MASTER THESIS

Methodological guideline to find suitable design patterns to implement adaptability

Ime Pijnenborg

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE

SOFTWARE ENGINEERING

EXAMINATION COMMITTEE
Prof. dr. ir. Mehmet Akşit
Dr. Maya Daneva

DOCUMENT NUMBER
1.5

UNIVERSITEIT TWENTE. APRIL 2016

Abstract

We have a rather broad definition of the term adaptability: Software S1 is considered more adaptable than software S2 if for example (a) it can satisfy evolutions in requirements with lesser effort than S2, (b) if it can more easily react to the changing contextual parameters, such as the execution environment, the platform, software usage patterns etc., and (c) if it is easier to carry out corrective operations such as bug fixes than S2.

It is a well-known fact that requirements of software systems continuously evolve [2]. Furthermore, more and more software systems are designed to cope with its changing context such as changing in usage patterns without losing its functionality. Also, many software systems are designed to operate in multiple platforms. Therefore we think that making software adaptable is a legitimate goal.

This project is inspired by the problem of implementing adaptability in object-oriented programs. Currently, software engineers adopt techniques for making software adaptable in ad hoc manner. What is new in this thesis, is that we offer a framework (a tool-shell) where it is possible to define a set of heuristic rules and to enter a set of patterns descriptions.

This framework can be programmed to help the designers in searching suitable patterns for their needs. For this, the designers must enter their current design into the tool and answer a set of questions. With the help of the current design and the answers that the designer provides, the tool suggests a set of patterns to be integrated with the current design.

We call this framework the ‘Design Pattern Recognition Framework’ (DPRF). We also use the term initial model for the current actual model of the client. The client is not satisfied with this model, because he or she needs some adaptability requirements which are not supported by the initial model.

In this case, we ask questions to the client, to detect its adaptability needs. The framework matches the adaptability needs with a set of patterns to determine which pattern is suitable for implementing the desired adaptability.

The framework then merges the client’s initial model with the inferred design pattern. This merged model represents the output of the framework and implements the detected adaptability need. The framework assists the designer by identifying a suitable design pattern for implementing adaptability. In this thesis, we demonstrate the working of the framework by applying it to an example application that has some adaptability need.

Preface

This thesis presents the results of the final assignment in order to obtain the degree Master of Science at the University of Twente. It has been a long and interesting road for me. Now I have completed my thesis, I would like to thank some people who helped me during this period.

First, I would like to thank Mehmet for being my first supervisor on behalf of the university. The numerous meetings really helped me to distinguish the main points from the side problems of this project and how to work more concrete. I have learned a lot thanks to your experience and feedback.

Second, I want to thank Maya, my second supervisor on behalf of the university for her positivity and for raising my awareness of the projects meaning and contribution.

Finally I would like to thank my family back home for their support and believe in my abilities to finish my study. Your support has helped me through the complete study at the university.

Ime Pijnenborg
Enschede, April 10, 2016

Table of contents

1. Introduction

1.1 Adaptability in object-oriented programs

We have a rather broad definition of the term adaptability: Software S1 is considered more adaptable than software S2 if for example (a) it can satisfy evolutions in requirements with lesser effort than S2, (b) if it can more easily react to the changing contextual parameters, such as the execution environment, the platform, software usage patterns etc., and (c) if it is easier to carry out corrective operations such as bug fixes than S2.

The process of enhancing adaptability in response to such changes involves finding solutions and integrating them into the system without deteriorating other relevant quality attributes. Of course the designer can also aim at anticipating the future changes.

Object-oriented languages have some features that allows the developers to implement adaptable programs [18]. For example these features are: separation of concerns, encapsulation, message passing, polymorphism, inheritance, and modular decomposition.

1.2 Problem statement of adaptability

Frequently, designers need to create adaptable software either based on anticipated future changes and/or based on new requirements. Currently, the designers have the following possibilities in carrying out such a task:

(1) They have experience in the adaptability features of the languages they adopt and they use this experience.

(2) They adopt a design method that emphasize adaptability.

(3) They select a suitable pattern from a pattern catalogue and integrate with their current design.

Unfortunately, there are no intuitive tools that can support the designers in assisting in these tasks. This thesis aims to support the designers in the third option.

1.3 Motivation

If the required adaptability of software can be anticipated, then right from the beginning of a project, it can be defined as a non-functional requirement. Alternatively, as a form of bug fixing, the designer may need to reengineer software to make it adaptable so that software functions as desired. A third possible option is that the requirements evolve. If the evolution was anticipated and software was designed accordingly, this may not be a problem. Otherwise re-engineering of software may be necessary.

In any of these three problem settings, the designer must find out a suitable solution pattern for his/her adaptability needs.

We offer a tool-shell that can be used to program a guideline that can help the designers to find suitable patterns for his or her needs and we call this tool-shell a framework. This framework should be programmable so that more and more patterns can be defined in the framework so that a large set of adaptability needs can be satisfied.

We assume that first the designer creates a UML model for his or her initial software which is not (yet) necessarily adaptable, which we call the client's initial model. We also assume that the designer is not satisfied with this model, because the adaptability requirements are not supported by the initial model. A framework may help designers to identify the needs.

1.4 Defining a framework that can assist the designers to find suitable patterns for his or her needs

This project aims to develop a framework for finding suitable design patterns. This project will only elaborate on design patterns as these are all modeling solutions for accomplishing adaptability.

This framework should help the designer to find a suitable pattern and this framework can help you to define a methodological guideline. A goal of this research is to develop a proof of concept in the form of a prototype that can assist the designer by identifying his or her needs and matching these to a set of patterns.

The goal of this project is not to fully implement all possible design patterns, but to develop a proof of concept that will demonstrate the process of matching and merging a design pattern to the designer's initial model. By describing the way to extend the framework with other patterns, this project will also offer the possibility of continuing to develop the framework.

This framework first applies a matching process between the designer's adaptability needs and a set of design patterns. Therefore, to successfully implement the matching and merging process, it is important to specify the initial model, needs, and design patterns in a formal manner.

1.5 Structure of the report

The structure of this thesis follows the approach taken in this study. The thesis is structured as follows:

Chapter 2 provides background information about subjects important to this research, including adaptability, evolution, design patterns, MACH, and rule-based expert systems.

Chapter 3 presents a model for adaptability that describes the four types of evolution in an abstract way. The four types of evolution are: non-overlapping extensions, specialization, overlapping extensions, and layered (meta) evolutions. Every type of evolution has a set of technologies and design patterns that can help implement this type of evolution. Of these patterns, the decorator pattern, is described in more detail.

Chapter 4 introduces the developed framework for finding the applicable design patterns. The ‘Design Pattern Recognition Framework’ (DPRF) is explained by presenting an overview of its components. The overview is divided into three parts: *Detecting the adaptability need*, *Pattern matching*, and *Merger*. These parts of this framework will be discussed in detail.

Chapter 5 presents an example application of the framework by describing how a suitable design pattern can be matched and merged with a designer’s initial model, according to the designer’s needs. The application will be divided into the same three parts of the framework as the overview in chapter 4.

Chapter 6 provides the conclusions of this study and suggestions for future work by defining the contributions of our work, the new questions that has appeared, and experiences. The suggestions for future work include a description of the tools to extend the framework with new patterns.

2. Background in object-oriented modeling

Here we present the relevant background work. By providing background information, the research activities will become clearer. This chapter presents background information about adaptability and evolution, the object-oriented paradigm, design patterns, Model Analysis and Checker (MACH), and rule-based expert systems.

2.1 Adaptability and evolution

Adaptability and evolution represent the main problem we address in this project. This section describes evolution of software and the way to adapt to it in more detail as a software solution can be modified and extended over time. An example scenario is depicted in Figure 2.1 [2].

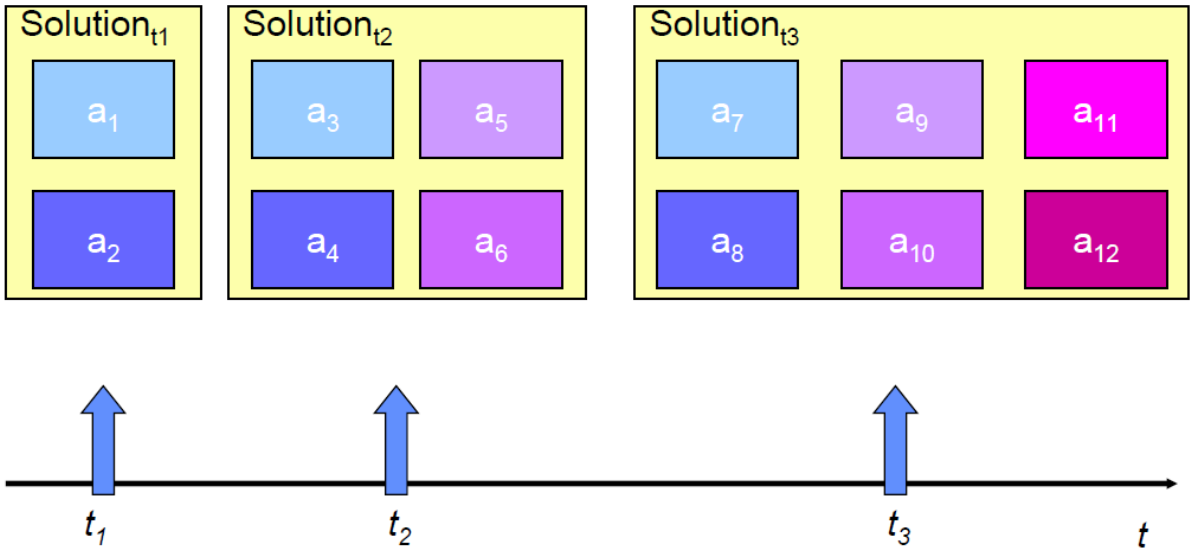

***Figure 2.1**: Example scenario for evolution [2]*

At time t1, the software solution has two components, A1 and A2. In this report we use the term component to represent any meaningful software abstraction that represents a significant concern. For example, in object-oriented languages, it may refer to an object.

The solution evolved at a time t2. The components, A1 and A2, have evolved into A3 and A4 and the solution is extended with the components A5 and A6. At time t3, the solution has further evolved, and consists of six components (A7, A8, A9, A10, A11, and A12).

Functional requirements and third-party technologies can change in time, which means the software has to change accordingly. It must adapt to these changes to perform tasks it was designed for [14]. According to Selim Ciraci, Pim van den Broek, and Mehmet Aksit, *"Software evolution for such changes involves finding solutions for these new set of requirements and integrating them into the system without affecting the quality of the system."* [9] They emphasize that it is important to maintain the quality of the software.

The business environment changes rapidly, therefore, it is important to adapt to possible changes in the future. Adaptable software deals with these possible changes in a way that it

can be altered or extended. To build adaptable software, it is important to *build the right thing*, to *build the thing right,* and to *support the next thing*. [11] Validating requirements can ensure that the right thing is built; by verifying and correcting the software, the developer can build correctly. There are several ways to support the next thing. A well-known approach is to build reusable software–blocks that can be used again in another place. A well-known example is an object in object-oriented languages.

Adaptable software can support evolution more efficiently as it anticipates changes in requirements or the environment.

2.2 The object-oriented paradigm

The goal of this project is to define a framework that can assist the designers to find suitable patterns for his or her needs with the help of object-oriented languages. Before elaborating the defined framework, we should describe the object-oriented paradigm. It is the most commonly used paradigm and it provides special constructs for adaptability.

The object-oriented programming paradigm was introduced in the late 1970s and early 1980s. This paradigm is based on the idea of 'objects', which are data structures that contain data (in the form of attributes) and behavior (in the form of methods) [18].

The object-oriented paradigm has become more popular as one can create a system with objects that can work independently. According to the University of Colorado, the system has become "*a network of objects collaborating to fulfill the responsibilities (requirements) of the system*" [4].

The objects are entities that combine data and behavior, also known as features. The data allows the object to keep track of its state, and the behavior allows it to function in the way it is supposed to function [23].

One of the characteristics of the object-oriented paradigm is that objects can act as instances of a class. It is possible to have multiple objects of the same class, each containing specific data although they have the same set of methods [18].

A powerful aspect of the object-oriented paradigm is the relationships between objects. For example, a class can have ***inheritance*** relationships. According to the University of Colorado, behavior and data associated with a superclass are passed down to instances of a subclass [4]. The subclass can add new behaviors and new data that are specific to the subclass, but it can also modify the inherited behavior from the superclass [15].

Figure 2.2 illustrates a small example of inheritance. Both the teacher and the student is a person; they both have the data-attributes: *id, first_name, surname, birthdate* and *phone number.* However, they also have data-attributes that are specific to the teacher and student. These data-attributes are defined in the subclasses *Teacher* and *Student.* This example also demonstrates how to modify the inherited behavior from the class *Person*. The subclasses override the function *getPersonDetails()* from *Person* and can change the implementation of this method.

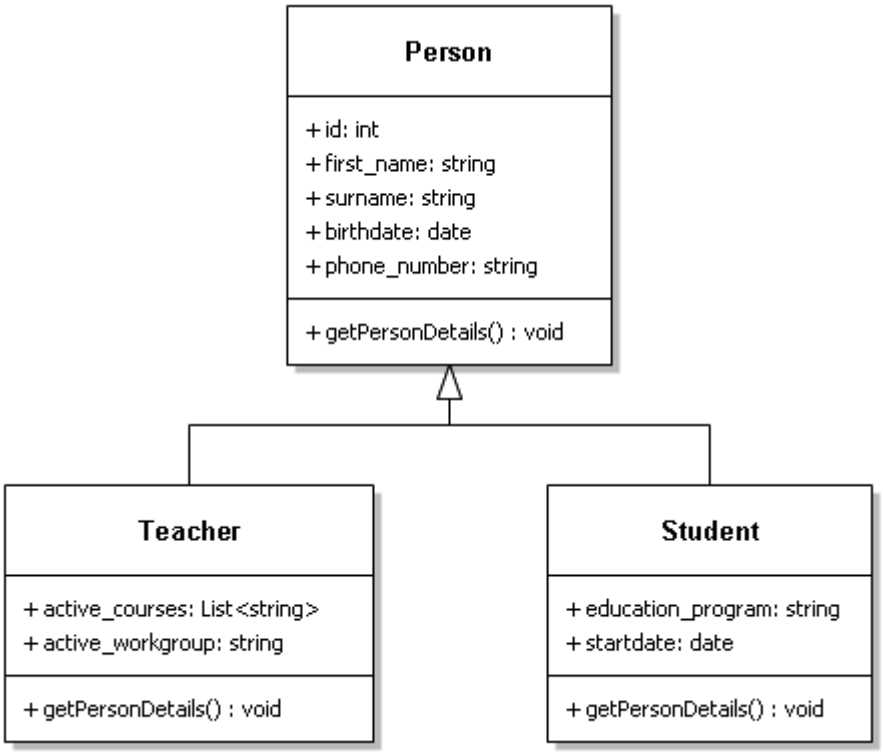

***Figure 2.2**: Example inheritance*

The object-oriented paradigm also has the ability to hide variables of a class; this is named ***encapsulation*** or ***information hiding*** [15]. The goal of encapsulation is to hide the variables of a class from other classes, by keeping them private. These variables can be accessed only through methods of their current class. The class should provide public setters and getters to modify and access the values of the hidden variables.

