Johannes Grindal Ervum

Proposing Kitchen Designs

A Case-Based Reasoning Approach

Master's thesis in Computer Science
Supervisor: Pieter Jelle Toussaint, Sindre Nyvoll
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Proposing Kitchen Designs: A Case-Based Reasoning Approach

Johannes Grindal Ervum

Supervisor: Pieter Jelle Toussaint, IDI
Co-supervisor: Sindre Nyvoll, Compusoft
Submission date: June 2019

Abstract

This thesis is motivated by an idea at Compusoft—a software company that develops a computer-aided design program to design kitchens. The idea is to reuse existing designs so that the system can propose possible solutions for a new kitchen. Case-based reasoning is a methodology within artificial intelligence that solves a new problem based on its experience—the main idea is that *similar problems have similar solutions*. In this thesis, however, the extended case-based reasoning view is used. This means, one measures the utility of a solution to a given problem. It is reasonable to believe that this approach has a great potential to propose kitchen designs. This thesis describes what case-based reasoning is and specifically its application in design. Additionally, it investigates related research in automatic furnishing and decision support systems in design. This thesis does not only describe a theoretical solution that is likely to work in practice—a working prototype is demonstrated and analyzed.

Preface

This thesis was written as the final deliverable of the 5-year Computer Science master's degree program with the specialization in software engineering at the Norwegian University of Science and Technology.

First of all, I would like to thank my supervisor Pieter Jelle Toussaint for supervising this project, for introducing me to case-based reasoning, and for giving me invaluable advice that improved the quality of this thesis significantly. I would also like to thank my co-supervisor Sindre Nyvoll for his support in the implementation of the system. Without you, or the resources that Compusoft provided in this project, this thesis would not include a working prototype of the system.

I would like to thank my family for always backing and having faith in me. I would also like to thank my girlfriend and friends I made here in Trondheim for making these years a great time. Last but not least, I would like to thank my friends at home whom I always been looking forward to meeting during the holidays.

Table of Contents

List of Tables

List of Figures

Abbreviations

AI	=	Artificial Intelligence
ASR	=	Architectural Significant Requirement
BLOB	=	Binary Large OBject
CAD	=	Computer-Aided Design
CBR	=	Case-Based Reasoning
CBD	=	Case-Based Design
DSS	=	Decision Support System
FR	=	Functional Requirement
GUID	=	Globally Unique IDentifier
RE	=	Requirements Engineering
RS	=	Recommendation System

Chapter 1

Introduction

This chapter introduces the problem and why it is desired to find a solution to it. Section 1.1 describes a short history in design and people's willingness to invest in renovations. Section 1.2 explains Compusoft's design program, Winner Design. Section 1.3 and 1.4 introduce a scenario, and challenges non-professionals meet in design. Section 1.5 describes Compusoft's motivation to their idea; at last, Section 1.6 contains the research questions for the master thesis.

1.1 Interest in Design

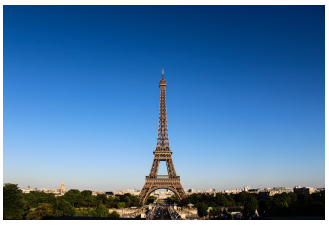
1.1.1 History

People have shown interest in architecture for decades. The Parthenon was built in Classical Greece (500–300 BC) and is seen as one the most famous buildings still existing from the ancient era. In fact, a full-scale copy of the Parthenon was built in Nashville as late as 1897. The Colosseum is a famous amphitheater in Rome and was built in 60–70 AD; much later in 1889, the Eiffel Tower was completed. This is just three examples of iconic structures in human history. Even today, countries are competing to build the most impressive buildings in the world.

Elsie De Wolfe wrote *The House In Good Taste* in 1913 [1]. According to *The New Yorker*, she invented the occupation interior decorator in this book. Although one can argue whether she was the inventor or not, one can say with certainty that she made people get interested in decorating their own homes. A decorator focuses on aesthetics when arranging furniture and decoration—the goal is to make the home look as good as possible. To achieve this, the decorator needs to be very creative and can freely decorate with few or none restrictions. An interior *designer* extends the scope of the decorator's circle of concern. In addition to aesthetics, they need to consider functional dependencies in the layout. For instance,

(a) The original Parthenon in Athens

(b) The Colosseum



(c) The Eiffel Tower

Figure 1.1: Historic structures

designing a kitchen may require a professional interior designer because of the complexity in the design.

1.1.2 Spending on Renovations

According to two surveys by Houzz, 58% of American and 56% of Canadian house owners renovated their homes in 2017 (full reports at [2, 3]). The results from these surveys are compared to the global trend in [4], which shows that the global trend is similar, but a bit weaker. The most popular room for renovation in both the US and Canada was the kitchen (31% and 28% of renovations); the median cost for a kitchen renovation was 11,000 USD and 12,000 CAD (≈9200 USD). The reports show that people are eager to renovate and are willing to spend a significant amount to do so.

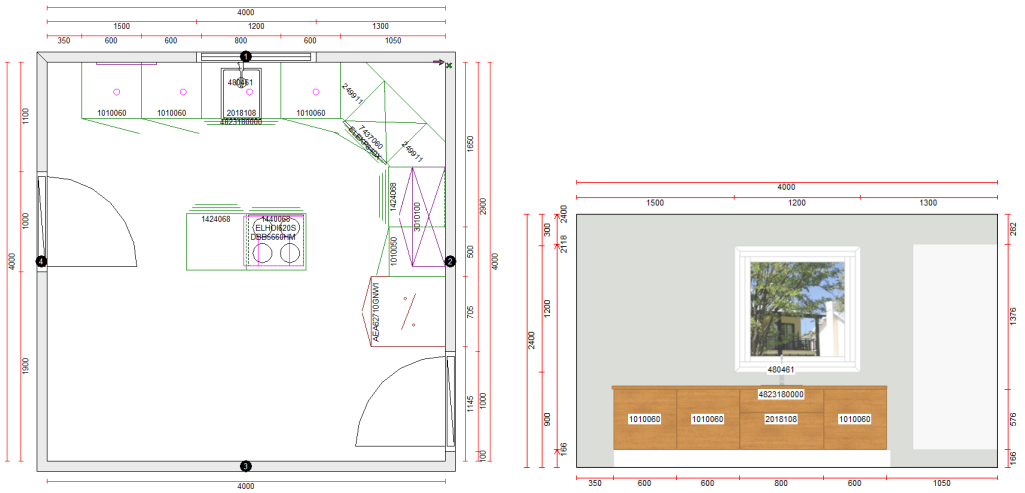
1.2 Winner Design

Winner Design is a computer-aided design (CAD) software for designing kitchens and was released in 1998 by Compusoft[1]. The software is used by kitchen retailers to tailor solutions for their customers. Currently, there are over 25,000 active Winner Design licenses which make Compusoft the market leader in Europe of this segment. One significant advantage of their program is the integration between different parts of a business transaction. In addition to creating a design, you get a complete list of all the items you need to realize it. The system provides an offer based on the articles, labor, delivery, and assembly. The manufacturer gets the order automatically if the customer accepts the offer. See Figure 1.2 for examples of what you can achieve with Winner Design.

Winner Design offers three different views: 2D, 3D perspective, and photorealistic. The 2D-view is the most technical one which shows appliances, worktops,

[1] https://www.compusoftgroup.com/worldwide

Figure 1.2: Sample designs from Winner Design



(a) Floor view **(b)** Wall view

Figure 1.3: 2D view in Winner Design

base cabinets, wall cabinets, etc., and their measurements. The user can choose between floor or wall view depending on where one wants to furnish (see Figure 1.3). The perspective view in 3D offers a better visualization (see Figure 1.4) of the real kitchen but contains no technical details. A photorealistic high-quality image can be generated from the perspective but does not support interaction.

1.3 Design Scenario

How each kitchen retailer operates differs but let us imagine one hypothetical case. You want to renovate your kitchen and visit your desired retailer and ask a professional interior designer for help. The designer needs some information

(a) Perspective view **(b)** Photorealistic view

Figure 1.4: 3D view in Winner Design

about your needs and measurements of the room. Depending on your preparation, you might have a design already in mind or are open for suggestions. Because you are prepared, the designer obtains your needs through an interview/need-analysis. The designer creates a kitchen design in Winner Design from your wants and shows you a perspective view of the kitchen. As a professional, the designer might identify possible flaws in your desired design and proposes alternative solutions which might be better. You need some time to rethink at home and receive prints of the different designs including price offers for each. The design session and copies are something the retailer gives you for free. After some consideration, you are satisfied with one of the alternatives and finalizes the deal.

1.3.1 Potential Problems

Some people are interested to see the potential of their kitchen but not willing to invest in a renovation. These ”customers” come to a retailer and get suggestions for possible designs. Retailers usually offer consultation and sketches for free which is a waste of time and resources if there is no chance of finalizing the deal. Another potential problem is people who receive the designs and use these to purchase the furniture from a cheaper vendor. This is a typical problem for physical shoe stores where people visit the store only to find the correct size and order them online.

1.3.2 Possible Solutions

An experienced designer might detect early whether a new customer is only curious or likely to make a purchase. A successful strategy would be to spend more time on the serious ones and make a quicker sketch for the curious. Creating designs from scratch is time-consuming, and a system that generates possible designs could be helpful. To make it harder for people to abuse free consultation and

sketches, retailers often print only the perspective view of the kitchen. This view does not contain details such as measurements and is harder to reuse directly.

1.4 Design Challenges for Non-Professionals

The first challenge a non-professional user faces when wanting to design something is to find an appropriate software or tool. There exist free CAD software on the internet, but these are often limited. Desired functionalities are only available for paying subscribers. Pen and paper would in many cases be more effective but lacks the details a CAD software provides. Assume one finds a program that supports the creation of kitchen designs. The next problem one faces is to learn how to use the program. For a non-technical user, this might be the end because CAD software is usually hard to use without prior experience or training. However, suppose the user found a CAD software and knows how to use it. The next challenge is how to design a kitchen. Professional designers use prior experiences and tacit knowledge. That is, the knowledge which is hard to write down or verbalize—which makes it difficult or impossible to transfer. For instance, the designer might recognize a good solution but cannot say precisely why it is a good one. Last but not least, if one actually managed to design a kitchen, one would have to find a manufacturer that provides the furniture. To summarize, it is clear that creating designs are considerably challenging for non-professional users.

1.5 Automatic Generation of Designs

Compusoft competes with other companies offering CAD software in kitchen design. To remain the market leader in this segment, they have to make sure their product offers more functionality and better usability than their competitors. As we have seen, non-professionals face several challenges in design. Hence, their customers are professional kitchen retailers. Compusoft believes a system that proposes kitchen designs automatically will be beneficiary for designers. It can bypass the tedious and time-consuming creation of simple layouts; it can serve as a starting point for more creative work. Additionally, it can enable non-professionals to generate kitchen designs themselves. Except for template-based designs, auto generation of kitchen designs would be the first of its kind in the market.

1.6 Research Questions

Compusoft has a large number of kitchen designs in their database which might be reusable when designing a new. The idea is that a customer can measure the size of the kitchen, make some requirements of the design, and get a design proposed automatically by the system. Case-based reasoning (CBR) is a problem-solving methodology that reasons about cases (prior experiences) to find a solution to a new problem. CBR is explained in more detail in Section 2.1. In this thesis, I will research whether a CBR system can propose kitchen designs and my research question are:

RQ1: How to design a system that proposes kitchen designs by using CBR?
RQ1.1: How should a case (kitchen design) be represented?
RQ1.2: How can a case be efficiently retrieved?
RQ1.3: How can a design be adapted into a new kitchen?

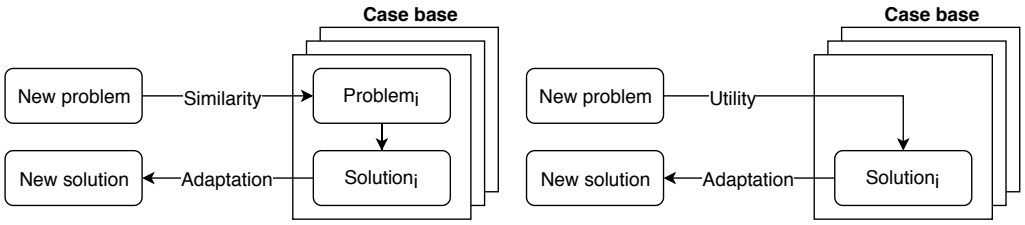
Chapter 2

Theory

Section 2.1 is based on a textbook by Richter and Weber [5] and one of the most cited foundational papers in CBR by Aamodt and Plaza [6]. Maher and Pou [7] address issues and applications of CBR in design—also known as case-based design (CBD). The general CBR theory is supplemented with this knowledge to make Section 2.1 more directed towards the design domain. Section 2.2 and 2.3 discuss related research—decision support systems in design and automatic furnishing. Section 2.4 contains a brief history of computer-aided design, and Section 2.5 describes some basic theory in kitchen design. Finally, I summarize the chapter and explain why CBD is an interesting approach to solve Compusoft's problem in Section 2.6.

2.1 Case-based Reasoning

CBR is a problem-solving methodology where the system reasons about cases—as the name indicates—which are experienced events and the solution for them. The main idea of CBR is that similar problems can be solved similarly. Aamodt and Plaza [6] identified four key processes: *retrieve*, *reuse*, *revise*, and *retain*, applied in that order. These processes form the so-called CBR-cycle (see Figure 2.5). A problem is first transformed into a specified data representation. Then the system searches the case base (where all cases are stored), and the most similar problem and its solution are retrieved. The proposed solution is reused either directly or after adaptations. Then the new solution is revised (evaluated) either internally or by an external expert. The new problem and its solution are at last retained as a new pair in the case base. Each step of the cycle requires general and/or domain-specific knowledge. The system uses this knowledge for reasoning and cases are stored in the memory for later reuse. CBR is classified as artificial



(a) Standard **(b)** Extended

Figure 2.1: Standard vs. Extended View

intelligence (AI) because it uses knowledge to reason about cases and learns by retaining them.

2.1.1 The Standard vs. The Extended View

The general idea of CBR is that similar problems have similar solutions. A case base is queried to find the most similar case(s). This is not very different from how people are solving problems manually. However, the approach has a weakness which limits the application domain. Let us say you recently opened a brand new car dealership. The first customer visits your dealership and tells you their preferences for the new car. You are using a CBR program and are now encountering a significant problem—the system has no past experiences. There is no lack of solutions (cars), but there are no prior problems to compare with the new customer's problem.

This challenge was the motivation for Bergmann et al. [8] to introduce the extended view. In the standard view, similarity functions compare two problem descriptions; in the extended, utility functions measure the expected utility of a solution (see Figure 2.1). The two approaches are in some sense related because the solution to the most similar problem is also anticipated to have the most utility. In fact, both approaches should propose the same solution, given the case base in the standard view is filled with sufficiently many cases and the solution set is equal. That a CBR system can operate without past problems is a significant advantage of the extended view. However, the new model has its own challenges. Comparing two problem descriptions is often simpler than estimating the utility of a solution. Problems are represented by the same model, while a problem and solution might be modeled very differently.

	Standard view	**Extended view**
A case contains	problem-solution pair	solution/product
Similarity between	old and new problem	problem and solution

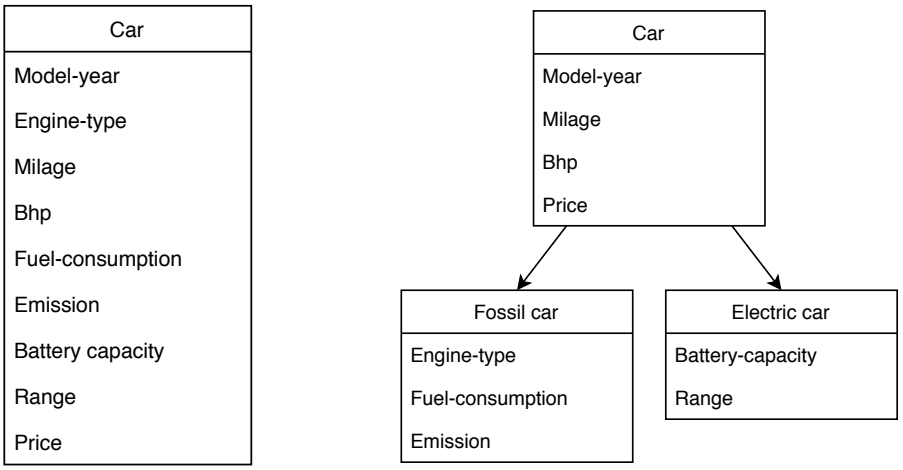
Table 2.1: Standard vs. Extended view

2.1.2 Basic CBR Elements

2.1.2.1 Case representations

A case is an experienced event that comprises a problem-solution pair. The problem is often in the real domain, and it is essential that the data representation reflects the complexity of the problem. A mismatch can be problematic because the retrieval process might not return the correct set of experiences from the case base. Cases can be represented in a wide variety of ways. Flat attribute-value is maybe the most popular choice because it is easy to understand, store, and retrieve (see Figure 2.2a). The disadvantage is that complex information is hard or impossible to represent. There are more advanced representations such as object-oriented and graph-based. Object-oriented representations allow a more compact style and might be even easier to understand but require more complex similarity functions. In Figure 2.2b we see that instead of storing all attributes related to a car in a single class (flat organization), attributes can be separated into sub-classes. For instance, battery capacity is not relevant to a fossil car, and emission is not to an electric—here, an object oritented representation allows a more compact structure. Notice that the cases only represent cars (or solutions) which means they are only usable in the extended view. All cases would need a problem part—where the car is the solution—to be useful for a standard CBR system.

One can design systems in different domains, for instance, in software, architecture, interior design, mechanical devices, structural systems, meal planning, etc. The representation of a design problem depends on which view you apply: you can write a rationale, draw a sketch, create a CAD model (a model created with a computer-aided design software), create a physical model, or any other approach. Maher and Andrés [9] argue that a common issue in design is that design cannot be described from a single point of view. You will have to consider the abstract, rationale, constraints among elements, trade-offs, decisions, etc. A case cannot comprise all these elements without compromising effectivity and/or efficiency. Hence, CBD systems tend to be domain specific and focus on parts of a problem.



(a) Flat attribute-value representation **(b)** Object-oriented representation

Figure 2.2: Case representations

2.1.2.2 Case bases

A case base is a special kind of a database that contains all the cases. In a relational database, the retrieval mechanism searches for exact matches between queries and records. In a case base, however, one tries to find the most similar or useful cases—which means one does not search for exact matches. The organization of the case base affects the effectivity and efficiency of the retrieval—sometimes one cannot have both. Kolodner [10] discusses different ways to organize the case library.

The first and simplest one is a flat structure (see Table 2.2). Each case is stored in a list, array, or file. It is easy to add a new one because you can just add it at the end of the list. Given the similarity measure is correct, you are guaranteed to retrieve the best case(s) because all cases are investigated. There are, however, a significant drawback with this organization. As the case base grows in size, retrieval gets quickly overwhelmed. The efficiency can be improved by searching in parallel or making the matching function more efficient, but the organization is still not suitable for large case bases.

