account in future work.

Resolving Issues

KonfiGr supports some approaches to resolve issues, but not every approach is available for every step type. The ability of an administrator to *select a final result* should be supported in all step types. Additionally, there should be more *algorithms* to automatically resolve issues. Moreover, the system should provide *recommendations for resolving issues*. Furthermore, resolved conflicts should be visible for users. Once *teams* exist, issues should be resolved according to compromises made in the past.

Currently, resolving an issue of a step sometimes changes the selection of a user. Thus, the results of related steps of this user is no longer usable, if this result is no longer possible due to constraints. In the future, it should be possible for users to have a vote on these steps again.

Recommender Systems for Knowledge Engineering

Recommender systems can be used to improve *knowledge engineering*. The *knowledge engineering bottleneck* is a major problem in *configuration*. Felfernig, Reiterer, et al., 2013 shows how recommenders can support *knowledge engineering* to avoid this problem. Currently, it is difficult for non-technicians to design complicated configurations. Clever recommendations could facilitate this problem.

In regard to all these suggestions, one can conclude that *KonfiGr* can be extended to many areas, because *group-based configuration* is a far-reaching and growing topic, which always offers new insights and thus new possibilities for improvements. The goal is to improve the application and to receive feedback by means of further tests and to further develop the system based on this feedback. In any case, *group-based configuration* is a topic that could become increasingly important in the near future. Therefore, research on this topic has to be continued.

Bibliography

Adomavicius, Gediminas and Alexander Tuzhilin (2005). "Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions." In: *IEEE Transactions on Knowledge & Data Engineering* 6, pp. 734–749 (cit. on p. 4).

Aggarwal, Charu C et al. (2016). *Recommender systems*. Springer (cit. on p. 4).

Ardissono, Liliana et al. (2003). "A framework for the development of personalized, distributed web-based configuration systems." In: *Ai Magazine* 24.3, pp. 93–93 (cit. on p. 4).

Burke, Robin (2000). "Knowledge-based recommender systems." In: *Encyclopedia of library and information systems* 69.Supplement 32, pp. 175–186 (cit. on p. 5).

Burke, Robin (2002). "Hybrid recommender systems: Survey and experiments." In: *User modeling and user-adapted interaction* 12.4, pp. 331–370 (cit. on p. 5).

Burke, Robin (2007). "Hybrid web recommender systems." In: *The adaptive web*. Springer, pp. 377–408 (cit. on p. 5).

Chen, Li and Pearl Pu (2007). "Hybrid critiquing-based recommender systems." In: *Proceedings of the 12th international conference on Intelligent user interfaces*. ACM, pp. 22–31 (cit. on p. 5).

Chen, Li and Pearl Pu (2012). "Critiquing-based recommenders: survey and emerging trends." In: *User Modeling and User-Adapted Interaction* 22.1-2, pp. 125–150 (cit. on p. 5).

Code First (2019). *PlanITpoker*. URL: https://www.planitpoker.com/ (visited on 01/29/2019) (cit. on p. 18).

Cöster, Rickard et al. (2002). "Enhancing web-based configuration with recommendations and cluster-based help." In: (cit. on p. 4).

Doodle AG (2019). *Doodle*. URL: https://doodle.com (visited on 01/24/2019) (cit. on p. 15).

Falkner, Andreas, Alexander Felfernig, and Albert Haag (2011). "Recommendation technologies for configurable products." In: *Ai Magazine* 32.3, pp. 99–108 (cit. on p. 4).

Felfernig, Alexander, Muesluem Atas, et al. (2016). "Towards group-based configuration." In: *International Workshop on Configuration 2016 (ConfWS'16)*, pp. 69–72 (cit. on pp. 1, 3, 11).

Felfernig, Alexander, Ludovico Boratto, et al. (2018). *Group Recommender Systems: An Introduction*. Springer (cit. on pp. 1, 3–8, 11–15, 18, 80).

Felfernig, Alexander and Robin Burke (2008). "Constraint-based recommender systems: technologies and research issues." In: *Proceedings of the 10th international conference on Electronic commerce*. ACM, p. 3 (cit. on pp. 4, 5).