Figure 2.3 is an example of encapsulation for the class *Person*. The variables are private, but there is a getter and setter for each variable.

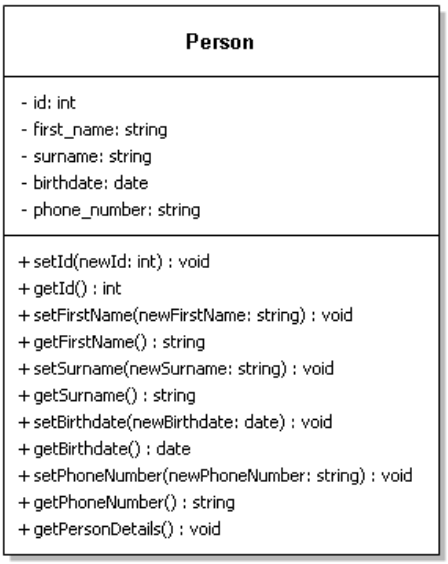

***Figure 2.3**: Example encapsulation*

Polymorphism is another powerful aspect of the object-oriented paradigm. Alan Shalloway explains that polymorphism is *"Being able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to"* [19]. Polymorphism is the ability of an object to take on many forms.

Polymorphism_example.java

```
public class Person {
	public void getPersonDetails() {
		System.out.println("This is a person.");
	}
}

public class Student extends Person {
	@Override
	public void getPersonDetails() {
		System.out.println("This is a student.");
	}
}

public class ClientApplication{
	public static void main(String[] args) {
		Person student = new Student();

		student.getPersonDetails();
	}
}
```

As illustrated above, the class Student (subclass) extends the class Person (superclass). In the class 'ClientApplication', which represents the client-program, we instantiate a variable 'student' of type Person. We assign a new Student object to the variable. When the client calls the overridden method 'getPersonDetails()' on the superclass variable, the subclass version of the method will be executed. The system will print: *"This is a student"*.

Another powerful aspect of the object-oriented paradigm is ***abstraction***. According to the University of Colorado, abstraction refers to "*The set of concepts that some entity provides you in order for you to achieve a task or solve a problem*" [4]. Abstraction allows the developer to write codes which work with abstract data structures. It is the concept of describing something in simpler terms (abstracting the details, in order to focus on what is important) [15].

We can demonstrate the concept of abstraction by using the example of *Person*, *Student* and *Teacher*. *Person* can be an abstract class to specify the default data and behavior of an object and let its subclasses (Student and Teacher) explicitly implement that functionality.

Abstraction_example.java

```
abstract class Person {
	public int id;
	public string first_name;
	public string surname;
	public date birthdate;
	public string phone_number;
```

```
        public void getPersonDetails();
}

public class Student extends Person {
        public void getPersonDetails() {
                System.out.println("This is a student.");
        }
}

public class ClientApplication{
        public static void main(String[] args) {
                Person student = new Student();

                student.getPersonDetails();
        }
}
```

The object-oriented paradigm offers numerous concepts and possibilities to model software. The use of objects together with the four important aspects of the paradigm (Inheritance, Encapsulation, Polymorphism, and Abstraction) gives the developer more options in the design of software architecture.

Various programming languages support the object-oriented paradigm; the most popular are Java, C++, C#, Delphi, PHP, Python, Visual Basic, and Objective C [18]. These languages have their own syntax, but it is possible to program in an object-oriented style.

2.3 Design patterns

In this project, we will define a framework that can help the designers to find suitable patterns for his or her needs. Hereby, in this section we will define what a design pattern is.

Patterns capture well-proven experience in software development and help to promote good design practice. A pattern can deal with specific recurring issues in the design or implementation of software. With the use of patterns, it is possible to construct software architecture with particular properties. [7]

Christopher Alexander explains that "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*" [3] This definition is based on patterns in buildings and cities, but is also applicable in software.

A design pattern has four essential elements: The context giving rise to a problem, the recurring problem, the proven resolution, and the consequences (advantages and disadvantages). [7][12] Design patterns can be used to model adaptability into software. [2]

2.4 Model Analysis and Checker (MACH)

The Model Analysis and Checker (MACH) is an experimental tool and it will be used in this project for the conversion of UML models into an internal data structure in Prolog. In this section we elaborate on the tool and describe the features it supports.

MACH is a tool to deploy and test advanced algorithms for analyzing UML models. [20] The tool's implementation language and existing code base uses Prolog (specifically SWI-Prolog), and the tool has a command-line User Interface (CLI). MACH is a flexible and lightweight framework for the loose integration of independent tools, and has a set of integrated tools.

MACH was developed in the *Department of Applied Mathematics and Computer Science*, Technical University of Denmark, in 2013 under the lead of Professor Harald Störrle. While developing the tool, they focused on academic stakeholders, but neglected industrial users. MACH users should have some understanding of models and the underlying concept to fully understand and use the tool.

The research work focuses on advanced operations on UML models that are beyond the scope of existing modeling tools. MACH offers an advanced analysis and checking procedures on UML models. The tool can be seen as proving ground for analysis techniques from recent research.

MACH loads models in XMI format and converts it to an internal data structure in Prolog. This internal data structure is represented as Prolog facts. When the model is converted to an internal data structure, it is possible to use commands to search for specific patterns or elements in models, and to inspect parts of models in detail.

The user can navigate through directories with a set of commands:

- *pwd* — Print working directory
- *cd [PATH]* — Change directory to [PATH]
- *ls [OPTION]* — Shows contents of working directory. Possible options:
  - *m*: list only models
  - *a*: list all files
  - *p*: list all Prolog files

To access/load the existing models in MACH, the models should be stored in a XMI format. MagicDraw is a modeling tool (which will be explained in more detail later in this section) and has a custom format (MDXML), which is based on the standard XMI model interchange format. MACH assumes the models are of the MDXML format.

The user can access existing models with the following commands:

- *open [FILE]* — Access file and interpret it as a model
- *open [FILE] as $ALIAS* — Access file, interpret it as a model and assign the name as a shorthand for the file
- *show all aliases* — List all currently defined aliases
- *show alias $ALIAS* — show details of the specified alias
- *clear all aliases* — Remove all alias declarations
- *clear alias $ALIAS* — Remove the specified alias declaration
- *show all opened models* — List all currently opened models.

SWI-Prolog

MACH requires SWI-Prolog, an open source implementation of Prolog. Prolog is a Logic-programming language based on formal logic [21]. It is expressed in terms of relations represented as facts and rules. A computation is initiated by running a query over these relations.

Prolog is helpful for MACH, because you can run a query to search for specific elements in models. With a query, questions are asked about the relations [6].

SWI-Prolog has a set of features, including libraries for constraint logic programming, multithreading, unit testing, GUI, interfacing, among other technologies. The development of SWI-Prolog started in 1987 by Jan Wielemaker at the University of Amsterdam.

Prolog is highly useful as it has a built-in backward chaining inference engine which can be used to partially implement some expert-systems [10].

Structure of Prolog facts

MACH will be used in this project to load models in XMI format and convert them to Prolog facts. Hereby, the structure of the facts should be the same to compare, infer, or match with other facts. The structure is explained based on an example class, illustrated in Figure 2.7.

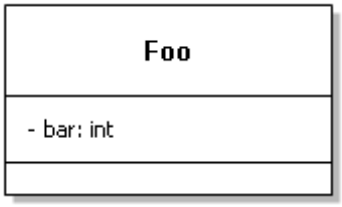

***Figure 2.5**: Example class 'Foo'*

Figure 2.5 demonstrates a class called 'Foo'. This class has a private property called 'bar' of the type 'int'. The representation of this class in Prolog looks like this:

```
me(class-1,[name-'Foo',ownedMember-ids([2])]).
me(property-2,[name-'bar', visibility-'private', type-'int']).
```

The first argument of the *me*-predicate is a pair of type and internal identifier (an integer). The second argument is a property list of tags for meta-attributes and their values. References to identifiers are marked with an *id* or *ids*-term. The attribute *'ownedMember-ids([2])'* of the class ´Foo´ indicates that this class has a subfact with an internal identifier of 2. This reference shows that the *ownedMember* is a property of the class.

The internal identifier is an integer and identifies the predicate. The type identifier represents the type of element. Possible type identifiers are: *model, comment, diagram, class, generalization, property, operation,* and *parameter.*

MagicDraw
MagicDraw is a visual UML, SysML, BPMN, and UPDM modeling tool. This dynamic tool facilitates analysis and design of object-oriented software and databases, publicly released in 1998 [17].

The Unified Modeling Language (UML) is the industry's standard general purpose modeling language for software engineering. MagicDraw supports the notation and semantics associated with the UML. The tool makes it possible for the user to create and edit UML diagrams, which are diagrams that follow the graphical notation of the UML.

MagicDraw saves UML diagrams as MDXML-files, which is required by MACH. Therefore, MagicDraw is an efficient and verified tool to accomplish the desired conversions. Other tools may work as well, but MagicDraw is verified by MACH developers.

2.4.1 Rule-based expert systems

As we use Prolog facts to represent UML models internally, we also use Prolog facts to implement the matching and merging process of the framework that we define in this project. A rule-based expert system is a technology that underlines this framework by using Prolog facts, and this section will elaborate on this.

An expert system is a system that corresponds to the decision-making capability of a domain expert. A rule-based expert system uses human expert knowledge to solve problems that occur in the real world. The system encodes expert knowledge into an automated system. These systems are the simplest form of artificial intelligence [1].

The basic structure of a rule-based expert system is illustrated in Figure 2.6. [16] The basic structure consists of a knowledge base, database, inference engine, explanation facilities, and user interface. The user represents the person that is seeking a solution to a particular problem and communicates with the user interface.

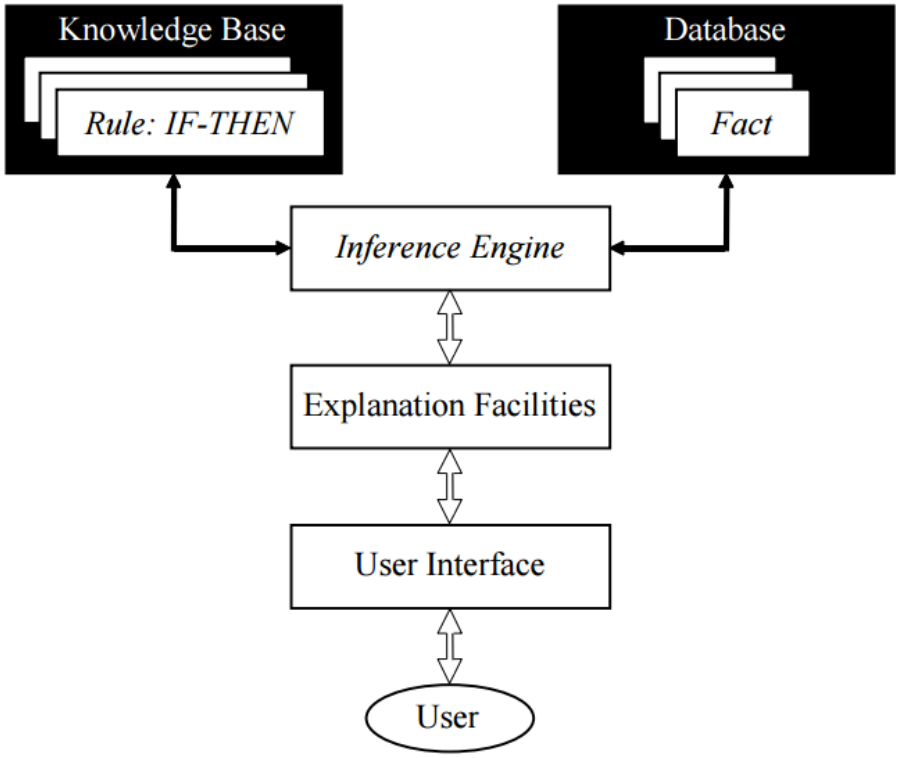

***Figure 2.6**: Basic structure of a rule-based expert system*

The **knowledge base** contains the domain knowledge that is needed for problem-solving. Domain knowledge is expressed as a set of rules, also described as a 'rule base'. This set contains all actions that should be taken depending on the set of facts. Each rule is represented as an IF-THEN statement, and is a conditional statement that connects conditions to actions or outcomes.

The **database** includes a set of facts used to match against the IF-part of rules in the knowledge base.

The **inference engine** performs the analysis and interpretation whereby the expert system finds an applicable solution. The inference engine links the given rules in the knowledge base with facts from the database.

A rule-based expert system also contains **explanation facilities** to clarify its reasoning and defend its conclusion. The explanation facilities enable the developer to ask the system why a specific fact is needed, and how a particular conclusion is reached.

The **user interface** is the communication layer between the user and the expert system. It is the means of communication between a user searching for a solution to a particular problem and the expert system.

Inference chains are produced by matching rules to facts. An inference chain reveals how an expert system applies the rules to find the right conclusion. There are two possible types of inference chains: forward chaining and backward chaining [16].

Forward chaining is data-driven reasoning. The reasoning starts from known data and progresses with that knowledge. Forward chaining is a technique for collecting data and inferring from it whatever is possible.

Backward chaining is goal-driven reasoning. The reasoning starts with a goal (hypothetical solution) and the inference engine attempts to find the rule(s) that might have the desired solution. Such rules must have the goal in their THEN-parts. Rules can be stacked to have subgoals that can confirm the main goal.

Decision tree

A decision tree is a good method for the implementation of the knowledge base. A decision tree can be used to classify a situation according to a set of decisions [22].