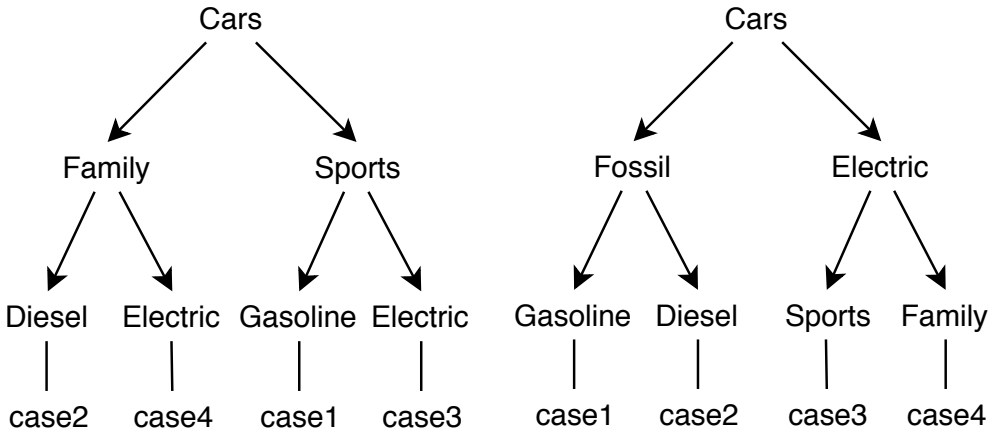
A shared-feature network is a more complex organization which overcomes the retrieval problem when the case base grows in size. When retrieving cases from this structure, one starts at the root node and chooses to traverse the best child based on a specific attribute which is specified at the node. For instance, a car buyer wants a new family car with an electric engine. Using the shared-feature network in Figure 2.3a, one chooses the branch containing family cars, then the electric car branch and the customer ends up with a Nissan Leaf (same

Attributes	**Case id**			
	case1	case2	case3	case4
Brand	BMW	Audi	Tesla	Nissan
Model	M3	A4	Model S P90D	Leaf
Model-year	2014	2015	2018	2018
Mileage (km)	47,426	28,968	0	0
Engine-type	Gasoline	Diesel	Electric	Electric
Bhp	424	190	525	150
Fuel-consumption (L/10km)	0.76	0.33	N/A	N/A
$C0_2$ emission (g/km)	204	99	N/A	N/A
Battery capacity (kWh)	N/A	N/A	90	40
Range (km)	N/A	N/A	473	378
Price (€)	38,071	22,966	97,000	33,529

Table 2.2: Flat case base organization

cases as in Table 2.2). In this situation, the customer got the best alternative, but this is not guaranteed in all situations. Consider a customer that wants a gasoline car, and if possible, a family car. The network will first explore the family car branch, then the diesel branch because diesel is more similar to gasoline than electricity. Hence, the reasoner proposes the Audi A4, although the BMW M3 is the best alternative. To enable shared-feature networks to support different reasoning goals one can build graphs having different prioritization of attributes. The graph in Figure 2.3b would successfully propose the BMW M3 for the second customer. The advantage of shared-feature networks is that it handles large case bases, but there are also a few disadvantages: it is not guaranteed to find the best solution, one possibly needs multiple shared-feature networks, and inserting new cases is a complex task.

The third structure is discrimination networks which are closely related to shared-feature networks but differs on some aspects. Both are graph-based, which limits the search space significantly. They differ on how branches are explored—discrimination networks use a question-answer approach as opposed to selecting the best child in shared-feature networks. All internal nodes hold the answer to the question in their parent node and a new question for their children. In Figure 2.4 one first ask what type of engine one wants and visits the node holding the answer. This sequence continues until a leaf node is reached. The prioritized order of the questions can be altered to support different reasoning goals. Discrimination networks have the same advantages and disadvantages as shared-feature networks but have a few extra of both. Retrieval is more efficient because selecting a branch (pick an answer) is easier than computing the best child. Choosing the wrong node



(a) Prioritization 1 **(b)** Prioritization 2

Figure 2.3: Shared-feature networks

early—resulting in a bad retrieval—is more likely in a discrimination network. A major drawback of both is how to deal with missing information; which branch should be chosen if there is no data to answer the question?

2.1.2.3 Similarity measures

The similarity measures—or utility measures in the extended view—are one of the core functions in CBR. As we saw in Section 2.1.1 one tries to find the most similar problem and retrieve its solution in the standard view; in the extended, one tries to assess the utility of a solution directly.

Definition 2.1. The similarity between two problems $p \in P$ is a function

$$\text{sim: } P \times P \to [0, 1]$$

A value close to zero indicates that the two problems are dissimilar, and a value close to one indicates that the two problems are almost equal. Measuring the similarity between two problems involves two steps: first, one computes the similarity between attributes separately; then a weighted sum of these indicates the global similarity—which is the so-called local-global principle. The weight of an attribute indicates its importance when comparing two problems. Different weighting schemes can enable comparisons based on different viewpoints—for example, in design, the view of the reasoner has a high impact of what is considered similar or not. Each attribute needs to have its own measure (or local similarity measure). For instance, two cars compared by model-year are dissimilar if one is 20 years older than the other. However, they are similar based on mileage if they only differ by 20 km.

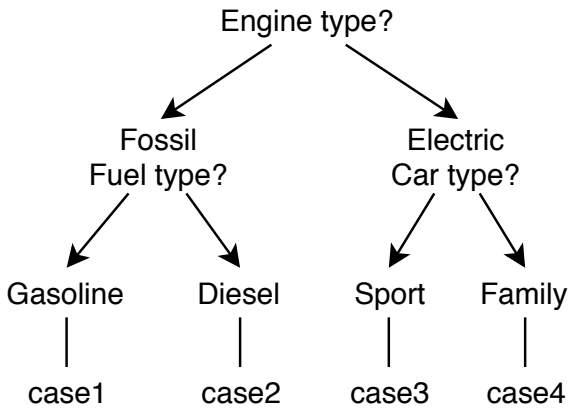


Figure 2.4: Discrimination Network

Definition 2.2. The utility of a solution $s \in S$ to a problem $p \in P$ is a function

$$\text{u: } P \times S \to [0, 1]$$

Utility functions are used in the extended CBR view to find the best solution to a problem in the case base. The problem and solution space are often in different domains—when a customer defines their requirements for a new car, they might use a different vocabulary than how a car is represented in the system. For example, the customer wants a gasoline car with low mileage, and it should be eco-friendly. An eco-friendly gasoline car could be one that has a low fuel-consumption and CO_2 emission, but the customer is not able to provide a specific number for any of them. The system needs to find a solution that is most useful for the customer. The global utility can be measured in the same way as with global similarity by the local-global principle.

Example 2.1. Computing similarity in the standard view

Table 2.3 compares a new car q (a Ford Focus) with the case base in Table 2.2. The goal is to find the most similar car to q and reuse the price for this car. The similarity between each attribute is computed with their local similarity measure; each attribute has a weight w which denotes its importance to the global similarity. In this example, $w = 1$ means it has a low impact, while $w = 6$ means it has a great impact. The global similarity is computed as a weighted sum

$$sim(q, c_i) = \frac{1}{\sum_{j=1}^{n} w_j} \sum_{j=1}^{n} w_j * sim(q_j, c_j)$$

and the results are

$$sim(q, c_1) = \frac{1}{37} * 15.0 \approx 0.41$$

Query q	Attributes	w	sim(q,c1)	sim(q,c2)	sim(q,c3)	sim(q,c4)
Ford	Brand	5	0	0	0	0
Focus ST	Model	6	0	0	0	0
2016	Model-year	4	0.8	0.9	0.8	0.8
15,934	Mileage	5	0.7	0.8	0.5	0.5
Gasoline	Engine-type	4	1	0.5	0	0
182	Bhp	6	0.3	0.9	0.2	0.7
0.35	Consumption	4	0.4	0.9	0	0
110	Emission	3	0.3	0.9	0	0
N/A	Battery cap.	6	-	-	-	-
N/A	Range	6	-	-	-	-
?	Price		38,071	22,966	97,000	33,529

Table 2.3: Case similarity measurement

$$sim(q, c_2) = \frac{1}{37} * 19.8 \approx 0.54$$
$$sim(q, c_3) = \frac{1}{37} * 6.9 \approx 0.18$$
$$sim(q, c_4) = \frac{1}{37} * 9.9 \approx 0.28$$

The Audi A4 is the most similar one to the Ford Focus ST, so its price is reused—namely €22,966.

2.1.2.4 Knowledge containers

The knowledge model in CBR comprises four different knowledge containers: the vocabulary, similarity, case base, and adaptation container. The vocabulary container contains all terms one needs to talk about something. For instance, we cannot discuss the positioning of furniture if ”furniture” is not in the dictionary. Describing possible actions for a real-world object with natural language may lead to infinitely many descriptions. Moreover, the object description itself varies from person to person depending on viewpoint, experience, knowledge, etc. Because of this, it is essential to identify the attributes that are necessary to create a data model—these attributes are a part of the vocabulary container.

The similarity container contains the knowledge and algorithms needed to compare cases. Depending on the complexity of the problem, the demand for domain knowledge ranges from none to high. Moreover, the complexity of the algorithms varies from distance functions (comparing numeric values) to sophisticated models (comparing symbolic values).

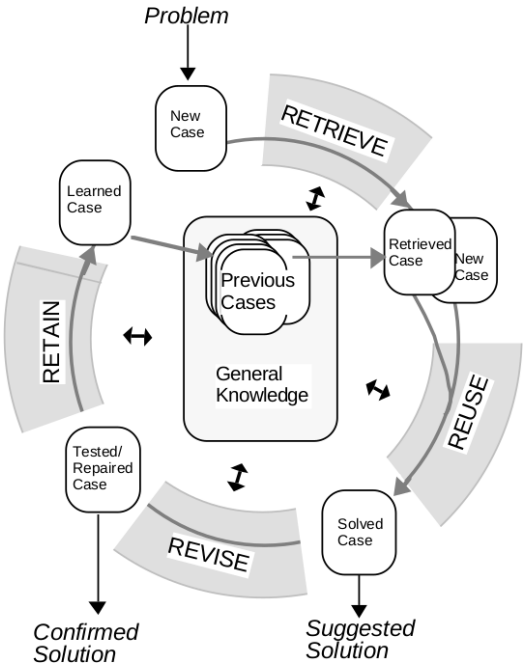


Figure 2.5: The CBR cycle presented by Aamodt and Plaza [6]

The case base container is just the case base itself and is the main source of knowledge in CBR.

The adaptation container comprises methods to transform problems and solutions. For instance, if you search for something in your favorite search engine and misspell a word, the system corrects the mistake automatically. Adaptation of solutions increases the number of problems the system can solve because unusable solutions can be modified and become useful. Applying adaptation successfully in a CBR system has a great impact on the performance.

2.1.3 The CBR Cycle

2.1.3.1 Retrieval

Case retrieval is the first process of the CBR cycle (see Figure 2.5). The task can be divided into feature identification, search, initially matching, and selection. Feature identification is partially solved with an appropriate case representation. Mapping the new problem into the case representation may in some situations be straightforward and in other more complicated. The complexity of the search method depends on how a case is represented; a more complex representation usually makes the search more complex as well. The MAC/FAC method introduced by Forbus et al. [11] is proved to be efficient when the search space is big. MAC

(many are called) is the first step which completes the initial matching. Here, a computationally cheap method is used to scan the case base; many potential cases are called/selected. FAC (few are chosen) is the second step, which uses similarity measures that rank the selected cases from most to least similar. At last, the best case is retrieved—usually the most similar one. As we saw in Section 2.1.2.2, the organization of the case base impacts retrieval.

Retrieval of designs can be done either informally or formally. An informal approach requires the user to browse the case base manually. The effectiveness depends on the richness and understandability of the indexing scheme (case representation). Formal retrieval accepts a problem description as input and retrieves similar cases automatically. The most popular method for computing similarity is comparing attribute-value pairs. Determining the weights for the global similarity can be really difficult for design, because the similarity between designs depends on the different views.

2.1.3.2 Reuse

Reuse is the second step of the cycle and is applied to the retrieved case. The optimal situation is when the suggested solution can be used directly on the new problem. This is normal if the task is to classify objects. In other domains, however, solutions often need to be adapted. Transformational methods modify the proposed solution based on rules while derivational methods modify the process that generated it. For instance, changing an attribute value is transformational reuse, and changing the input values for a formula is an example of derivational. Making good use of adaptations can increase the performance of a CBR system drastically and may propose plausible solutions even with a small case base.

A design case often comprises many constraints. Here, the adaption can be carried out by solving a constraint satisfaction problem (CSP). Retrieving a case gives a valid solution in the old context but might violate constraints in the new. It becomes valid by solving the new CSP. Another approach is to leave the adaptation to the user, as in *Archie II* (see Section 2.1.4). Adaptation is, in general, a difficult task, and is especially challenging in design. Quality evaluation in design is difficult because of the lack of formal knowledge in this domain.

2.1.3.3 Revise

Revise is the third step of the cycle and is an evaluation of the proposed solution's quality. In some situations, this can only be simulated or estimated in beforehand, and other times it can only be measured afterward. Revision is usually performed outside the system because the solution is applied in the real environment and cannot be tested accurately in advance. A satisfiable solution can be stored in the

case base; an unsatisfiable one has to be repaired by user input or domain-specific knowledge before the case is stored in the case base. The error might expose a significant liability of the CBR system which requires not only an improvement of the solution but to the entire system.

2.1.3.4 Retain

Retain is the last step of the CBR cycle. Some systems store all cases, while other store only those that increase the competence or performance of the system. The competence of a case base is defined by how many problems it can solve—a new case that solves the same problem as another case does not increase the competence. The performance of a system is how fast it can retrieve and propose a solution to a new problem. Storing every case can lead to the so-called swamping problem which means the cost for finding a solution overcomes the value one gets from it. How one stores the cases depends on the design of the system. Each problem-solution pair can be retained as a new case, or an old one can be modified to subsume both the new and old problem—given they had the same solution. In the extended view, one only retains the solutions because the problem is irrelevant. A solved case can also help to improve the indexing structure, i.e., update which features are essential in distinguishing cases. An improved indexing structure will, in theory, increase performance and/or the effectivity of the system.

2.1.4 Applications of CBR in Design

2.1.4.1 Archie-II

Archie is a case-based design aid system that lets the user create a conceptual design [9]. The user writes a partial description of a building which is queried in the case base. Similar buildings and their documentation are displayed in the program. This is called issue discovery because it provides the user with inspiration from previous cases. The user can use this information to identify relevant issues. For instance, one wants to design a library; prior cases reveal that natural light and privacy are important in reading areas. The building needs windows to allow light to pass through, and windows compromise privacy, i.e., conflicting issues. Archie has a case base for such concerns which helps the user to make trade-off decisions. When the user has completed the conceptual design, Archie can evaluate the quality of it. However, this does not work very well because the evaluation of designs is a very complex task. Archie is only an assisting program, so no plausible designs are proposed—the users have to create it themselves.

2.1.4.2 FABEL

FABEL was a project that took place between 1992 and 1996 in Germany. Angi Voss was the project manager of a team of over 20 members in addition to several students who contributed to the project; she describes her experiences in Maher and Pu [7]. The program they implemented supports multiple methods for retrieval, adaptation, assessment, and elaboration. She recommends that one should experiment with competitive and complementary methods because the interpretation of complex designs is subjective. FABEL offers the user 13 different retrieval functions where each returns a different set of cases. The advantage with several options for retrieval is that it enables different reasoning goals. For instance, when they researched what makes two designs similar they got a different answer every time—indicating that the similarity between designs depends on which spectacles one uses. When developing a CBD system, she recommends creating prototypes early to verify that the system can solve the task.

2.1.5 Issues in CBD

K. Richter [12] investigates in her paper why CBR has not been as successful in architecture as one would think. She identifies highly recognized academic CBD systems such as FABEL, SEED, and DYNAMO, and questions why they have not succeeded outside the scientific world. She concludes in her paper that the main reason for the limited success of CBR in design is the lack of accessibility and acquisition of knowledge—that is, availability of existing designs and creation of new. General findings in her research show that architects are skeptical to reuse or publish creative work in fear of plagiarism and copyright violations. Additionally, they value originality and are afraid that reusing designs can decrease their creativity. A study by Heylighen and Neuckerman [13], however, shows that students increased the quality of their designs when exposed to several past cases; they also proved that their creativity was not decreased when using a CBD application.

2.2 Decision Support Systems

2.2.1 DSS in E-Commerce

The goal of a decision support system (DSS) is to provide support to a user in decision making. In several situations, the problem is complicated, or the user is overwhelmed with information. For instance, you have to make a series of choices if you want to renovate the kitchen. Some decisions can open up new opportunities in the design and at the same time create limitations—so-called trade-off decisions. You might become uncertain because you are not sure which choice is the

right one. With lacking decision support, you might postpone or permanently give up on the renovation. This is a lose-lose situation for both you and the potential seller because you will not get a new kitchen and the seller lost a buyer. A proper DSS would avoid this situation by giving you appropriate guidance throughout the buying process. Traditionally you could go to a kitchen retailer and speak with a professional, but getting expert help in e-commerce is more challenging.

Netflix uses a recommendation system (RS) to help the users in choosing series and movies based on their preferences. There is a difference between DSSs and RSs, but they are in some sense similar. A DSS is like a framework where the user can analyze graphs, statistics, figures, etc. that helps the user to make a decision; a recommender system makes an educated guess what you like based on what the system knows about you. Netflix recommends content to the users to increase the user experience. For example, *Lord of the Rings: The Two Towers* may be recommended if you watched *Lord of the Rings: The Fellowship of The Ring* and gave positive feedback for it. Likewise, Amazon gives recommendations of products you may be interested in. Additionally, customers review and rate products which are used by new customers to choose a product.

2.2.2 DSS in Design

2.2.2.1 DSS in House Customization: A Hybrid Approach

Juan et al. [14] proposed in 2006 a DSS that enables house buyers to customize their homes before the house is built, in other words, pre-sale housing. At the time of writing, this strategy dominated the housing market in Taiwan, but a recurring problem was that newly built houses often were modified because the buyer was not satisfied with the layout. House suppliers saw a business opportunity in offering the customers to participate in the design process. Success depended on the ability to communicate and they had problems because the customers did not get all necessary information and decision support to communicate their needs. Juan et al. [14] propose a system that makes this process more effective and efficient.

First, the buyer enters their preferences which include budget, area, spatial needs, housing layout, and interior fittings and finishes. Then the requirements are transformed such that they can be used for retrieval from a case base. The k-nearest neighbors method is used to find one or a small set of similar cases. All local similarities are computed by a formula, and the global similarity (similarity between cases) is a weighted sum of the local. The weights are user-defined from an analytic hierarchy process (AHP). How this works is not described or referenced in the paper. Then a genetic algorithm (GA) applied to the proposed solution(s) to find the near-optimal one(s). Describing GAs are outside the scope of this thesis, but the general concept is that one tries to mimic the evolution

of nature. The fittest individuals (solutions) survives and forms new generations (new solutions) where mutations occasionally happen. The final solution can be retained in the case base for reuse.

They conclude that a GA enhances the general CBR system. It was applied in an experimental project in pre-sale housing in Taiwan, and it helped to resolve some of the root problems.

2.2.2.2 DSS in Pre-sale Housing

Juan et al. [15] have written a paper where they have implemented a DSS using CBR that helps the customer to choose a house in the pre-sale house market. The system focuses on the Asian market where the Feng Shui theory is important for house buyers. They believe that fulfilling it will give success in life. For instance, half of the house buyers in Taiwan seeks advice from a Feng Shui expert before making a decision. The Form School focuses on the external factors: environment, location, and price—the Compass school focuses on the internal factors: area, needs, and layout. Both these schools have designed a model that is used in the DSS to evaluate these factors. For example, the room layout and for what purpose a room is used (guest room vs. storage room) influence the Feng Shui score based on your *Kua* number. This number is scientifically computed from gender and year of birth.

Although the theory itself is interesting, how they applied CBR to this problem is more interesting from a technical point of view. The case representation consists of both numeric and symbolic attributes which make the similarity measure more complex. The first task of the CBR cycle is to retrieve one or a set of similar cases. They apply a hybrid approach of inductive indexing and nearest neighbor search. Inductive indexing uses a decision tree which may be a more known concept [16]. It is effective when the case base is large, and sets of cases can be ignored because not all subtrees are explored. When the initial search is completed, they apply nearest neighbor search to find the most similar cases—not very different from the MAC/FAC principle described in Section 2.1.3. Attributes used in the inductive search are left out of the nearest neighbor search. The similarity between numeric values are computed, and symbolic values are compared by a model from the Form school. The local similarities are finally aggregated by a weighted sum, where the weights are user-defined by an AHP.

They conclude that their DSS significantly improves the housing decision and communication between the customer and Feng Shui experts in the pre-sale stage.