Felfernig, Alexander, Gerhard Friedrich, Dietmar Jannach, et al. (2015). "Constraint-based recommender systems." In: *Recommender systems handbook*. Springer, pp. 161–190 (cit. on p. 5).

Felfernig, Alexander, Gerhard Friedrich, Monika Schubert, et al. (2009). "Plausible repairs for inconsistent requirements." In: *Twenty-First International Joint Conference on Artificial Intelligence* (cit. on p. 4).

Felfernig, Alexander, Lothar Hotz, et al. (2014). *Knowledge-based configuration: From research to business cases*. Newnes (cit. on pp. 1, 2, 10, 11, 35, 37, 38).

Felfernig, Alexander, Michael Jeran, et al. (2014). "Basic approaches in recommendation systems." In: *Recommendation Systems in Software Engineering*. Springer, pp. 15–37 (cit. on p. 4).

Felfernig, Alexander and Gerald Ninaus (2012). "Group recommendation algorithms for requirements prioritization." In: *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*. IEEE, pp. 59–62 (cit. on p. 20).

Felfernig, Alexander, Stefan Reiterer, et al. (2013). "Recommender Systems for Configuration Knowledge Engineering." In: *Configuration Workshop*. Citeseer, pp. 51–54 (cit. on pp. 8, 82).

Felfernig, Alexander, Johannes Spöcklberger, et al. (2018). "Configuring Release Plans." In: *Proceedings of the 20th Configuration Workshop*. CEUR-WS. org (cit. on p. 3).

Felfernig, Alexander, Martin Stettinger, et al. (2014). "Towards Open Configuration." In: *Configuration Workshop*. Citeseer, pp. 89–94 (cit. on pp. 1, 3, 8).

Felfernig, Alexander, Juha Tiihonen, et al. (2018). "20 th International Configuration Workshop." In: (cit. on p. 2).

Felfernig, Alexander, Christoph Zehentner, et al. (2011). "Group decision support for requirements negotiation." In: *International Conference on User Modeling, Adaptation, and Personalization*. Springer, pp. 105–116 (cit. on p. 3).

Graz University of Technology (2018). *Choiclaweb*. URL: `https://www.choiclaweb.com` (visited on 01/24/2019) (cit. on p. 14).

Jameson, Anthony, Stephan Baldes, and Thomas Kleinbauer (2004). "Two methods for enhancing mutual awareness in a group recommender system." In: *Proceedings of the working conference on Advanced visual interfaces*. ACM, pp. 447–449 (cit. on p. 3).

Jannach, Dietmar et al. (2010). *Recommender systems: an introduction*. Cambridge University Press (cit. on p. 4).

Koren, Yehuda and Robert Bell (2015). "Advances in collaborative filtering." In: *Recommender systems handbook*. Springer, pp. 77–118 (cit. on p. 5).

Mark Otto, Jacob Thornton (2019). *Bootstrap*. URL: `https://getbootstrap.com/docs/3.3/` (visited on 01/28/2019) (cit. on p. 31).

Mathew, Praveena, Bincy Kuriakose, and Vinayak Hegde (2016). "Book Recommendation System through content based and collaborative filtering method." In: *2016 International Conference on Data Mining and Advanced Computing (SAPIENCE)*. IEEE, pp. 47–52 (cit. on p. 5).

McCarthy, Kevin et al. (2006). "Group recommender systems: a critiquing based approach." In: *Proceedings of the 11th international conference on Intelligent user interfaces*. ACM, pp. 267–269 (cit. on pp. 5, 18).

Microsoft (2019). *Microsoft Forms*. URL: `https://forms.office.com` (visited on 01/24/2019) (cit. on p. 16).

Ninaus, Gerald et al. (2014). "INTELLIREQ: Intelligent Techniques for Software Requirements Engineering." In: *ECAI*, pp. 1161–1166 (cit. on p. 20).

Pivotal Software (2019). *Spring Boot*. URL: `https://spring.io/projects/spring-boot` (visited on 01/28/2019) (cit. on p. 30).