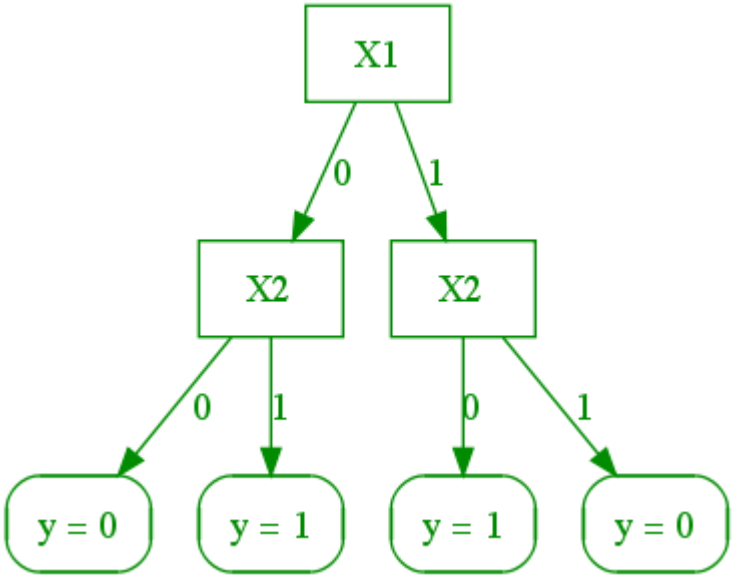

***Figure 2.7**: Decision tree example*

A decision tree is an intuitive structure that looks like a flowchart and consists of decision nodes. After going through the decision nodes, the decision tree will end with a suggestion. In Figure 2.7, X1 and X2 are variables that can be false (0) or true (1). The value of variable y depends on the values of X1 and X2. The decision nodes represent the IF-THEN statements in the knowledge base according to given facts (X1 and X2), like one of the nodes in the example:

$$IF\ (X2\ =\ 0)\ THEN\ y\ =\ 0$$

3. A model and design patterns for adaptability

Evolution can have a impact on a software solution which is why it is important to implement adaptability in the software [8]. In this chapter, we will define the formula to model adaptability and make divisions in evolution, stated as evolution semantics. For each type of evolution, we define a set of object-oriented technologies and design patterns that can be used to implement this. We elaborate on the decorator pattern.

The formula to model adaptability into software can be defined as follows [2]:

$$S_{n+1} = (S_2, S_3, \ldots, S_4) \oplus S_1$$

Explanation of the formula:

S_{n+1}	This is the solution that you need
$(S_2, S_3, \ldots, S_n)$	These are the solutions (or solution) you need to add
$\oplus$	This is how you add it
S_1	This is the solution that you have

The software that has to evolve (solution S1) can be *anticipated* or *unanticipated*. This indicates if the software is prepared for changing requirements and/or future technology. Anticipated software is software that is designed in a way that can be replaced or extended in runtime. Software that is unanticipated is software that is without considering changes in the future. This means when requirements or technology changes, the existing software should adapt to these changes, which indicates that it should be recompiled.

Anticipated software can be divided into two types: *Just good* and *Too much*. *Just good* indicates that the software correctly anticipates desired changes. *Too much* indicates that the software anticipates too many possible changes. This causes significant overheads and may result in a lack of robustness.

Possible changes in functional requirements or technologies are anticipated by determining a line between the fixed part and the adaptable part of the software. It is important to know which components should be variable and which components should remain fixed. When this is clear, the software anticipates numerous possible changes.

Evolution can be divided into four types: non-overlapping extensions, specialization, overlapping extensions, and layered (meta) evolutions [2]. The following section, section 3.1, will describe these according to the formula for modeling adaptability. Section 3.2 will describe the decorator pattern that can be used to implement adaptability.

3.1 Evolution semantics

3.1.1 Non-overlapping extensions

Non-overlapping extensions are an extension to the existing solution. Figure 3.1 depicts an extended software system [2].

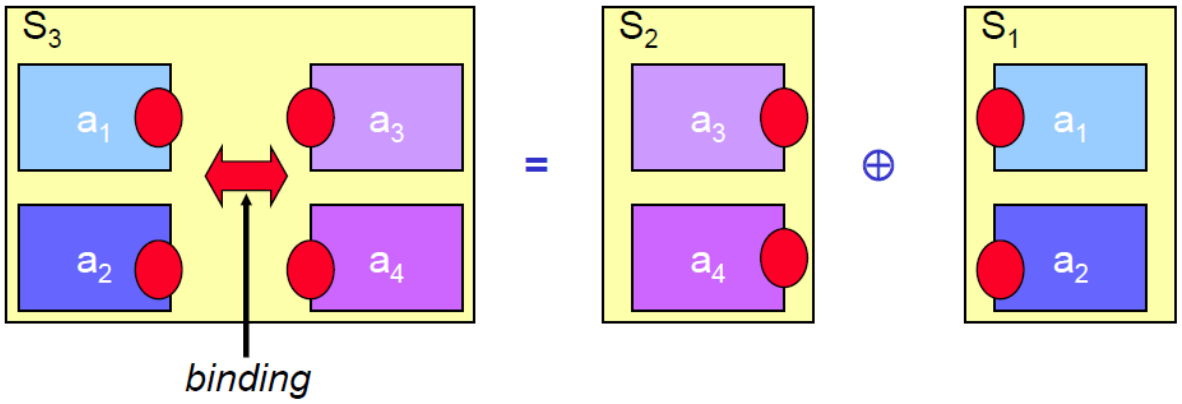

***Figure 3.1**: Example of non-overlapping extensions [2]*

Here the most right rectangle S1 represents a system to be extended. This system has two components, A1 and A2, which are represented as two inner rectangles. The extension interfaces of these two components are represented by two ellipses on the left of these rectangles.

S2 represents a software system that is to be added to the existing system. The (+) is a composition operator, which is necessary to integrate S1 and S2. Like S1, S2 has two inner components (A3 and A4) with their own extension interface.

We define this as a non-overlapping extension because S1 and S2 do not share software components. S2 will be bound to S1. The binding of solutions S1 and S2 means that components A1 and A2 will connect with components A3 and A4. This connection can be implemented with the help of various technologies and design patterns.

To implement a non-overlapping extension, four design patterns can be used: decorator, proxy, mediator, and adapter. Other object-oriented technologies can be used to implement this type of evolution: glue code, script, and transformation. In the following we will define these patterns and other object-oriented technologies.

Decorator pattern
The decorator pattern is used to attach additional responsibilities dynamically to individual objects instead of an entire class. Decorators provide a flexible alternative to subclassing for extending functionality [12].

Proxy pattern
The proxy pattern can be used to create a wrapper to cover the main object's complexity from the client. The pattern provides a surrogate or placeholder for another object to control access to it [12].

Mediator pattern
By using the mediator pattern, the communication complexity between multiple objects or classes will be reduced by providing a mediator class which handles all communication between them [12].

Adapter pattern
An adapter pattern is applied to use an existing class and its interface does not match the one you need. The pattern lets classes work together that otherwise could not because of incompatible interfaces [12].

Glue code
Glue code is a dedicated program that replaces and/or binds the interfaces of modules. The standard practice is to keep logic out of the glue code and leave that to the code blocks it connects to [2].

Script
Script is a dedicated interpreter-based language that installs modules and configures calls among them [2].

Transformation
Transformation transforms the solution into a new solution. This can be implemented with the help of transformation rules [2].

3.1.2 Specialization

Specialization is the extension of the current solution which serves as a replacement. Figure 3.2 illustrates a software system which is extended [2].

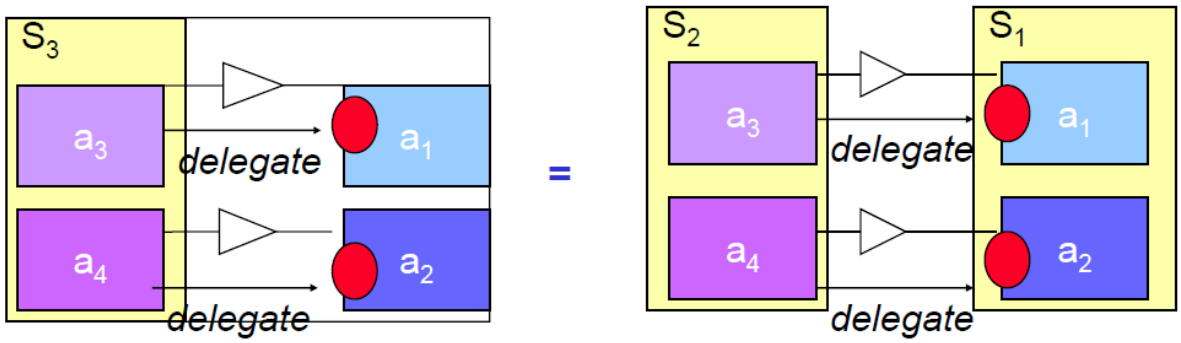

***Figure 3.2**: Example of specialization [2]*

In Figure 3.2, a software system is depicted which is extended with the use of a delegation mechanism. S1 represents a system to be extended. This system has two components, A1 and A2. S2 represents a software system that is to be added to the existing system. Like S1, S2 has two inner components (A3 and A4).

We define this extension as specialization, because components in S2 replace components in S1 with the help of a delegation mechanism. Delegation refers to one object relying upon another to provide a set of functionalities. The components in S1 hand over tasks to components in S2.

It will become possible to substitute components in S1 with components in S2, resulting in the desired solution S3. This mechanism can be implemented with the help of various technologies.

To implement the specialization, three design patterns can be used: bridge, strategy, and command. In addition, other object-oriented technologies can be used to implement this type of evolution: inheritance and true delegation. In the following we will define these patterns and other object-oriented technologies.

Bridge pattern
The bridge pattern can be used to decouple an abstraction from its implementation so that the two can vary independently by providing a bridge structure between them [12].

Strategy pattern
By using the strategy pattern, a class behavior or its algorithm can be changed at runtime. With this pattern, objects are created which represent different strategies and a context-object with different behaviors per strategy [12].

Command pattern
The command pattern is used to issue requests to objects without knowing anything about the operation that is being requested or the receiver of the request. It encapsulates a request as an object, thereby allows one to parameterize clients with different requests and support undoable operations [12].

Inheritance
You can extend behavior through inheritance with the help of super calls, which is similar to the decorator pattern, although it is through the inheritance hierarchy [2].

True delegation
This technology is about transitive reuse with support of self calls, similar to the delegation-based language *self* [2].

3.1.3 Overlapping extensions

Overlapping extension is a type of evolution where at least one software component is shared between the current solution S1 and the solution S2 to add. Figure 3.3 illustrates an extended software system [2].

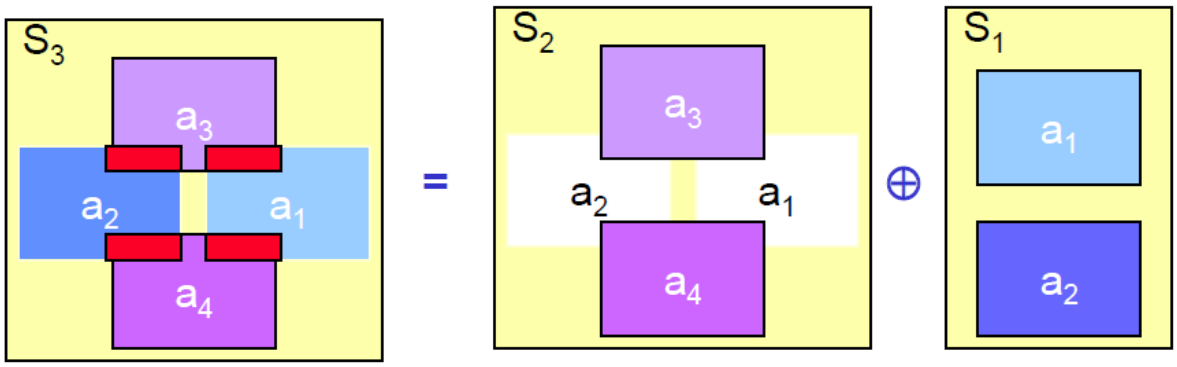

***Figure 3.3**: Example of overlapping extensions [2]*

S1 represents a system to be extended. System S1 has two components, A1 and A2. S2 represents a software system that is to be added to the existing system. System S2 has two components, A3 and A4.

We define this extension as an overlapping extension because S2 has components that overlap some functionality of the components in S1. One or multiple software components are shared between the solutions S1 and S2.

Merging the components can be quite difficult, because it contains overlapping functionality. Nevertheless, with the help of various technologies it will become possible to extend solution S1 with solution S2.

To implement the overlapping extension, two design patterns can be used: bridge and strategy. These patterns have been described in section 3.1.2. Other object-oriented technologies can be used to implement this type of evolution: programming, override, and transformation. In the following we will define object-oriented technologies.

Programming
By reprogramming and recompiling the solution that you have, it is possible to extend it with the solution to add [2].

Override
With this technology, you can replace parts of the program using inheritance to override existing methods [2].

Transformation
This technology is about transforming the software solution into a new software solution [2].

3.1.4 Layered (meta) evolutions

The solution, S2, that we want to add as a layered (meta) evolution can be seen as a layer on top of the current solution S1. Figure 3.4 depicts an example software system which is extended with an extra layer [2].

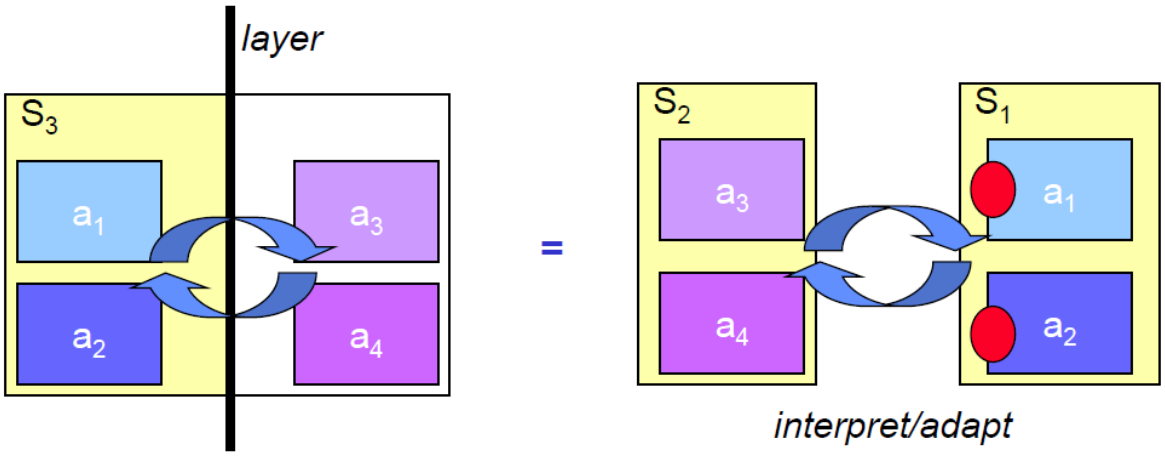

***Figure 3.4**: Example of layered (meta) evolutions [2]*

The right rectangle S1 represents a system to be extended. This system has two components, A1 and A2. S2 represents a particular (meta) layer to be added to the existing system. Like S1, S2 has two inner components (A3 and A4).

We define this type of evolution as layered (meta) evolutions, because the components in S2 interpret or adapt the existing components in S1. This means that the components in S2 describe the components in S1 on a higher (meta) level.

When the layers interact with each other, it will result in the solution S3. This layered evolution can be implemented with the help of various technologies.

To implement the layered (meta) evolutions, one design pattern can be used: the command pattern, described in section 3.1.2. Besides the command pattern, other object-oriented technologies can be used to implement this type of evolution: interpretation, compilation, reflective features, and transformation. Transformation technology is discussed in section 3.1.3. In the following we will define interpretation, compilation and reflective features.