2.3 Automatic Furniture Arrangement

2.3.1 Automatic Furnishing using CBR and Floor Fields

A recent paper (2018) by Song et al. [17] proposes a web-based application for automatic generation of plausible designs. Their algorithm divides the scene in up to four different modes: *coupled*, *enclosed*, *matrix*, and *circular*. The coupled mode can be further divided into furniture-furniture and furniture-room coupled mode. This mode sets constraints for angles and distances between objects. For example, the TV should face directly towards and not be too far away from the sofa. The enclosed mode requires that furniture is first placed up to the wall and can then be moved along it. This mode is applied for example when placing a shelf in the room. Matrix mode handles furniture that is placed in a matrix-like manner. A typical example is placing desks in a classroom. The circular mode puts furniture in a circular form.

The coupled mode is solved by CBR or potentially recursive CBR. A case consists of one parent furniture and *n* identical child elements. The name of a case is on the form (parent furniture, child furniture, number of children) and the problem description is a vector $q = (l_0, h_0, w_0, l_1, h_1, w_1)$ where l, h, and w denote the length, height, and width of the parent and child furniture. The nearest neighbor is found from the set of cases with an identical name. The solution of the retrieved case contains a mathematical model, and the parameters from this model are assigned to the new problem. Recursive CBR is applied if the child of the parent object has its own children. For instance, a table can have a PC as a child, and the PC has a mouse. The PC is first placed on the table, then the mouse is placed relative to the PC.

The enclosed uses a floor field to place furniture. Energy fields near the walls are positive, and along a path the energy is negative. A path is a straight line between two doors, two windows, or a door and a window. Placing furniture close to positive energy fields is desired, and the optimal solution is the one with the maximum energy score.

The matrix and circular mode use somewhat simple methods. The vertical and horizontal distance between furniture is equal for all furniture in the matrix mode. In the circular, furniture is placed evenly around a round object.

To evaluate the quality of the generated designs, they consider the users' opinion. If the algorithm placed A number of pieces and the user re-arranged B of them, then the layout accuracy is computed as $LA = 1 - B/A$. A different method they propose is to print the 2D design and ask the users which pieces they would have moved. The two approaches are not very different and should give approximately the same results. An average layout accuracy can easily be computed from the individual evaluations.

They conclude that their algorithm is fast and effective and can meet functional requirements in indoor design.

2.3.2 Other approaches

Yu et al. [18] propose a solution using an optimization algorithm. A cost is computed based on important ergonomic factors. For instance, a TV facing away from the sofa has a higher cost than a TV facing towards it. They use simulated annealing which is an iterative optimization-based algorithm that finds a local minimum or maximum. Because the goal is to minimize the total cost, the simulation should find the local—or even better—the global minimum. Kán and Kaufmann [19] proposed in 2017 an algorithm that uses a genetic algorithm for cost optimization. The cost function considers aesthetics, ergonomics, and functionality. The same authors published a paper one year later that uses a greedy algorithm [20].

2.4 Computer-Aided Design

2.4.1 History

Sketchpad was the first computer-aided design (CAD) software invented by Ivan Edward Sutherland in his Ph.D. thesis in 1963 [21]. The designer used a ”light pen” to draw directly on a computer screen, and the drawings were highly accurate and supported complex designs. As you can see in Figure 2.6, the sketching is not very different from the use of pen and paper on a sketch board.

In the 1970s most CAD software was still 2D replacements for physical sketch boards, but there were some interest and research in 3D design. The decade saw significant advances in other aspects of CAD software, especially in fundamental geometric algorithms. General improvements in computer science were also notable. For instance, the computational power of computers increased substantially, prices and operating costs for computers decreased, new high-level programming languages and a more efficient operating system called UNIX was introduced.

In the 1980s IBM released its first PC and Apple released its first Macintosh. Autodesk released its first version of AutoCAD for PC just one year after IBM released their first PC. Although PCs became more popular than UNIX workstations, software on workstations offered much higher image quality than PCs. Workstations had much higher computational power, but as we know, this was only temporary. The market share for AutoCAD increased during the decade and is still one the most known CAD software today. For a more detailed history of CAD from the 1960s until 2004 see [22].

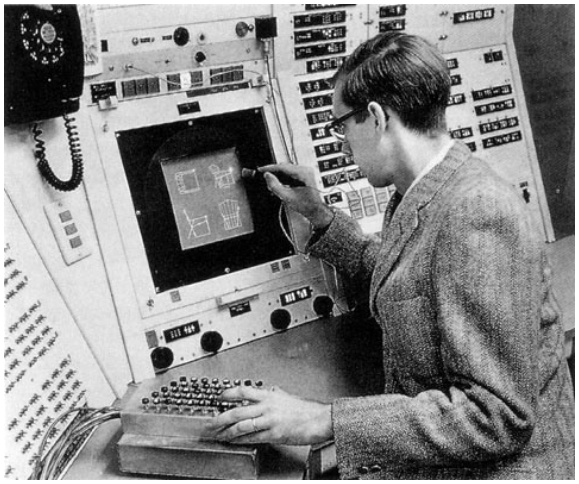


Figure 2.6: Ivan Sutherland demonstrates Sketchpad.
Source: http://history-computer.com/ModernComputer/Software/Sketchpad.html

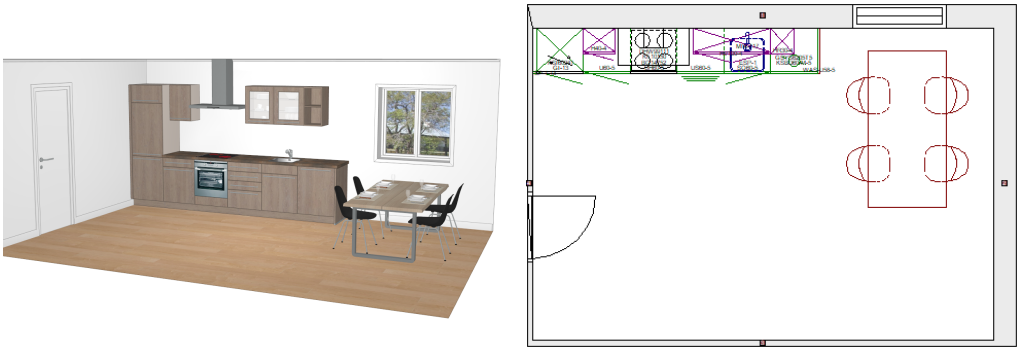
2.5 Theory in Kitchen Design

To find explicit knowledge about how to design kitchens was more challenging than expected. In Norway, it seems like a formal degree in interior design is not a requirement to work as a designer. I found some interesting requirements when browsing for open positions as a kitchen designer in Norway. Retailers often seek a person that has relevant experience, the right personality, good IT competence, and is interested in kitchen design. The National Kitchen & Bath Association in the United States offers various certification levels—they argue that certified designers are more attractive for employment and can show customers a proof of their competence[1]. A questionnaire by Taha et al. [23] shows that architects value experimental knowledge—that is, experience is crucial. The theory in the following sections should be seen more like guidelines because designers can—to some degree—choose their own style.

2.5.1 The Basic Shapes

A widely accepted theory is the idea of the five basic shapes: *single-wall*, *two-wall or corridor*, *L-shape*, *U-shape*, and *G-shape*.

[1] https://nkba.org/info/certification (Accessed 28.12.2018)



(a) 3D View **(b)** 2D plan

Figure 2.7: Single wall layout

2.5.1.1 Single-wall

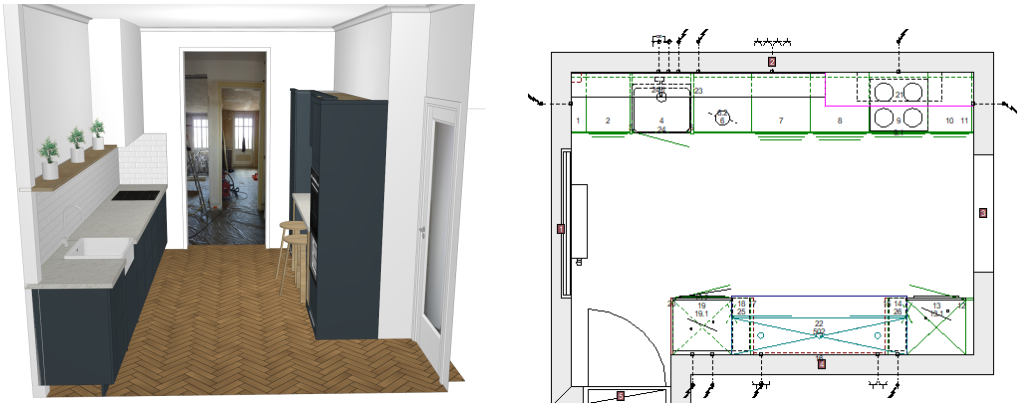
The single wall kitchen has everything on one wall (see Figure 2.7). This is suitable for small kitchens and is likely the cheapest layout to realize—it is area effective which makes more room for a dining table etc. The disadvantages of this layout are the limited workspace, and it may be problematic to be used by two or more persons at the same time.

2.5.1.2 Two-wall

The two-wall—or corridor—layout is similar to single-wall but has workspaces on two opposite sides of the kitchen (see Figure 2.8). This layout has obviously more workspace area and storage space than the single-wall but has one crucial disadvantage—the space between the workspaces gets to some degree unusable. For instance, placing a kitchen table in the middle will interfere with the workflow.

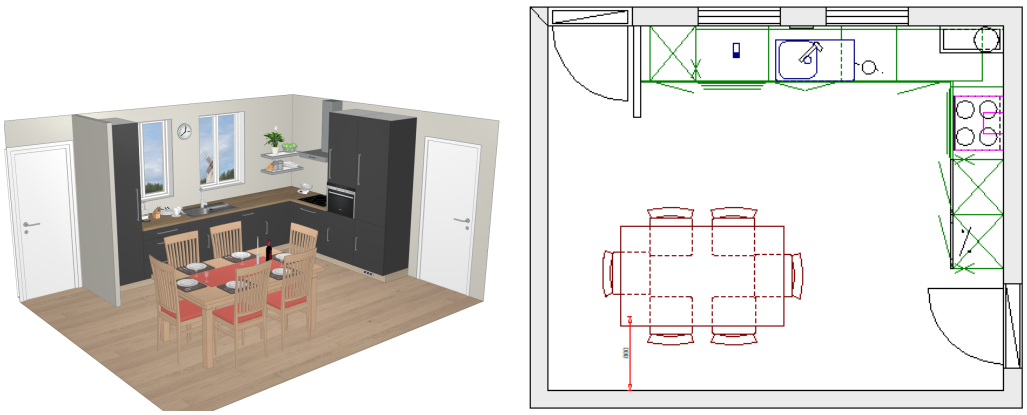
2.5.1.3 L-shape

The L-shaped kitchen is formed like the letter L (see Figure 2.9). It allows a continuous design—unlike the two-wall layout—which may be more practical if you want to have a dining table in the kitchen. Additionally, it is a quite simple design that is easy to realize.



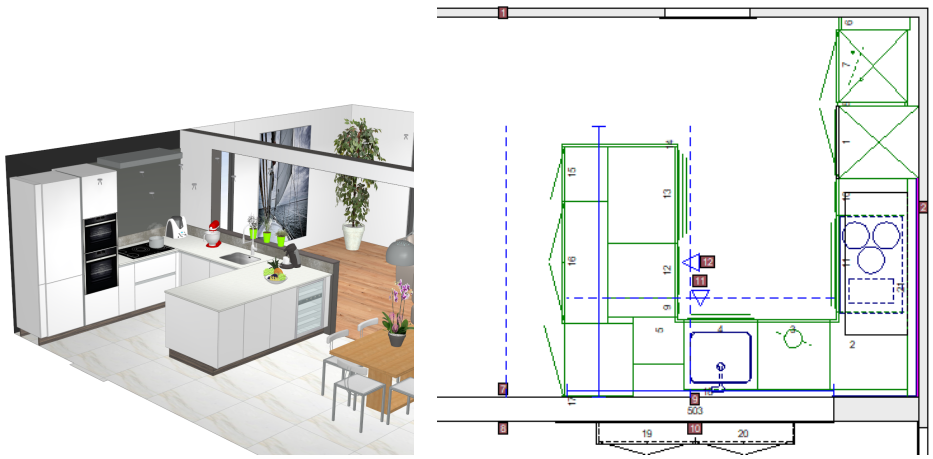
(a) 3D View **(b)** 2D plan

Figure 2.8: Two-wall layout



(a) 3D View **(b)** 2D plan

Figure 2.9: L-shape layout



(a) 3D View **(b)** 2D plan

Figure 2.10: U-shape layout

2.5.1.4 U-shape

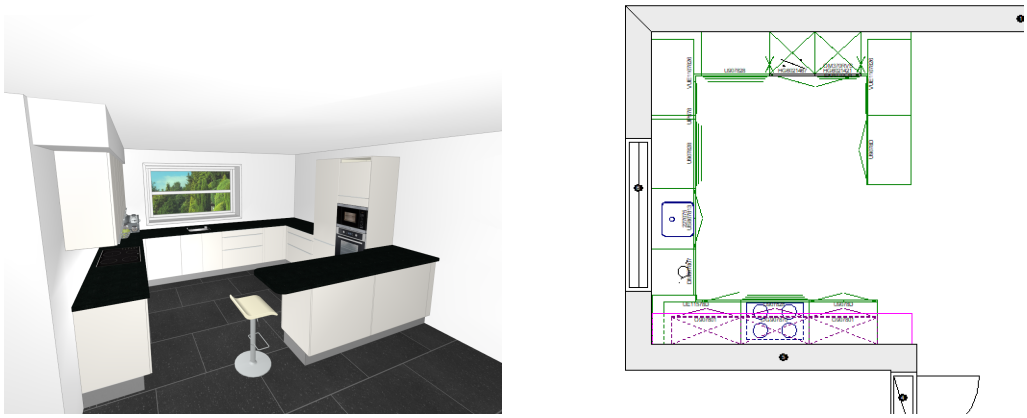
The U-shaped kitchen is a continuous solution that is suitable for larger kitchens (see Figure 2.10). This layout supports separate workspaces which enables two persons to cook together without stepping on each other's toes. Price-wise it is prone to be more expensive because you will need more furniture and countertop to complete the design.

2.5.1.5 G-shaped

The G-shape kitchen is similar to the U-shape but contains an additional piece at the end of the U to complete the G (see Figure 2.11). This piece can, for instance, serve as an additional spot to eat a smaller meal. The disadvantage of this layout is that the kitchen might seem enclosed by the furniture. Having a large kitchen could help to reduce this effect—in fact, both the U and G-shaped layouts are not suitable for small kitchens.

2.5.2 Kitchen Island

A kitchen island is one or a set of furniture that is not placed against a wall or beside another piece of furniture—you must be able to walk around it. It can be added to each of the basic shapes but requires that there is enough space for it. The size of the island can vary from a small zone for chopping vegetables to a larger one which also have appliances. The cost of a kitchen island depends on what it contains. Having a sink on it leads to moving the drain away from the wall



(a) 3D View **(b)** 2D plan

Figure 2.11: G-shape layout

which is costly; similarly, installing an extractor hood on an island usually costs more than placing it on the wall. The upside is that one can make better use of the limited kitchen area. However, an island is not suitable for a small kitchen.

2.5.3 The Kitchen Triangle

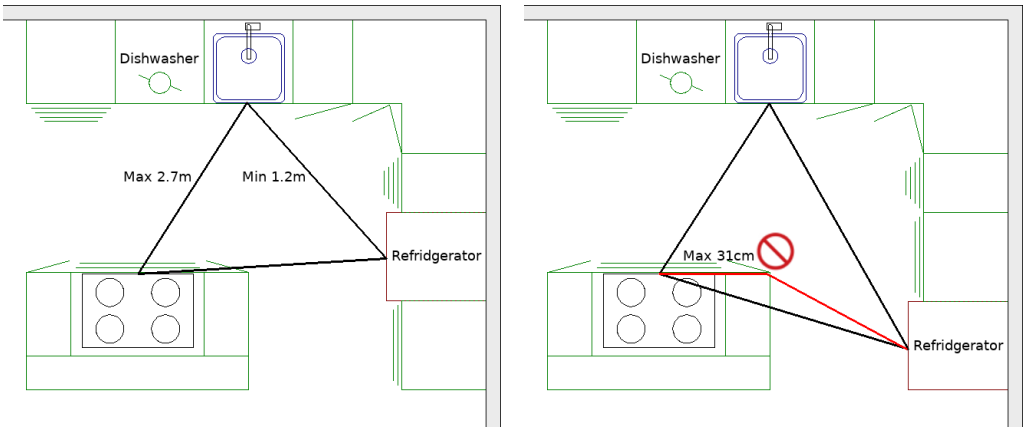
The kitchen triangle is a widely known concept for kitchen designers that has a great impact on the cook's workflow. In NKBA's planning guidelines [24] it is described what it is and which constraints that must be satisfied. Each corner of the triangle represent a primary work center—a work center is a major appliance and its surrounding landing area. A landing area is a space that is needed to keep things temporarily—for instance, a spot to put the cooking tray when taking it out of the oven. The length of each leg should be in between 1.2–2.7m and the perimeter of the triangle should not exceed 7.9m. Figure 2.13a shows an example of a kitchen with three work centers: cooking surface, preparation/cleanup surface, and the refrigerator. Additionally, no object can intersect the triangle by more than 31cm (see Figure 2.13b). The triangle can be extended if the kitchen contains more than three primary stations—in this situation, the additional work center should not be further away than 2.7m to the other stations.

2.6 Summary

The paper by K. Richter [12] raises a concern that CBD applications have limited success in the real world—the most successful ones are in the academic field. So

(a) One-wall with island **(b)** U with island

Figure 2.12: Layouts with island



(a) Triangle OK **(b)** Triangle not OK

Figure 2.13: The Kitchen Triangle

why would it be successful in kitchen design? Or, why would it *not* succeed? A simple—probably too simple—answer for the second question is the absence of prior success in CBD; K. Richter pointed out that the lack of accessibility and acquisition of designs were the main reason. At Compusoft, I have access to a database that contains a large set of kitchen designs created by professionals, which does at least increase the chances of a successful CBD program based on her concern about this issue. The case base in CBR is like an experience memory; relevant designs can be retrieved for the architect when they face a similar design problem—in kitchen design, this could be a room which has a similar layout. A retrieval-only system would only show potential solutions, while a more complex CBR program could also adapt a design to fit the new room. As we saw that in *Archie II*, the user only got support in the design process; the user had to create the designs themselves. It is clear that adapting designs is far more challenging than just retrieving them. The minimum requirement for the CBD system I shall develop in my master should be: to define proper case representations and be able to retrieve potential designs for a new kitchen. It would be a huge plus if the system also could adapt them.

Chapter 3

Method

This chapter describes how I worked when writing my master thesis and developing the CBR application. The working method and how to describe it are inspired by the textbook by R. Wieringa [25] about design science methodology.

3.1 Type of Research Problem

R. Wieringa [25] defines two main categories for research problems: design problems, and knowledge questions. The goal of a design problem is to create an artifact that answers the question raised by the research problem. Knowledge questions seek answers to the research problem but do not require an implementation—a theoretical answer is sufficient. In this thesis, it is clear that we have a design problem because the answer to the research problem ”How to design a system that proposes kitchen designs by using CBR?” will be presented by a proof-of-concept prototype.

3.2 The Design Cycle

Design problem research iterates through the design cycle which consists of three activities namely problem investigation, treatment design, and treatment validation. It is a part of a larger one called the engineering cycle that includes treatment implementation and implementation evaluation as well (see Figure 3.1). In this project, only the design cycle was performed because the last two steps of the engineering cycle require transferring the artifact into the real world for use and evaluation; in other words, as the presented artifact in this thesis is only a prototype, it stays within the academic/development boundary.