Reinecke, Katharina et al. (2013). "Doodle around the world: online scheduling behavior reflects cultural differences in time perception and group decision-making." In: *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, pp. 45–54 (cit. on p. 15).

Ricci, Francesco, Lior Rokach, and Bracha Shapira (2011). "Introduction to recommender systems handbook." In: *Recommender systems handbook*. Springer, pp. 1–35 (cit. on p. 4).

Sabin, Daniel and Rainer Weigel (1998). "Product configuration frameworks-a survey." In: *IEEE Intelligent Systems and their applications* 13.4, pp. 42–49 (cit. on p. 2).

Schafer, J Ben et al. (2007). "Collaborative filtering recommender systems." In: *The adaptive web*. Springer, pp. 291–324 (cit. on p. 5).

Stettinger, Martin (2014). "Choicla: Towards domain-independent decision support for groups of users." In: *Proceedings of the 8th ACM Conference on Recommender systems*. ACM, pp. 425–428 (cit. on p. 14).

Stettinger, Martin and Alexander Felfernig (2014). "Choicla: Intelligent decision support for groups of users in the context of personnel decisions." In: *Proceedings of the ACM RecSys' 2014 IntRS Workshop*, pp. 28–32 (cit. on p. 14).

Stettinger, Martin, Alexander Felfernig, Gerhard Leitner, and Stefan Reiterer (2015). "Counteracting anchoring effects in group decision making." In: *International Conference on User Modeling, Adaptation, and Personalization*. Springer, pp. 118–130 (cit. on p. 14).

Stettinger, Martin, Alexander Felfernig, Gerhard Leitner, Stefan Reiterer, and Michael Jeran (2015). "Counteracting serial position effects in the choicla group decision support environment." In: *Proceedings of the 20th international conference on intelligent user interfaces*. ACM, pp. 148–157 (cit. on p. 14).

Stumptner, Markus (1997). "An overview of knowledge-based configuration." In: *Ai Communications* 10.2, pp. 111–125 (cit. on pp. 2, 4).

The Thymeleaf Team (2019). *Thymeleaf*. URL: `https://www.thymeleaf.org` (visited on 01/28/2019) (cit. on p. 31).

Thüm, Thomas, Sebastian Krieter, and Ina Schaefer (2018). "Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators." In: *20 th International Configuration Workshop*, p. 1 (cit. on p. 22).

Tiihonen, Juha and Alexander Felfernig (2010). "Towards recommending configurable offerings." In: *International journal of mass customisation* 3.4, pp. 389–406 (cit. on p. 4).

Tiihonen, Juha and Alexander Felfernig (2017). “An introduction to personalization and mass customization.” In: *Journal of Intelligent Information Systems* 49.1, pp. 1–7 (cit. on p. 2).

Tran, Trang et al. (2016). “An extension of CHOICLA User Interfaces for Configurable Products.” In: *RS-BDA’16 Workshop* (cit. on p. 14).

Trewin, Shari (2000). “Knowledge-based recommender systems.” In: *Encyclopedia of library and information science* 69.Supplement 32, p. 180 (cit. on p. 5).

Tseng, Hwai-En, Chien-Chen Chang, and Shu-Hsuan Chang (2005). “Applying case-based reasoning for product configuration in mass customization environments.” In: *Expert Systems with Applications* 29.4, pp. 913–925 (cit. on p. 4).

Van Meteren, Robin and Maarten Van Someren (2000). “Using content-based filtering for recommendation.” In: *Proceedings of the Machine Learning in the New Information Age: MLnet/ECML2000 Workshop*, pp. 47–56 (cit. on p. 5).

Wang, Yue and Daniel Yiu-Wing Mo (2018). “The Effect of Default Options on Consumer Decisions in the Product Configuration Process.” In: *20 th International Configuration Workshop*, p. 31 (cit. on p. 38).

Zhang, Linda L (2014). “Product configuration: a review of the state-of-the-art and future research.” In: *International Journal of Production Research* 52.21, pp. 6381–6398 (cit. on p. 2).

Zhang, Linda L et al. (2014). “Open Configuration: a New Approach to Product Customization.” In: *Configuration Workshop*, pp. 75–79 (cit. on p. 8).