Interpretation
Interpretation is about processing a program and/or runtime environment [2].

Compilation
Compilation states a compiling program, like application generators, language compilers, and stub-generators [2].

Reflective features
This technology includes the use of reflective features of the language to examine and modify the structure and behavior of the solution at runtime [2].

3.2 Design patterns for adaptability

As described in the previous section, there are seven design patterns that can be used to implement adaptability. In this research project, we only elaborate on the Decorator pattern to illustrate the method for applicable design patterns, instead of working out all design

patterns. The decorator pattern is a design pattern that can be used for non-overlapping extensions.

We now describe the Decorator pattern in detail, by defining the applicability, solution, advantages, and disadvantages.

3.2.1 Decorator pattern

Applicability
The decorator pattern can be used when it is important to add encapsulated responsibility to individual and existing objects transparently without affecting another object. These responsibilities can be removed again without affecting other objects [12].

This pattern extends classes without the use of subclassing. Subclassing adds behavior at compile-time and the change affects all instances of the original class, which makes it a less suitable solution. The pattern is suitable when it is desirable to add new behavior at runtime.

Solution
The solution of the decorator pattern can appear in various forms. The UML-diagram of the most basic form is depicted in Figure 3.5 and has four participants: ‘Component’, ‘IComponent’, ‘Decorator’ and ‘ConcreteDecorator’.

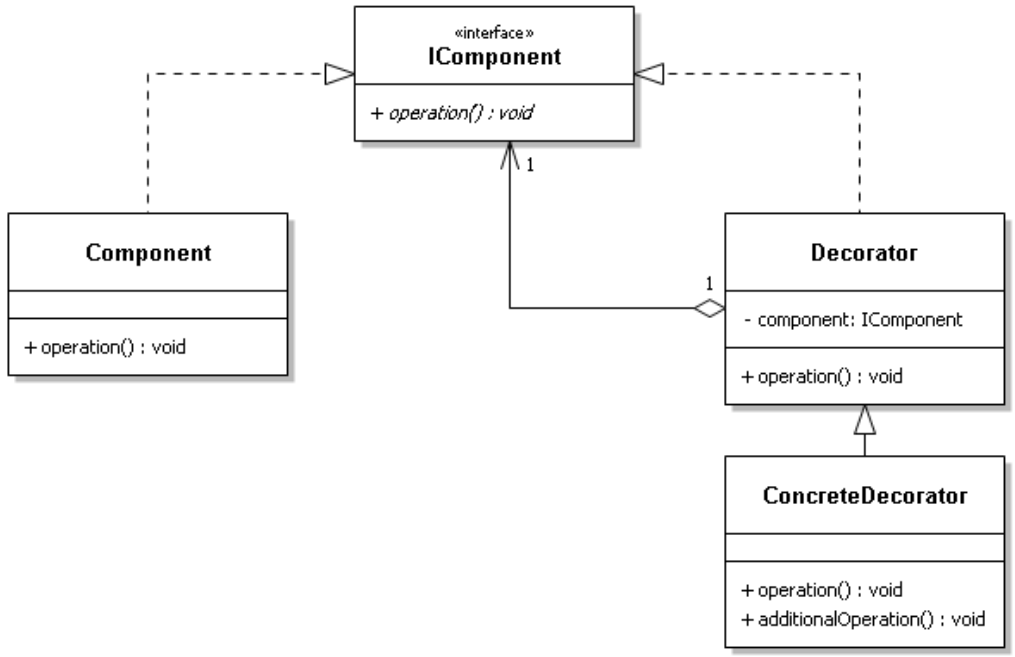

***Figure 3.5**: UML-diagram of decorator pattern*

Participants:
- *Component*
  - Component is the object to which additional behavior can be added.
- *IComponent*
  - IComponent is the interface of the class Component to which additional behavior can be added dynamically.

- *Decorator*
    - Maintains a reference to Component and defines an interface that is to conform to IComponent
- *ConcreteDecorator*
    - Extends the behavior of Component by adding additional functionality

Collaboration:
Component and IComponent are classes in the existing solution where additional behavior is desired. The Decorator maintains a reference to Component and defines an interface that is the same as IComponent. By creating one or multiple ConcreteDecorators, the behavior of Component can be extended. The ConcreteDecorator calls the method operation() from Component and the additional extension additionalOperation().

Advantages and disadvantages
Advantages:

- *Flexibility*, by extending messages and behavior at runtime. It is also possible to extend the individual object with as much additional behavior as the developer desires, by adding more ConcreteDecorators.
- *Not much anticipation of software needed,* because no complex customizable class is needed for additional behavior. The software only has to anticipate the Decorator-class. The behavior can be extended at runtime by incrementally adding ConcreteDecorators.

Disadvantages:

- *Decorator and ConcreteDecorators are not identical*, from an object identity point of view. This means that it is not possible to rely on object identity of the decorators.
- *Many small objects,* because the decorator has many small objects that look alike. The difference between these small objects is the interconnection between each other. This can result in a complex design, which is difficult to debug.

4. A framework to find applicable design patterns

In this chapter, we will describe the framework that we designed to find applicable patterns. We call this ‘Design Pattern Recognition Framework’ (DPRF). This framework is our contribution to the problem statement as we described in chapter 1. With DPRF, it will become clear which design pattern can be used in the most effective way to implement adaptability into the initial model of the designer. DPRF assists the designer by finding an applicable pattern in his situation and according to his needs.

Section 4.1 defines the overview of the framework with the relevant components and techniques in a UML activity diagram. Sections 4.2, 4.3, and 4.4 specify and explain the components and techniques, divided into three parts (adaptability need, pattern matching, and merger).

4.1 Overview of the Design Pattern Recognition Framework

This section will describe the overview of DPRF by introducing a UML activity diagram. This diagram is illustrated in Figure 4.1.

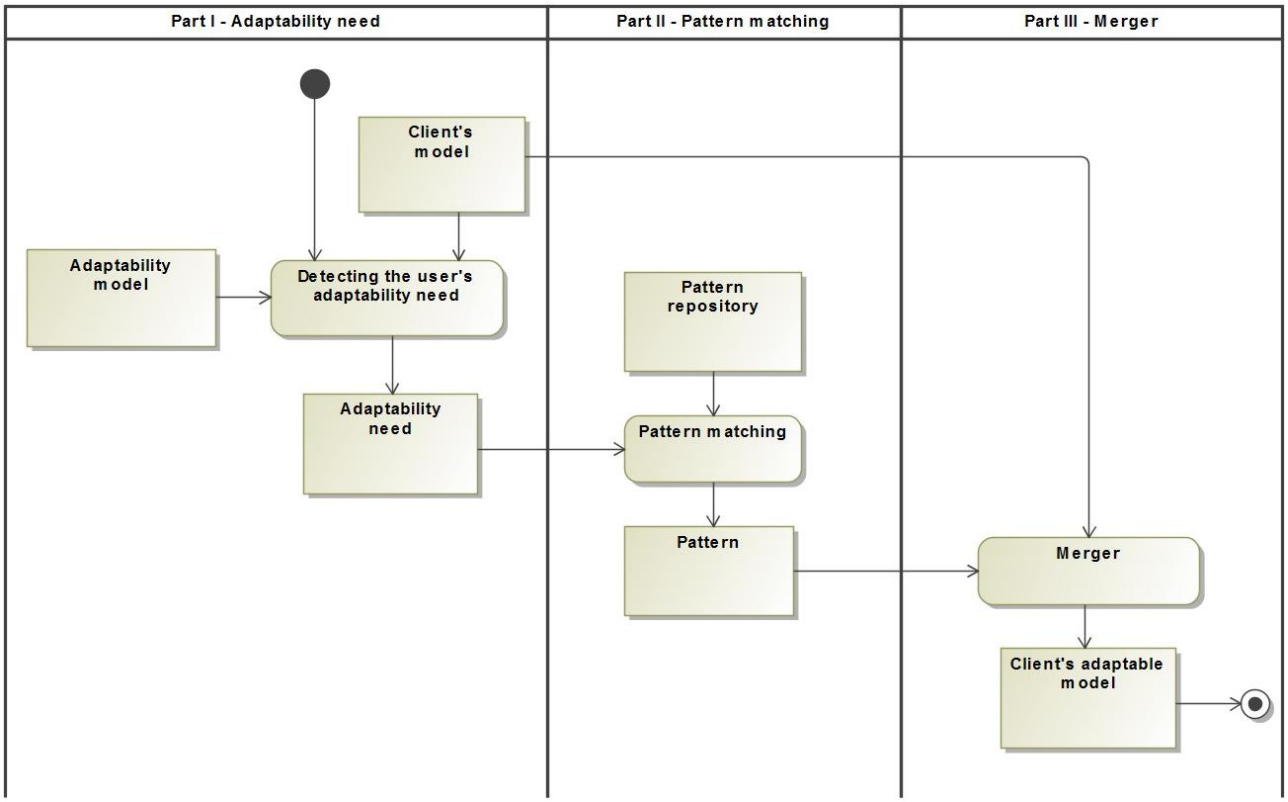

***Figure 4.1**: UML activity diagram of DPRF*

The overview is divided into three parts: *Adaptability need* (part I), *Pattern matching* (part II), and *Merger* (part III).

Part I focuses on the detection of the user's adaptability need by applying a matching process between the client's model and adaptability model. The black circle at the top represents the starting point of the activity and points to the first activity ('*Detecting the user's adaptability need*'). The client's and adaptability model serves as input to this activity. The client's model is a set of Prolog facts and represents the user's initial UML model internally. The adaptability model includes generic adaptability features that can be identified by the user. With the help of a decision tree, the user can detect its adaptability need.

Part II contains a pattern repository that exists in a set of design patterns. Each pattern has a subset of adaptability features that can be implemented by the pattern. This subset is matched with the output of the decision tree, to find a suitable pattern according to the user's need.

Rule-based expert system, as described in section 2.4.1, is the technology that underlines the first two parts of the framework. The knowledge base is represented by the adaptability need and the pattern repository represents the database. DPRF performs the analysis and interpretation as an inference engine, to find a suitable pattern.

Part III represents the merging process between the client's model and the derived pattern. The merger unites the pattern with the client's model to fulfill the user's adaptability need. This activity results in the client's adaptable model.

4.2 Part I - Detecting the adaptability need

In this section, we describe the relevant components in part I. To detect the user's adaptability need, the framework uses the client's model and adaptability model as input. This process results in a subset of adaptability needs.

The client's model serves as input for the framework. It represents the UML model of the existing application which has some adaptability problem. The client's model must be represented internally using Prolog facts in a particular structure as described in section 2.5.

The adaptability model includes generic adaptability features and is based on the four types of evolution described in section 3.1. These generic features describe the types of adaptability that can be recognized by the user. The process to detect the user's adaptability need is the identification of a set of adaptability features from the adaptability model. This set of features is the output of the process and can be represented as the user's adaptability need.

In the following, we will define and describe the features step-by-step. Each feature is represented within a rectangle and is numbered from 0 to 6.

```
0 "We have a method m1()."
```

Feature 0 states that there is a method in the client's model that relates to the adaptability problem. We refer to this method as 'm1()'.

```
1 "The implementation of method m1() may change according to the context at runtime."
```

Feature 1 indicates that the set of statements that represents the implementation of method m1(), may change. The program should decide at runtime which set of statements is used for the implementation of the method. This decision depends on the context.

```
2 "The method m1() can be divided into sub-methods."
```

Feature 2 declares that the implementation of method m1() can be divided into multiple sets of statements.

```
3 "The implementation of the change means sequentially cascading a selected set of sub-methods
  together."
```

Feature 3 states that implementing the desired change means sequentially cascading one or multiple sub-methods. This set of sub-methods will give the desired implementation of the method.

```
4 "Each sub-method is implemented in a different object."
```

Feature 4 indicates that each sub-method has its own object. This means that sub-methods are separated from each other at an object level.

```
5 "The existing object will not be affected."
```

Feature 5 declares that the existing object will not be affected in any way during the implementation of the adaptability solution. This means that the existing object in which the problem applies, does not have to be recompiled.

```
6 "Change is implemented by running a configuration program which sequentially cascades sub-
  methods."
```

Feature 6 states that the desired change will be implemented by running a configuration program. This configuration program sequentially cascades a selected set of sub-methods.

To detect the user's adaptability need based on the client's and adaptability model, we use a decision tree. The user will run the decision tree and identify adaptability features. The decision tree is depicted in Figure 4.2.

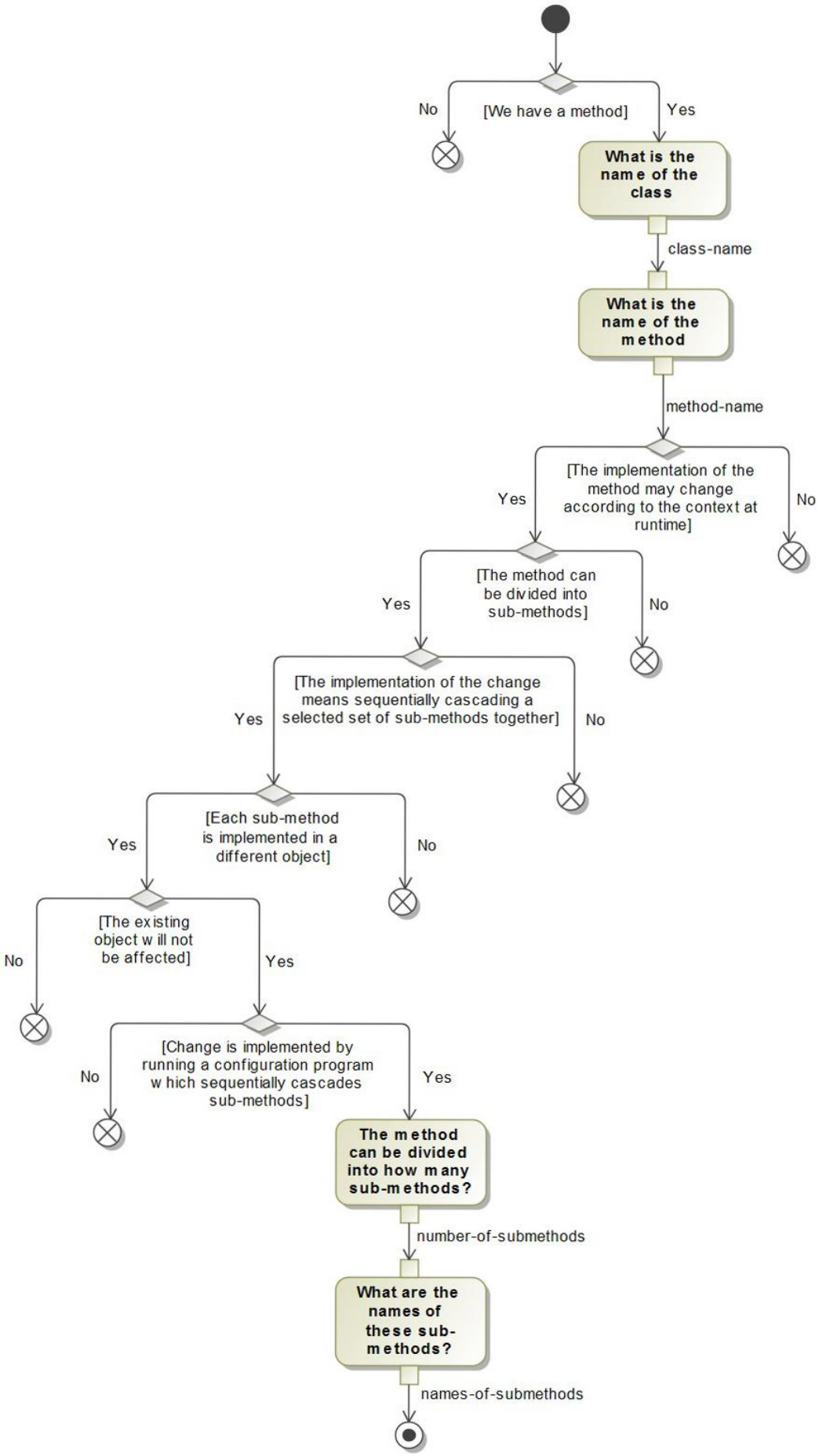

***Figure 4.2**: Decision tree to detect the user's adaptability need*

The decision tree starts with the black circle at the top of the figure. The arrow from this starting point indicates the direction of flow. A diamond-shaped symbol represents a decision with two possible outcomes that are represented by outgoing flows. A rounded rectangle represents an action. These actions are used by DPRF to ask for additional information from the user. The white circle with a black cross sign indicates that the flow is ended, and this occurs when the framework can't process the decisions to a particular design pattern. The white circle with an inner black circle at the bottom of the tree represents the final node and states that the decision tree is executed successfully.