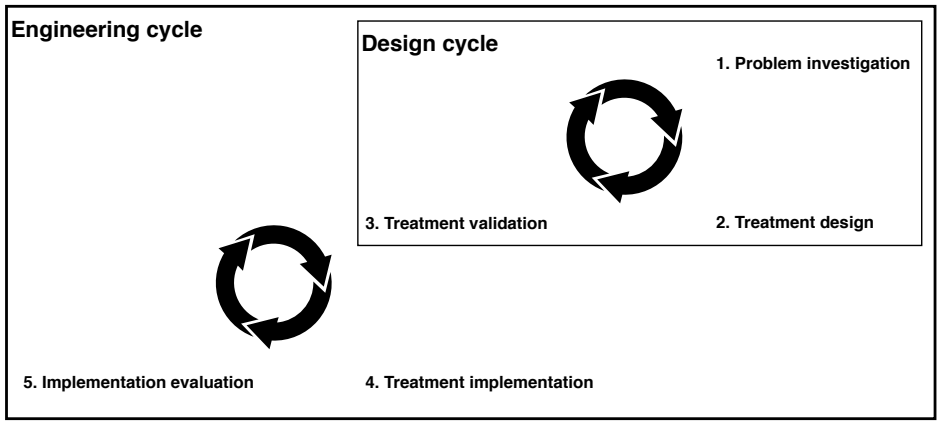


Figure 3.1: The engineering cycle

3.2.1 Problem Investigation

In the first step of the design cycle, a real-world problem must be identified that someone desires a solution for, and at the same time are willing to allocate resources for the development of the artifact—i.e., identify stakeholders. The resources Compusoft allocate in this project are a developer, Sindre Nyvoll, who is the co-supervisor of my thesis, and they cover the operational costs such as storage and deployment costs of the application.

I had several discussions with the developers at Compusoft to find a problem that would be either advantageous for them internally or for their customers. As the research problem of my thesis clearly shows, we settled for a system that will propose kitchen designs. An alternative thesis problem was to design a system that would analyze the database of the CAD objects and try to identify similar objects. This could help to manage the database such that duplicates are removed, and it would make it easier for designers to replace a piece of furniture with a similar one. The reason why the autoplanning feature was preferred, was that Compusoft believes that this feature will improve their services the most. As we saw in the introductory chapter, some people are only curious to see the potential of their kitchen but not willing to invest in a renovation. The designers could spare time in suggesting designs to these people by using the autoplanning system instead of designing kitchens from scratch. Additionally, the service can be made available online which would enable people to explore designs at home.

When the problem was identified and described, I had to find a professor at the university that would like to supervise my master thesis. After a meeting with Pieter Toussaint, he accepted the request to be my supervisor. At that point, there were no constraints on how to design the system. Together with Pieter, we found an academic approach to solve the problem such that the solution yields

new knowledge in the scientific field, not only a new artifact that is just valuable for Compusoft and/or their customers. The approach we chose was to research how CBR can be used to propose kitchen designs.

3.2.2 Treatment Design

In the second step of the design cycle, one must specify the requirements of the artifact and verify whether these contribute to the stakeholders' goals. To satisfy a requirement, one can use available treatments or design new ones. R. Wieringa [25] prefers the term *treatment* to *solution* because the treatment of a problem may or may not be successful while a solution indicates that the problem is solved successfully. Designing a treatment means the same—at least for a software engineer—as to implement a piece of software that is supposed to satisfy a set of requirements.

I spent a considerable amount of time in the first iteration of the design cycle. I had to specify the requirements of the system as well as to check whether these contributed to the goals of the stakeholders. The goals of the stakeholders were derived in the problem investigation—Compusoft's goal is a system that can propose designs, while the academic goal is to research how CBR can be used to realize it. Requirements can be divided into three categories: functional requirements, quality attributes, and constraints. The functional requirements specify what the system must be able to do. The quality attributes define the quality or how this functionality is implemented, while the constraints specify decisions about the software architecture the architect must respect [26]. I discussed the requirements of the system with the developers at Compusoft and interviewed experienced designers at Røskaft Kjøkken in Trondheim. The outcome of these activities is described in Section 4.1.2, and the critical ones are: the application must be accessible as a web-service, it must be developed using the ASP.NET Core framework with C#, and the designs must be proposed using CBR.

When the requirements were elicited, I investigated available treatments which include searching for available technologies, frameworks, and programming libraries that would make the design easier to realize. When designing my own treatments, I identified distinct modules, or sub-parts of the system, and designed them one at the time. Because I had little to none experience in C# and cloud development, in particular, I had to acquire knowledge in both these domains. Learning C# was not that difficult because the programming concepts are similar as in Java, which I have experience in, and Microsoft has great documentation for the language. By using the ASP.NET Core framework with Azure as the cloud provider, it was pretty straightforward to deploy the application as a web-service.

3.2.3 Treatment Validation

When validating the treatments, the designer has to investigate whether it is likely that the goals of stakeholders would be achieved if the artifact was implemented (treatment implementation is the fourth step of the engineering cycle). Note that the word implementation in this context means transferring the artifact into the real world. If the implementation of the artifact is not considered to contribute to the goals, a new iteration of the design cycle is executed. As the design cycle can be iterated many times, not all requirements must be treated at once. Hence, one can apply more treatments over time until the implementation of the artifact will realize the goals of the stakeholders.

To validate the treatments, I used NUnit[1] tests or tested them manually with Postman[2]. NUnit is a framework for unit testing of .NET applications (I used C#) that is useful when the output is known for a given input. If the test class is well written, one can be quite confident that an algorithm produces the correct output for any valid input. However, applying unit testing does not always guarantee that the tested parts are free for bugs. Postman is a client to write and execute HTTP requests. As the system must be available as a web-service (see requirements in Table 4.2), I tested the endpoints of the service using Postman. To assess whether the implementation (transferring it to the real world) of the artifact would realize the goals of the stakeholders, I created requests for designs to different rooms and examined whether the proposed kitchen designs made sense. A final assessment should be done by a domain expert to ensure that the design suggestions are any good.

[1] https://nunit.org/
[2] https://www.getpostman.com/

Chapter 4

Solution

This chapter describes the solution of the application that proposes kitchen designing using case-based reasoning. Section 4.1 describes what requirements are, and Section 4.2 describes the requirements of the solution and how these were elicited. Section 4.3 specifies the commercial off-the-shelf (COTS) products the system uses. Section 4.4 describes the software architecture using the 4+1 view model by P. Kruchten [27], and at last, Section 4.5 discusses some issues of the solution.

4.1 Requirements

This section describes what software requirements are and how the requirements in this projected were elicited.

4.1.1 Classification

According to P. Clements et al. [26], requirements encompass these three categories:
1. Functional requirements. The functional requirements specify what the system must be able to do, and how to behave and react to runtime stimuli. The project owner, end users, and other stakeholders that are interested in the functionality of the program may contribute to the derivation of these requirements. Additionally, the required tasks not visible from a black-box perspective are also seen as functional requirements.

2. Quality attribute requirements. The quality attribute requirements define the overall qualification of the system or functional requirements. That the system must have a high degree of usability or be interoperable are examples of system

qualifications; that a function must respond to stimuli within a specified period or handle erroneous input are examples of qualifications of the functional requirements.

3. Constraints. A constraint is a decision about the architecture the architect must accept and incorporate in the design. Examples of constraints are: a specific programming language or framework must be used, the system must be available on Android or iOS, or the system must be interoperable with legacy software.

4.1.1.1 Architectural Significant Requirements

ASRs are not a category within requirements but a collection of the requirements that have a significant impact on the architecture. If the absence of a requirement may result in a drastically different architecture, then the requirement is an ASR. Functional requirements cannot be ASRs because they do not specify anything about how the functionality is implemented—they only describe what the system must be able to do. Both quality attributes and constraints have the potential to affect the final design of the architecture significantly, and hence they can be ASRs.

4.1.2 Requirements Elicitation

A textbook by Zowghi and Coulin [28] reviews the state-of-the-art (2006) requirements engineering (RE) process research, key areas of RE, and presents empirical evidence and experience from practice in the industry. They define requirement elicitation as the process of seeking, uncovering, acquiring, and elaborating requirements. The goal of the process is to uncover and understand the users' needs and communicate these to the developers. They discuss elicitation approaches such as interviews, questionnaires, task analysis, domain analysis, introspection, group work, brainstorming, etc. In this project, the requirements were acquired through interviews and group work.

4.1.2.1 Interviews

Interviews can be performed in three different ways: structured, semi-structured, or unstructured. Structured interviews require the interviewer to have knowledge of the subject so that questions can be formed in advance. Unstructured interviews are preferred if the purpose of the interview is to not only elicit the requirements but gain critical knowledge of the application domain.

I performed a semi-structured interview with an experienced kitchen designer at Røskaft Kjøkken in Trondheim, i.e., I prepared a few questions but let the in-

terviewee speak freely. This helped me to gain insight into how a designer thinks, what information they require from their customers and thoughts about autoplanning of designs. The interview revealed that a designer needs to know, not surprisingly, the layout of the kitchen and position of walls, windows, drain, and power outlets. The interviewee made it clear that designers work differently in how they transform the users' needs into a final design—some start with a pen and paper sketch, some design directly in Winner Design, some starts with a guide through the kitchen showrooms, and so on. Using the autoplanner must be more efficient than designing from scratch to be usable by the designer.

4.1.2.2 Group Work

Group work promotes cooperation between the stakeholders. The effectiveness depends on the cohesion within the group, i.e., the ensemble of the participants have expertise in the whole application domain. Additionally, it is critical that each group member is allowed to speak freely and express their opinions.

The group work in this project consisted of the development manager and two developers at Compusoft, and me as the role of an architect and developer. The two developers work in different application domains; one works with the development of Winner Design, and the other mainly works with cloud-based development. The development manager is the project owner and has a great influence on high-level requirements.

One of Compusoft's goals is to make their applications available in the cloud. Winner@Web is a lightweight planning version of Winner Design that lets users design their kitchens online, and they offer cloud-based storage of designs. It is required that this project, i.e., the autoplanning feature, is designed as a cloud application. Because the devlopers at Compusoft already use and have experience in the Microsoft ASP.NET—a framework for cloud-based development—it is both wanted and logical to choose this framework.

4.2 Requirements of the Solution

This section lists and categorizes the elicited requirements of the solution in this project. The ASRs, which are a subset of the quality attributes and constraints, are listed in Table 4.1.

4.2.1 Quality Attributes

P. Clements et al. [26] discuss a comprehensive list of quality attributes and describe the most popular ones in detail, which are availability, interoperability,

ASR	Description
1	CBR must be used to propose designs.
2	The application must be available as a web-service.
3	ASP.NET Core must be used to develop the web-service.

Table 4.1: ASRs

modifiability, performance, security, testability, and usability. As the kitchen designer made clear in the interview, the autoplanning feature must be more efficient than designing from scratch—hence, performance is required for the solution. Because end users are going to use the autoplanner, usability is also important. Attributes such as security, modifiability, and availability are relevant but are not considered as critical in the prototype.

4.2.1.1 Performance

The stimulus in the general performance scenario is the arrival of a periodic, sporadic, or stochastic event, and the system responds by processing it. Performance can be measured by the latency (time to process an event), whether it is processed within a deadline or not, the throughput, jitter (variation in latency), and miss rate. For the autoplanning feature, latency and throughput are critical. If it takes too much time for the CBR to return solutions, then it is likely that the perceived usefulness of the feature will diminish. If it turns out that it is time-consuming to propose solutions, then the client should at least be informed by this before the task is executed. To test if the system is more efficient than designing from scratch, one can compare the time used to design the kitchen manually versus using the autoplanner. The outcome of the test is not final because the quality of the design must be acceptable as well—time consumption is not the only critical factor.

4.2.1.2 Usability

The autoplanning feature is supposed to make the design process more efficient for professional designers, and enable non-professionals to generate their own designs. P. Clements et al. [26] define the stimulus of the general usability scenario to be an action where the end user wishes to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or configure the system. The system should either provide the user with needed features or anticipate the user's needs as the response to the stimulus. The usability of the system is somewhat connected to the performance of it. If the design process is slower

ID	Description
1	CBR must be used to propose designs.
2	The application must be available as a web-service.
3	The ASP.NET Core framework must be used to develop the web-service.

Table 4.2: Constraints

than designing from scratch, the perceived usefulness of the system is low for professionals. For a non-professional, however, the performance is not that critical because they are not able to design kitchens themselves. Of course, if it takes too much time, the non-professional will also abandon the process.

Consistency in software design lets users that are known with one application reuse their experiences when trying to learn a new one. For instance, the design of every Apple product is similar which makes it easier for a user to learn how to use a new Apple product if they are familiar with one of their other products. Drawing the room for the autoplanning system should be similar as in Winner Design because a designer familiar with this program can use the autoplanner without learning a new design tool.

4.2.2 Constraints

The purpose of the master thesis is to research how CBR can be used to propose kitchen designs. Hence, the first constraint is that the solution must use CBR. As the outcome of the group work shows, the application must be available as a web-service and implemented using the ASP.NET Core framework. The constraints are listed in Table 4.2.

4.2.3 Functional Requirements

The functional requirements of the system are listed in Table 4.3. They are separated into three different categories, namely Client, API, and CBR. The client requirements specify the necessary functionality to construct a CBR problem. The API requirements specify needed functionality to use the case-based reasoner over the web. And at last, the CBR module must at least be able to perform the retrieve and retain processes of the CBR-cycle.

FR	**Description** — It must be possible to ...
1.1	draw the room that shall be furnished.
1.2	select which walls one wants to furnish.
1.3	select desired shape of the design (see Section 2.5.1).
1.4	search for designs suggestions.
1.5	import the chosen design to the room.

(a) User requirements

FR	**Description** — It must be possible to ...
2.1	upload a solution (.json extension).
2.2	upload a kitchen design (.drw extension).
2.3	upload a kitchen design preview (.jpeg extension).
2.4	download a solution.
2.5	download a kitchen design.
2.6	download a kitchen design preview.
2.7	delete a solution.
2.8	delete a kitchen design.
2.9	delete a kitchen design preview.
2.10	retrieve solutions for a problem.

(b) Web API requirements

FR	**Description**
3.1	The reasoner must be able to retrieve solutions.
3.2	The reasoner must be able to retain solutions.

(c) CBR requirements

Table 4.3: Functional Requirements

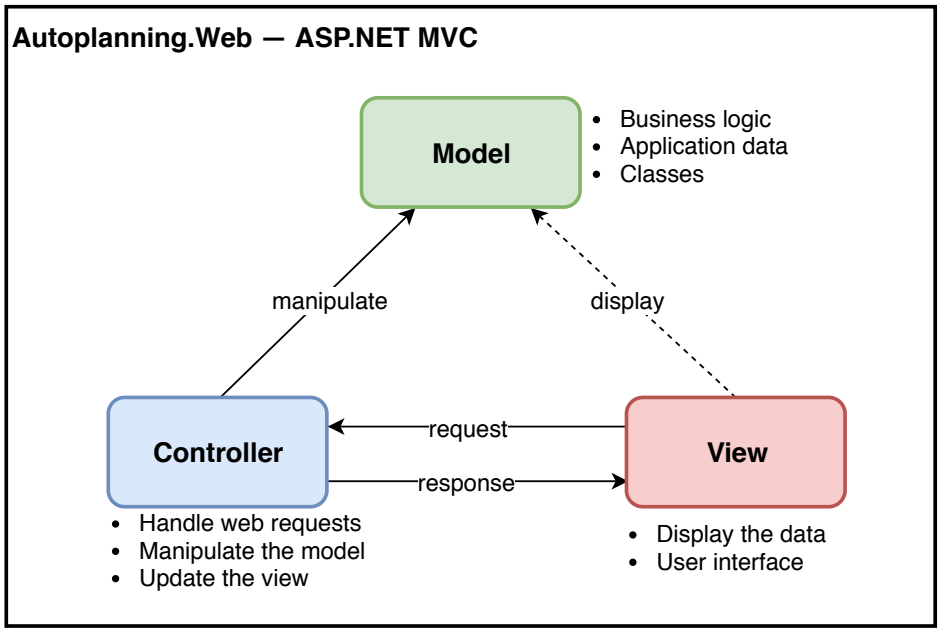


Figure 4.1: ASP.NET MVC

4.3 COTS

COTS is short for commercial off-the-shelf which are third-party hardware or software that can be used to solve a problem or complete a task. These can be open source, and hence free, but may also come at a cost. In this solution, these four COTS products are used: ASP.NET Core, Azure Storage, Azure CosmosDB, and Azure App Service.

4.3.1 ASP.NET Core

The ASP.NET Core framework is free to use. It supports three different programming languages namely C#, F#, and Visual Basic. A natural choice in this project is C# because the developers at Compusoft is familiar with this language and I am experienced in Java—C# and Java are similar which makes it easier to learn the other if one master one of them. The framework is designed to make use of the Model-View-Controller (MVC) architectural pattern.

4.3.1.1 ASP.NET MVC

Figure 4.1 shows how the three parts (Model, View, and Controller) communicate with each other and their responsibilities.

The model contains business logic, the application data, and the classes. When using an external database or storage, it is the model's job to read and store data. The controller transforms user input or events into method calls that are passed to the model. In the context of web services, the controller(s) implements the server endpoints which are reachable by HTTP requests. The ASP.NET framework handles the routing of requests (mapping from URL to method in the controller) as well as functionality to return the response. The view is the user interface—typically a website for web services. One data model can have zero to many views; each displays the same data but from a different point of view. For instance, the data can be displayed as a histogram, bar chart, pie chart, etc.

The advantage with MVC is that it applies separation of concerns, i.e., each part is responsible for a distinct task, and a modification in one of parts should not propagate (too much) to other. Significant changes in the model would, of course, impact the controller but changes to the view would not affect the model. Because the views can be modified without side-effects, they can be optimized to ensure high usability without changing anything of the back-end. This is significant because as we saw in the previous section, usability is a requirement of the solution.

4.3.2 Azure Blob Storage

Azure Blob Storage is a cloud-based storage offered by Microsoft that can store unstructured data, and the solution in this project will use this storage for previews and drw files—Blob is short for Binary Large OBjects. The maximum storage capacity for an account is 20 PB, and the storage can handle up to 20,000 requests per second. The egress limit, i.e., the maximum transfer rate from the server to the client, is 50 Gbps, and the ingress limit, i.e., the maximum incoming transfer rate the server can handle, is 10 Gbps. To reduce latency, Microsoft offers premium storage that uses SSD (solid-state drives), or one can enable Azure CDN (content distribution network) that reduces latency by having the data physically closer to the client. It is reasonable to assume that the performance of this storage will not be a bottleneck to the performance of the autoplanner. For more information about Azure Blob Storage see their website[1].

4.3.3 Azure Cosmos DB

Azure Cosmos DB is a globally distributed database service that supports document, key-value, wide-column, and graph databases. The CBR case base in the solution will use a document database—MongoDB to be more specific. Microsoft

[1] https://docs.microsoft.com/en-gb/azure/storage/blobs/storage-blobs-overview

guarantees a latency <10 ms for reads and <15 ms for indexed writes operations at the 99th percentile, which means only one percent of the requests exceed 10 ms for reads or 15 ms for writes. To improve performance, MongoDB uses shards that distribute documents in a collection according to some shard key. This key should be chosen carefully such that the documents are evenly distributed among the shards. Cosmos DB uses a currency called request units (RUs) and must be reserved by a database. Reading a 1 kb document costs 1 RU and writing a 1 kb document costs 5 RU. For more information about Azure Cosmos DB see their website[2].

4.3.4 Azure App Service

Azure App Service is a service for hosting of web apps, REST APIs, and mobile back-ends. It can provide authentication, security, load balancing, out and up-scaling, and it can integrate the development and deployment via its DevOps service. There are various pricing tiers, and one can choose the appropriate one according to the requirements of the application. The cheapest option is suitable for development and testing, while the premium options are more suitable for high performance and security applications. Scaling up means that the app is backed with better hardware, and scaling out means that one creates more instances of the app. For more information about the Azure App Service see their website[3].

4.4 Architectural Views

This section uses the 4+1 model by P. Kruchten [27] to describe the architecture of the solution. It consists of the *development*, *logical*, *process*, and *physical* view where each one describes the system from a different viewpoint. The ”plus one” contains the *scenarios*, also known as user stories. It is not considered as an independent view because it does not give any extra information—its purpose is to explain the behavior of the program to stakeholders not familiar with or interested in the details of the development.

4.4.1 Development View

The purpose of the development view is to visualize the relations between the different modules and submodules in the system and their dependencies. The package diagram in Figure 4.2 shows that four new packages will be created in this project namely Autoplanning.Web, KitchenCBR, and their test packages.

[2]https://azure.microsoft.com/en-us/services/cosmos-db/

[3]https://azure.microsoft.com/en-us/pricing/details/app-service/windows/

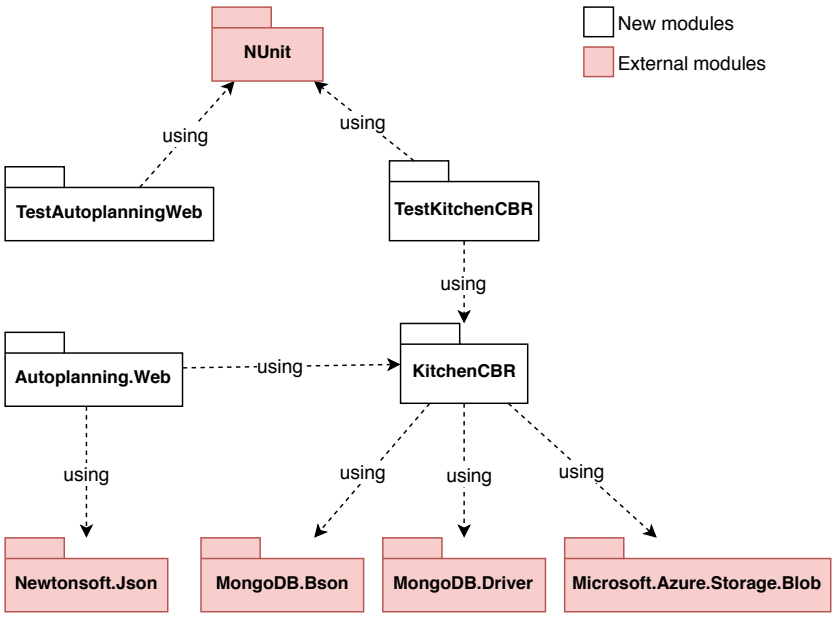


Figure 4.2: Package diagram

TestKitchenCBR and TestAutoplanningWeb test the KitchenCBR and Autoplanning.Web packages respectively using the NUnit framework.

The KitchenCBR module is responsible for performing the CBR methodology; in the first version of the program, it will retrieve solutions from the case base and retain new ones—the reuse and revise step of the cycle must be performed manually by the user. Such a system is called a retrieve-only CBR. If possible, the module will be responsible for the reuse of designs in later releases. I.e., the system inserts a design into the given room.

The CBR methodology requires an implementation of a case base but does not specify which technology to use. MongoDB is a NoSQL technology which provides high performance, scalability, and modifiability. MongoDB stores BSON (binary JSON) documents and does not require any information about the internal structure of them. An advantage of this design is that the representation of the cases can be modified without changing anything with the database setup. With a SQL database, one will have to add attributes to an existing table or create a new one if there are significant changes to the case structure.

The case base will be realized using MongoDB. Hence, the KitchenCBR module depends on the MongoDB.Bson and MongoDB.Driver libraries. The Bson module provides the Binary JSON (Bson) class, and the Driver module offers functionality to communicate with the database.

The design preview (.jpeg) and CAD file (.drw used in Winner Design) should be stored outside the case base to reduce the size of a case. This will improve

efficiency because working with smaller cases is faster, and the preview and CAD model will anyways not influence the output of the CBR retrieval.

The project will use Microsoft Azure Storage, a cloud-based solution, to store previews and binary drw files. The Microsoft.Azure.Storage.Blob module implements methods to interact with the storage unit and uses BLOBs (Binary Large OBjects) for uploading and downloading documents. Microsoft has detailed documentation on how to use the library in C#.

The Autoplanning.Web module implements the web endpoints of the CBR application by using the ASP.NET Core framework. This module enables communication between the client and the KitchenCBR module with HTTP requests. As we saw in Section 4.3.1.1, the ASP.NET framework has built-in support for the MVC pattern. The web package depends on the KitchenCBR package because the model uses the KitchenCBR methods. Newtonsoft.JSON is used to convert JSON-objects to and from .NET (C#) objects.

In the first version of the system, the web service is only an API that lets the client communicate with the KitchenCBR. This means that the client can send a request to the API to retrieve solutions, but needs to handle the response themselves. The idea is that Winner Design is used to draw the room, send the request to the API, and display the returned solution(s). It is desired that the application can be available as a stand-alone web service without the usage of Winner Design as a supportive tool in later versions.

4.4.2 Logical View

The logical view focuses on the implementation of the system and contains class diagrams and interfaces. Developers can use this view to see which methods the interfaces exposes and the methods and properties of the classes. This section contains a detailed description of how the CBR cases are represented, which addresses research question 1.1 (see Section 1.6).

4.4.2.1 Interfaces

The KitchenCBR module exposes three interfaces namely ICaseBase, IStorage, ICbr (see Figure 4.3). ICbr is the main interface that is supposed to be used by external applications that want to access the case-based reasoner. The other two interfaces are related to storage of cases and binary data used by a CBR class. An advantage of this design is that the CBR class does not need to know how data are stored or what kind of technology is used. For instance, the case base can be stored in a local file system, in a MongoDB database, or in a MySQL database—switching from to another will not affect the CBR class because it relies on the interface of ICaseBase. The methods are marked as Async which means they

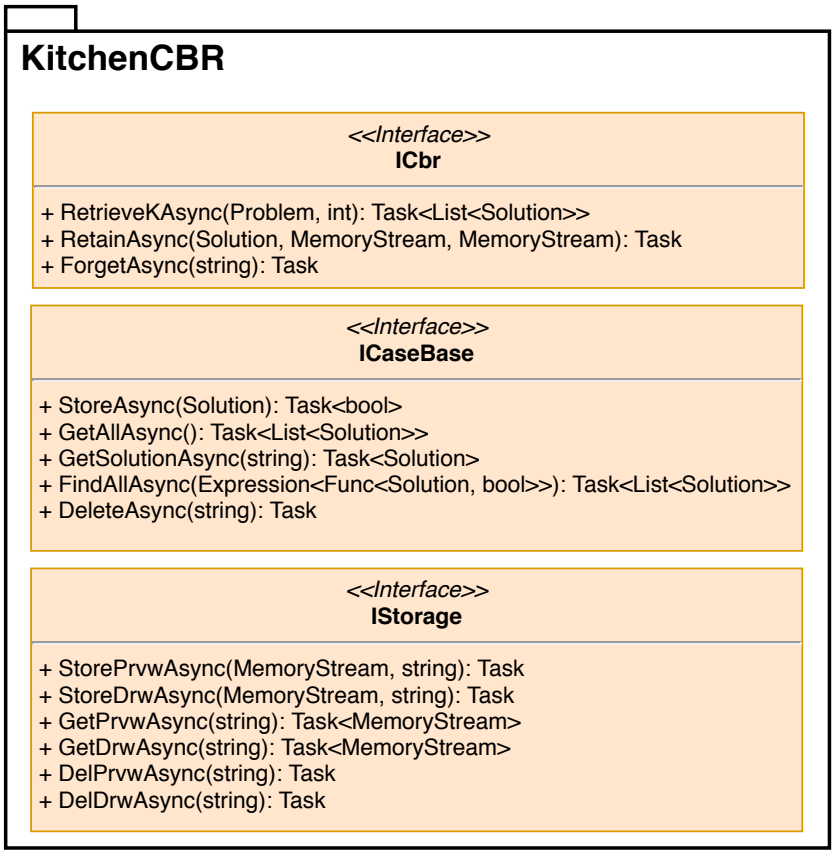


Figure 4.3: KitchenCBR interfaces

should be implemented using asynchronous programming. Async methods have a great increase in performance over synchronous ones when doing IO operations[4].

The interface of the web service, or API, is listed on the left-hand side in Figure 4.4 and the resources on the right-hand side. The last three endpoints execute CBR methods and the other gets, puts, or deletes resources on the server. There is an issue with this design regarding RESTful design which is discussed in Section 4.5. For a detailed explanation of each endpoint see Table 4.5.

[4]https://msdn.microsoft.com/en-us/magazine/dn802603.aspx accessed 27.03.19

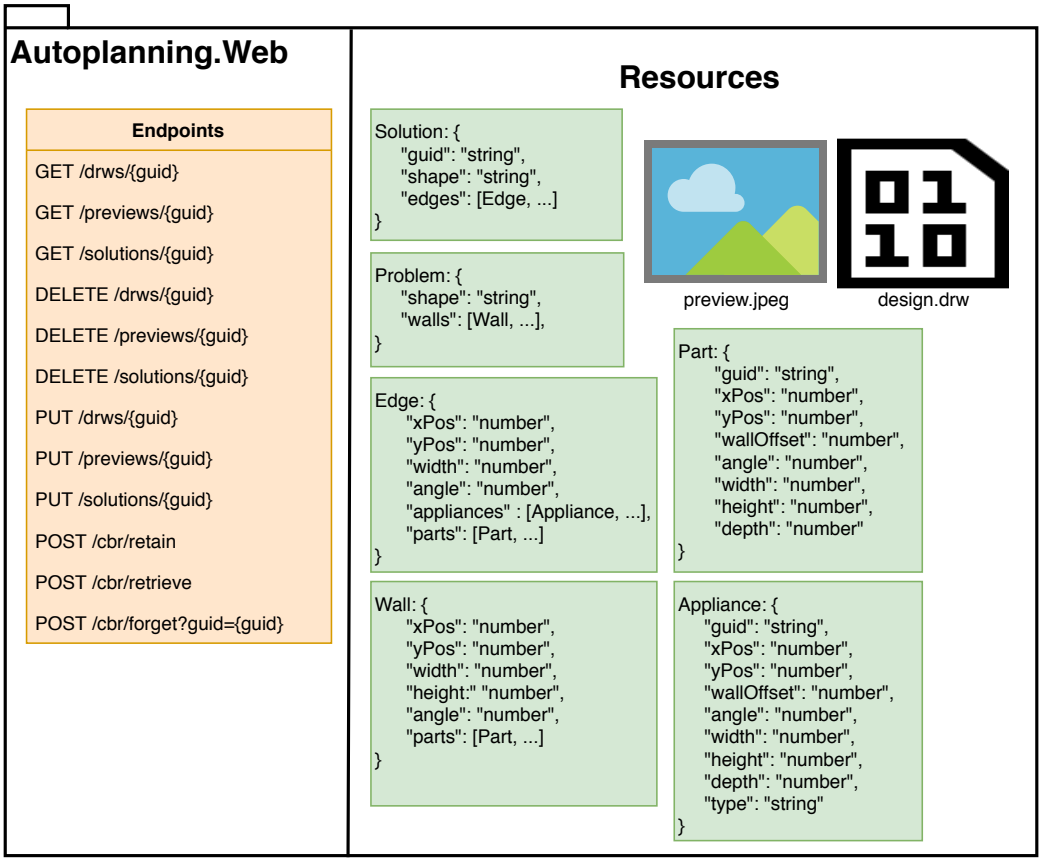


Figure 4.4: Autoplanning.Web endpoints and resources

GET /previews/{guid} – Get the solution's preview image	
Path vars	{guid} – The id of the solution
Request vars	none
Body	none
Response	200 OK image/jpeg
	404 NOT FOUND

(a) Get the solution's preview image

GET drws/{guid} – Get the solution's drw file	
Path vars	{guid} – The id of the solution
Request vars	none
Body	none
Response	200 OK application/octet-stream
	404 NOT FOUND

(b) Get the solution's drw file

GET solutions/{guid} – Get the solution in json format with this guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	none
Response	200 OK application/json {Solution}
	404 NOT FOUND

(c) Get solution

DELETE /previews/{guid} – Delete the preview with this guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	none
Response	200 OK
	404 NOT FOUND

(d) Delete preview

DELETE /drws/{**guid**} – Delete the drw with this guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	none
Response	200 OK
	404 NOT FOUND

(e) Delete drw

DELETE /solutions/{**guid**} – Delete the solution with this guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	none
Response	200 OK
	404 NOT FOUND

(f) Delete solution

PUT /previews/{**guid**} – Store a preview with the given guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	multipart/form-data [*.jpeg]
Response	201 CREATED
	400 BAD REQUEST

(g) Store preview

PUT /drws/{**guid**} – Store a drw with the given guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	multipart/form-data [*.drw]
Response	201 CREATED
	400 BAD REQUEST

(h) Store drw

PUT /solutions/{guid} – Store a solution with the given guid	
Path vars	{guid} – The id of the solution
Request vars	none
Body	application/json {Solution}
Response	201 CREATED
	400 BAD REQUEST

(i) Store solution

POST /cbr/retrieveK?k={k} – Execute CBR retrieval	
Path vars	none
Request vars	{k} – Number of cases to retrieve
Body	application/json {Problem}
Response	200 OK application/json { ”solutions”: [Solution, ...] }

(j) Execute CBR retrieval

POST /cbr/retain – Retain a new solution. Stores case, preview, and drw file.	
Path vars	none
Request vars	none
Body	multipart/form-data [*.json, *.jpeg, *.drw]
Response	200 OK
	400 BAD REQUEST

(k) Execute CBR retain

POST /cbr/forget?guid={guid} – Deletes case, preview, and drw file.	
Path vars	none
Request vars	{guid} – The id of the solution
Body	none
Response	200 OK
	404 NOT FOUND

(l) Execute CBR forget

Table 4.4: Autoplanning.Web API

4.4.2.2 Classes

As the class diagram in Figure 4.5 shows, IStorage is realized with AzureBlobStorage and ICaseBase with MongoDB. The six classes Solution, Problem, Edge, Wall, Appliance, and Part are used to construct the CBR cases—Solution and Problem are the two central ones, while the others are means to build them in an object-oriented fashion. Table 4.5 describes the attributes of all the classes.

Note: A GUID is a randomly generated 128-bit string, i.e., it can give up to 2^{128} unique identifiers. In practice, a collision will occur before all unique ones have been generated because of the birthday paradox. It will happen with the probability of 50% if one generates $1.17 * \sqrt{n} + 1$ identifiers, where n is the number of possible assignments (see Appendix A.1 for proof). If approximately $2.16 * 10^{19}$ Guids are generated, then there is a 50% chance that two of them are equal. However, this application will never even come close to this amount of identifiers.

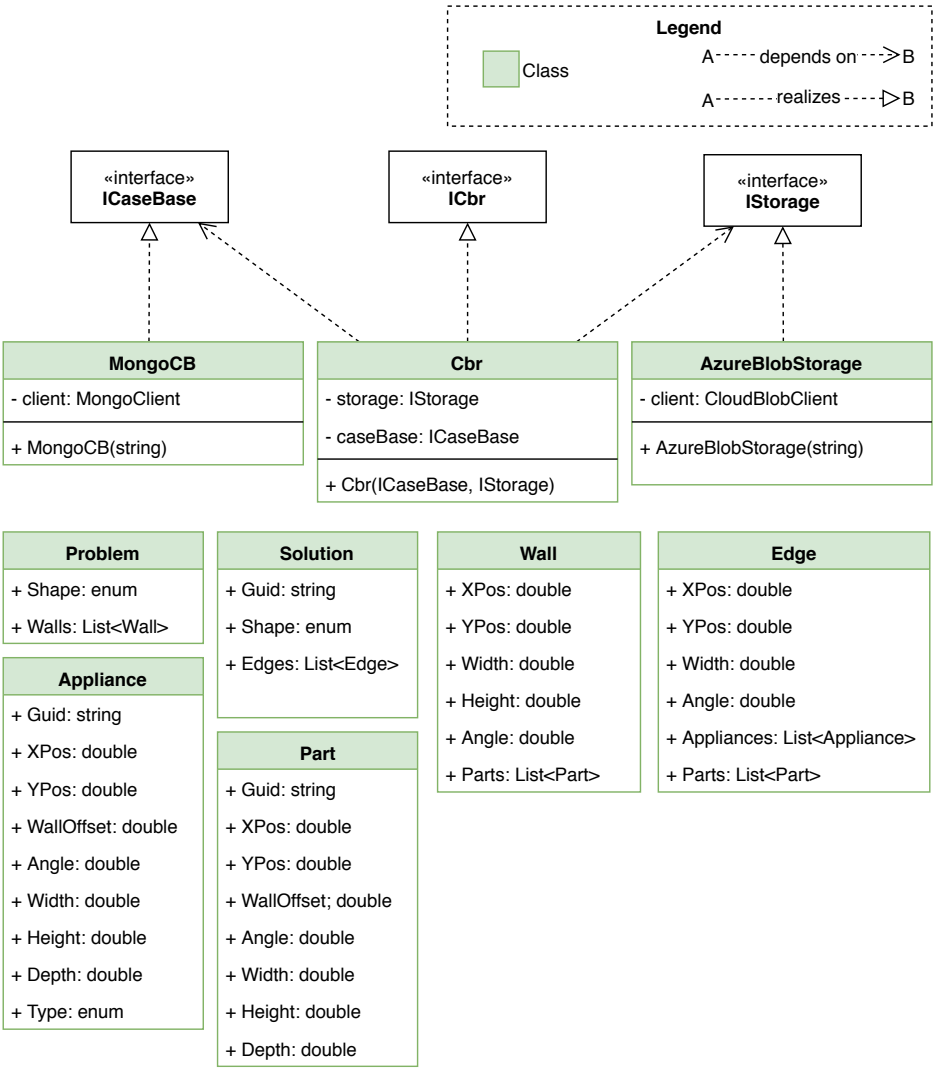


Figure 4.5: KitchenCBR class diagram

Solution

Attribute	Description
Guid	A globally unique identifier.
Shape	Enum type that represents the shape of the kitchen (see Section 2.5.1).
Edges	A list that contains the edges of the solution. An edge is a continuous line of furniture.

(a) Solution

Problem

Attribute	Description
Walls	A list that contains the walls of the kitchen that the user wants to furnish.
Shape	What kind of kitchen shape the user wants.

(b) Problem

Edge

Attribute	Description
XPos	X-coordinate of its start position.
YPos	Y-coordinate of its start position.
Width	The width of the edge.
Angle	The angle of the edge around the z-axis (cf. Figure 4.6).
Appliances	A list of appliances on this edge.
Parts	A list of furniture on this edge.

(c) Edge

Wall

Attribute	Description
XPos	X-coordinate of its start position.
YPos	Y-coordinate of its start position.
Width	The width of the wall.
Height	The height of the wall.
Angle	The angle of the wall around the z-axis (cf. Figure 4.6).
Parts	A list of furniture (windows, doors, ..) on this wall.

(d) Wall

Appliance

Attribute	Description
Guid	Identifier of this part (furniture) in the drw file.
XPos	The x position of this appliance.
YPos	The y position of this appliance.
WallOffset	Specifies the offset from the wall for this appliance.
Angle	Rotation of the appliance relative to the parent edge around the z-axis.
Width	The width of this appliance.
Height	The height of this appliance.
Depth	The width of this appliance.
Type	What kind of appliance this is (Refrigerator, Cooker Top, ...).

(e) Appliance

Part

Attribute	Description
Guid	Identifier of this part (furniture) in the drw file.
XPos	X position of the part.
YPos	Y position of the part.
WallOffset	Specifies the offset from the wall for this part.
Angle	Rotation of the part relative to the parent edge around the z-axis.
Width	The width of this part.
Height	The height of this part.
Depth	The depth of this part.

(f) Part

Table 4.5: KitchenCBR class descriptions

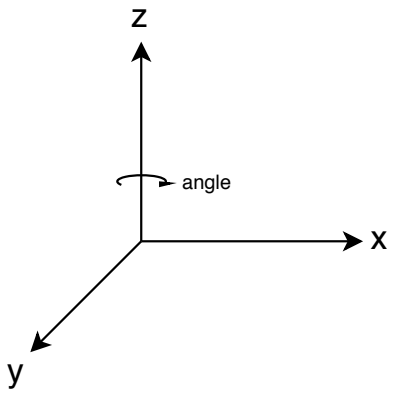


Figure 4.6: The coordinate system used in Winner Design is left-handed where the z-axis points up and the x-axis points to the right. Rotation in a left-handed coordinate system is clock-wise.