DPRF runs the decision tree and guides the user to identify his adaptability need. The framework asks the user to make a set of decisions and requests additional information.

4.3 Part II - Pattern matching

In this section, we describe the pattern matching process between the pattern repository and the adaptability need. The adaptability need is a set of adaptability features according to the user's need as described in the previous section.

The pattern repository represents a set of design patterns. The patterns are represented internally as Prolog-facts. Each pattern is defined as a set of facts to represent its structure. Each element and relations between the elements is represented. We use the MACH tool as a means to generate the Prolog facts, based on the UML diagram of the design pattern. Internally, MACH has a library of Prolog facts which are used in the representation of diagrams.

To match the pattern with the user's adaptability need, we define the related adaptability features. For each pattern, a subset of adaptability features from the adaptability model is designed. This subset represents the needs for adaptability that can be implemented by the pattern. This states that each pattern in the pattern repository consists of two parts: the internal representation and the related adaptability features.

The related adaptability features are represented as Prolog facts to implement the pattern matching process. The related features are stated as follows:

$$adaptability\_feature(P, F, Y).$$

'adaptability_feature' is a Prolog predicate, where *P* represents the name of the pattern, *F* represents the identifier of the adaptability feature, and *Y* represents the description of the feature. *Y* is not relevant to the matching process, as it does not identify the feature or pattern.

The user's adaptability need is also represented as a set of Prolog facts. These facts are stated as follows:

$$adaptability\_need(F, Y).$$

'adaptability_need' is the second Prolog predicate, where *F* represents the identifier of the adaptability feature and *Y* represents the description of the feature. Y is irrelevant, as it does not identify the adaptability feature.

DPRF will find the pattern that has the same adaptability features as defined in the adaptability need. Referring to the stated Prolog facts, DPRF matches *F* between both predicates. There is a match when all the adaptability needs are stated as adaptability features (same *F*) and are related to the same pattern, *P*.

If there is a match, the related pattern *P* serves as an output of the matching process and is identified as a suitable design pattern to implement the desired adaptability need.

4.4 Part III - Merger

In this section, we describe the merging process between the client's model and a suitable design pattern. The client's model is already recognized and represented internally in the first part of DPRF. By executing the decision tree, the framework knows to which element the adaptability problem applies.

DPRF merges the pattern with the client's model by transforming the client's model into an adaptable model. This transformation applies transformation rules to produce the adaptable model, which represents the solution to the user's adaptability problem.

The framework consolidates the internal representation of the pattern with the internal representation of the client's model, by finding the object(s) in the pattern model to be replaced by object(s) in the client's model. DPRF transforms generic objects of the pattern into instantiated objects according to the client's model and adaptability need.

The decision tree as described in section 4.2, asks the user to identify the object(s) or methods of the client's model to merge with the pattern in preparation of the merging process.

The transformation rules that are applied on the client's model are dependent on the pattern structure, which states that every pattern has its own set of transformation rules.

The output of the merger is the internal representation of the client's adaptable model. It represents the implementation of the user's adaptability need with a suitable pattern.

5. Application of Design Pattern Recognition Framework

In this section, with the help of DPRF, we will describe how an appropriate pattern can be identified and applied to address a specific adaptability problem that may appear in applications.

In section 5.1, we will describe an application by defining the requirements and UML model. In section 5.2, we will describe the detection of the user's adaptability need by defining the evolution scenario and adaptability facts related to these needs. In section 5.3, we will describe the matching of a pattern in the pattern repository with the user's adaptability needs. In this section we will represent the decorator pattern internally and define adaptability features that can be implemented by the pattern. In section 5.4, we will describe the merging process between the client's model and the derived pattern by using transformation rules.

5.1 An example application

We elaborate the application of DPRF with an example of an application for an airline to administer flight seats. The requirements of this application are defined in Listing 5.1.

```
Requirement 1 - "Maintain a list of all flight seats."
Requirement 2 - "Get the details of a flight seat."
Requirement 3 - "Print the details of a flight seat."
Requirement 4 - "Get the price of a flight seat."
```

***Listing 5.1**: Requirements example application*

The architecture of the application is designed on the basis of requirements as a UML model, illustrated in Figure 5.1.

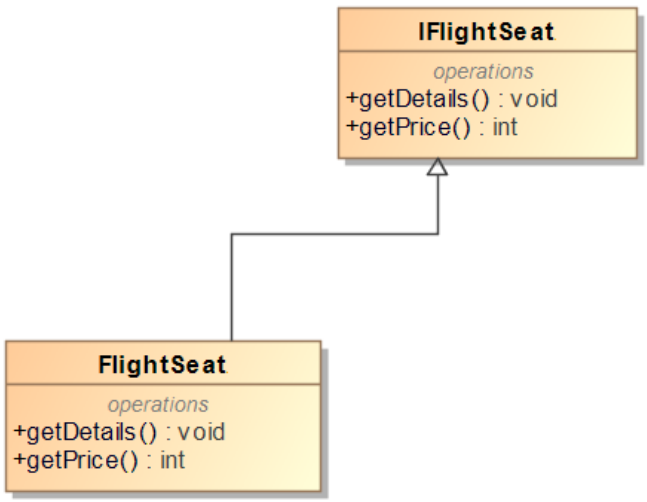

***Figure 5.1**: UML-model in example application*

The example application has two classes: FlightSeat and IFlightSeat. IFlightSeat is the interface of FlightSeat with two methods: ‘getDetails()’ and ‘getPrice()’. The method ‘getDetails()’ prints the details and the method ‘getPrice()’ returns the price of the flight seat.

The UML model represents the client’s model of DPRF. The client’s model is the internal representation of the designer’s initial UML model. MACH loads the diagram and converts it to a set of Prolog facts.

In the following, step-by-step, we will describe how the diagram in Figure 5.1 is represented as Prolog facts. Each fact is represented within a rectangle and is numbered from 0 to 13. The fact number is outside the rectangle on the left hand side.

```
0 client_model(model-0,[annotation-id(1),ownedMember-ids([1,2,3,9]),name-'Clients-model']).
```

‘client_model’ is provided to the MACH tool and states that this fact refers to the client’s model. Every time MACH generates a Prolog representation of a UML diagram, it generates the fact ‘model’ with an index starting from 0.

Each time a new fact is generated, the index number is incremented. The expression between the brackets ‘[’ and ‘]’ defines the properties of the fact. Here, the keyword ‘annotation’ represents the facts that are attached to this model as metadata. The value ‘id(1)’ of this property refers to the fact with an index of 1.

The keyword ‘ownedMember’ refers to sub-facts, which are represented as ‘1, 2, 3, 9’ between the brackets ‘[’ and ‘]’. The keyword ‘name’ is generated by the tool and indicates that this is the client’s model.

```
1 client_model(comment-1,[body-'Author:IPIJNENB',annotatedElement-id(0)]).
```

The fact ‘comment’ is used to indicate the author of the model as a comment. The keywords ‘body’ and ‘annotatedElement’ refer to the content of the comment and the corresponding model ID, respectively.

```
2 client_model(diagram-2,[name-'UML-diagram',visibility- (public),ownerOfDiagram-eee_1]).
```

The fact ‘diagram’ indicates that the name of the diagram is ‘UML-diagram’ and the visibility of the diagram ‘(public)’ and the owner of the diagram is ‘eee_1’.

```
3 client_model(class-3,[ownedMember-ids([4,5,7]),name-'FlightSeat']).
```

The fact ‘class’ states that the facts that are referred to by ‘4, 5, 7’ are related facts and the name of the class is ‘FlightSeat’.

```
4 client_model(generalization-4,[general-'9']).
```

The fact ‘generalization’ indicates that there is a generalization relation between the class that refers to this fact and the class that is referred to by the identifier ‘9’. Since this fact was referred to by the previous fact, the generalization relation is defined between class FlightSeat (declared by fact 3) and the class referred to by the identifier 9.

```
5 client_model(operation-5,[ownedMember-ids([6]),name-getDetails,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class FlightSeat and states that the fact that it is referred to by the number ‘6’ is a related fact. The fact also declares that the name of this operation is ‘getDetails’.

```
6 client_model(parameter-6,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value (type void).

```
7 client_model(operation-7,[ownedMember-ids([8]),name-getPrice,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class FlightSeat and the name of this operation is ‘getPrice’.

```
8 client_model(parameter-8,[visibility- (public),direction-return,type-int]).
```

The fact ‘parameter’ was referred to by fact 7 and states that the operation returns a value of type ‘int’.

```
9 client_model(class-9,[ownedMember-ids([10,12]),name-'IFlightSeat']).
```

The fact ‘class’ states that the name of the class is ‘IFlightSeat’ and the facts that are referred to by the numbers ‘10’ and ‘12’ are related facts.

```
10 client_model(operation-10,[ownedMember-ids([11]),name-getDetails,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class IFlightSeat and the name of this operation is ‘getDetails’.

```
11 client_model(parameter-11,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value (type void).

```
12 client_model(operation-12,[ownedMember-ids([13]),name-getPrice,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class IFlightSeat and the name of this operation is ‘getPrice’.

```
13 client_model(parameter-13,[visibility- (public),direction-return,type-int]).
```

The fact ‘parameter’ was referred to by fact 12 and states that the operation returns a value of type ‘int’.

Listing 5.2 provides an overview of all Prolog facts together as a Prolog program that represents the UML diagram of the example application.

```
0  client_model(model-0,[annotation-id(1),ownedMember-ids([1,2,3,9]),name-'Clients-model']).
1  client_model(comment-1,[body-'Author:IPIJNENB',annotatedElement-id(0)]).
2  client_model(diagram-2,[name-'UML-diagram',visibility- (public),ownerOfDiagram-eee_1]).
3  client_model(class-3,[ownedMember-ids([4,5,7]),name-'FlightSeat']).
4  client_model(generalization-4,[general-'9']).
5  client_model(operation-5,[ownedMember-ids([6]),name-getDetails,visibility- (public)]).
6  client_model(parameter-6,[visibility- (public),direction-return,type-int]).
7  client_model(operation-7,[ownedMember-ids([8]),name-getPrice,visibility- (public)]).
8  client_model(parameter-8,[visibility- (public),direction-return]).
9  client_model(class-9,[ownedMember-ids([10,12]),name-'IFlightSeat']).
10 client_model(operation-10,[ownedMember-ids([11]),name-getDetails,visibility- (public)]).
11 client_model(parameter-11,[visibility- (public),direction-return]).
12 client_model(operation-12,[ownedMember-ids([13]),name-getPrice,visibility- (public)]).
13 client_model(parameter-13,[visibility- (public),direction-return,type-int]).
```

***Listing 5.2**: Client’s model in example solution*

5.2 Part I - Detecting the adaptability need

This section elaborates the detection of the user’s adaptability need. In section 5.2.1, we define the evolution scenario by defining new requirements. In section 5.2.2, we identify adaptability facts that are related to the needs of the user.

5.2.1 Evolution scenario

Assume that the requirements of the example application have changed so that flight seats now include additional features, like WiFi or DVD. The set of new requirements is defined in Listing 5.3.

```
Requirement 5 - “Ability to add additional features to the flight seats.”
Requirement 6 - “Ability to depend the details of the flight seat on the number of additional
features.”
```

***Listing 5.3**: New requirements example application*

Requirement 5 states that it should be possible to include additional features to flight seats, like WiFi or DVD, depending on the context. Requirement 6 states that the details of the flight seat changes, depending on the number of additional features.

The existing application, as described in section 5.1, does not support the new requirements. We would like to assist the modeler to find which pattern can be used to implement the change.

5.2.2 The application of the facts related to the adaptability needs

As depicted and described in section 4.2, a Prolog-based decision tree is used to define the user's adaptability needs. The decision tree will ask the user questions to determine the desired adaptability features. By instantiating generic features in the adaptability model, specific features that refer to the client's model are defined.

The following figures illustrate the decision tree, with explanations of all the instantiated adaptability features and the process used to detect the user's adaptability needs.

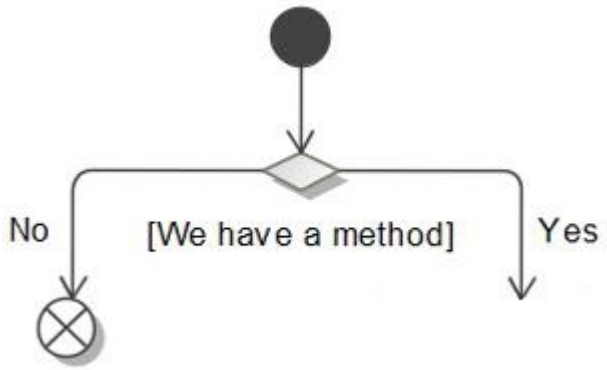

***Figure 5.2**: Decision tree: Step 1*

Figure 5.2 illustrates Step 1 of the decision tree. The decision tree starts with a decision. This decision represents the first adaptability feature "We have a method", determining if there is a method in the client's model that relates to the adaptability problem. The adaptability problem of the example application refers to the method getDetails(), which indicates that this adaptability feature is applicable in the user's situation, so *Yes* is the answer.