4.4.3 Process View

The purpose of the process view is to describe and visualize the flow of the program. Activity diagrams show the sequence of actions that are performed when an activity is initiated. These diagrams should be readable by stakeholders who are not interested in implementation details but rather in an abstract view of what happens when completing a task. Sequence diagrams can also show the flow of the program but are implementation specific; these diagrams display the method calls, which are valuable to stakeholders that want to understand how a task is solved at a more technical level—mostly used by the developers. This section includes both activity and sequence diagrams to satisfy both needs.

4.4.3.1 Activity Diagrams

Figure 4.7 depicts the activity diagram of the CBR retrieve process. First, the web service receives an HTTP request for designs suggestions for a given problem (room and requirements). If the request is invalid, an error message will be returned, and the process is terminated. Else, the web service initiates a task in the CBR module to retrieve designs. When the solutions are found, the CBR module ranks them from best to worst based on their utility for the specific problem. At last, the API returns the proposed solutions to the user.

Figure 4.8 shows the activity flow of the retain process. As for retrieval, the task is initiated by an HTTP request from the client, however, this time for storing a new case. An error message is returned if the request is invalid or incomplete, and the process is terminated. If it is valid, the CBR will first retain the case in the case base and afterwards store the preview and drw file in the storage. At last, a

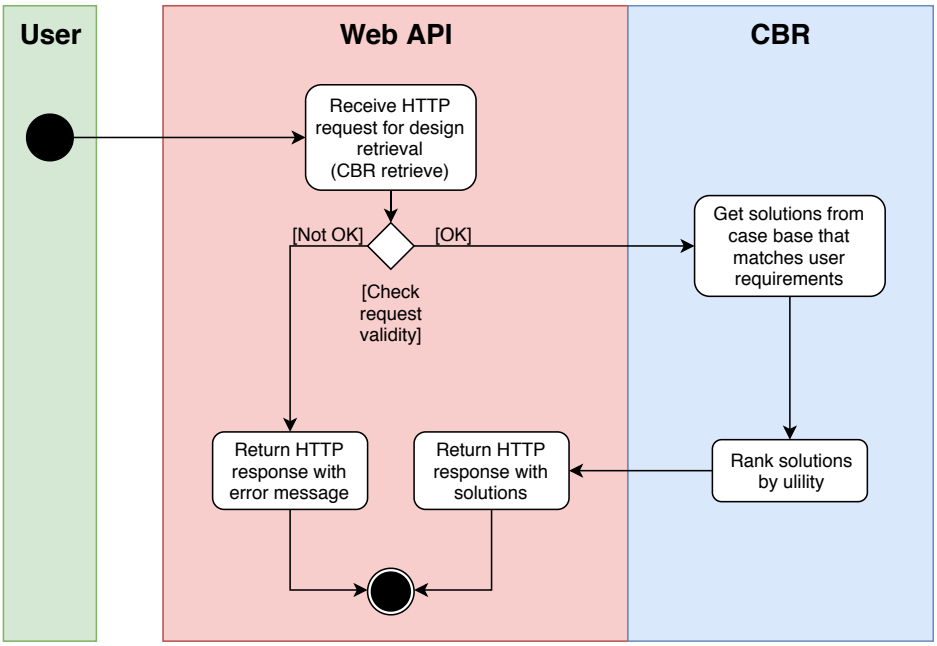


Figure 4.7: Activity diagram that shows the activity flow of the retrieval process

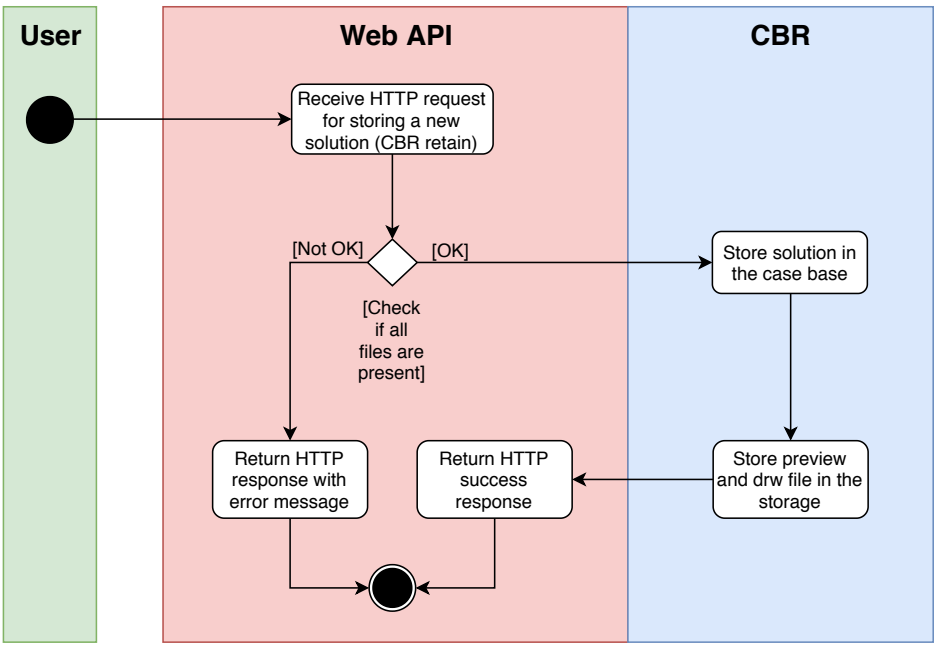


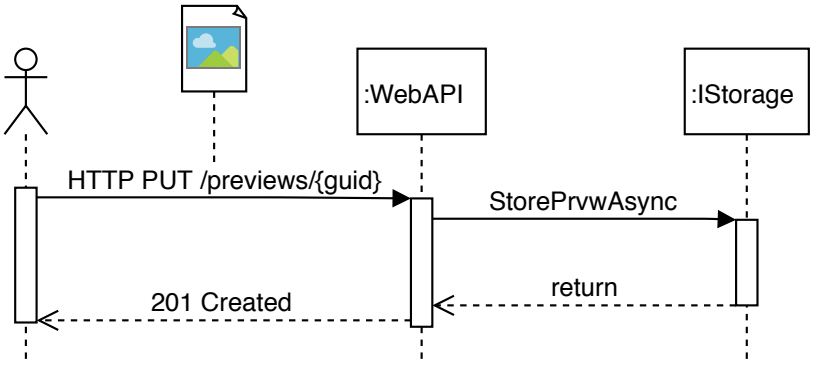
Figure 4.8: Activity diagram that shows the activity flow of the retain process

response is sent to the client that informs the action was successfully completed.

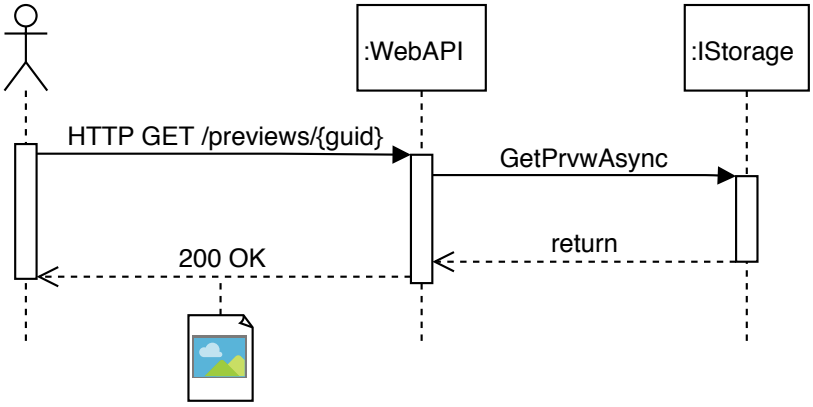
4.4.3.2 Sequence Diagrams

Figure 4.9 depicts three sequence diagrams of the possible interactions on the /previews endpoint on the API. As stated earlier, these diagrams show the method calls in the process flow. First, an HTTP request is sent to the server which includes one of the HTTP verbs PUT, GET, or DELETE together with a specific endpoint, which in this case is /previews/{guid}. The first part of the endpoint, namely previews, refers to a collection of preview images, and the guid part refers to one specific resource in this collection. When putting a new image to the server, the image must be added to the request body as seen in Figure 4.9a. The Web API calls the correct IStorage method based on the HTTP verb of the request—the name of the message maps directly to one of the methods exposed by the interface in Figure 4.4. The user is notified with a 201 Created response if the preview was uploaded successfully, a 200 OK response with the image if the GET verb was used, and a 200 OK if the deletion was successful.

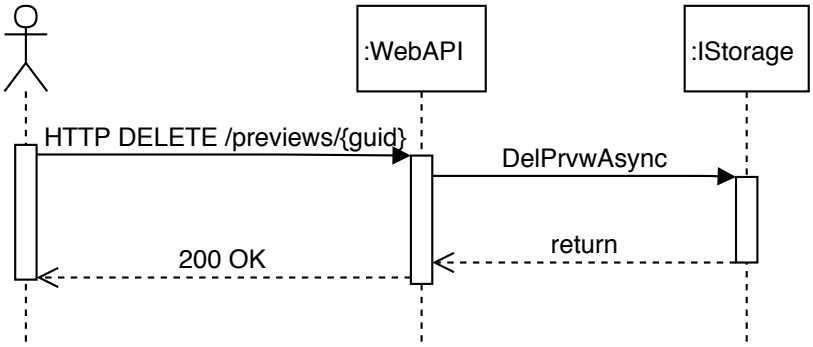
Figure 4.10 shows the process of retaining a complete set of a solution, preview and drw file on the /cbr/retain endpoint. The request is transmitted with the HTTP POST verb, and the request body contains the three files that will be uploaded to the server. First, a RetainAsync message is sent to the ICbr class which executes three asynchronously methods on the IStorage and ICaseBase classes. The sequence diagram shows the advantage of using asynchronous over synchronous methods; instead of sending a message and wait for it to complete before the next message is sent, one can send multiple messages at the same time and wait until all of them are completed. The user is notified with a 200 OK response if the request was processed successfully.



(a) Put a preview into the storage



(b) Get a preview from the storage



(c) Delete a preview from the storage

Figure 4.9: Sequence diagrams that show the program flow when putting, getting, or deleting previews from the server.

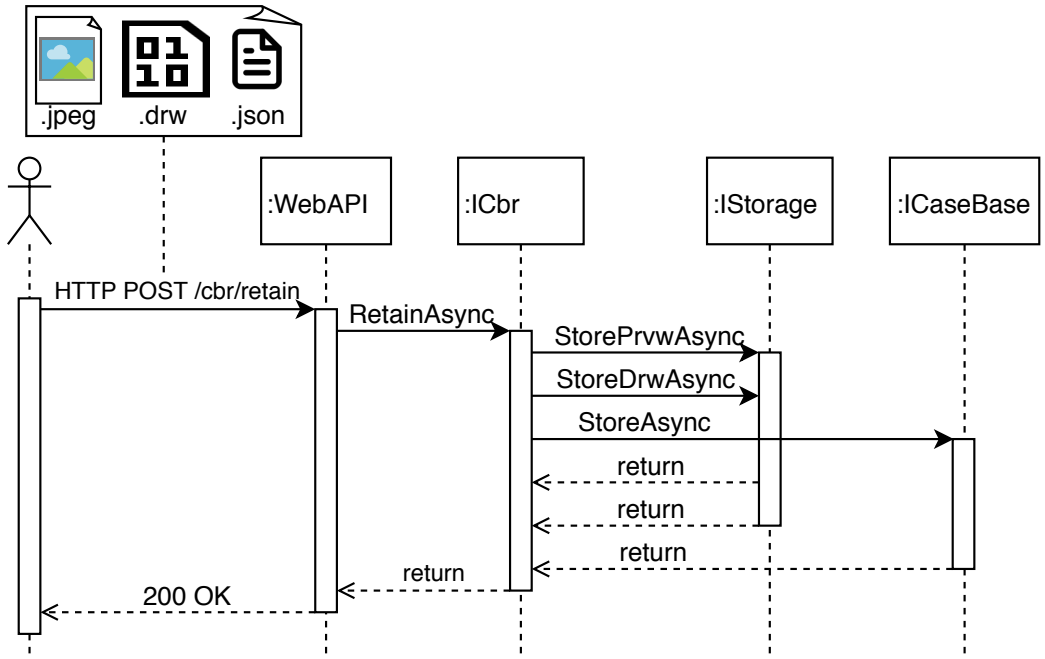


Figure 4.10: Sequence diagram that shows the program flow when using the /cbr/retain endpoint.

4.4.4 Physical View

The physical view is used to describe the physical deployment of the application by using deployment diagrams. These diagrams show the different nodes (computers, servers, databases), how these communicate with each other, and what piece of software which is deployed on them.

Traditionally, each development team had at least one person responsible for deploying new releases onto the nodes. With the DevOps (development operations) practice, however, writing code and deploying it have become a more integrated process. In the Azure DevOps environment, one connects a VCS (version control system) repository and each time a developer push changes to master, a build is executed. A release can be deployed when a trigger condition is met, for instance, each time a build has succeeded, it can be scheduled, or initiated manually. This enables the developers to focus more on developing the application and less on deployment.

Figure 4.11 shows the deployment diagram of the system. The KitchenCBR and Autoplanning.Web modules are deployed on Microsoft's Azure Web App Service—it is not known which physical computer(s) Microsoft use for running the application. The machine can be shared with other applications, or one can pay a premium to get higher performance. Their website describes the possible

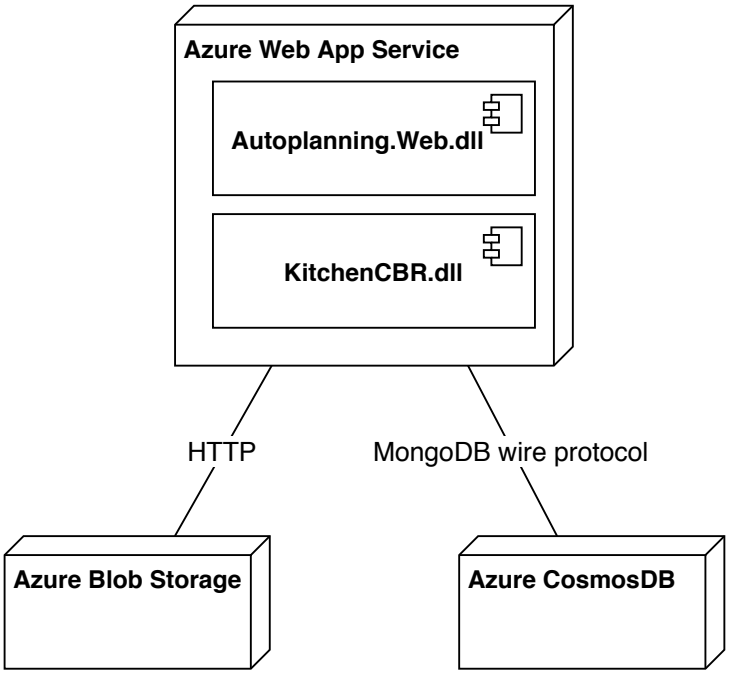


Figure 4.11: Deployment Diagram

options and the pricing of them[5].

The case base is implemented using MongoDB, but any technology could have been used. CosmosDB uses the MongoDB wire protocol to enable communication between the application and the database. Similar to the Web App Service, Microsoft offers various pricing levels depending on the desired features and performance of the database. The different options and their prices can be found on their website[6].

Azure Blob Storage is a service that enables storing BLOBs by using the HTTP protocol. Again, it is not known which physical machine or server that stores the data. The Blob Storage have different pricing levels that depend on the required storage capacity, and the performance of read and write operations. Prices can be obtained on their website[7].

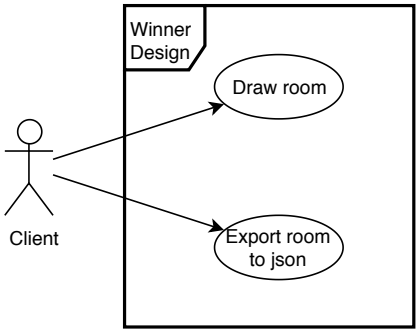
4.4.5 Scenarios

This view shows how the system appears for a user from a black-box perspective. It hides the implementation details, the activity flow behind the scenes, or anything

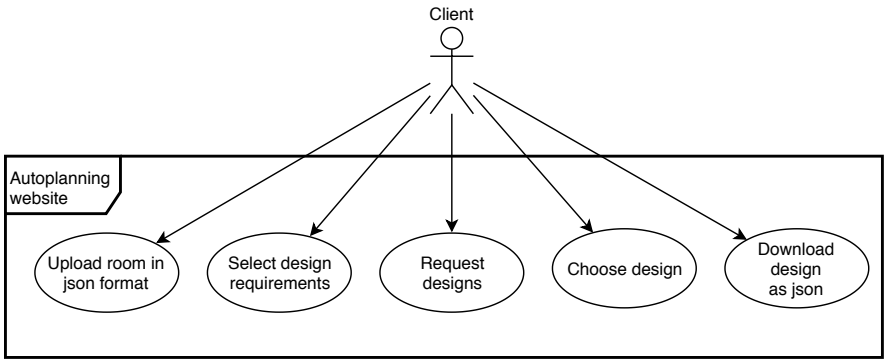
[5]https://azure.microsoft.com/en-us/pricing/details/app-service/windows/ accessed 25.03.19
[6]https://azure.microsoft.com/en-us/pricing/details/cosmos-db/ accessed 25.03.19
[7]https://azure.microsoft.com/en-us/pricing/details/storage/blobs/ accessed 25.03.19

else which is not visible for the user.

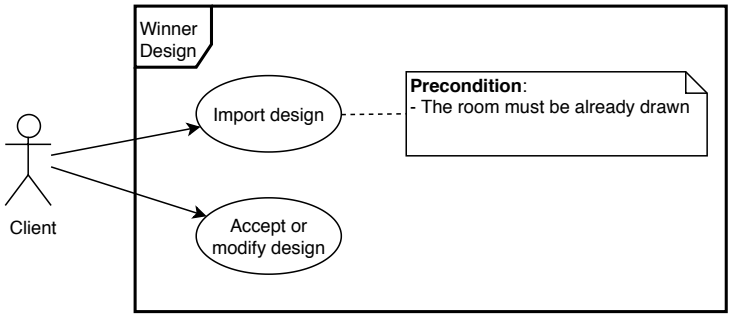
The diagram in Figure 4.12 shows the design process from start to end and separates the scenario into three sub-scenarios. First, the user has to draw his or her room in Winner Design and export it in a json format. Then, the user uploads this json file to the autoplanning website and specifies the requirements of the desired design and requests solutions. The system retrieves some designs with their associated previews that the user can choose and download. At last, the user imports the design in Winner Design which will apply it to the room they drew earlier.



(a) Draw room and export room in Winner Design



(b) Get designs from the autoplanner



(c) Import the design in Winner Design

Figure 4.12: This diagram shows a scenario that consists of three sub-scenarios. **(a)** The user draws the room. **(b)** The website finds solutions. **(c)** The user imports the selected design in Winner Design.

4.5 Issues

This section describes the identified concerns about the architectural design and the problem (autoplanning) in general.

4.5.1 REST vs. RPC

REST is an abbreviation for REpresentational State Transfer and was introduced by Roy Fielding in his doctoral dissertation in 2000 [29]. He describes a network-based architecture that is one of the most popular choices when designing web applications. There is an issue when deploying the CBR application on the web in a RESTful way because it violates the uniform interface constraint in the REST architecture. This interface specifies that all interfaces on the web service must fulfill these four constraints: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state. Design previews, drws, and solutions (the json representation of them) are resources and can fulfill all the requirements of the uniform interface. But how can the CBR methods in the cycle be executed in a RESTful way? For instance, searching for a resource in the REST architecture requires a SQL-like query, which means it is not possible to query a resource with another type. For instance, one cannot search for solutions with a problem in the CBR domain.

A remote procedure call (RPC) executes a procedure on a remote host. This seems to be the optimal choice when retrieving cases with the case-based reasoner, but it leads to an issue in the architecture. Mixing REST and RPC into a combined API can lead to confusion for both the client and the developers.

4.5.2 The Complexity of Autoplanning

In a perfect world, all sorts of kitchen designs can be proposed by the CBR—it can handle unusual room layouts and extraordinary designs. With the limited resources in this master project, however, it is critical to get some possible results rather than a production-ready software. A challenge is to find a case representation such that the CBR gets some results, and these designs must be either usable directly or after minor modifications. Making too high expectations of the CBR might result in too complex case representations and overthinking about odd room shapes and unusual designs.

4.5.3 Modifiability

This is somewhat related to the complexity of autoplanning because it requires careful thought about possible features and changes that can be implemented in

the future. Although getting some results are the top priority of the project, it is critical to design the system such that it is possible to build on it later. Let us say a prototype of the system successfully proposes suitable designs. What is now the cost to add a new feature or change the case representation? Is it enough to update the case representations and utility metrics? How do the changes impact the rest of the system? How can the existing cases be updated with new attributes? This concern should be a motivation to design the architecture after best practices and strive for modifiability.

Chapter 5

Results

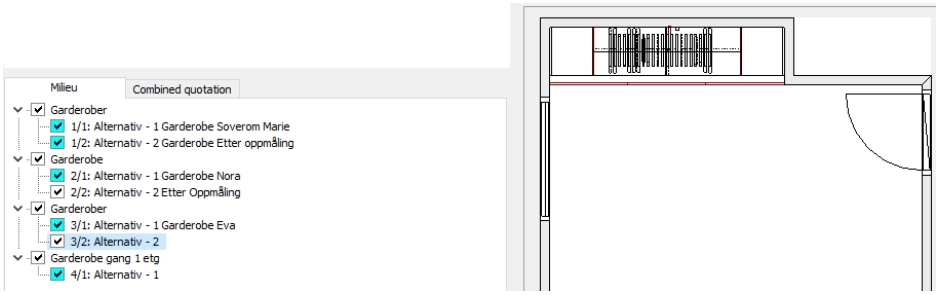
This chapter describes how the case base was constructed, or rather, populated with cases (kitchen designs). The CBR retrieval method of the system is explained, and its performance is analyzed. The front-end (website) of the application is presented, which describes how you can use the autoplanner to retrieve kitchen designs. At last, there are some experiments to show what the autoplanner is capable of doing.

5.1 Building the Case Base

A case-based reasoner without a case base cannot propose any solutions, because it can only propose solutions from its memory. K. Richter [12] discussed in her paper why case-based design applications seldom succeeded outside the academic field. The main reason for this was the lack of accessibility and acquisition of knowledge. With this, she means access to existing designs and the creation of new.

I had access to a database that contained 1853 Winner Design projects for the construction of the case base in this project. Additionally, each project could have multiple alternatives—an alternative is one design—where each one was a candidate for the case base. The true number of usable kitchen designs is uncertain because some projects had zero useful designs while some had several. Figure 5.1 shows two projects where one has plenty of kitchen designs, while the other has none. As Winner Designs is capable of designing kitchens, bathrooms, and wardrobes, some projects may not have any kitchen designs at all (as in Figure 5.1b). However, I assume that the number of available designs is more than enough to build a sufficiently large case base. My co-supervisor Sindre Nyvoll created an export feature for Winner Design that identifies what kind of shape

(a) Project with plenty of kitchen designs



(b) Project for wardrobes

Figure 5.1: Winner Design projects

the kitchen has (One wall, two walls, L, etc.), exports the design to a JSON represented format as specified in Section 4.4.2, and uploads it to the case base by using the /solutions endpoint of the Web API. I exported 45 One Wall and 55 L-shaped kitchen designs with this extension. It took some days to export these because I had to make sure that each export was correct, such that the case base would not be filled with erroneous cases. The total size of the 100 cases adds up to 1.15 MB—which is approximately 11.8 KB per case.

5.2 The Retrieval Method

Retrieval is the first process of the CBR-cycle and takes a problem as input. First, the method queries the case base to get all cases with a kitchen shape that matches the required shape by the problem. This corresponds to the many-are-chosen principle by the MAC/FAC methodology introduced by K. Forbus et al. [11]. For instance, if the user wants an L-shaped kitchen, the system should not propose any other shape. Hence, it is unnecessary to evaluate solutions with an incorrect shape. A utility metric which calculates the utility of a solution to the problem is applied to all the selected solutions from the first stage. The k solutions (the user

(a) Solution uses plenty of the available space

(b) Solution does not make good use of the available space

Figure 5.2: Two different designs in the same room

decides k) with the highest utility are sorted from high to low, and the user can choose the desired solution.

5.2.1 The Utility Metric

One of the research questions of my thesis is how to retrieve cases efficiently. Having this in mind, I tried to design the utility metrics such that the computational cost for each measurement is as low as possible. The algorithm computes the average of two local utility metrics—one computes the ratio of the used area of the available area of the wall, and the other evaluates whether the properties of the kitchen triangle are preserved. These properties are described in Section 2.5.3. The general idea is that using more of the available space is better than leaving parts of the wall unfurnished. Figure 5.2 shows two different designs in the same room. It is reasonable to believe that the largest is the best one. Hence, the utility measure should give a higher score to the larger kitchen. The metric does not accept a large intersection between blocked regions of the wall and the solution—i.e., collisions between furniture and window, doors, etc. As a consequence, a utility of zero is given for solutions that do not fit the input room. The global utility of the kitchen is the average of the edge utility (how much space is used), and the quality of the kitchen triangle. For instance, if 60% percent of the wall is furnished, and the triangle formed by the work centers almost complies with the kitchen triangle (gives a triangle utility of 0.75). Then the global utility would be $0.5 * (0.6 + 0.75) = 0.675$ given for that solution. Code snippets for measuring the edge and triangle utitlities are listed in Appendix A.2.

5.2.2 Adaptations

Adaptation is typically applied after a solution is selected (see Figure 2.1). The retrieval method in this system, however, performs a few simple adaptations before measuring the utility of a solution. For instance, an L-shaped solution is translated into the corner of the input room before the utility is measured. Remember that one utility metric measured the ratio of used area of the wall. The value of this metric depends on the positioning of the furniture. Additionally, the solution must be placed correctly in the room before one can verify whether it collides or not with unavailable parts (doors, windows, etc.) of the wall.

One could argue whether these minor adaptations are significant enough to be labeled as CBR adaptations—the layout does not change at all. However, the positions of the parts are included in the case representation, and by modifying them, one is strictly speaking adapting the cases.

5.2.3 Performance

The performance of the retrieval method is one of the research questions in my thesis. Additionally, it is also a requirement of the system that it must be faster to use the autoplanner than designing kitchens from scratch (see Section 4.1). To measure the performance of the CBR retrieval, I will do two things. First, I will investigate how much time it takes to measure the utility of a solution. Then, I will evaluate the Time To First Byte (TTFB), which indicates the responsiveness of the web server—that is, I will assess the TTFB of the RetrieveK endpoint of the API.

First, the performance of computing the utility of one wall designs is measuring using the problem in Listing 5.1. The width of the wall is set to 10,000 mm so that the utility metric cannot break early because the kitchen design is wider than the room; if the width of the kitchen is wider than the wall, a utility score of zero will be given without further measurements. It took 3 ms to compute the utility of 45 one wall solutions—which corresponds to approximately 0.07 ms per case or 14 286 cases per second.

Listing 5.1: Problem: One wall shape

```
{
    "shape": "ONE_WALL",
    "walls": [
        {
            "xPos": 0,
            "yPos": 0,
            "width": 10000,
```

```
            "height": 2400
        }
    ]
}
```

The performance of measuring L-shaped kitchens is computed using the problem in Listing 5.2. For the same reason as with one wall designs, the width of the walls is set to 10,000 mm to make sure that the utility of all the cases is measured. It took 7 ms to compute the utility of 55 L-shaped kitchens—which corresponds to approximately 0.13 ms per case or 7692 cases per second. It is not surprising that it takes about twice as much time to measure the utility of an L-shaped kitchen compared to a one wall design. A likely explanation is that the algorithm to compute the ratio of used wall space and collision detection is executed twice instead of once.

Listing 5.2: Problem: L shape

```
{
    "shape": "L",
    "walls": [
        {
            "xPos": 0,
            "yPos": 0,
            "width": 10000,
            "height": 2400
        },
        {
            "xPos": 10000,
            "yPos": 0,
            "width": 10000,
            "height": 2400,
            "angle": 90
        }
    ]
}
```

So, the performance of the utility measure should be more than fast enough to propose designs quickly. Let us take a look at the TTFB of the RetrieveK endpoint. When there have not been any requests for a while, Azure CosmosDB (the case base) frees up some resources and goes to some sort of idle mode. This makes the first request a bit slower because there is some overhead in firing up the service again. The overhead varies, but it usually takes up to a second extra for the first request. If the database is not idle, TTFB for the RetrieveK endpoint is around

200–250 ms for the problem in Listing 5.1 and 250–300 ms for the problem in Listing 5.2—which means it takes around 4.4–5.6 ms per case the system has to reason about before it can return the results. If the retrieval method spends 5 ms on average per case, the autoplanner could theoretically retrieve 200 cases per second.

5.3 Front End

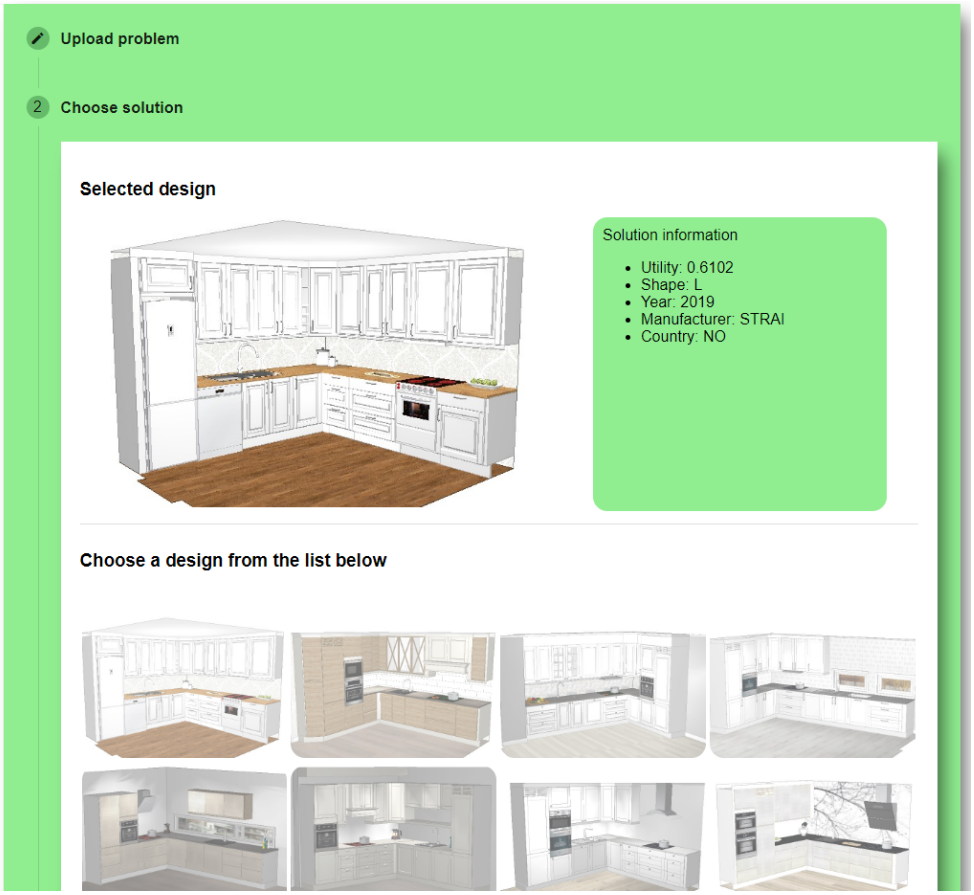
As promised in Chapter 4, the web service is available as a web service. The API can be accessed with Postman[1] or any other client that lets you create an HTTP request. For the sake of usability, I created a website that makes it easier to try out the autoplanner. If you visit http://autoplanning-webapp-dev.azurewebsites.net, you will see the landing page in Figure 5.3a. First, you will have to upload a Problem as a JSON file. You could use one of the problems in Listing 5.1, 5.2, A.3.1, A.3.2, A.3.3, A.3.4—you must copy the case and store it as a json file. Note that only One Wall and L shaped designs can be proposed at this time. By clicking the next button, the website will propose the best solutions based on their utility. Click on a preview in the list to show its utility and other information about the design. When you have chosen a solution, click next, and you will have the possibility to download the solution. There is no point in doing so without Winner Design installed because you cannot import it in any other program this program. If you had it, you could import the design and check how it looks in your kitchen.

5.3.1 Integrated Process in Winner Design

There is a feature in Winner Design that can use the autoplanner without taking the detour through the website. First, one draws the room. Then, one chooses which walls one would like to furnish—in this case, only one wall (see Figure 5.4a). The program will then access the retrieveK endpoint of the web service to get design suggestions and display these to the user (see Figure 5.4b). At last, the chosen kitchen design is imported automatically and placed on the specified wall(s) from the first stage (see Figure 5.4c).

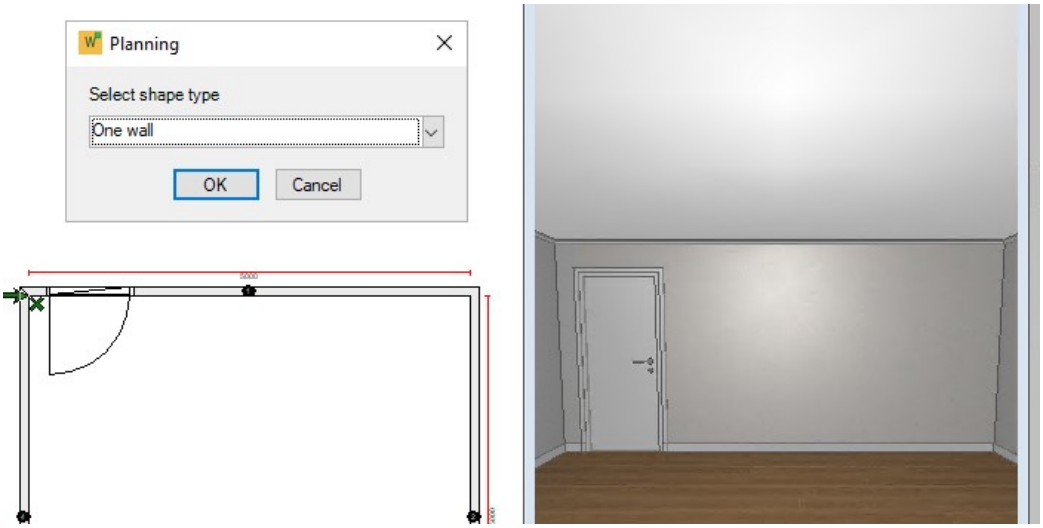
[1] https://www.getpostman.com/

(a) Autoplanning landing page

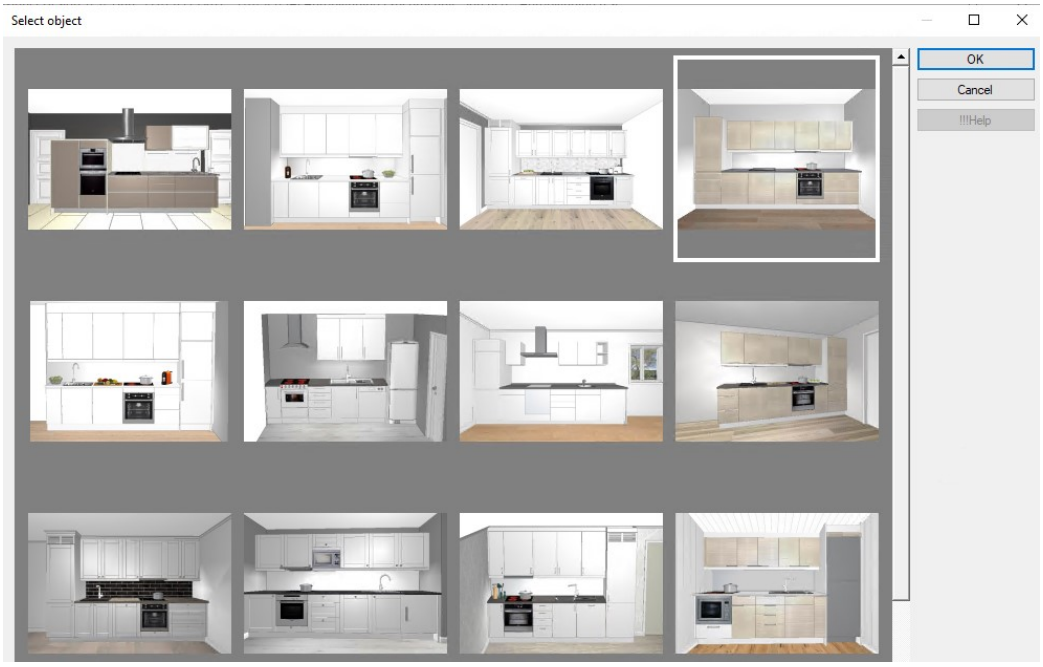


(b) Design suggestions

Figure 5.3: The Autoplanning Website



(a) Draw room and select wall(s) that should be furnished



(b) Select kitchen design

(c) Chosen kitchen design is imported automatically

Figure 5.4: Integrated process of autoplanning in Winner Design

5.4 Experimental

In this section, I will use four different problems to test the autoplanner. The two first problems are kitchens where the user wants a one wall design. The first room of these two does not have any obstacles—that is, the whole wall is available for furnishing. The other room has the same dimensions as the first but has a door on the left-hand side. The two last problems are kitchens where an L-shape is desired. As with the two first, one room has all the wall space available while the other has a door, and additionally, a window.

5.4.1 Problem 1: One Wall

Figure 5.5 shows a subset of the suggested solutions when retrieving cases with the problem in Listing A.3.1 in the Appendix. The empty room that should be furnished is displayed in Figure 5.5a among three of the design alternatives. The solution with the highest utility has a score of 0.8161, and as you can see in the figure, it fits quite well. The solution in Figure 5.5c uses most of the available wall area, but the appliances are placed too close to each other to get a good score for the kitchen triangle. Strictly speaking, it is not possible to construct a triangle from three points on a line. However, the same guidelines of the triangle can be used to evaluate whether the work centers are too close or too far away from

(a) Input room

(b) Utility 0.8161

(c) Utility: 0.4844

(d) Utility: 0.0875

Figure 5.5: Results for Problem 1

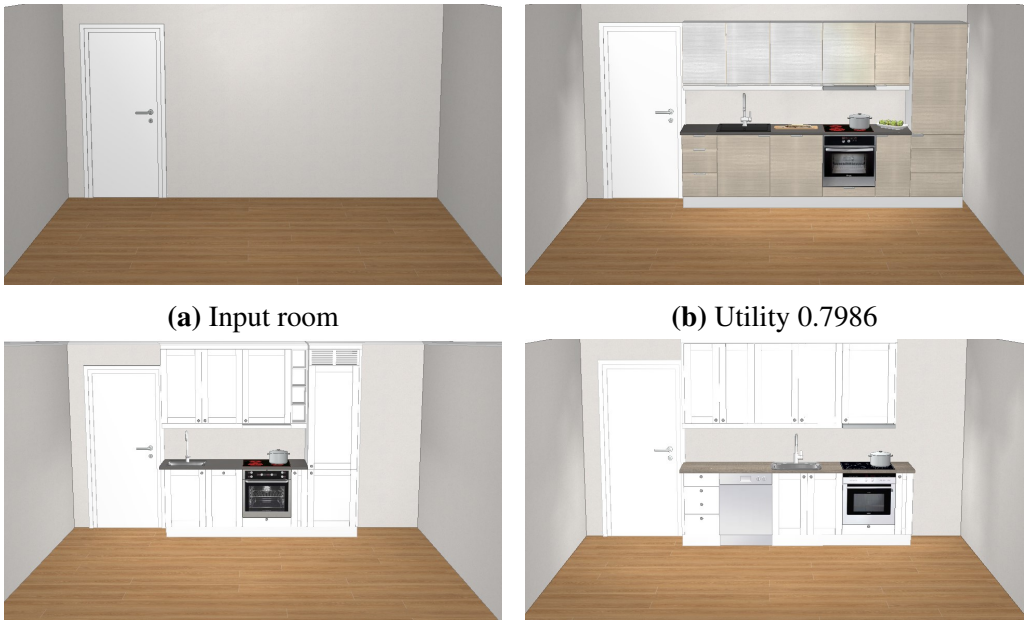
each other. Not surprisingly, the last kitchen has a low utility score because of its minimal size.