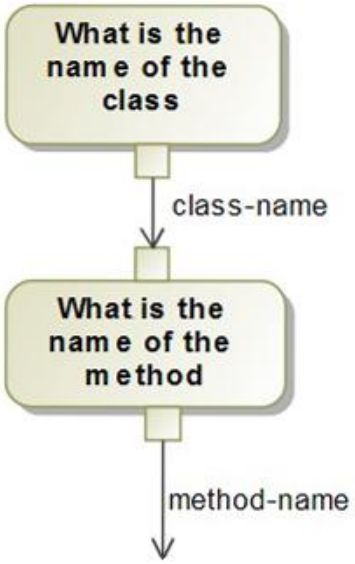

***Figure 5.3**: Decision tree: Step 2*

Figure 5.3 illustrates Step 2 in the decision tree. The two rounded rectangles represent two actions. The first action asks a question of the user: "What is the name of the class?" The small rectangle attached to the action represents a 'pin' and serves as an input or output to

an action. With the first action, the pin represents the output of the question, labeled *class name*. The second action depicts a second question to the user, with *class name* as an input. When this step is completed, the name of the class and method that is referred to by the adaptability problem are defined. Using the example, the name of the corresponding class is *FlightSeat* and the name of the corresponding method is *getDetails*().

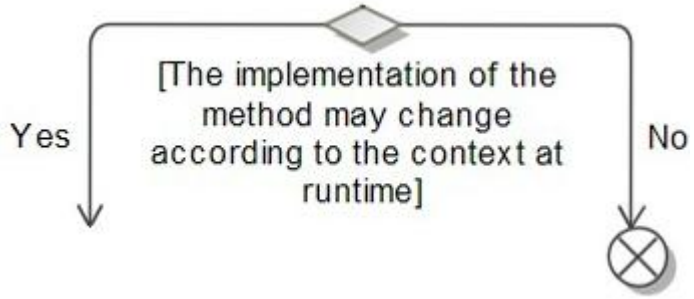

***Figure 5.4**: Decision tree: Step 3*

Figure 5.4 illustrates the Step 3 in the decision tree and represents the adaptability feature. The decision is whether the statement “The implementation of the method may change according to the context at runtime” is true or not. In the example, this would mean that the implementation of the method *getDetails()* in the class *FlightSeat* may change at runtime. The adaptability problem of the example indicates that this is the case, as details of a flight seat may change depending on the number of additional features. The user will therefore choose *Yes* for this decision.

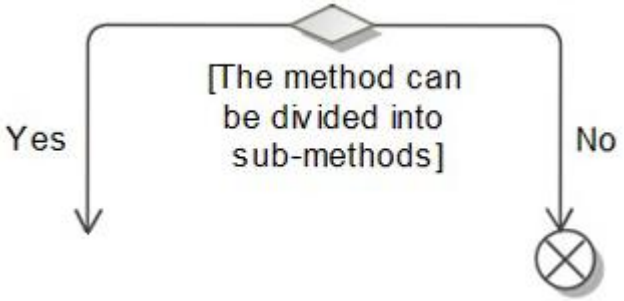

***Figure 5.5**: Decision tree: step 4*

Figure 5.5 depicts Step 4 in the decision tree. The user must choose whether the adaptability feature “The method can be divided into sub-methods” is applicable to his situation. Using the example, this adaptability feature states that the method *getDetails()* can be divided into a number of sub-methods. Because every additional feature has its own details, the method can be divided per feature and the user will choose *Yes*.

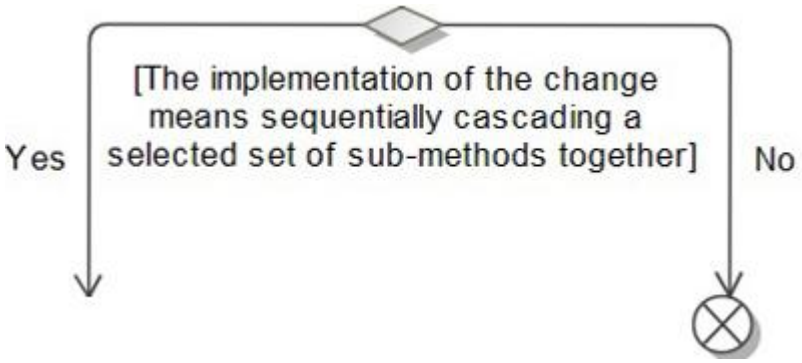

***Figure 5.6**: Decision tree: Step 5*

Figure 5.6 illustrates Step 5 in the decision tree, and lets the user choose if the implementation of the change means sequentially cascading a selected set of sub-methods together. In the example application, an instance of a flight seat has a set of additional features. A flight seat can have zero, one, or two additional features and the context determines the selected set. As this adaptability feature is applicable in this instance, the user will choose *Yes*.

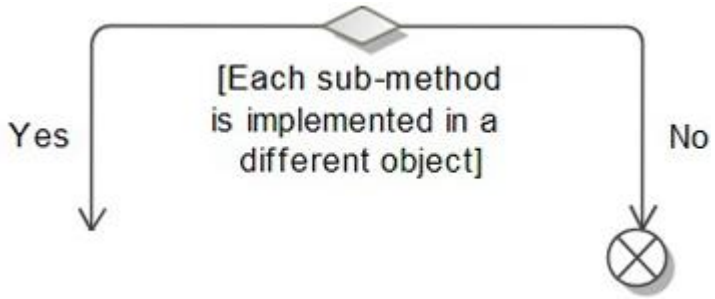

***Figure 5.7**: Decision tree: Step 6*

Figure 5.7 depicts Step 6 of the decision tree, which identifies another adaptability feature: “Each sub-method is implemented in a different object”. For the user this is less interesting, but it is important to know if this is fine to the user. In the example application, the additional features can be implemented in a different object, which means the user will choose *Yes*.

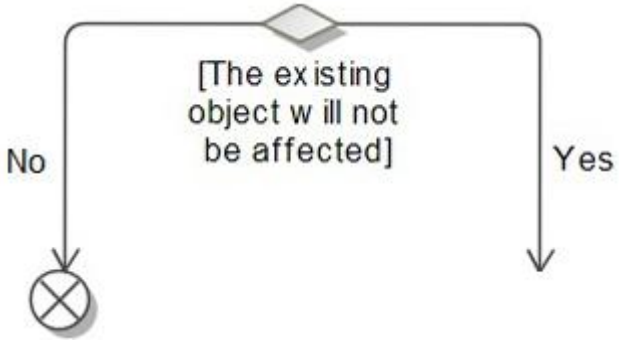

***Figure 5.8**: Decision tree: Step 7*

Figure 5.8 illustrates Step 7 of the decision tree, which asks the user if the existing object will not be affected by the implementation of the change. In the example application, the user desires that the *FlightSeat* object should not change, as it is important for the system to keep the existing functionality as it is. This indicates that the user will choose *Yes* for this decision.

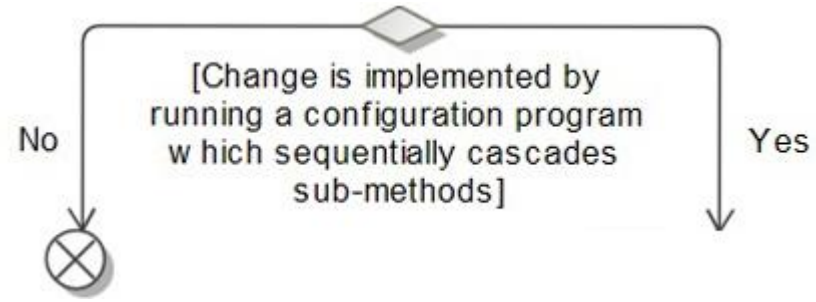

***Figure 5.9**: Decision tree: Step 8*

Figure 5.9 depicts Step 8 of the decision tree, which asks the user to assess another adaptability feature: *Change is implemented by running a configuration program which sequentially cascades sub-methods.* In the example application, the client-program serves as a configuration program and determines which additional features will be added to the flight seat. It also defines the order in which the features are added. As this adaptability feature is applicable for the user, he or she will choose *Yes*.

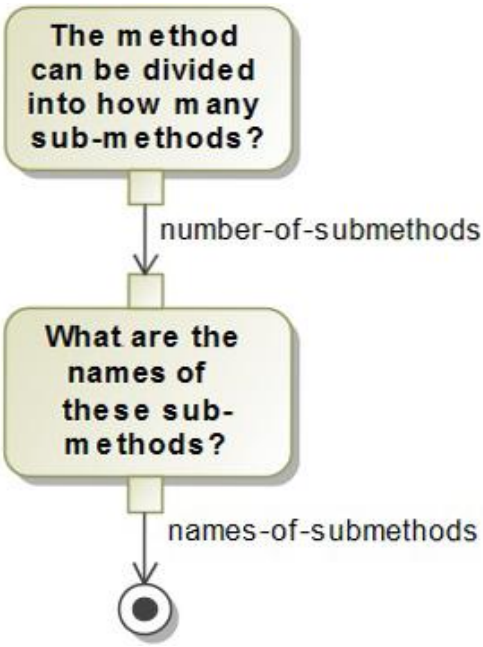

***Figure 5.10**: Decision tree: Step 9*

Figure 5.10 illustrates Step 9 the decision tree, which is the final step. It asks the user for additional information that is needed for the implementation of the pattern. This step involves two actions that must take place before the flow arrives at its final state. The first action asks a question of the user: “The method can be divided into how many sub-methods?” The second action asks another question: “What are the names of these sub-methods?”

The user will identify the number of sub-methods, by identifying the number of additional features that are possible. In the example application there are two additional features (DVD and WiFi), which means that the method can be divided into two sub-methods with the names *WiFi* and *DVD*. This step is the final step in the decision tree and returns the user’s adaptability needs.

The user's adaptability needs is a set of adaptability features that apply to the user's needs and initial model. The adaptability needs is the output of the executed decision tree. The set of adaptability features that is identified by the user of the example application is depicted in Listing 5.4.

```
adaptability_need(adaptability-feature-0, "We have a method m1()").

adaptability_need(adaptability-feature-1, "The implementation of method m1() may change
according to the context at runtime").

adaptability_need(adaptability-feature-2, "The method m1() can be divided into sub-methods").

adaptability_need(adaptability-feature-3, "The implementation of the change means sequentially
cascading a selected set of sub-methods together").

adaptability_need(adaptability-feature-4, "Each sub-method is implemented in a different
object").

adaptability_need(adaptability-feature-5, "The existing object will not be affected").

adaptability_need(adaptability-feature-6, "Change is implemented by running a configuration
program which sequentially cascades sub-methods").
```

***Listing 5.4**: Adaptability need in example solution*

5.3 Part II - Pattern matching

In the previous section, we defined the user's adaptability need. In this section we define the matching process between the pattern repository and the user's adaptability need. First, we describe the pattern repository, using the Decorator pattern. We represent each pattern in the pattern repository internally and define the adaptability features that can be identified by the patterns. Second, we will define the matching process between the pattern repository and the user's adaptability need.

5.3.1 Pattern repository: Decorator pattern

As described in section 3.2.1, the decorator pattern can be implemented in various forms. One form is illustrated as a UML diagram in Figure 5.11, which we refer to as 'Variation 1'.

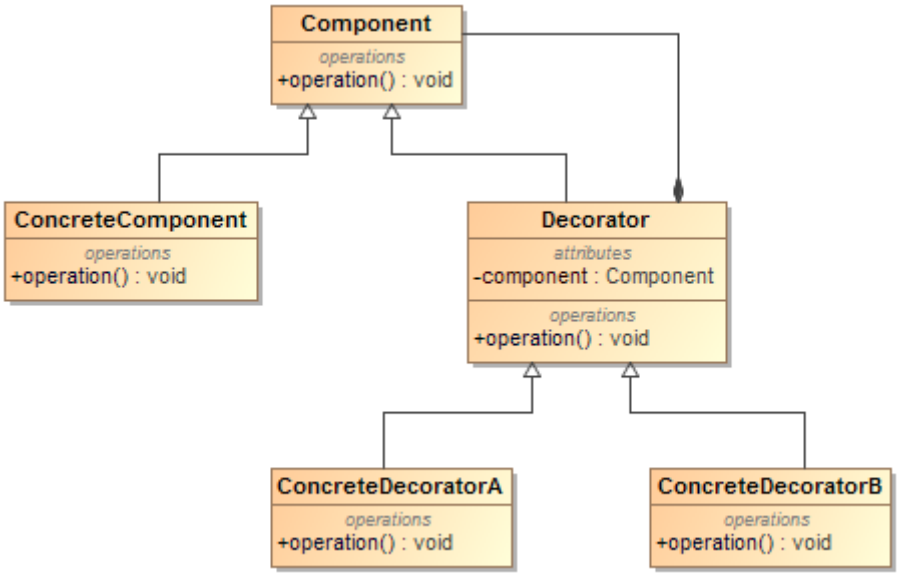

***Figure 5.11**: Model of decorator pattern (variation 1)*

We use the MACH tool as a means to generate pattern representations internally. As explained in section XXX, MACH uses the Prolog language to represent UML diagrams. Each UML diagram is therefore a Prolog program. Each element and relations between the elements are represented as Prolog facts.

In the following, we describe how the diagram in Figure 5.11 is represented as Prolog facts. Each fact is represented within a rectangle and is numbered from 0 to 20. The number of a fact is shown outside the rectangle on the left.

0
```
d_1(model-0,[annotation-id(1),ownedMember-ids([1,2,3,9,13,14,18,22]),name-'UML-model']).
```

'd_1', on the left side of the expression, is provided to the MACH tool and refers to 'variation 1' of the decorator pattern. In the set of properties, the keyword 'annotation' represents the facts that are attached to this model as metadata. This property refers to the fact with an index of 1. The keyword 'ownedMember' refers to sub-facts, represented as '1, 2, 3, 9, 13, 14, 18, 22'. The keyword 'name' indicates that this is a UML model.

1
```
d_1(comment-1,[body-'Author:IPIJNENB',annotatedElement-id(0)]).
```

The fact 'comment' is used to indicate the author of the model as a comment. The keywords 'body' and 'annotatedElement' refer to the content of the comment and the corresponding model ID, respectively.

2
```
d_1(diagram-2,[name-'UML-diagram',visibility- (public),ownerOfDiagram-eee_1]).
```

The fact 'diagram' indicates that the name of the diagram is 'UML-diagram', the visibility of the diagram '(public)', and the owner of the diagram is 'eee_1'.

```
3 d_1(class-3,[ownedMember-ids([4,5,6,7]),name-'Decorator']).
```

The fact ‘class’ states that the facts that are referred to by ‘4, 5, 6, and 7’ are related facts and the name of the class is ‘Decorator’.

```
4 d_1(generalization-4,[general-'9']).
```

The fact ‘generalization’ indicates that there is a generalization relation between the class that refers to this fact and the class that is referred to by the identifier ‘9’. Since this fact was referred to by the previous fact, the generalization relation is defined between class Decorator (declared by fact 3) and the class referred to by the identifier ‘9’.

```
5 d_1(property-5,[name-component,visibility-private,type-'9']).
```

The fact ‘property’ declares the ‘private’ attribute ‘component’, which has the type of the class declared by the fact ‘9’.

```
6 d_1(property-6,[visibility-private,type-'9',association-'_4361',aggregation-composite]).
```

The fact ‘property’ declares the ‘private’ composite-association between class Decorator and the class which is declared by the fact ‘9’. The identifier of this relation is defined with the identifier ‘4361’.

```
7 d_1(operation-7,[ownedMember-ids([8]),name-operation,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class Decorator and states that the fact that is referred to by the number ‘8’ is a related fact. The fact also declares that the name of this operation is ‘operation’.

```
8 d_1(parameter-8,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value (type void).

```
9 d_1(class-9,[ownedMember-ids([10,11]),name-'Component']).
```

The fact ‘class’ states that the facts that are referred to by the numbers ‘10 and 11’ are related facts and the name of the class is ‘Component’.

```
10 d_1(property-10,[visibility-private,type-'3',association-'_4361']).
```

The fact ‘property’ declares the ‘private’ association between class Component and the class which is declared by the fact ‘3’. The identifier of this relation is defined with the identifier ‘4361’.

```
11 d_1(operation-11,[ownedMember-ids([12]),name-operation,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class Component and the name of this operation is ‘operation’.

```
12 d_1(parameter-12,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value.

```
13 d_1(association-13,[memberEnd-ids([10,6])]).
```

The fact ‘association’ declares the association between class Decorator and the class Component. The fact states that the facts that are referred to by ‘10 and 6’ are related facts that represent both ends of the association.

```
14 d_1(class-14,[ownedMember-ids([15,16]),name-'ConcreteComponent']).
```

The fact ‘class’ states that the facts that are referred to by ‘15’ and ‘16’ are related facts and the name of the class is ‘ConcreteComponent’.

```
15 d_1(generalization-15,[general-'9']).
```

The fact ‘generalization’ declares the generalization relation between the class that refers to this fact and the class that is referred to by the identifier ‘9’. Since this fact was referred to by fact 14, the generalization relation is defined between class ConcreteComponent and the class declared by the fact ‘9’.

```
16 d_1(operation-16,[ownedMember-ids([17]),name-operation,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class ConcreteComponent and the name of this operation is ‘operation’.

```
17 d_1(parameter-17,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value.

```
18 d_1(class-18,[ownedMember-ids([19,20]),name-'ConcreteDecoratorB']).
```

The fact ‘class’ states that the name of the class is ‘ConcreteDecoratorB’ and the facts that are referred to by ‘19’ and ‘20’ are related facts.

```
19 d_1(generalization-19,[general-'3']).
```

The fact ‘generalization’ declares the generalization relation between class ConcreteDecoratorB and the class declared by the fact ‘3’.

```
20 d_1(operation-20,[ownedMember-ids([21]),name-operation,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class ConcreteDecoratorB and the name of this operation is ‘operation’.

```
21 d_1(parameter-21,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value.

```
22 d_1(class-22,[ownedMember-ids([23,24]),name-'ConcreteDecoratorA']).
```

The fact ‘class’ states that the name of the class is ‘ConcreteDecoratorA’ and the facts that are referred to by ‘23’ and ‘24’ are related facts.

```
23 d_1(generalization-23,[general-'3']).
```

The fact ‘generalization’ declares the generalization relation between class ConcreteDecoratorA and the class declared by the fact ‘3’.

```
24 d_1(operation-24,[ownedMember-ids([25]),name-operation,visibility- (public)]).
```

The fact ‘operation’ declares the ‘public’ operation of class ConcreteDecoratorA and the name of this operation is ‘operation’.

```
25 d_1(parameter-25,[visibility- (public),direction-return]).
```

The fact ‘parameter’ was referred to by the previous fact and states that the operation returns no value.

Listing 5.5 will give an overview of all Prolog facts together as a Prolog program that represents the UML diagram of the Decorator pattern (variation 1).

```
0  me(model-0,[annotation-id(1),ownedMember-ids([1,2,3,9,13,14,18,22]),name-'UML-model']).
1  d_1(comment-1,[body-'Author:IPIJNENB',annotatedElement-id(0)]).
2  d_1(diagram-2,[name-'UML-diagram',visibility- (public),ownerOfDiagram-eee_1]).
3  d_1(class-3,[ownedMember-ids([4,5,6,7]),name-'Decorator']).
4  d_1(generalization-4,[general-'9']).
5  d_1(property-5,[name-component,visibility-private,type-'9']).
6  d_1(property-6,[visibility-private,type-'9',association-'_4361',aggregation-composite]).
7  d_1(operation-7,[ownedMember-ids([8]),name-operation,visibility- (public)]).
8  d_1(parameter-8,[visibility- (public),direction-return]).
9  d_1(class-9,[ownedMember-ids([10,11]),name-'Component']).
10 d_1(property-10,[visibility-private,type-'3',association-'_4361']).
11 d_1(operation-11,[ownedMember-ids([12]),name-operation,visibility- (public)]).
12 d_1(parameter-12,[visibility- (public),direction-return]).
13 d_1(association-13,[memberEnd-ids([10,6])]).
14 d_1(class-14,[ownedMember-ids([15,16]),name-'ConcreteComponent']).
15 d_1(generalization-15,[general-'9']).
16 d_1(operation-16,[ownedMember-ids([17]),name-operation,visibility- (public)]).
17 d_1(parameter-17,[visibility- (public),direction-return]).
18 d_1(class-18,[ownedMember-ids([19,20]),name-'ConcreteDecoratorB']).
19 d_1(generalization-19,[general-'3']).
20 d_1(operation-20,[ownedMember-ids([21]),name-operation,visibility- (public)]).
21 d_1(parameter-21,[visibility- (public),direction-return]).
22 d_1(class-22,[ownedMember-ids([23,24]),name-'ConcreteDecoratorA']).
23 d_1(generalization-23,[general-'3']).
24 d_1(operation-24,[ownedMember-ids([25]),name-operation,visibility- (public)]).
25 d_1(parameter-25,[visibility- (public),direction-return]).
```

***Listing 5.5**: Prolog model of decorator pattern (variation 1)*

5.3.2 Pattern matching

In this section, we define the matching process between patterns in the pattern repository and the user's adaptability need by applying matching rules.

The Prolog model of a pattern, like the Prolog model of the decorator pattern represented in Listing 5.5, is a graphical representation of the UML diagram. This representation only shows elements and the relations between these elements. The representation does not provide an explicit meaning of a decorator pattern. A class diagram only shows static relationships and does not indicate the dynamic properties of the decorator pattern. In the decorator pattern, extensions of a method happen at runtime.

We determine all the needs for adaptability that can be implemented by the pattern by identifying adaptability features. This defined set of adaptability features is a subset of the adaptability model. In the following, we describe the subset of corresponding adaptability features of the decorator pattern defined as Prolog facts.

```
0  adaptability_feature(d_1, adaptability-feature-0, "We have a method m1()").
```

Adaptability feature 0 is linked to the decorator pattern (d_1), because the pattern can be applied for adding encapsulated responsibility to some method.

```
1  adaptability_feature(d_1, adaptability-feature-1, "The implementation of method m1() may change
   according to the context at runtime").
```

Adaptability feature 1 is related to the pattern, because the decorator-object can create one or multiple concrete decorators. These decorators can be added at runtime to change the implementation of some method.

```
2 adaptability_feature(d_1, adaptability-feature-2, "The method m1() can be divided into sub-
methods").
```

Adaptability feature 2 is associated with the pattern, as the additional implementation of some method can be divided into one or more concrete decorators.

```
3 adaptability_feature(d_1, adaptability-feature-3, "The implementation of the change means
sequentially cascading a selected set of sub-methods together").
```

Adaptability feature 3 is linked to the pattern, as the pattern gives the possibility to select a set of concrete decorators according to the requirements of the client program.

```
4 adaptability_feature(d_1, adaptability-feature-4, "Each sub-method is implemented in a
different object").
```

Adaptability feature 4 is related to the pattern, as every concrete decorator is a different object. The sub-methods are represented by concrete decorators, which indicates that this feature is associated with the decorator pattern.

```
5 adaptability_feature(d_1, adaptability-feature-5, "The existing object will not be affected").
```

Adaptability feature 5 is linked to the pattern, as the existing component-object will not be affected. The implementation of the component-object will stay the same.

```
6 adaptability_feature(d_1, adaptability-feature-6, "Change is implemented by running a
configuration program which sequentially cascades sub-methods").
```

Adaptability feature 6 is related to the pattern, because the client contains a configuration program that determines which concrete decorators should be executed. The configuration program also defines the order in which it is to be executed.

We now define the matching process between the pattern repository and the adaptability need. As described in section 4.3, DPRF will find the pattern that has the same adaptability features as those defined in the adaptability need. The adaptability need is defined in Listing 5.4 and consists of adaptability features 1, 2, 3, 4, 5, and 6.

DPRF confirms if these features are defined in the pattern repository by using Prolog queries. DPRF has various queries to ask the repository if there is a pattern that contains the adaptability features 1 to 6. The framework will return ‘d_1’ as an output of the matching process, as the decorator pattern contains all adaptability features defined at the beginning of this section.

5.4 Part III - Merger

In this section, we describe the merging process between the pattern, which is matched with the user's adaptability needs, and the client's model. In the previous section the decorator pattern was matched with the adaptability needs. Now the merging process between the decorator pattern and the client's model will be defined as described in section 5.1. This will result in an adaptable model. Section 5.4.1 describes the merger and section 5.4.2 defines the resulting model.

5.4.1 Merger

As described in the overview of DPRF, the framework consolidates the internal representation of the pattern with the internal representation of the client's model to create the client's desired adaptable model.

The internal representation of the decorator pattern is defined in Listing 5.5. DPRF finds one or more objects in the pattern model that can be replaced by objects from the client's model to unite the two models.

The decorator pattern decorates a method that is defined in the objects *Component* and *ConcreteComponent*, which indicates that these two objects are replaceable by objects of the client's model.

As we described in section 5.2.2, DPRF asked the user for the names of the class and method that needs to be decorated. The user stated that the name of the class is *FlightSeat* and that the method of the class has the name *getDetails*. DPRF also requested some additional information as preparation for the merging-process, such as the number of decorators with their corresponding names.

In order to merge the client's model with the decorator pattern, three new classes will be defined by DPRF: FlightSeatDecorator, DVDDecorator and WiFiDecorator. FlightSeatDecorator represents the decorator class. DVDDecorator and WiFiDecorator represent the concrete decorators.

DPRF implements FlightSeatDecorator as a subclass of IFlightSeat. FlightSeatDecorator contains an attribute with the name flightSeat of type IFlightSeat and inherits the methods getDetails and getPrice. DVDDecorator and WiFiDecorator are both subclasses of FlightSeatDecorator and they both inherit the two methods as well.

Now the user has the ability to implement the additional details in the method getDetails of DVDDecorator and WiFiDecorator. The user can decide if the software program first executes the initial implementation of the method or if it first executes the new additional set of statements.

5.4.2 Client's adaptable model

The internal representation of the client's adaptable model is defined in Listing 5.6. This model represents the output of the framework. This model is the consolidation of the decorator pattern and the client's model to implement the user's adaptability needs.

```
0  adaptable_model(model-0,[annotation-id(1),ownedMember-ids([1,2,3,9,15,23,29,35]),name-
   'clients_adaptable_model']).
1  adaptable_model(comment-1,[body-'Author:IPIJNENB',annotatedElement-id(0)]).
2  adaptable_model(diagram-2,[name-'UML-diagram',visibility- (public),ownerOfDiagram-eee_1]).
3  adaptable_model(class-3,[ownedMember-ids([4,5,7]),name-'FlightSeat']).
4  adaptable_model(generalization-4,[general-'9']).
5  adaptable_model(operation-5,[ownedMember-ids([6]),name-getDetails,visibility- (public)]).
6  adaptable_model(parameter-6,[visibility- (public),direction-return]).
7  adaptable_model(operation-7,[ownedMember-ids([8]),name-getPrice,visibility- (public)]).
8  adaptable_model(parameter-8,[visibility- (public),direction-return,type-int]).
9  adaptable_model(class-9,[ownedMember-ids([10,11,13]),name-'IFlightSeat']).
10 adaptable_model(property-10,[visibility-private,type-'15',association-'_4453']).
11 adaptable_model(operation-11,[ownedMember-ids([12]),name-getDetails,visibility- (public)]).
12 adaptable_model(parameter-12,[visibility- (public),direction-return]).
13 adaptable_model(operation-13,[ownedMember-ids([14]),name-getPrice,visibility- (public)]).
14 adaptable_model(parameter-14,[visibility- (public),direction-return,type-int]).
15 adaptable_model(class-15,[ownedMember-ids([16,17,18,19,21]),name-'FlightSeatDecorator']).
16 adaptable_model(generalization-16,[general-'9']).
17 adaptable_model(property-17,[name-fligh Seat,visibility-private,type-'9']).
18 adaptable_model(property-18,[visibility-private,type-'9',association-'_4453',aggregation-
   composite]).
19 adaptable_model(operation-19,[ownedMember-ids([20]),name-getDetails,visibility- (public)]).
20 adaptable_model(parameter-20,[visibility- (public),direction-return]).
21 adaptable_model(operation-21,[ownedMember-ids([22]),name-getPrice,visibility- (public)]).
22 adaptable_model(parameter-22,[visibility- (public),direction-return,type-int]).
23 adaptable_model(class-23,[ownedMember-ids([24,25,27]),name-'DVDDecorator']).
24 adaptable_model(generalization-24,[general-'15']).
25 adaptable_model(operation-25,[ownedMember-ids([26]),name-getDetails,visibility- (public)]).
26 adaptable_model(parameter-26,[visibility- (public),direction-return]).
27 adaptable_model(operation-27,[ownedMember-ids([28]),name-getPrice,visibility- (public)]).
28 adaptable_model(parameter-28,[visibility- (public),direction-return,type-int]).
29 adaptable_model(class-29,[ownedMember-ids([30,31,33]),name-'WiFiDecorator']).
30 adaptable_model(generalization-30,[general-'15']).
31 adaptable_model(operation-31,[ownedMember-ids([32]),name-getDetails,visibility- (public)]).
32 adaptable_model(parameter-32,[visibility- (public),direction-return]).
33 adaptable_model(operation-33,[ownedMember-ids([34]),name-getPrice,visibility- (public)]).
34 adaptable_model(parameter-34,[visibility- (public),direction-return,type-int]).
35 adaptable_model(association-35,[memberEnd-ids([10,18])]).).
```

***Listing 5.6**: Internal representation of client's adaptable model*

Listing 5.6 represents the internal structure of a UML diagram. The corresponding UML diagram is illustrated in Figure 5.12.