5.4.2 Problem 2: One Wall with Door

The result when retrieving solutions with the problem in Listing A.3.2 is quite similar to the result of problem 1 (see Figure 5.6). The dimensions of the room are the same but a door has been inserted on the left-hand side of the wall. Note that the highest rated design from the first experiment is not suggested because it is too large when there is a door in the room. The lowest rated kitchen looks quite similar to the second best but it has a significantly lower utility score because it has no refrigerator. The user can, of course, modify the chosen design to fit the room even better. For instance, for the lowest rated kitchen in this experiment, a refrigerator could be added to the right side of the oven.

5.4.3 Problem 3: L-shape

The problem in Listing A.3.3 seeks an L-solution on the two walls in Figure 5.7a. The utility of the suggested solutions in this experiment looks to be violating the idea that a larger kitchen is better than a smaller one. The highest rated kitchen for



(a) Input room **(b)** Utility 0.7986

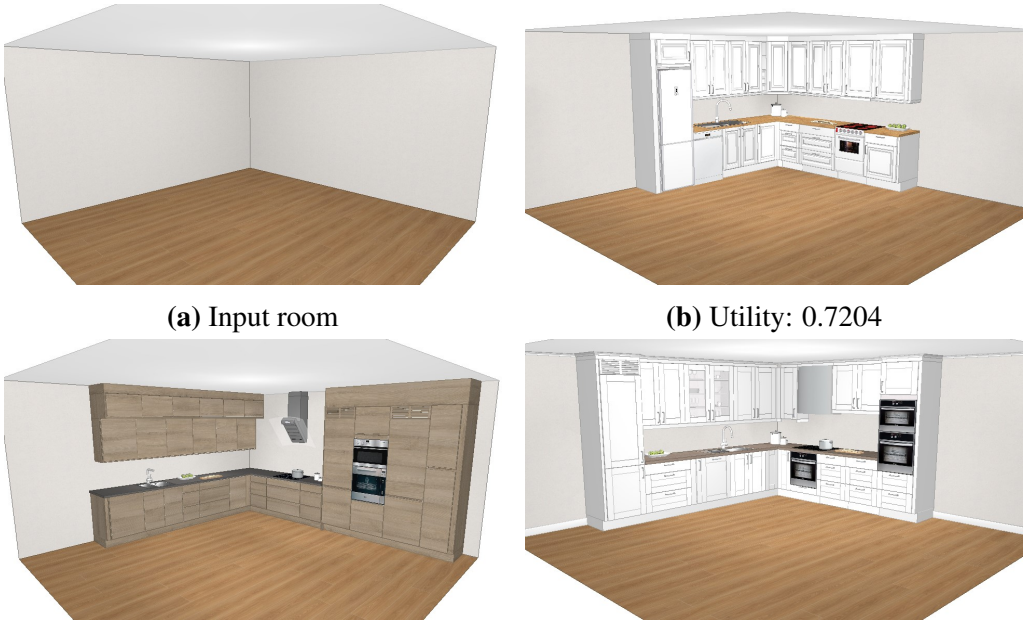
(c) Utility: 0.4288 **(d)** Utility: 0.2549

Figure 5.6: Results for Problem 2

Solution	Edge Utility	Kitchen Triangle Utility	Global Utility
(b)	0.4407	1	0.7204
(c)	0.5794	0.75	0.6647
(d)	0.4889	0.75	0.6194

Table 5.1: Utility calculation

this problem has a utility score of 0.7204 while the two other seems to be better, although their utilities are 0.6647 and 0.6194. Let us take a look into the reason behind the utility of each kitchen. Table 5.1 shows the edge utility and kitchen triangle utility for each of the three solutions. As can be seen in the table, the highest rated solution has the lowest edge utility but the highest for the kitchen triangle. Because the global utility is computed from the average of these two metrics, the kitchen in Figure 5.7b comes out on top. The weighting of the two utility metrics might have to be adjusted, such that the global utility indicates the usefulness of a kitchen to a higher degree. It is unlikely that the edge utility will get a full score because this means that 100% of the available wall space must be furnished, while a full score for the kitchen triangle is more common. Hence, the current weighting may be biased against designs that do not follow the triangle guidelines.



(a) Input room

(b) Utility: 0.7204

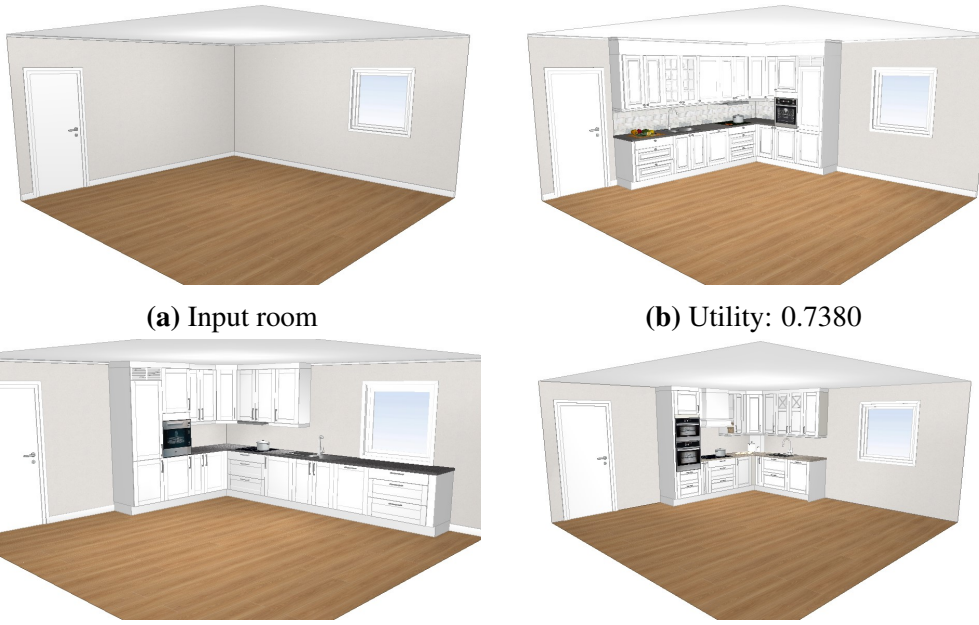
(c) Utility: 0.6647

(d) Utility: 0.6194

Figure 5.7: Results for Problem 3

5.4.4 Problem 4: L-shape with Door and Window

Figure 5.8 shows a subset of the suggested solutions when querying the case-based reasoner with the problem in Listing A.3.4. This experiment shows that the system is capable of suggesting designs that avoid unavailable parts of the wall. For instance, the kitchen in Figure 5.8b displays a kitchen that utilizes the area below the window. Again we see that the lowest rated kitchen in Figure 5.8d has a low utility score, not only because of its small size but because it has no refrigerator.



(a) Input room

(b) Utility: 0.7380

(c) Utility: 0.7280

(d) Utility: 0.1906

Figure 5.8: Results for Problem 4

Chapter 6

Conclusion

This thesis shows that it is possible to propose kitchen designs by using a case-based reasoning approach, which was the goal of this thesis. The three sub-research questions were: how to represent the cases—or kitchen designs, how to efficiently retrieve cases, and how to adapt an existing design for a new kitchen. Section 4.4.2 about the logical view of the architecture presents how the designs are represented, and here comes a short summary of it.

The design of a kitchen has one of the following shapes: One wall, two walls, L, U, or G. An edge corresponds to a line of furniture, so for one wall, there is one edge. For two walls there are two edges, and for L and U there are three edges—G solutions have four. Each edge has a list of parts (furniture) that are placed relative to that edge. The position of a part and its physical dimensions are stored in its representation. This is basically all the necessary data to be able to retrieve designs.

To be able to retrieve kitchens, a representation of the problem is needed as well. The representation of a problem is quite similar to the solution. It has an attribute that specifies the desired shape of the solution and a corresponding number of listed walls that should be furnished—i.e., if an L-solution is wanted, two walls must be provided in the problem representation. Each wall has a width and a height and optionally a list of parts that specify unavailable areas of the wall, typically doors and windows. The retrieval method uses two utility metrics: edge utility and kitchen triangle utility. The edge utility calculates how much of the available area of the wall is used, and the idea is that the more, the better. The utility of the kitchen triangle is based on to which degree the kitchen design complies with the design guidelines described in Section 2.5.3. Finally, the global utility of a solution to a problem is the average of these two metrics. The performance of the retrieval algorithm is analyzed in Section 5.2.3, and it is estimated that the CBR can retrieve around 200 cases per second. That includes getting the

cases from the case base and utility measurement. The CBR can calculate the utility of 14286 one wall designs per second and 7692 L-shaped solutions per second. Hence, most of the latency comes from getting them from the case base.

The autoplanner can be used in three different ways. The first a and most technical one is to communicate with the API manually. This can be achieved by using Postman or any other client to create HTTP requests. Secondly, there is a website that lets the user upload a CBR problem (room and requirements) and get design proposals in return. The last, and probably the best way, is to use the integrated feature in Winner Design (given you have a license for the software). Here you can draw the layout of the room, select walls that should be furnished, request design suggestions, select the desired design, and import the chosen kitchen—everything integrated into one process.

Bibliography

[1] Elsie De Wolfe. *The House in Good Taste*. The Century Co., 1913.

[2] Houzz and home - survey us. `http://st.hzcdn.com/static/econ/18HouzzandHome1.pdf`. Accessed 18.10.2018.

[3] Houzz and home - survey canda. `http://st.hzcdn.com/static/econ/HHCA18.pdf`. Accessed 18.10.2018.

[4] Houzz and home - survey global. `http://st.hzcdn.com/static/econ/Houzz&Home2018GlobalUSRenovationActivityUS.pdf`. Accessed 18.10.2018.

[5] Michael M Richter and Rosina O Weber. *Case-based reasoning*. Springer, 2016.

[6] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

[7] Mary Lou Maher and PEARL Pu. *Introduction to the issues and applications of case-based reasoning in design*. Lawrence Erlbaum Associates, Mahwah, NJ, 1997.

[8] Ralph Bergmann, M Michael Richter, Sascha Schmitt, Armin Stahl, and Ivo Vollrath. Utility-oriented matching: A new research direction for case-based reasoning. In *Professionelles Wissensmanagement: Erfahrungen und Visionen. Proceedings of the 1st Conference on Professional Knowledge Management. Shaker*, 2001.

[9] Mary Lou Maher and A Gomez de Silva Garza. Case-based reasoning in design. *IEEE Expert*, 12(2):34–41, 1997.

[10] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann, 1993.

[11] Kenneth D Forbus, Dedre Gentner, and Keith Law. Mac/fac: A model of similarity-based retrieval. *Cognitive science*, 19(2):141–205, 1995.

[12] Katharina Richter. What a shame-why good ideas can't make it in architecture: A contemporary approach towards the case-based reasoning paradigm in architecture. In *FLAIRS Conference*, 2013.

[13] Ann Heylighen and Herman Neuckermans. Dynamo: dynamic architectural memory on-line. *Educational Technology and Society*, 3(2):86–95, 2000.

[14] Yi-Kai Juan, Shen-Guan Shih, and Yeng-Horng Perng. Decision support for housing customization: A hybrid approach using case-based reasoning and genetic algorithm. *Expert Systems with Applications*, 31(1):83–93, 2006.

[15] Yi-Kai Juan, Sheng-Fen Chien, and Yi-Jhen Li. Customer focused system for pre-sale housing customisation using case-based reasoning and feng shui theory. *Indoor and Built Environment*, 19(4):453–464, 2010.

[16] Kyung-shik Shin and Ingoo Han. A case-based approach using inductive indexing for corporate bond rating. *Decision Support Systems*, 32(1):41–52, 2001.

[17] Peihua Song, Youyi Zheng, Jinyuan Jia, and Yan Gao. Web3d-based automatic furniture layout system using recursive case-based reasoning and floor field. *Multimedia Tools and Applications*, pages 1–29, 2018.

[18] Lap Fai Yu, Sai Kit Yeung, Chi Keung Tang, Demetri Terzopoulos, Tony F Chan, and Stanley J Osher. Make it home: automatic optimization of furniture arrangement. -, 2011.

[19] Peter Kán and Hannes Kaufmann. Automated interior design using a genetic algorithm. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology*, page 25. ACM, 2017.

[20] Peter Kán and Hannes Kaufmann. Automatic furniture arrangement using greedy cost minimization. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 491–498. IEEE, 2018.

[21] Ivan E Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[22] Cad software history. http://www.cadazz.com/cad-software-history.htm. Accessed 22.10.2018.

[23] D Taha, S Hosni, M Suermann, and B Streich. The role of cases in architectural practice and education–moneo: An architectural assistant system. *Proceedings of the ASCAAD*, 2007.

[24] National Kitchen and Bath Association. *NKBA Kitchen and Bathroom Planning Guidelines with Access Standards*. John Wiley & Sons, 2016.

[25] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[26] Paul Clements, Rick Kazman, and Len Bass. *Software Architecture in Practice*. Pearson, 2013.

[27] Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

[28] Didar Zowghi and Chad Coulin. *Requirements elicitation: A survey of techniques, approaches, and tools*, pages 19–46. Springer, 2005.

[29] Roy Thomas Fielding. Rest: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*, 2000.

Appendix

A.1 Proof of hash collision probability

The proof is extracted from the lecture notes in Cryptography at RWTH Aachen University[1].

10.1. SECURITY OF HASH FUNCTIONS 83

Proposition 10.3. *k objects are randomly put into n bins. Let $p_{k,n}$ denote the probability that no bin contains two or more objects (there is no collision). Then*

$$p_{k,n} = \frac{n(n-1)(n-2)\dots(n-k+1)}{n^k} \leq \exp\left(-\frac{k(k-1)}{2n}\right).$$

Proof.

$$\begin{aligned} p_{k,n} &= \frac{\text{number of collision free assignments}}{\text{number of all possible assignments}} = \frac{n(n-1)(n-2)\dots(n-k+1)}{n^k} \\ &= 1 \cdot \left(1-\frac{1}{n}\right)\dots\left(1-\frac{k-1}{n}\right) = \exp\left(\sum_{i=0}^{k-1}\ln(1-\frac{i}{n})\right) \\ &\leq \exp\left(-\sum_{i=0}^{k-1}\frac{i}{n}\right) = \exp\left(-\frac{k(k-1)}{2n}\right). \end{aligned}$$

In the proof we use, that $\ln x \leq x-1$ for $x \geq 0 \Leftrightarrow \ln(1-x) \leq -x$ for $x \leq 1$. □

The name „birthday paradox“ comes from the following famous example: Let $n = 365$ (days) and $k = 23$ (people). Assume that birthdays are uniformly distributed. Then it holds, that the probability, that at least two people have birthday on the same day is bigger than $\frac{1}{2}$, since

$$p_{23,365} \leq \exp\left(-\frac{23 \cdot 22}{2 \cdot 365}\right) \approx 0.499998.$$

In general it holds that $p_{k,n} \leq \frac{1}{2}$ if $k \geq \sqrt{2n\ln 2}+1 \approx 1.17\sqrt{n}+1$, since

$$\begin{aligned} k-1 \geq \sqrt{2n\ln 2} &\implies \frac{(k-1)^2}{2n} \geq \ln 2 \\ \implies p_{k,n} \leq \exp\left(-\frac{k(k-1)}{2n}\right) &\leq \exp\left(-\frac{(k-1)^2}{2n}\right) \leq \frac{1}{2}. \end{aligned}$$

[1] https://www.ti.rwth-aachen.de/teaching/cryptography/lecture_crypto1_ws0708.shtml

A.2 Source Code

```
// Compute distances between the work centers
var ab = new Vector2(workcenters[1].X - workcenters[0].X,
   workcenters[1].Y - workcenters[0].Y).Length();
var bc = new Vector2(workcenters[2].X - workcenters[1].X,
   workcenters[2].Y - workcenters[1].Y).Length();
var ca = new Vector2(workcenters[0].X - workcenters[2].X,
   workcenters[0].Y - workcenters[2].Y).Length();
var total = ab + bc + ca;

// Count number of fulfilled criteria of the kitchen
   triangle (allow 10% slack)
double cnt = 0;
if (total <= 7920)
{
    cnt += 1;
}
if (ab >= 1080 && ab <= 2970)
{
    cnt += 1;
}
if (bc >= 1080 && bc <= 2970)
{
    cnt += 1;
}
if (ca >= 1080 && ca <= 2970)
{
    cnt += 1;
}

return cnt / 4d;
```

Listing A.2.1: Triangle utility metric

```
// Compute distances between the work centers
var ab = new Vector2(workcenters[1].X - workcenters[0].X,
   workcenters[1].Y - workcenters[0].Y).Length();
var bc = new Vector2(workcenters[2].X - workcenters[1].X,
   workcenters[2].Y - workcenters[1].Y).Length();

// Count number of fulfilled criteria of the kitchen
   triangle (allow 10% slack)
int cnt = 0;
```

```
if (ab + bc <= 7920)
{
    cnt += 1;
}
if (ab >= 1080 && ab <= 2970)
{
    cnt += 1;
}
if (bc >= 1080 && bc <= 2970)
{
    cnt += 1;
}

return cnt / 3d;
```

Listing A.2.2: Triangle utility metric for one wall

```
/// <summary>
/// Computes the ratio between used area of the available
   area of the wall
/// </summary>
private double EdgeUtility(Wall wall, Edge edge)
{
    var usedRegion = edge.GetUsedRegion();
    var availableRegion = wall.GetAvailableRegion();

    // Intersect used region with available region
    usedRegion.Intersect(availableRegion);

    var availableArea = ComputeRegionArea(availableRegion);
    var usedArea = ComputeRegionArea(usedRegion);

    if (availableArea > 0)
    {
        return usedArea / availableArea;
    }
    else
    {
        return 0;
    }
}
```

Listing A.2.3: Edge utility

A.3 CBR Problems

```
{
    "shape": "ONE_WALL",
    "walls": [
        {
            "xPos": 0,
            "yPos": 0,
            "width": 4000,
            "height": 2400
        }
    ]
}
```

Listing A.3.1: Problem 1

```
{
  "shape": "ONE_WALL",
  "walls": [
    {
      "xPos": 0,
      "yPos": 0,
      "width": 4500,
      "height": 2400,
      "parts": [
        {
          "guid": "door",
          "xPos": 200,
          "width": 1000,
          "height": 2100
        }
      ]
    }
  ]
}
```

Listing A.3.2: Problem 2

```
{
    "shape": "L",
    "walls": [
        {
            "xPos": 0,
            "yPos": 0,
            "width": 5000,
            "height": 2400
        },
        {
            "xPos": 5000,
            "yPos": 0,
            "width": 5000,
            "height": 2400,
            "angle": 90
        }
    ]
}
```

Listing A.3.3: Problem 3

```
{
    "shape": "L",
    "walls": [
        {
            "xPos": 0,
            "yPos": 0,
            "width": 5000,
            "height": 2400,
            "parts": [
              {
                "guid": "door",
                "xPos": 200,
                "width": 1000,
                "height": 2100
              }
            ]
        },
        {
            "xPos": 5000,
            "yPos": 0,
            "width": 5000,
            "height": 2400,
            "angle": 90,
            "parts": [
              {
                "guid": "window",
                "xPos": 3000,
                "yPos": 900,
                "width": 1200,
                "height": 1200
              }
            ]
        }
    ]
}
```

Listing A.3.4: Problem 4

Johannes Grindal Ervum

Proposing Kitchen Designs: A CBR Approach

NTNU
Norwegian University of
Science and Technology