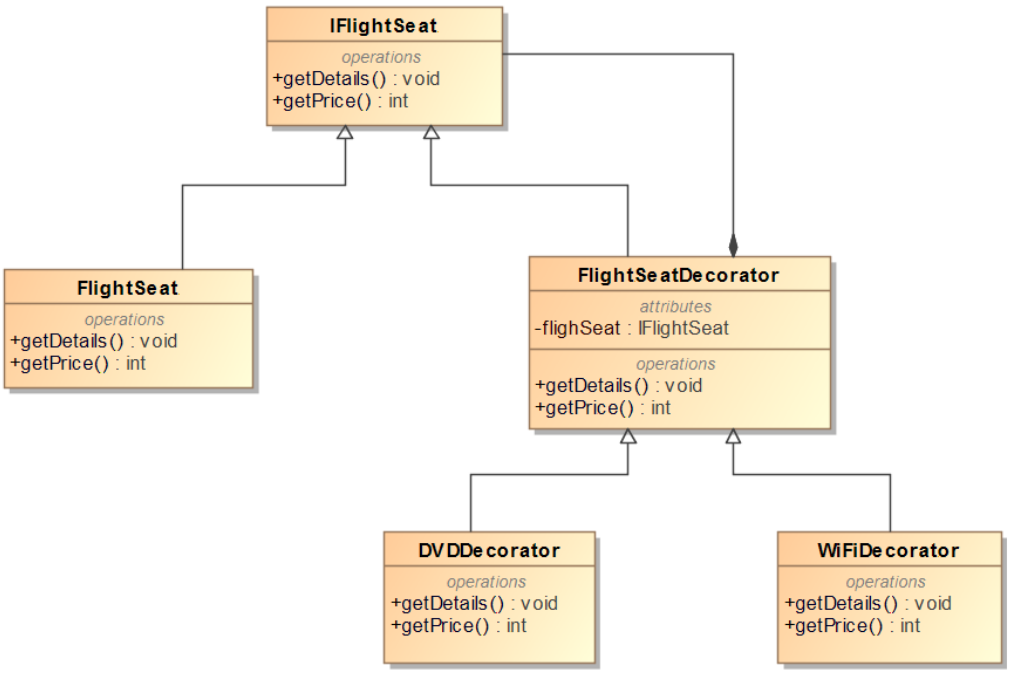

***Figure 5.12**: Client's adaptable model as UML diagram in example solution*

6. Conclusions and future work

The research presented in this thesis has raised new questions. In section 6.1, we will describe the conclusions by identifying the contributions of this project. In section 6.2, we will describe the several lines of research arising from this work, which should be pursued further. In section 6.3 we will reflect on personal experiences and discuss the lessons that have been learned.

6.1 Contributions

This research contributes to the area of software evolution and adaptability in object-oriented languages. It makes three specific contributions, as described below.

(1) A framework to find applicable patterns to implement adaptability

We designed a framework called DPRF that assists the designer in modeling and developing adaptable software. A designer has an initial model and some need to implement adaptability, but it is difficult and unclear how to model this correctly. The designer can use DPRF to find applicable design patterns for implementing the adaptability.

This framework can be used to program a guideline that can help the designers to find suitable patterns for his or her needs. DPRF transforms the client's initial model into a model that is adaptable according to his identified adaptability needs. The client's adaptable model is designed to accommodate new requirements or changes in the technology.

(2) A prototype that implements part of the framework

We developed a prototype that implements a part of the framework. It is developed in Prolog and implements the detection of the user's adaptability need and the pattern matching. This prototype represents a proof of concept to test if the framework can be implemented to serve as a valuable system.

The prototype consults the client's model and adaptability model as a set of Prolog facts. It asks questions of the user according to the decision tree presented in section 4.2. The prototype saves the identified adaptability features in a local database, to match them with the pattern repository.

Thereafter, the prototype consults a file with a set of Prolog facts. This set contains the internal representation of the decorator pattern and serves as the pattern repository. The prototype consults a second file, which contains adaptability features that can be implemented by the decorator pattern. The prototype runs a set of queries to match these adaptability features with the features that are identified by the user. It results in a pattern that is most appropriate for implementing the desired adaptability.

The prototype does not implement the merging process of a suitable pattern with the client's model.

(3) An example of a client's application to demonstrate the framework

The third contribution of this project is to demonstrate the working of the framework by using an example. This example depicts a model for an airline to administer flight seats and describes an evolution scenario. A new set of requirements indicates that the client's model should adapt.

By going through every decision in the decision tree, the framework detects the client's adaptability needs as it identifies a set of adaptability features. To explain the matching process the pattern repository is first described, using the Decorator pattern. Second, the adaptability features that can be implemented by the pattern are identified. In the example, DPRF matches the Decorator pattern with the client's adaptability needs.

Finally, we describe how to merge the Decorator pattern with the client's model by identifying objects in the pattern model that can be replaced by objects from the client's model to unite the two.

6.2 Future work

We designed a solution for the adaptability problem, as formulated in Chapter 1, by providing a framework for designers to model and develop adaptable software correctly. However, there are still some questions that should be pursued. In this section, we describe the several lines of research arising from this work.

(1) Extending the pattern repository with new patterns
In this project, we elaborated only on the decorator pattern. The pattern repository should contain more design patterns for DPRF to be of significant value to the user.

Figure 6.1 illustrates a UML activity diagram that depicts how a developer can extend the pattern repository with new patterns.

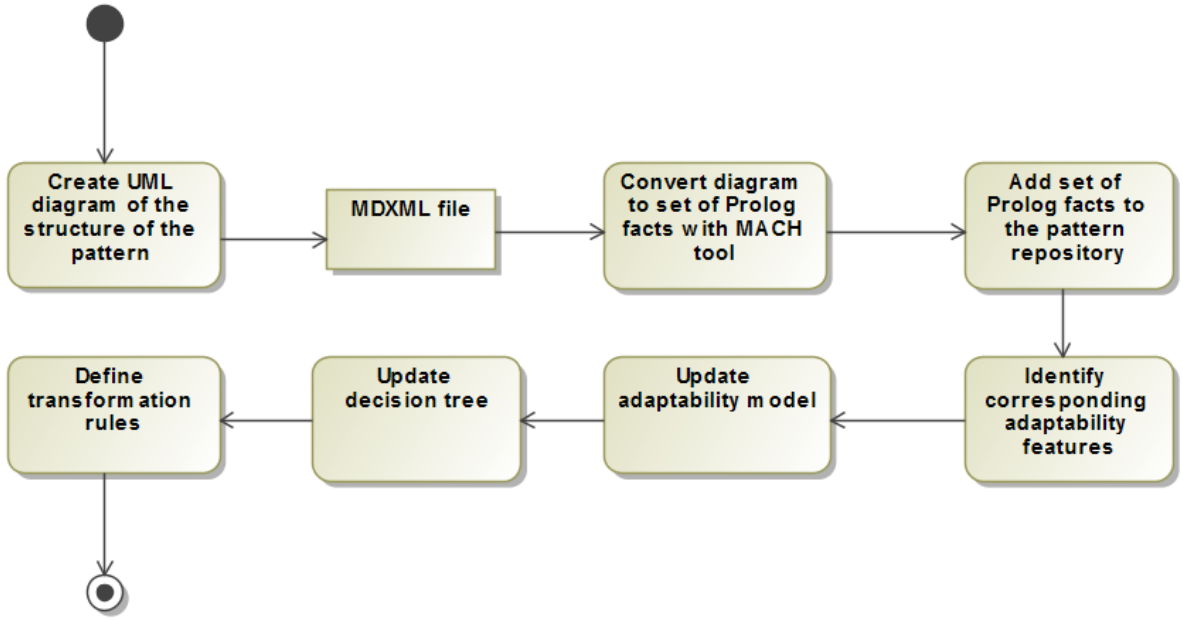

***Figure 6.1**: UML activity diagram for expanding the pattern repository*

The developer needs to create a UML diagram of the structure of the pattern in an MDXML format. This diagram can contain generic names for the elements. This MDXML diagram needs to be converted to a set of Prolog facts using the MACH tool. The developer must give the Prolog predicate a unique identifier that indicates which pattern is represented. This results in the internal representation of the pattern.

The pattern repository is one file with Prolog facts for all defined patterns. The generated set of Prolog facts must be added to the pattern repository by including them to the file.

For DPRF to match the pattern with the user's adaptability needs, it is necessary to define the related adaptability features. It is presumable that the new pattern has adaptability features that are not yet defined in the adaptability model. The developer must identify all corresponding adaptability features from the adaptability model and define the features that need to be added to the model. When new adaptability features are defined, the adaptability model needs to be updated to be representative for all patterns in the repository.

DPRF detects the user's adaptability needs by executing a decision tree. This tree must contain all adaptability features in order to find the corresponding design pattern. The decision tree needs to be updated when a new pattern is introduced. The tree will implement the set of adaptability features for the new pattern as a possible flow.

To complete the extension, transformation rules for the new pattern should be defined in order for DPRF to transform the client's model to the client's adaptable model. This set of rules needs to be defined to identify the corresponding elements between the pattern and the client's model.

The desire to extend this framework has raised the question of whether perhaps it is too costly to extend DPRF with all patterns and their variations, because it is necessary to define

specific adaptability features and formulate unique merging rules for each pattern variation. Future research should take this in consideration.

(2) Reducing ambiguity by adding other criteria
For DPRF to recognize suitable patterns, it is very important that the adaptability features be unambiguous. Ambiguous adaptability features will have a negative effect on the matching process.

In this project we reduced ambiguity by defining explicit adaptability features, to ensure that the correct (and only one) pattern will be recognized. However, to reduce ambiguity even more, other criteria can be added to the design patterns in the matching process.

In this hypothetical situation, the user can identify suitable design patterns by defining which details he or she finds important. For example, when the user wants the fastest solution, DPRF can match this with the patterns in the pattern repository.

(3) Prolog to UML conversion
DPRF merges the internal representation of a suitable design pattern with the internal representation of the client's model. This means that two sets of Prolog facts will be merged into one. This resulting set is the internal representation of the client's adaptable model, according to his or her adaptability needs.

However, DPRF does not support the conversion from Prolog to a UML model. MACH is a tool that can convert UML models to a set of Prolog facts, but unfortunately there is no information about converting in the opposite direction.

To convert the client's adaptable model into a visual UML-model, a tool should be found or developed that can implement this conversion. This tool should have the same transformation rules as the MACH tool, only reversed.

(4) Elaborate on the merging process
Further future work could be to elaborate on the merging process. At this time, DPRF needs custom transformation rules to merge the design pattern with the client's model. It takes a great deal of time to define all rules in the right way.

A good technique for merging the two models is missing and can be elaborated as future work. More research can be conducted to find the best generic technique to implement the merging process. It is important that the designer need configure as little as possible, to reduce the amount of time and effort required of the designer.

(5) Complete the prototype
As we described in section 6.1, we developed a prototype that implements a part of the framework. This prototype has not yet implemented the merging part of the framework. Future work can be to complete this implementation, which would complete the prototype.

The prototype already consults the internal representation of the client's model and the internal representation of a suitable design pattern. The goal of the merger is to unite the two models into one, by using custom transformation rules or other techniques.

6.3 Lessons learned

The most important lesson learned in this project is the importance of designing a good construction to develop the framework incrementally. I found that it was much better to start with one building block and continue working on this, rather than trying to elaborate on the whole project from the beginning. It turned out to be better to start with something concrete, and then to raise the level of abstraction in the project.

Working on this project helped me to improve my writing on an academic level. I am especially grateful to Mehmet for his assistance.

Designing a pattern recognition framework was valuable experience. The concept of matching and merging a pattern to address an adaptability problem is challenging, but very interesting. The goal of this project has been achieved, and a new tool to assist designers has been developed, as DPRF assists the designer to implement adaptability.

References

[1] Abraham, A. (2005). Rule Based Expert Systems. *Handbook of measuring system design*.

[2] Aksit, M. (n.d.). *PATO-course*. Architecture design for evolution [PowerPoint slides].

[3] Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: towns, buildings, construction* (Vol. 2). Oxford University Press.

[4] Anderson, K. (2015). *The OO Paradigm*. Retrieved March 27, 2016, from https://www.cs.colorado.edu/~kena/classes/5448/f12/lectures/02-ooparadigm.pdf.

[5] Bennett, K. H., & Rajlich, V. T. (2000, May). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 73-87). ACM.

[6] Bratko, I. (2001). *Prolog programming for artificial intelligence*. Pearson education.

[7] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). A system of patterns: Pattern-oriented software architecture.

[8] Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., & Tan, W. G. (2001). Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, *13*(1), 3-30.

[9] Ciraci, Selim and Broek, Pim van den and Aksit, Mehmet (2007) *A Taxonomy for a Constructive Approach to Software Evolution*. Journal of Software, 2 (2). pp. 84-97. ISSN 1796-217X

[10] Clocksin, W., & Mellish, C. S. (2003). *Programming in PROLOG*. Springer Science & Business Media.

[11] Fayad, M., & Cline, M. P. (1996). Aspects of software adaptability. *Communications of the ACM*, *39*(10), 58-59.

[12] Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[13] Perry, D. E. (1994, September). Dimensions of software evolution. In *Software Maintenance, 1994. Proceedings., International Conference on* (pp. 296-303). IEEE.

[14] Mens, T., & Eden, A. H. (2005). On the evolution complexity of design patterns.*Electronic Notes in Theoretical Computer Science*, *127*(3), 147-163.

[15] Meyer, B. (1988). *Object-oriented software construction* (Vol. 2, pp. 331-410). New York: Prentice hall.

[16] Negnevitsky, M. (2002). *Rule-based expert systems* [PowerPoint slides]. Retrieved from http://staff.informatics.buu.ac.th/~krisana/975352/handout/Lecture02.pdf

[17] No Magic. (2016). *MagicDraw*. Retrieved February 23, 2016, from https://www.nomagic.com/products/magicdraw.html

[18] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. E. (1991, October). *Object-oriented modeling and design* (Vol. 199). Englewood Cliffs, NJ: Prentice-hall.

[19] Shalloway, A., & Trott, J. R. (2004). *Design patterns explained: a new perspective on object-oriented design*. Pearson Education.

[20] Störrle, H. (2014). *MACH 0.94 (subsonic)*.

[21] Störrle. H. (n.d.). *MACH*. Retrieved March 7, 2016, from http://www2.compute.dtu.dk/~hsto/tools/mach.html

[22] University of Pennsylvania. (n.d.). *Decision Trees*. Retrieved February 26, 2016, from https://alliance.seas.upenn.edu/~cis520/wiki/index.php?n=Lectures.DecisionTrees

[23] Wolfgang, P. (1994). *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley.