MASTER

"Digitale koopgoot"
An environment for online decision support systems

Berkers, Teun; Raedts, I.G.J.

Award date:
2003

MASTER'S THESIS

"Digitale Koopgoot"

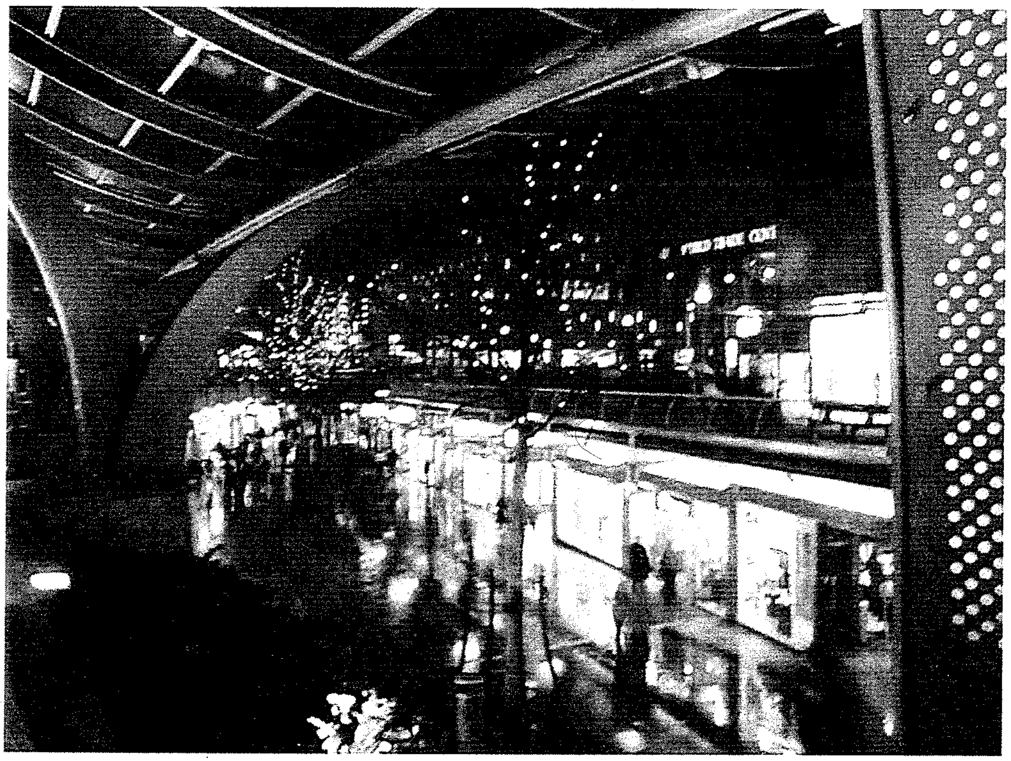
An environment for online decision support systems

by

Teun Berkers
Ivo Raedts

"Digitale Koopgoot"
An environment for online decision support systems

Master's thesis of
Teun Berkers
Ivo Raedts



Institute	Technische Universiteit Eindhoven
Coach	prof. dr. K.M. van Hee
Period	february – november 2003
Company	Deloitte
Department	Vakdirectoraat Consultancy
Coach	dr. ir. C.W. van den Herik

Abstract

Decision support systems are information systems designed to interactively support all phases of a user's decision making process. By presenting decision support systems as online solutions, a new distribution channel of services can be created. In the field of customer-related services, many services still require an expert to perform tasks that could easily be automated. As a result one of our objectives is: the design and implementation of one or more decision support systems. The other objective is: the design and implementation of a technical architecture that is needed to realize a new distribution channel for online decision support systems: the "Digitale Koopgoot".

In chapter 2, we have categorized decision support systems into five different types of systems that are relevant to the Koopgoot: benchmarking tools, case-based reasoning tools, generic queuing models, decision trees, value calculators. All these types are explained.

Chapter 3 describes the business objectives of project and presents the functional requirements for both the online decision support systems and the Koopgoot. Also some use case scenarios are presented to describe the requirements of the Koopgoot in an informal and easy understandable way.

Chapter 4 describes a top-down approach of the logical model of the system. We present a model of the global system that can be decomposed into numerous subsystems. The subsystems that exist in the environment will be modelled and described. In order to describe the communication between the 'Koopgoot' and its the subsystems we will describe two specific scenarios.

In chapter 5 we derive the technical architecture from the technical requirements and the environment model, explained in chapter 4. Subsequently we map the functional components that are described in chapter 4 to the technical components, after which these technical components are described.

Contents

List of Figures

List of Tables

1 Introduction

The master program 'Computer Science and Engineering' at the 'Technische Universiteit Eindhoven' is concluded with a graduation period. The study was done in the field of Information Systems, with architecture of information systems as area of expertise. We have carried out this graduation study at Deloitte.

1.1 Background

Decision support systems are information systems designed to interactively support all phases of a user's decision making process. By presenting decision support systems as online solutions, a new distribution channel of services can be created. The two main online services that are currently offered by Deloitte are salary management service and accounting. Since Deloitte has no official division for online services, a pilot project was initiated. Because the standard project procedures can be quite time consuming and hinder project progress, a more clandestine approach was preferred to demonstrate the possibilities of online decision support systems in a short period of time. To characterize this approach, project secretary Dick van der Net of Deloitte Rotterdam came up with "de Koopgoot" as working title for the project. "De Koopgoot" is a nickname of a famous underground shopping mall located in Rotterdam. The Dutch word "Koopgoot" literally means 'shopping gutter', which refers nicely to the characteristics of our project: a clandestine (underground) approach to realize a new distribution channel (gutter) to sell services.

1.2 Problem description

Small and Medium Enterprises, individuals, local government and non-profit organizations often have a need for professional advise when important decisions have to be made or certain tasks have to be fulfilled. Many of them use various services provided by Deloitte to obtain this professional advice. Despite the fact that many of these Deloitte clients have a similar type of problem to be solved, Deloitte can improve in delivering the solution in a more efficient (innovative) way. Deloitte needs to find a more efficient way to service these clients by utilizing other media available, like the internet. For this new kind of customer-related services it's interesting to look at important differences in the field of operational management between the service sector and other sectors, like the industrial sector where most of the production processes are automated. In the field of customer-related services however, many services still require an expert to perform tasks that could easily be automated. When a new automated and more efficient type of customer-related services can be realized, Deloitte can service its clients in a more efficient way and possibly reach a new group of potential customers that is not using Deloitte services in its current form.

1.1 Objectives

Our study can be split up into two objectives:

1. The design and implementation of a technical architecture that is needed to realize a new distribution channel for online decision support systems.
   To realize this technical architecture, the functional and technical requirements of the system have to be formulated. Subsequently, the architecture has to be designed and implemented, satisfying the requirements.

2. The design and implementation of one or more decision support systems for this environment.
   We will try to determine what type of services could be added to the environment. Subsequently we will formulate requirements for the services. Based on the formulated requirements, one or more appropriate examples of decision support systems will be implemented as a proof of concept to the environment.

2 Decision Support Systems

2.1 Introduction

Decision support systems or decision making support systems are information systems designed to interactively support all phases of a user's decision making process. The support that is offered can be direct or indirect and it can be for a single user or group usage.
In this chapter we will briefly present a theoretical basis for decision support and the architectures that have been proposed to deliver the theory in practice.

Because of the importance to individual, group and organizational success, information systems research has examined ways to improve support for decision making for the last three decades. The research has generated a wide variety of information systems to provide the necessary support. In this period there has been an evolution from simple data access and basic reporting to a more complex analytical, creative and even artificial intelligent support for decision making [HW1996]. Much of the research however in the field of decision making support systems has taken place in various specific disciplines, such as computer science, information systems, management science and psychology, resulting in a situation where researchers and practitioners in one discipline have often been unaware of important developments in the others. This has resulted in a large variety of independent individual systems that support decision making in a fragmentary and often incomplete manner [MFG2003].

We can distinguish a variety of decision support system types. The architecture of a system that calculates to which subsidies a company is entitled differs from the architecture of a system that can highlight weaknesses in a production line. The architectures of decision support systems can differ on many points:

- The way the user input is retrieved. In most cases the user will have to enter all input and await the result. But in other cases the user will enter only some input and based on this input, the next input form is presented to the user. When all required input is collected, the result is presented to the user.
- The type of result that is presented to the user. The results can be presented in many different forms. It can be an advice, a value or list of values, a comparison and many more.
- The type of calculation that is applied to the user input. Some types of decision support systems use databases to perform different types of queries, other types of decision support systems use knowledgebase's, and other decision support systems only use calculations or algorithms to generate the results.

We will try to categorize decision support systems into five different types of systems that are relevant to the Koopgoot. It is understood that more categories can be named, but our aim is to provide a framework for online services useful to the Koopgoot. In the Koopgoot, we distinguish the following five types of decision support systems:

- Benchmarking tools
- Case-based reasoning tools
- Generic queuing models
- Decision trees
- Value calculators

In the following paragraphs, the specified types will be described in detail.

2.2 Benchmarking

In today's highly competitive and rapidly changing global economy organizations have been forced to consider a wide variety of innovative management philosophies and techniques. One such technique that has been used extensively is benchmarking. Benchmarking can be defined as "… a systematic and continuous measurement process; a process of continuously measuring and comparing an organisation's business process against business leaders anywhere in the world to gain information which will help the organisation to take action to improve its performance" [LePr1995].

Although benchmarking in business organisations is a relatively new concept and practice, it has rapidly gained acceptance worldwide as an instrument of continuous improvement in the context of total quality management (TQM). In a survey conducted by Voss et al. (1997) involving a sample of over 600 European manufacturing companies, it is shown that increased levels of benchmarking use were associated with higher levels of both adopting of best practices and operational performance. Based on the results of the study, the authors proposed a relationship model between learning, benchmarking, understanding and performance. In this model, benchmarking, as part of a learning process of an organization leads in the same time to:

1. an improvement in a company's understanding of its strengths and weaknesses;
2. higher levels of performance.

Benchmarking practices can be generically classified according to the nature of the object of study of the benchmark into three types of benchmarking:

- process benchmarking: used to compare operations, work practices and business processes;
- product benchmarking: used to compare products or services;
- strategic benchmarking: used to compare organisational structures, management practices and business strategies. This type of benchmarking bears some resemblance to process benchmarking.

It is also interesting to note that in the USA and Europe many organisations exists that promote the use of benchmarking, such as the International Benchmarking Clearing House or the European Network for Advanced Performance System (ENAPS), which provide benchmarking databases and assistance in identifying partners [CDM2002].

To describe the basic structure of the different types of decision support systems, we will present a Petri net[1] model of the type. The model starts with a user entering his first input form. To keep the models simple we assume that the user input is correct. Based on the user's input a query is generated and executed on a database containing benchmarking data. If the results are useful, they will be presented to the user. Otherwise, it is explained to the user why the results are not useful and what can be done to obtain useful results.

[1] Further reading about the technique of Petri nets: [HSVW2003], [Hee1994], [GV2003]

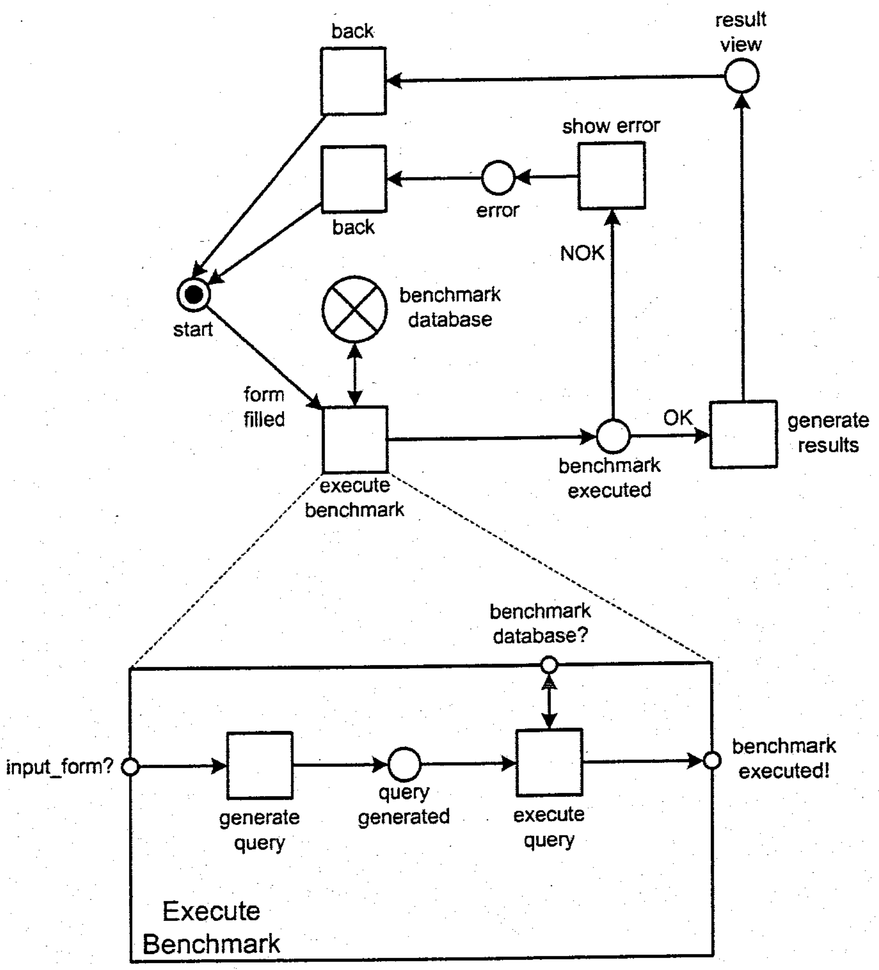


Figure 2.1. Benchmarking model

2.3 Case-Based Reasoning

In case-based reasoning (CBR), a reasoner remembers previous situations similar to the current one and uses them to help solve the new problem. Remembered cases are used to suggest a means of solving a new problem, suggest a means of adapting a solution that doesn't quite fit, warn of possible failures and to interpret a situation. [Kolodner1993]

Research by psychologists and cognitive scientists has proven that humans routinely use CBR in their decision processes. While humans commonly use CBR, they suffer from an inability to consistently recall the appropriate set of prior cases, distinguish between important and unimportant features and deal with incomplete and uncertain information in current problems [Kolodner1991],[AD1998]. It would therefore be useful to consider building CBR systems which are geared towards aiding the CBR processes of humans.

Case-based reasoning is useful to people who know a lot about a task and domain because it gives them a way of reusing hard reasoning they have done in the past. It is however equally useful to those who know little about a task or domain. Finally, case-based reasoning is also useful when knowledge is incomplete. Logical systems have trouble dealing with this type of situation because they want to base their answer on what is well known and sound. Case-based reasoning provides a method for dealing with incomplete knowledge: a case-based reasoner can make assumptions to fill in missing knowledge, based on experience and use this to start with. Most of the time solutions that are generated this way won't be optimal but if the reasoner evaluates the proposed solutions correctly this methodology presents a way to generate answers easily [Kolodner1993].

CBR working in brief

In a CBR application, a user enters details about a case (person, job function, problem, etcetera). Subsequently, the CBR application searches the database for similar cases using an intelligent algorithm. Every found case is assigned a match value that defines the degree of similarity to the input case. If a sufficient number of matching cases is found, a list of best matching cases is presented to the user, together with their most important properties. A user can select one or more cases from the list and study them in detail.

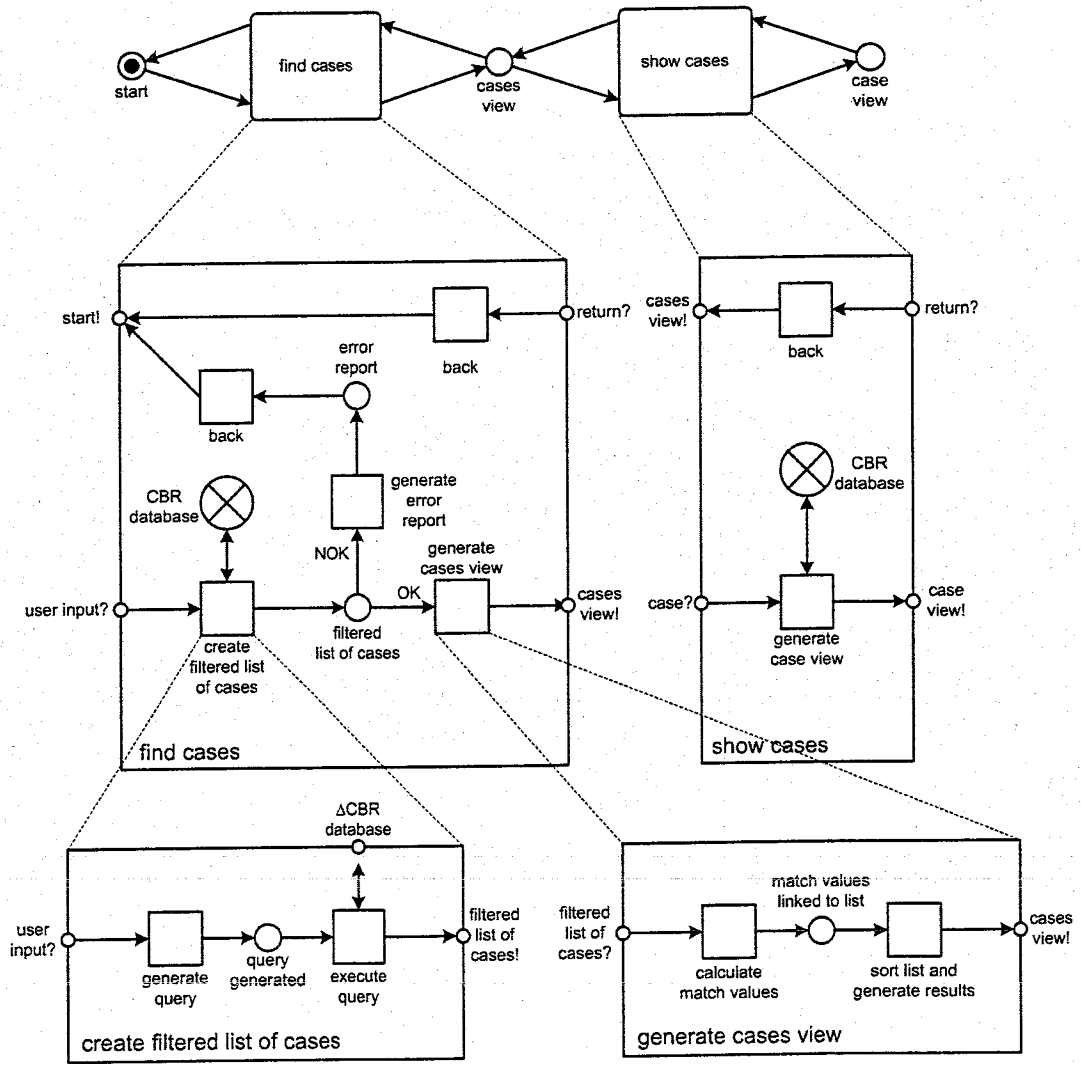


Figure 2.2. Case-Based Reasoning model

Based on the user's input which describes his case, a query is generated. This query might filter on some properties. This query is executed on the CBR database after which the number of remaining cases is considered. If this number is too small it is shown to the user that there are not enough 'matching' cases and suggestions are given to prevent this in the next query. If an appropriate number of cases is left, for all remaining cases a match value is calculated that indicates the degree of similarity. Subsequently the cases are ranked on their match value, after which a number of best matching cases are presented to the user in a list, displaying the most important properties of these cases. The user can select one of these cases, to view that case in detail.

CBR Example:
A company wants to hire a person for a specific function, but has no idea about the salary that is common for such a function. In the input form, the user fills in optional case properties like age, gender, education, function description, branch, area, and so on. If the input criteria lead to enough similar cases, a list of best matching cases is presented showing properties like age, function description, salary and match value. If the user selects a case, all details are shown. Based on the returned results the user should have a good indication of the appropriate salary for the person they want to take into employment.

2.4 Generic Queuing Models

Consider any system that has a capacity C, representing the maximum rate at which it can perform work. Assume that R represents the average rate at which work is demanded from this system. One fundamental law of nature states that if R<C then the system can handle he demands placed upon it, whereas if R>C then the system capacity is insufficient and all the unpleasant and catastrophic effects of saturation will be experienced. However, even when R<C we still experience a different set of unpleasantness that come about because of the irregularity of the demands. [Kleinrock1975]

Queues arise from two sources. The first is the unscheduled arrival time of the customers. The second is the random demand (duration of service) that each customer requires of the system. The characterization of these two unpredictable quantities (the arrival times and the service times) and the evaluations of their effect on queuing phenomena form the essence of queuing theory. [Kleinrock1975]

Briefly, a queuing model is one in which you have a sequence of items (such as people) arriving at a facility for service. There are a lot of different queuing models, but the performance characteristics of all models are based on four quantities [EGSMW1998]:

1. The number of items waiting in the system: the number of items currently being served, as well as those waiting for service.
2. The number of people in the queue: the number of items waiting for service.
3. The waiting time in the system: the interval between when an individual item enters the system and when it leaves the system. This interval includes the service time.
4. The waiting time in the queue: the time between entering the system and the beginning of service.

When a model is completely described the following items are described [EGSMW1998]:

1. Arrival process.
   The arrival process describes how the times, between the intervals of arriving items, is distributed. When not much is known of the arrival process, a default distribution will often be used. This default distribution is a special kind of the exponential distribution: the Poisson distribution. A Poisson distributed process requires only one parameter to specify the arrival process: *the mean arrival rate*. The mean arrival rate describes how many items arrive on average during a specified period of time.

2. Service process.
   The service process describes how the time, that the service takes, is distributed. When not much is known of the service process, a exponential distribution will again be used. Again only one parameter to specify the arrival process: *the service rate.* The mean service rate describes how many times a service is completed on average during a specified period of time.
3. Queue size.
   The limit of the number of items that can wait in the queue.
4. Queue discipline.
   The queuing discipline describes the order in which the items are served. In many situations the first come, first serve discipline is used. Other situations might use priorities or other algorithms to describe the order in which the items are served.
5. Time Horizon.
   The system operates continuously over the specified horizon.
6. Source Population.
   The source population describes the number of available items that are eligible to join the queue. In most situations this number is assumed as infinite.

To facilitate communication among those working on queuing models, D.G. Kendall proposed a taxonomy based on the following notation:

A/B/s

Where:

A = arrival distribution
B = service distribution
s = the number of servers

Different letters are used to designate certain distributions. Placed in the A and B position, they indicate the arrival or the service distribution, respectively. The following conventions are in general use:

M = exponential distribution
D = deterministic number
G = any distribution

The often used default model consists of a single server queue with exponential arrival intervals and service times. This is called the M/M/1 model.

By linking the output of a model to the input of another model, two models can be serialized. This can be formulated in other words: the output of the first model describes the arrival process of the second model. In real-life this is also a common situation and a lot of situations will be based on more than just one model. If production lines are considered, the total queuing model can consist of multiple sub models. All models of the total queuing model have their own service distribution properties. The arrival process of a specific model is described by the output of the preceding processes. Figure 2.3 shows a queuing model consisting of six queuing models. The performance characteristics can be determined for a single queuing model and for the total queuing model.

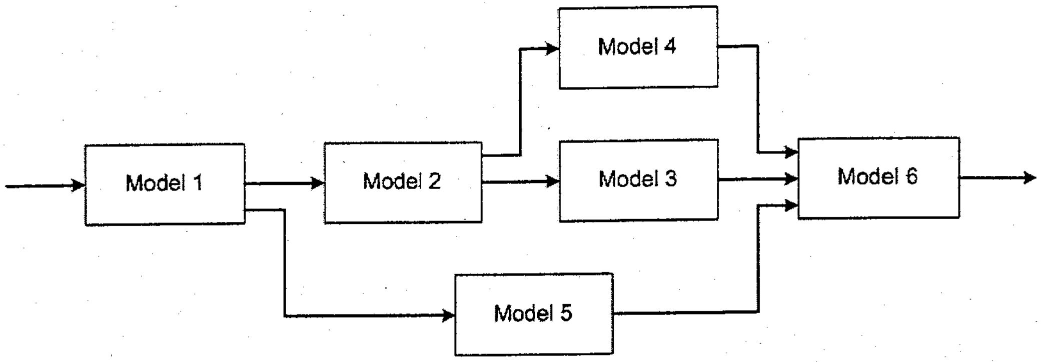


Figure 2.3. Linked queuing models

In a decision support model of the type queuing models, a lot of different results can be wanted. In general the four quantities are determined based on the properties of the queuing model. The main issue will be how changes to the model will influence the result. It will be all about the streamlining of the process. A trade-off will often have to be made between costs and performance.

The results can be determined on two ways: calculation and simulation. Calculations in models containing determined distributions will be simpler than calculations of models containing exponential distribution. But when other distributions are used, which are described with a mean interval time and a variance, the calculations become more complicated. In situations consisting of more sub models, the calculations will become even more complicated. When situations become too complicated, simulations can be a solution. Some test runs of the system can be done and the results, consisting of the four quantities, can be averaged over the number of runs. In this case it should be questioned what number of test runs is required to get a useful result.

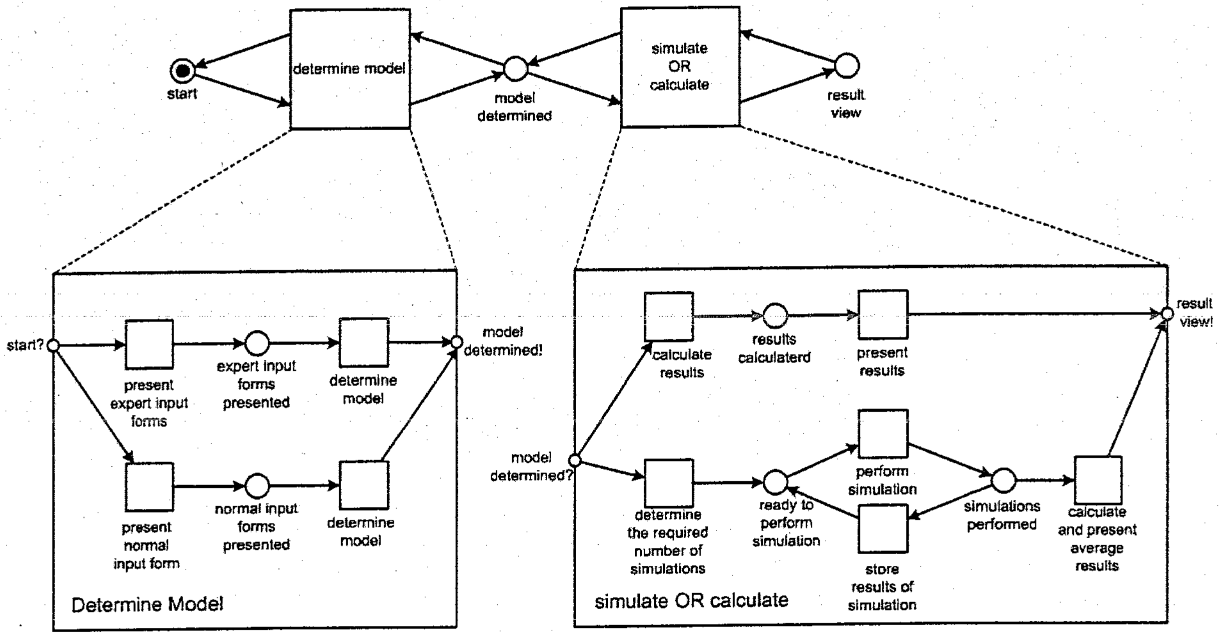


Figure 2.4. Generic Queuing model

Based on the users input form(s), the model(s), describing the situation are determined. A normal user will have to answer a lot of questions, before model is determined. Expert users should be able to user other input form(s) than normal users, where they can define the model(s) by specifying the types and values that define the model. After the model is determined, the results (the four quantities) have to be determined by simulation or calculation. Subsequently the results are presented to the user, who wants to see how certain changes to the model affect the results. The user should learn about his queuing situation by experimenting with the model so that he can make a better trade-off between performance and costs.

Generic Queuing example:
A manager of a supermarket wants to determine how many cash desks should be open so that the average time his customers have to wait will less than one minute. To answer this question, a generic queuing decision support system is used. The first question asked by the system is whether he wants to specify the model himself (expert mode) or want to let the system determine the model based on some questions understandable to a user without knowledge of queuing systems. Since this manager is no queuing expert, he chooses to use normal mode and answers some questions that are presented to him. When all questions are answered the model is determined. After the model is determined, the manager can enter the specific parameters required for the determined model. Subsequently the results are determined. This can be done by calculation if there is a calculation module implemented for the determined model. Otherwise a specific number of simulation runs are executed and the average results are presented. The manager studies the results, which indicate that the average time his customers have to wait is nearly two minutes. Subsequently the manager can adjust the number of cash desks and study the outcome again. After trying different numbers of cash desks, the manager should have an indication how many desks should be opened in order to keep the average waiting time for his customers below one minute.

2.5 Decision trees

A decision tree is a representation of a decision case that can be used to display and order the conditions and related actions that are required in the decision making process of the case. Decision trees can be used to structure complex decisions. A decision tree is a directed acyclic graph satisfying the following properties: [Wets1998]

- There is exactly one node, called the root, which no edges enter.
- Every node except the root has exactly one entering edge.
- There is a unique path from the root to each node.

In a decision tree, there are two types of nodes: decision nodes and leaves. Decision nodes specify a test which should be carried out on the value of a feature (user input). Each possible outcome of the test results in a branch of the decision tree. Leaves are the terminal nodes of the tree. They specify to which class an instance belongs. In Figure 2.5, an example of a decision tree is given. The example describes a simplified procedure for selecting potential locations for a supermarket [Wets1998].

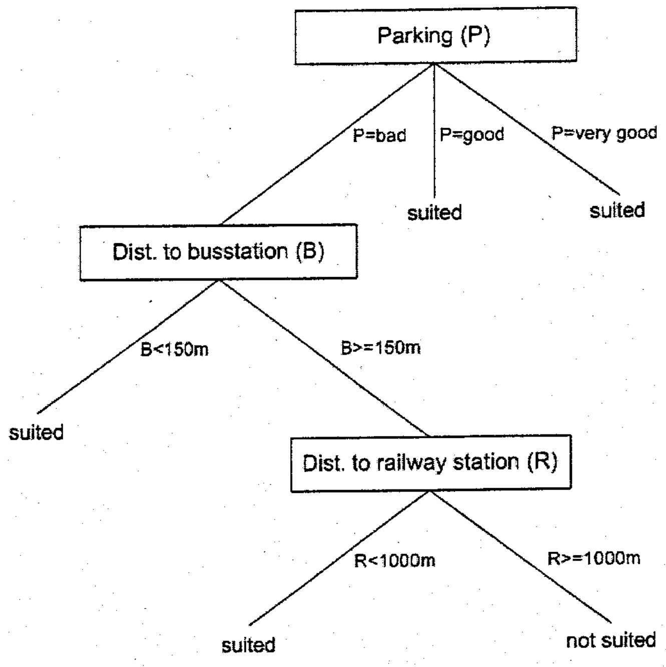


Figure 2.5. Example of a decision tree

The usefulness of decision trees manifests itself in many fields of application: in the application of procedures, in the verification of completeness, consistency and correctness, and in the documentation of complex voluminous knowledge.

In the application of procedures, decision trees can help speed up the decision making process, caused by the fact that decision trees are condition-oriented whereas normal text is action-oriented. Subsequently, every condition only has to be described once. Furthermore, decision trees help improving the level of correctness of the decisions. This is related to the fact that fewer questions have to be answered and the trees can only be interpreted in one way.

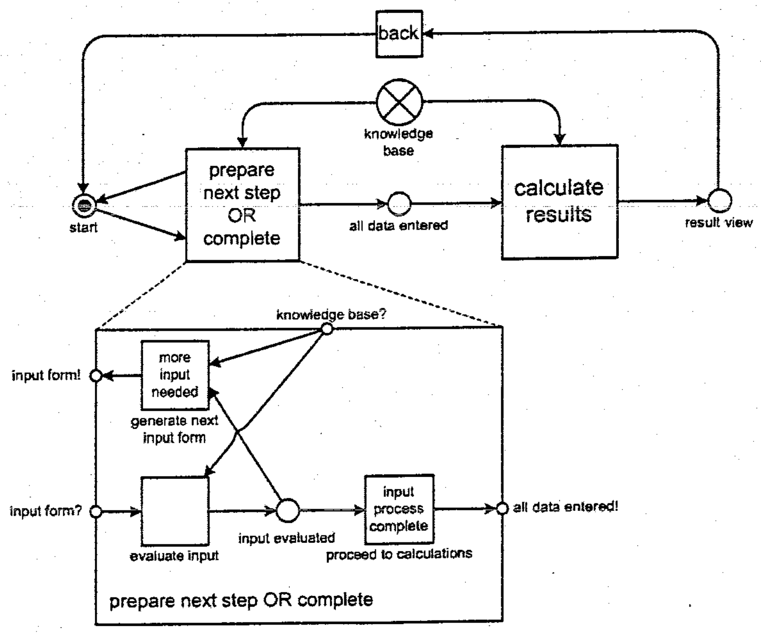


Figure 2.6. Decision tree model

After the user has completed the input form, a knowledge base is used to determine whether there is more input needed or not. If more input is needed from the user, the knowledge base is used to determine what information is needed so that the next input form can be generated. If no more input is needed, the knowledge base is used to determine or calculate the results.

The user should also be able to perform a what-if analysis that allows the user to roll-back one or more steps. To keep the model simple we have not modeled the what-if analysis.

Decision Tree example:
A trader wants to import goods from a foreign country. When importing goods from some countries there are a lot of different kind of taxes (For example: anti-dump taxes) that might have to be paid. To determine which taxes should be paid, a lot of rules have to be considered. This might be a complicated and time-consuming process, especially when different parts of the goods originate from different countries. When an application based on a decision tree is used, the user only has to answer a sequence of questions relevant to his situation, to determine the results. No additional paperwork has to be studied: the system will determine the relevant input that has to be supplied and therefore decrease the chance of misinterpretation.

2.6 Value Calculators

Value Calculators are the most general category of decision support systems considered in our outline. With value calculators, users fill in an input form and the result will be calculated without the use of databases or knowledge bases. The intelligence behind the processing of the results will only contain algorithms or calculations.
A Return On Investment (ROI) calculator is an good example of a frequently used value calculator. After the user has entered the input form, a result is generated describing how long it takes for an investment to pay of.

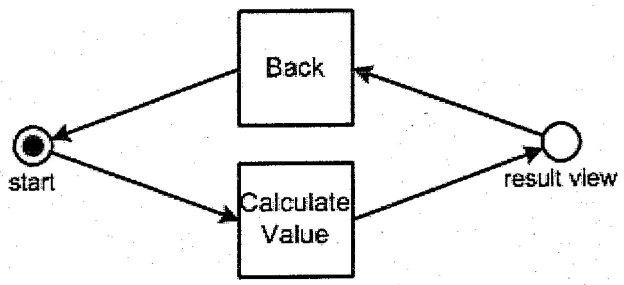


Figure 2.7. Value Calculation model

Because value calculators are a very general type, the resulting model is very simple.

3 Business model

3.1 Introduction

Deloitte offers a wide range of professional services to its clients. Despite the fact that many of these clients have similar types of problems to be solved, Deloitte can improve in delivering the solution in a more efficient (innovative) way. We try to find a more efficient way to service these clients by utilizing the internet. If a specific category of customer-related services can be automated, Deloitte can service its clients in a more efficient way and possibly reach a new group of potential customers who is not willing to use Deloitte services in its current form.
We intend to offer a group of services as online decision support systems. The services in question can be either new services or existing traditional services brought to the customers via the web. Both types of services are highly characterized by customer 'self-service'.

3.2 Objective

The business objective is to provide an environment for online services that
- offers better value to (potential) clients;
- services clients more efficiently;
- positions Deloitte as 'knowledge powerhouse' in the market.

More specifically, the introduction of online decision support systems could lead to the following benefits:
- Higher and faster availability. It is no longer necessary for a client to make an appointment with a consultant or other expert when he needs professional advice. Using this new form of online service, it would be possible to obtain the advice almost instantly at any time and at any place, without the help of an expert.
- Lower barrier to acquire service. Users can now just pick a service from the site.
- Better and faster advice. Many services require databases or knowledge bases. An online decision support system would allow a client to directly interact with the database and extract useful information in a short period of time. A decision support system can also lead to a better advice as the result or advice can be based on the extensive intelligence the system can contain.
- Better efficiency. If a traditional service is offered online in the form of a decision support system, a consultant or advisor will not be required anymore to provide the advice or solution in most of the cases. His previous vital servicing role will be reduced to an advisory role: only when a client requires additional advice after using the online decision support system, the consultant has to be contacted. This could also result in lower prices for the services, but a more profound view on this matter would be beyond the scope of this study as defined in paragraph 3.3.

The mentioned benefits should eventually attract new customers who are drawn by the new type of services but it is beyond the scope of this project to express a balanced opinion on this subject.

Besides looking at the numerous advantages that can be gained by the introduction of online decision support systems, we also want to pay attention to some disadvantages the introduction of such a system can have. It is useful to be acquainted with possible difficulties and issues so they can receive particular attention in the design and implementation phase of the system.

Possible disadvantages of online decision support systems that have to be considered are:

- Bad advice due to bad input. Because the results of an online decision support system are based on the input supplied by the user, incorrect input can lead to incorrect advice. As there is no professional advisor present to assist the user in this process, the chances of incorrect input are larger compared to a traditional type of service. This issue should receive special attention during the design and implementation of the system. Stringent requirements for the user interface in general and the user input in specific must be formulated to reduce the chance of input errors.
- No specific advice targeted at a client's unique situation. Not all types of services are suitable for implementation as decision support system. Some traditional services provide an advice that is tailored to the client's specific situation. An online decision support system is unsuitable to solve this type of complex and often unique problems. We could however turn this disadvantage into an advantage: for this type of services, the decision support system technique could be used to develop an online version of the service with limited functionality. The tool can be used as 'lead generator' for the existing traditional service: it will provide a general advice and a reference to the traditional service if the client has a need for a more specific advice tailored to his own situation.

3.3 Scope

This study will focus on the technical architecture that is needed to realize a new distribution channel for online decision support systems.

3.4 Functional requirements for the environment

The environment that is required to host the online services has to meet the following requirements:

- Easy and free access
  Users should be able to get access to the portal without delay or human intervention. Parts of the environment must be accessible to everyone without logging in, other parts should only be accessible to authorized users. An easy registration mechanism is required that allows users to fill in a form and get a login account.
- Customization options
  If a user is logged on to the environment, he should be able to customize general site settings and be able to update his profile.
- Easy navigation
  The services should be presented in an orderly fashion, classified into categories that

are useful to the end-user. The following two types of categories should enable easy navigation through the services:

1. industry-based service navigation.
2. function-based service navigation.

Both navigation options should be available at every place of the environment.

- Support for different methods of payment
  The environment must support different payment options for a single service. It should be possible to:
  1. subscribe to a service for a specified period.
  2. allow single usage of a service after paying a specified amount.
- Online payment
  To implement the payment options described above, it should be possible to make online payments. Credit cards should at least be supported as an instrument of payment.
- Service preview options
  The option to add a preview or demo to a service should be present, to give an impression of the features of a service.
- Administrative options
  It must be easy for administrators to add, edit and delete services, manage users and manage service subscriptions.

3.5 Functional requirements for the services

Services that are to be distributed as online decision support systems should meet the following requirements:

- Common look and feel
  All of the online decision support systems should have the same natural and convenient look and feel. Because user input is such an important part of a decision support system, the user interface should be well-organized to avoid confusion and erroneous input.
- User interface tailored to target group of the service
  The user interface of the online decision support system should be aimed at the users of the system. Comments, control labels and explanations must be comprehensible to the users of system. The end results must also be presented in an understandable way.
- Use of intelligence
  An online decision support system should use a database, knowledge base, algorithm, intelligent calculation or a combination of those to provide a professional advice.
- Protection of intelligence
  The databases, knowledge bases, algorithms or calculations that are used to determine the results, should not be accessible to the users. Because the service is presented online, security is an important issue. Unprotected client side scripts or applets should be avoided for non-trivial functionality.
- Accessibility
  An online decision support system should only be accessible to those who have obtained the rights to use it.
- Single sign-on
  To meet the previous requirement, a security mechanism has to be implemented in the service. However, we do not want to force a user to enter his credentials twice (once when logging on to the Koopgoot and once when starting the service). The service security mechanism should be able to check the validity of the started service session without user interaction.
- Fulfill to one of the case types
  It should be possible to classify the service as one of the case types that is defined in chapter two.

3.6 Use case scenarios

In this paragraph we present a number of *use case* scenarios to describe the requirements of the Koopgoot in an informal and easy understandable way. The use cases describe the sequence of interactions between actors and the system. We will only describe scenarios for the main flow through a use case for a number of essential system functions, the use cases in this paragraph will therefore not all of the behavior required of the system.

3.6.1 Registering to the system

Table 3.1. Koopgoot Registration

Koopgoot Registration *actor: normal user*
enter site
select register option
fill in registration form
submit registration form
receive activation mail
activate account

In this scenario a user wants to create an account to obtain login access to the system. The user selects the "register" option from the Koopgoot welcome page and completes the registration form, filling in his personal details and e-mail address. If the form is completed correctly, the system will send an activation mail to the user's e-mail address that contains a link the user has to click to activate his account. This procedure will help to prevent the usage of an illegal e-mail address.

3.6.2 Starting a service

Table 3.2. Starting a service

Starting a service *actor: normal user*
enter site
log in
select function or industry
select service
select pay for single usage
confirm payment
start using service

The second use case describes the required user actions to start a service. After logging in to the Koopgoot, a category of services can be selected. The Koopgoot offers two types of categories: services classified by function and by industry. When a category is selected, the user can browse the list of services for this category and select a service. In this use case, we presume the service is not free and the user has no subscription to it. The user chooses to pay for the selected service and has to confirm the payment. If he has enough credits available, his balance is decreased and the service is started in a new window.

3.6.3 Subscribing to a service

Table 3.3. Subscribing to a service

Subscribing to a service *actor: normal user*
enter site
log in
select function or industry
select service
select subscribe option
select desired subscription
confirm payment

The scenario above describes the option to subscribe to a service. When a service has been selected, the actor can choose to subscribe to the service concerned. One or more subscription options are presented to define the duration and price of the subscription. If the amount of credits available is sufficient the user will be subscribed to the service for a period of time set by the selected subscription option.

3.6.4 Increasing credits

Table 3.4. Increasing the amount of credits

Increasing the amount of credits *actor: normal user*
enter site
log in
select increase credits
enter amount
confirm amount
enter credit card details
confirm credit card payment

In order to start non-free services, the user's amount of credits must be sufficient. To increase the amount of credits, the 'increase credits' option can be selected from any place in the system, under the condition that the user is logged in. An amount can be entered and confirmed after which the user is redirected to the external payment service provider site to complete the credit card payment.

3.6.5 Adding a service

The administrative user is another actor who uses the system. Administrative users use the same portal to enter the Koopgoot, but have additional features available when logged in.

Table 3.5. Adding a service

Adding a service *actor: administrative user*
enter site
log in
select 'Service management'
select 'Add new service'
enter service details
select industry or industries
select function
add service

After logging in, the 'Service management' option can be selected. In the 'Service management' screen, a new service can be added by selecting the 'Add new service' option. After entering the service details, the service can be assigned to one or more industries and to a function group. Subsequently the service can be added to the system. In addition to this scenario, the 'Service Management' option also provides functionality to change or delete services.

4 Logical architecture

4.1 Introduction

This chapter describes a top-down approach of the logical model of the system using Petri nets[2]. We present a model of the global system that can be decomposed into numerous subsystems. The description of the logical system will start with a high-level model of the environment. Subsequently, the subsystems that exist in the environment will be modelled and described. The subsystems are independent to a large extend; however they must be able to communicate with one another. The communication between the subsystems will be described in the last two paragraphs, in which we will introduce two specific scenarios to illustrate the communication between 'de Koopgoot' and its services and the communication that takes place in the event of an online payment.

4.2 Environment

The environment for our online decision support systems consists of 'de Koopgoot', the internet portal that can be seen as the interface that enables clients to connect to the online decision support systems. Furthermore the environment consists of services, representing the online decision support systems that can be started, and a Payment Service Provider (PSP) that enables clients to pay for the services they want to use. These clients will be referred to as 'users' in the remaining part of this chapter. Deloitte OnlinePay is needed to communicate with the Payment Service Provider. Deloitte OnlinePay presents an interface for the 'Koopgoot' to enable online payments. Subsequently, no PSP-specific communication is done in 'de Koopgoot' other than redirecting the client to the PSP-site to begin the payment and provide the PSP with a return page to which the user is returned when the payment procedure has ended. We will discuss this in more detail in paragraph 4.10.

The resulting high-level model is displayed in Figure 4.1. The interfaces between the subsystems are tokens that enter or exit the subsystems via places, which are connected to the subsystems via place fusion. We have omitted the place labels for the interface places that connect the Client subsystem to other subsystems as the user interaction arcs are so numerous, describing all of them at this high level would make the model unnecessarily complex. Of course, user interaction will be described in more detail at a lower level.

[2] Further reading about the technique of Petri nets: [HSVW2003], [Hee1994], [GV2003]

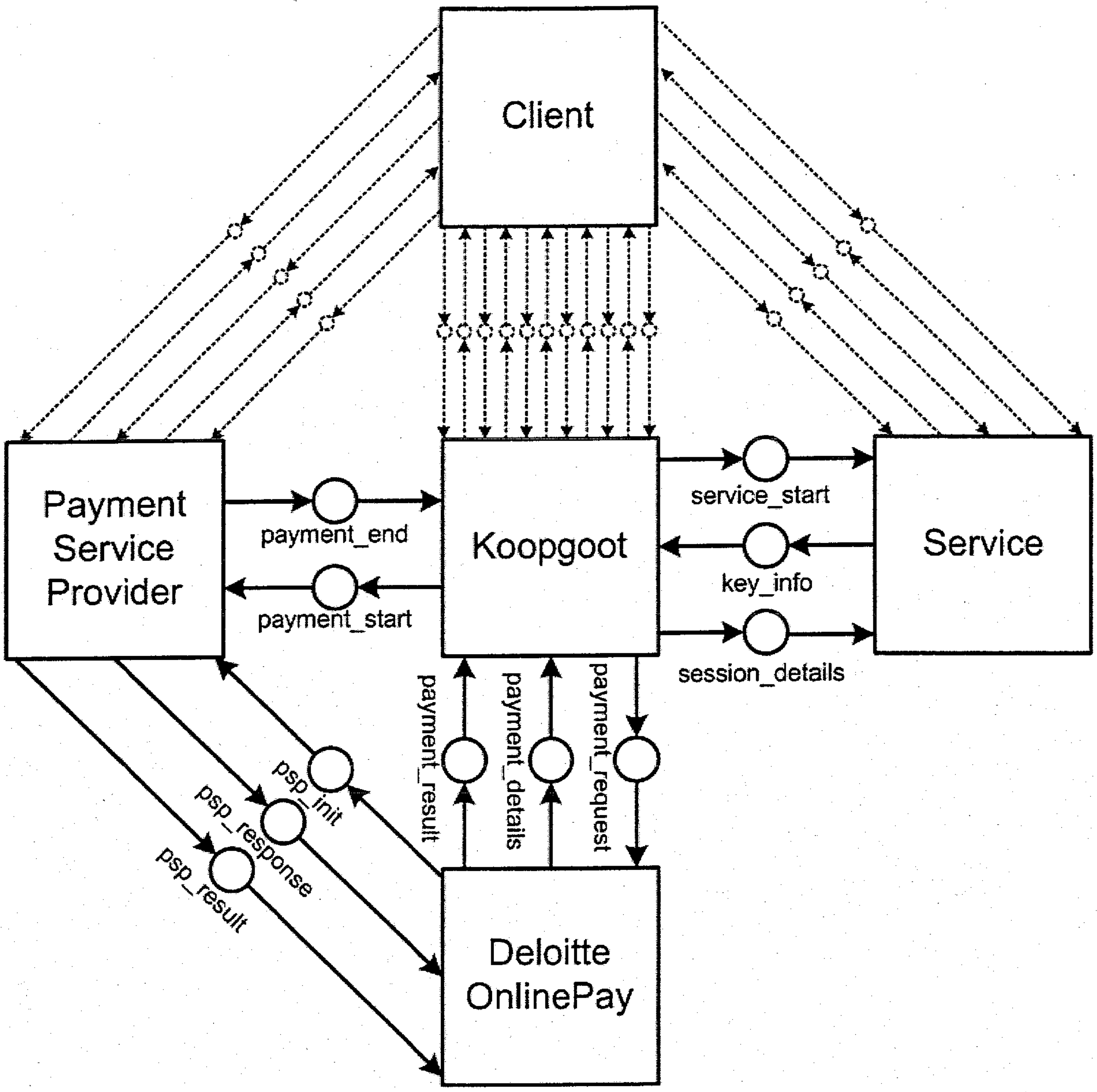


Figure 4.1. Environment model

In the next paragraphs, the specific subsystems will be decomposed and described. We will start with the Koopgoot system and all its underlying subsystems. Subsequently, the Service subsystem will be covered in paragraph 4.4. The service model will however be bounded to the functionality that is required to enable single-sign on for a service. This sign on procedure of a service will be followed by the actual functionality of the service, of which we have presented a broad outline in chapter 2.

4.3 'De Koopgoot'

In a nutshell, the 'Koopgoot' system enables users to browse through lists of categorized services and select and start one, under the right conditions. A user that is active in 'de Koopgoot' can be either authorized or not authorized. Both types of users have access to certain public 'Koopgoot'-features, like browsing the list of services and starting free services or demo's. There are however features that are only accessible to authorized users: a user must log on to make payments, use his online credits or view and edit his profile. The Koopgoot model is build around two general states in which a user can be active in the system: *not logged in* and *logged in*. Later on in the paragraph we will study the two states in more detail and show that they are in fact a fusion of more states.

To maintain a well-organized model of the Koopgoot system, we have split up the model into three separate models, each representing a subset of the complete model. The union of the three models will therefore be a complete model of the Koopgoot subsystem. The first model contains the numerous user-input arcs while the second model contains the user-output arcs. Finally a third model is introduced in which we use a "super place" to model the system's exceptional exit flows as an addition to the system's normal exit flows present in the first two models.

Finally, the system also includes functionality to manage the content of Koopgoot. A model of the management functionality will be presented in paragraph 4.7.

As the architecture of the Koopgoot is fairly complex, it is interesting to look at the soundness of the system, to guarantee the absence of deadlock and garbage left behind in the system. We will attempt to proof that the system is 1-sound in paragraph 4.8.

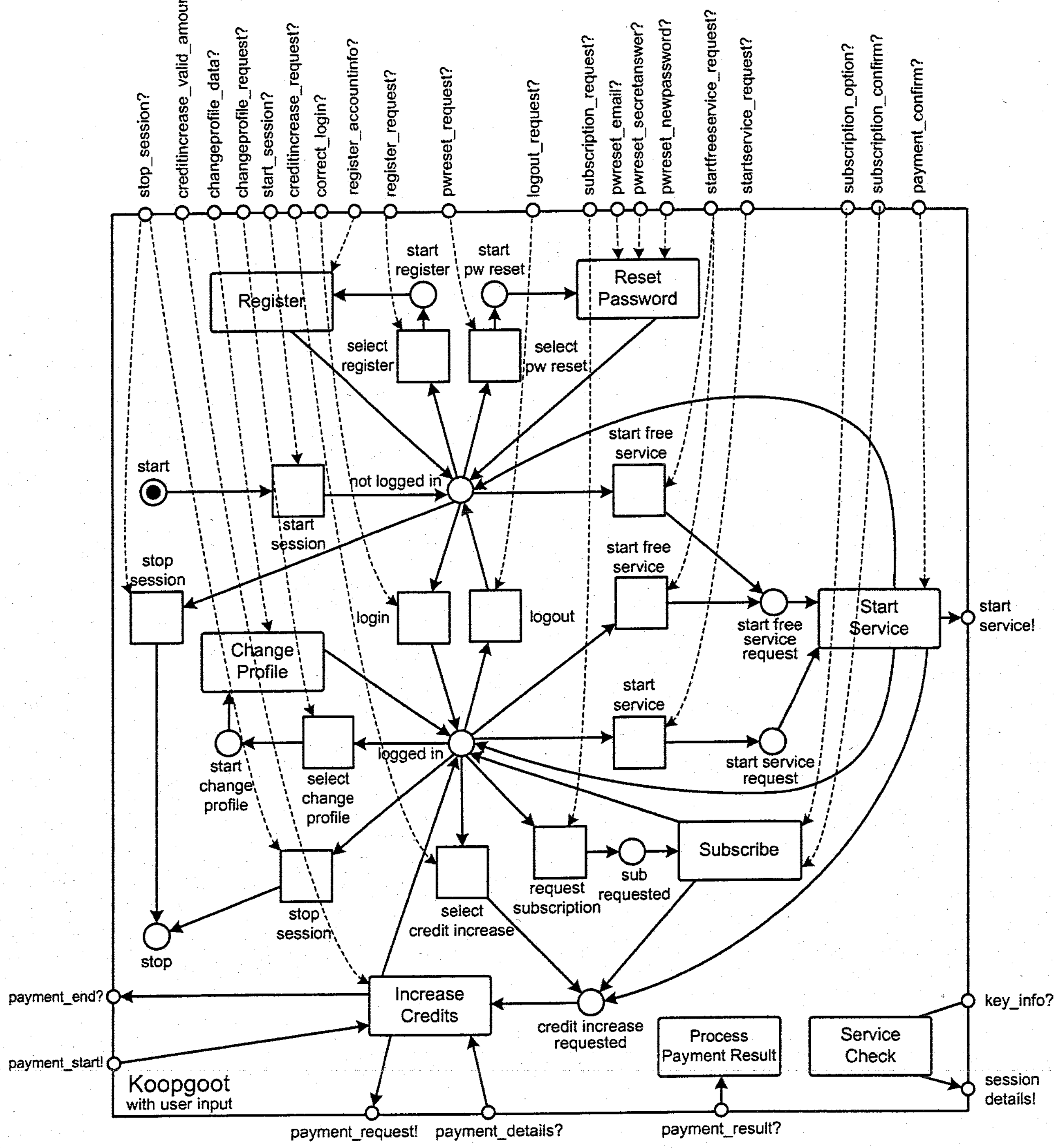


Figure 4.2. Koopgoot with user input

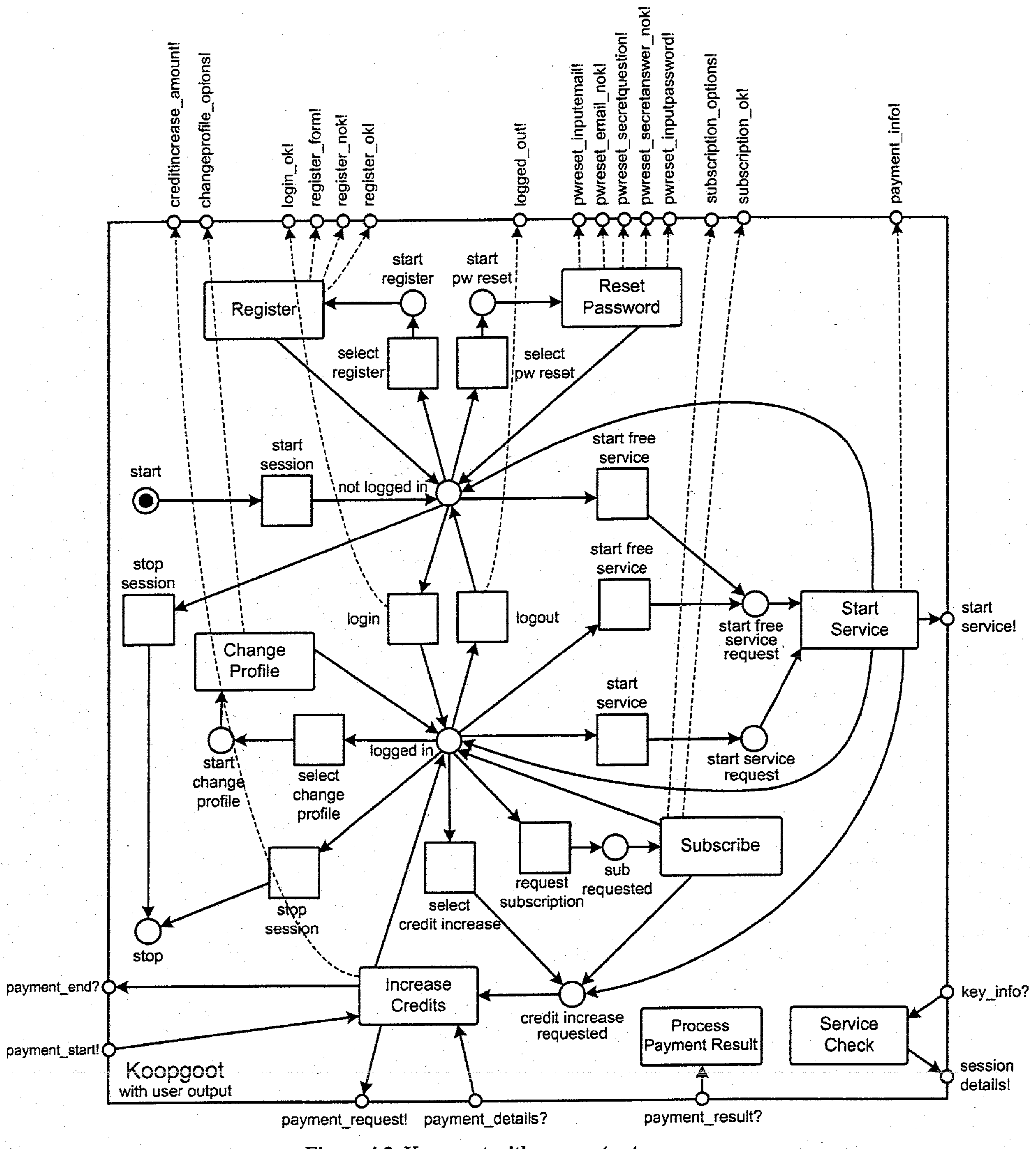


Figure 4.3. Koopgoot with user output

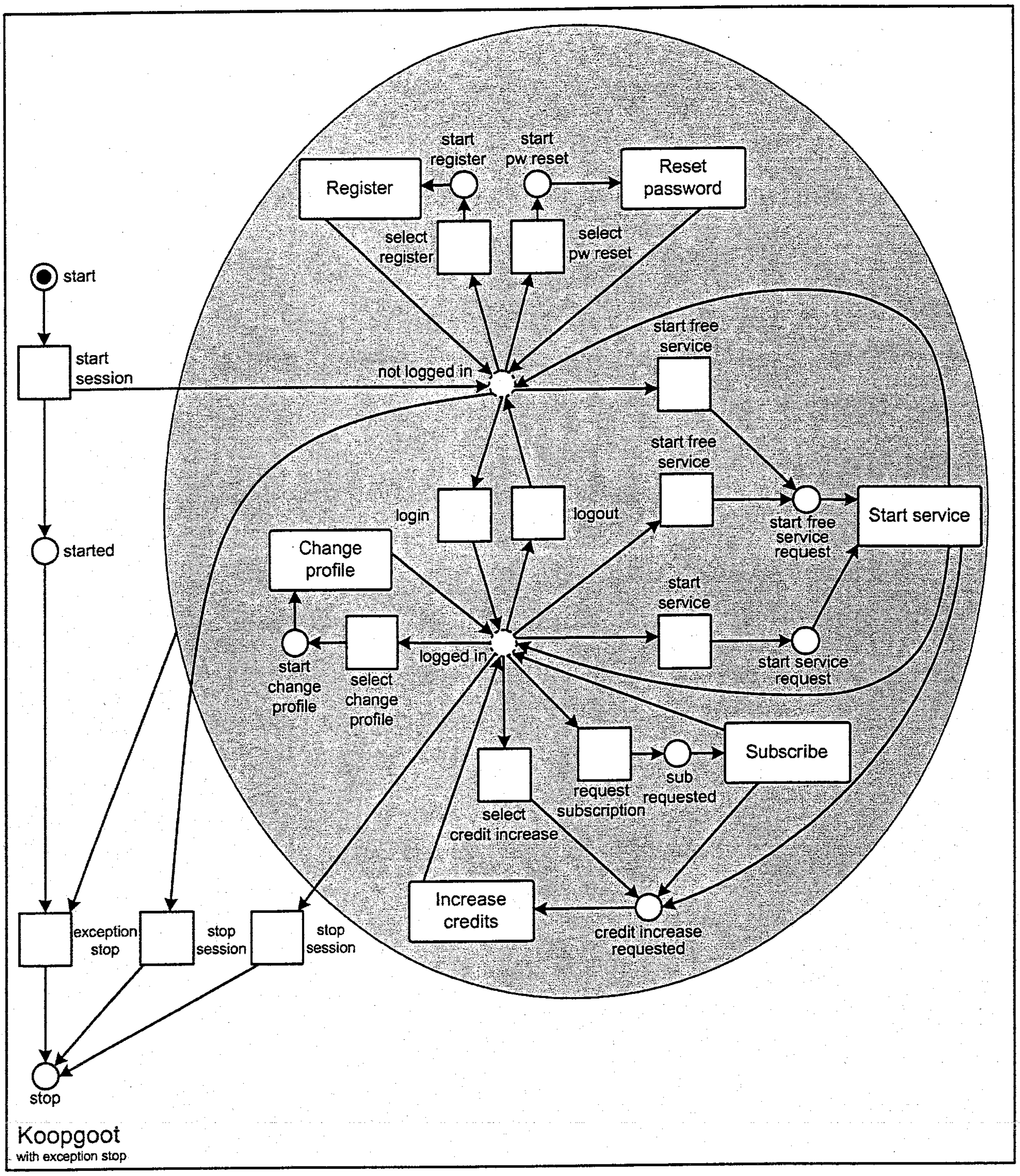


Figure 4.4. Koopgoot with "super place"

The "super place" in Figure 4.4 does not include the dotted places *not logged in* and *logged in*, as they have a 'normal exit' procedure available via the connected *stop session* transitions. Note that the model presented in Figure 4.4 is not a complete model of the 'Koopgoot' subsystem but only a subset to describe the Koopgoot's exit procedure.

The two main states of the system, *not logged in* and *logged in* can now be described in more detail. Both places can be seen as a fusion of two more specific states: *startview* and *serviceview*. The *startview* state represents a state in which an overview of all service categories is presented. To browse the services of a specific category, the user can enter the *serviceview* state by selecting an industry of function.

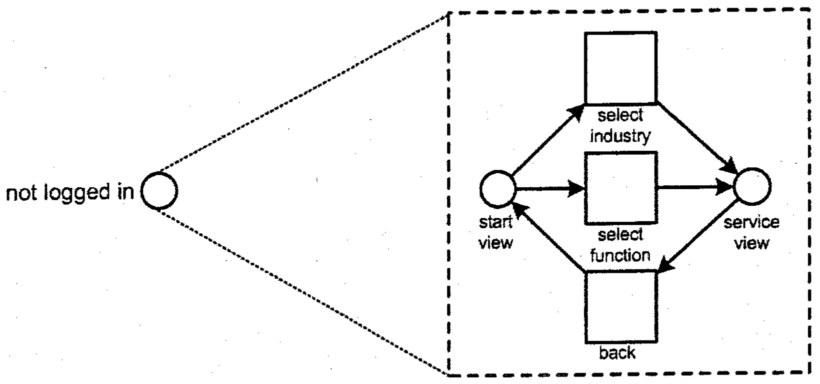


Figure 4.5. Not logged in

The logged in state contains the exact same two states and also has the same transitions available to switch between them.

Now that we have presented a model for the Koopgoot system, we will use the next subparagraphs to describe its subsystems. We will not discuss the other transitions in the Koopgoot model and consider them as atomic actions.

4.3.1 Register

The 'Register' subsystem contains the functionality that is needed for users to register in order to get a login account for the Koopgoot. A user is asked to supply personal details including an e-mail address that is unique to the system, after which an activation e-mail is sent to the user's e-mail address. The e-mail contains a link that the user has to click on to activate his account.

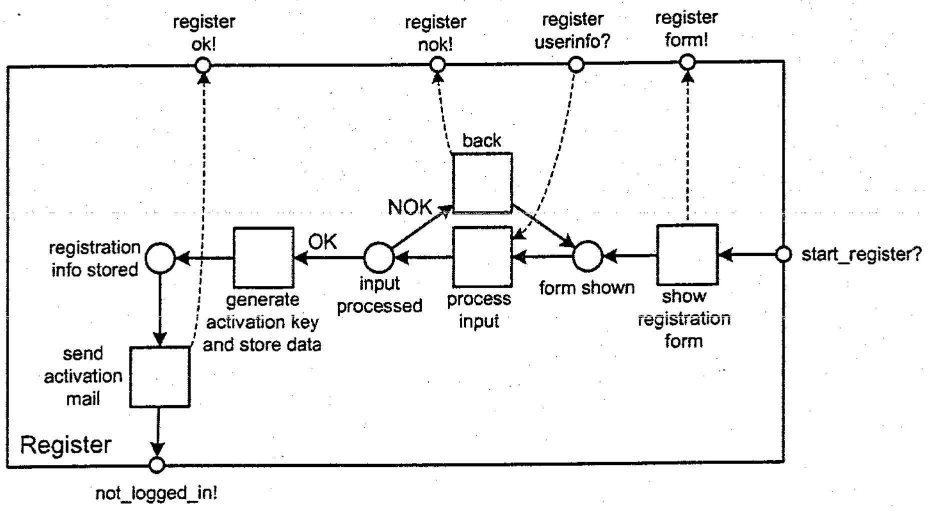


Figure 4.6. Register

4.3.2 Reset password

The ‘Reset password’ workflow is described in Figure 4.7.

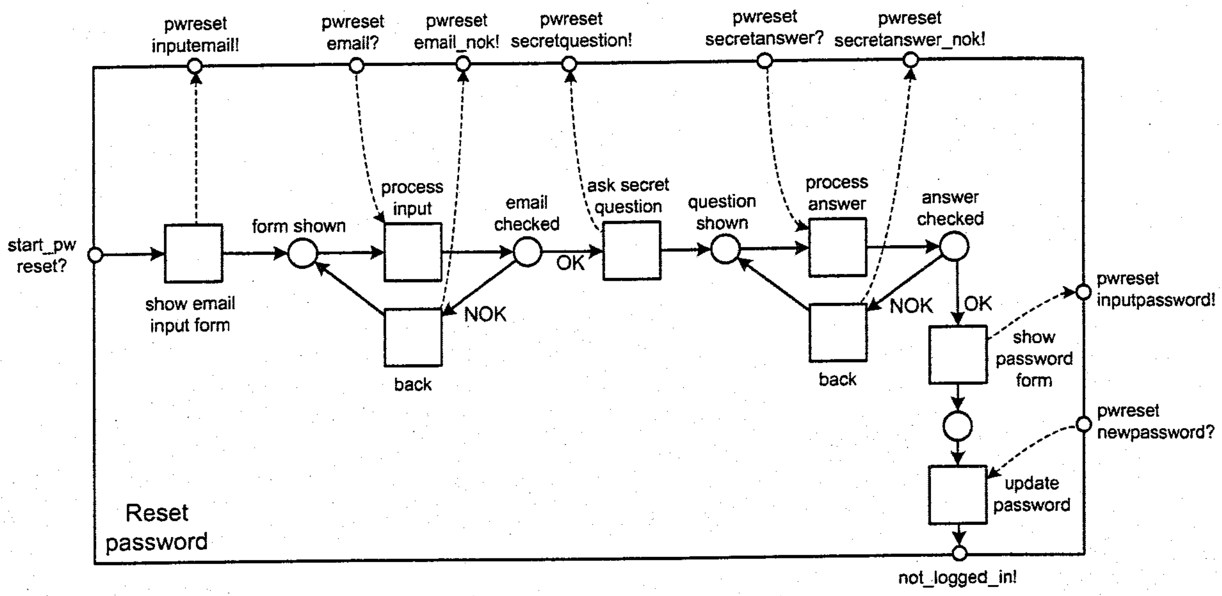


Figure 4.7. Reset password

4.3.3 Start service

The ‘Start Service’ component (Figure 4.8. Start service) handles the service start procedure. The component can start two types of services: free services and non-free services. A request to start a free service will arrive at pin *start_free_service_request?*. This request does not require further processing: the free service is started immediately in a new browser session and the user is returned to his previous state in the Koopgoot.

A request to start a non-free service arrives at the *start_service_request?* pin. The first transition that fires will check if the user has a valid subscription to the service. If this is the case, the system will enter the *access granted* state. If not, the user has to pay first, to enable single usage of the service, using his online credits. The online credit balance will be checked and if there are insufficient credits available, the system will exit the ‘Start service’ component at the *increase_credits!* pin and enter the ‘Increase credits’ subsystem. If the user has sufficient credits available, he will be prompted to pay for the service. If the user completes the payment, the *access granted* state is reached.

From this state, the transition *generate session key* is fired that generates a random key for the current session and stores it in the database ServiceAccess table, together with other session-specific data like the sessionId, serviceId and timestamp. As a result, the state *key stored* is reached, enabling transition *start service with parameters in new window* to fire. This transition composes the parameters sessionKey and sessionID that have to be passed to the service and then redirects the user to the service. As a result, two tokens will leave the *Koopgoot* subsystem: one through pin *start_service!* that will enter the *service* subsystem at the *start_service?* pin and one through pin *return_logged_in* that will return to the *logged_in* state.

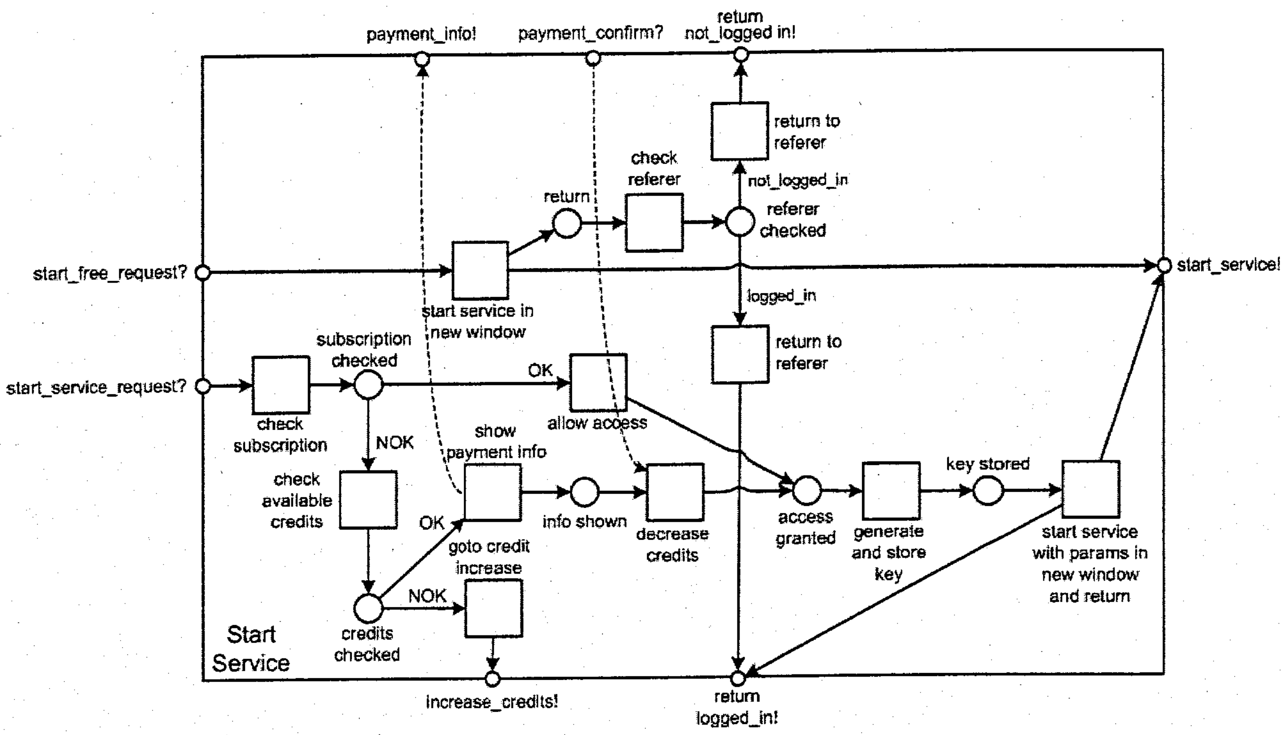


Figure 4.8. Start service

4.3.4 Increase credits

The Increase Credits component (Figure 4.9. Increase Credits) contains three subsystems that all access the Payment store. The component is activated via the *credit_increase_requested?* pin, that is connected to the Payment Init subsystem. Payment Init initializes a payment and has two possible paths:

- Payment Init is started with an amount supplied. A payment record will be created in the Koopgoot system and a payment request will be sent to the OnlinePay subsystem.
- Payment Init is started without any parameters. The user will be prompted to enter an amount, after which a payment request for this amount will be sent to the OnlinePay subsystem.

By introducing the case analysis above, the Increase Credit component can be started either by another subsystem when a specific credit increase is required, or manually by a user who wants to increase his credits.

The response from the OnlinePay subsystem to the payment request is handled by the Payment Start component. The response is processed and linked to the Payment record that has been created in the Payment Init component, after which the user is redirected to a specific location at the PSP-site. (In fact, this location is supplied by the response from the OnlinePay component)

When the payment at the external PSP system is ended, a token will enter the Payment End subsystem, via its *payment_end?* pin. Subsequently, the status of the payment concerned is checked. Depending on the status of the payment, the Payment End component is exited or the page will be refreshed to re-check the status.

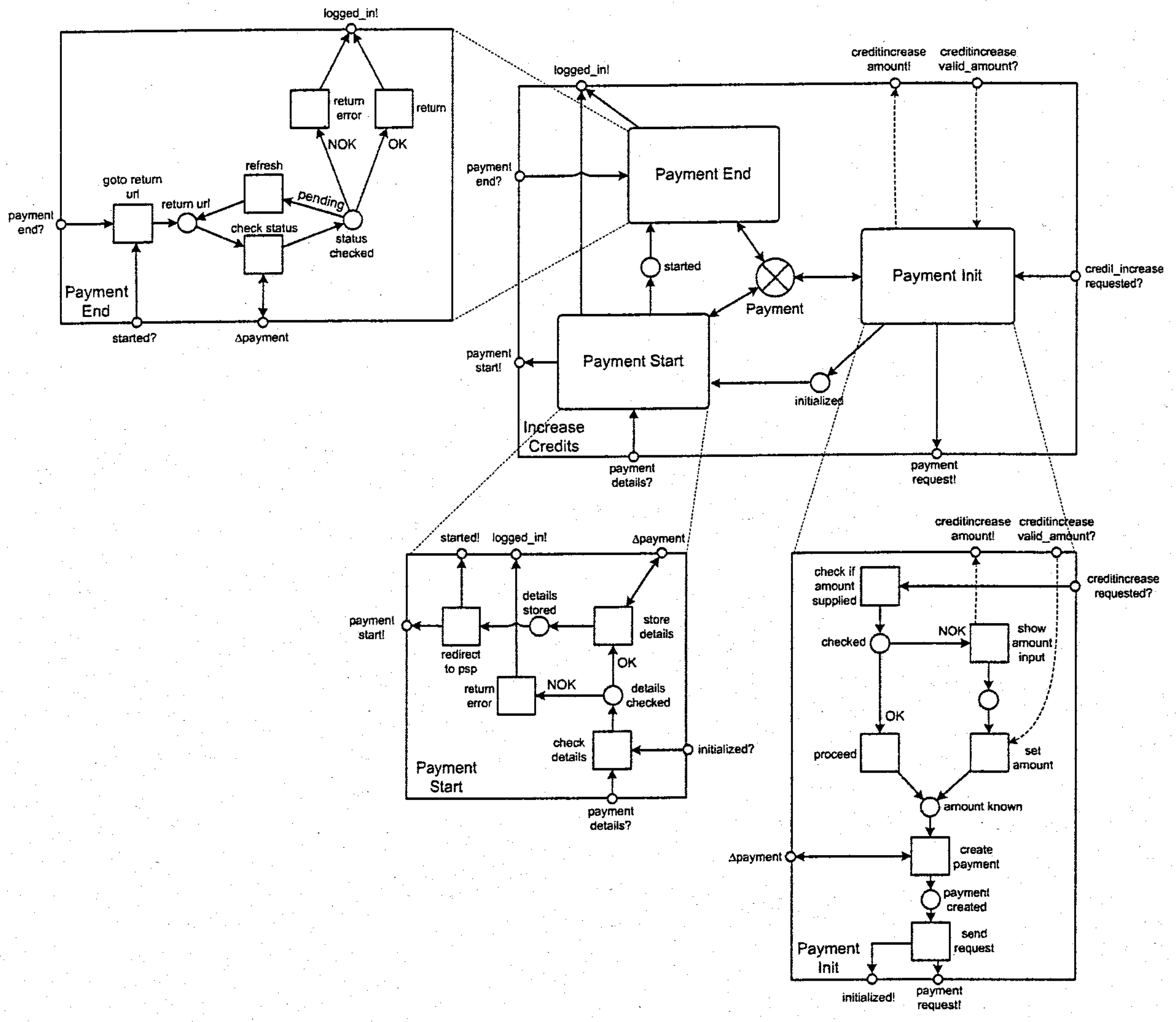


Figure 4.9. Increase Credits

4.3.5 Process Payment Result

Process Payment Result (Figure 4.10) handles the payment result messages that are received from the OnlinePay subsystem. A payment result arrives at the *payment_result?* pin and is linked to an existing payment item in the Payment store. The status of the existing item is updated and a credit increase will follow if and only if the received status represents a successfully completed payment.

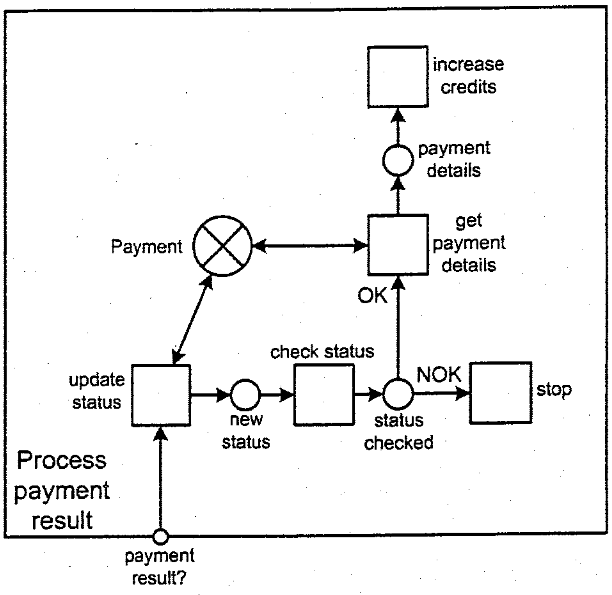


Figure 4.10. Process payment result

4.3.6 Subscribe

The Subscribe component handles a subscription request for a specified service. It contains the basic user interaction to determine the desired subscription type and starts the 'Increase credits' subsystem if a user has insufficient credits available for the specified service subscription. If the amount of credits available is sufficient, the user will be subscribed to the specified service and his online credits will be decreased.

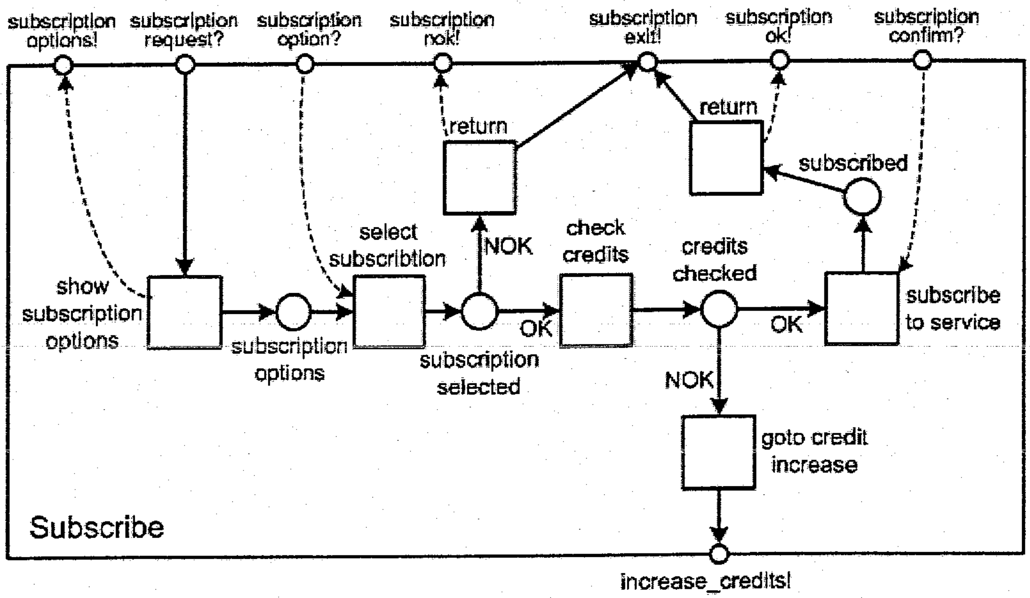


Figure 4.11. Subscibe

4.3.7 Service check

The actual check to see if a service may be started is performed by the *Service Check* subsystem located in the *Koopgoot*. This component is initiated from the outside by the Service subsystem via the *key_info?* pin. The data received at the key_info? pin will be described in the next paragraph (4.4). Transition *verify key info* will check the validity of the received data and after this check, a response will be sent back to the service.

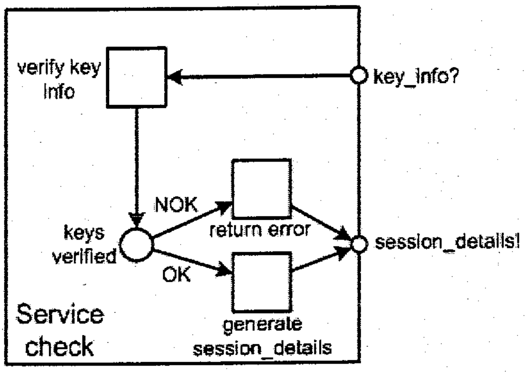


Figure 4.12. ServiceCheck

4.4 Service

The Service subsystem is another part of our proposed environment (Figure 4.1. Environment model), representing a service that can be started from 'de Koopgoot'.

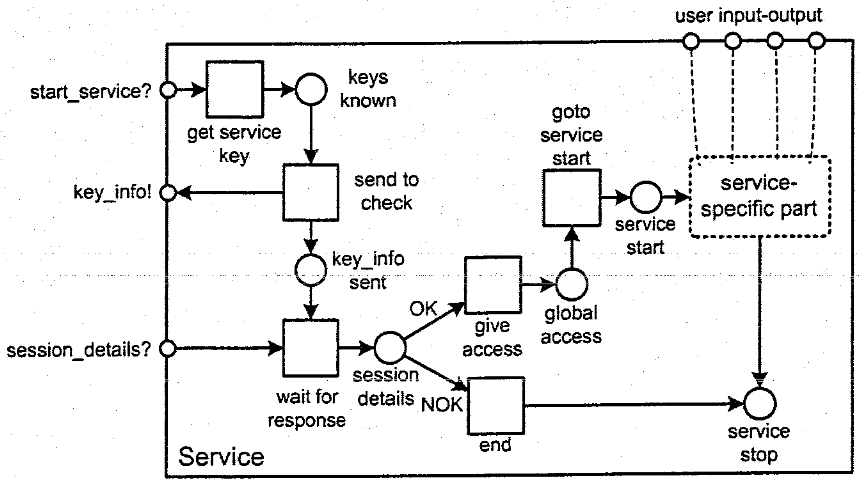


Figure 4.13. Service model

The subsystem is initiated via the *start_service?* input pin, representing a browser session pointing to the service location with the correct set of parameters. The transition *get service key* retrieves the service's unique serviceKey that is stored somewhere in the service subsystem. After this, the state *keys known* is reached, where the service has the following keys available:

- serviceKey, required for service authentication;
- sessionKey, taken from passed parameters, required for session authorization;
- serviceId, taken from passed parameters, required for session authorization.

The following transition, *send to check*, takes these three keys as input and sends them to the *Koopgoot* to check their validity. *Send to check* also produces a token in state *key_info sent*, that is one of the two tokens needed to fire transition *wait for response*. This transition also needs the response from the *Koopgoot* system to fire, that will be received from the *session_details?* pin.

When the *wait for response* transition is fired, a token will be produced in state *session_details*, representing the *session_details* received from the *Koopgoot* subsystem. If *session_details* contains no valid data, the precondition NOK will be true and the system will end. If *session_details* contains valid session data, the *give access* transition will fire because its precondition OK will be satisfied. The *give access* transition is modelled to set global variables in the service application to guarantee access to all parts of the service. Subsequently, the actual service will be started. The functionality of this subsystem is beyond the scope of this chapter, we refer to chapter 2 for an outline of the service functionality we have in mind.

4.5 Deloitte OnlinePay

In this paragraph we will present a model of the Deloitte OnlinePay system. This external system has been developed in consultation with the ICTS group of Deloitte and will be used by the Koopgoot system to communicate with the PSP. It enables Deloitte to use the online payment functionality for other and future portal applications in a uniform manner.
We will not go into specific details of the OnlinePay component, because they are not relevant to the Koopgoot itself, but we will only describe the trivial processes of the system that are important to understand the process of online payment.

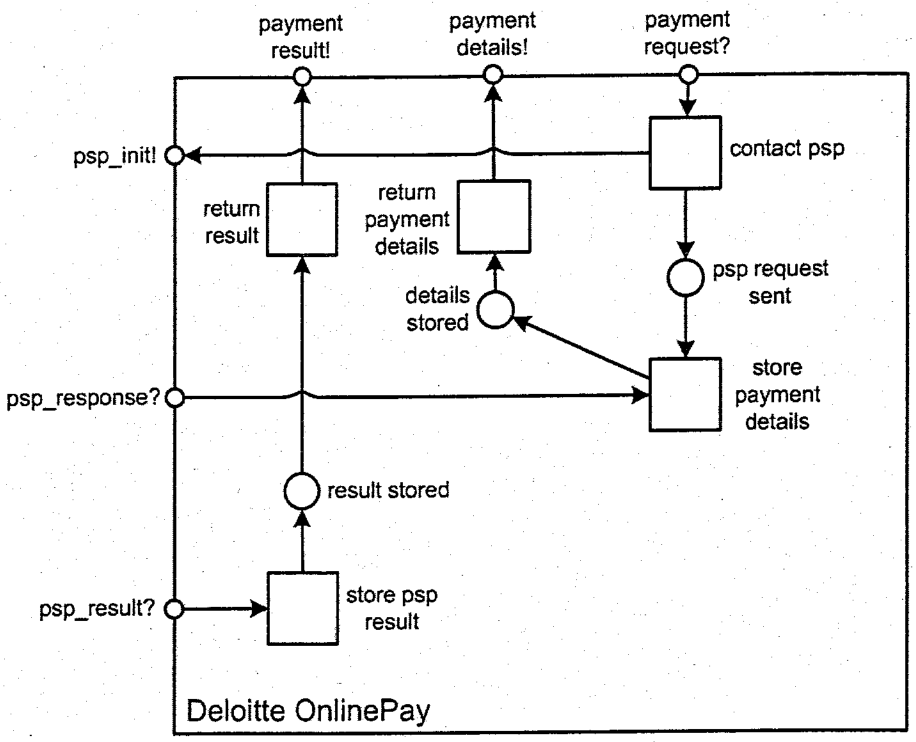


Figure 4.14. Deloitte OnlinePay

When a payment request -containing an amount, user data and return url- is initiated from Koopgoot, the OnlinePay component will compose and send a payment request to the PSP. Subsequently, OnlinePay will wait for a confirmation response from the PSP to confirm that a payment can take place. A response will be sent to the Koopgoot via the *payment_details!* pin, including the relevant information for the Koopgoot to start the payment:

- a *SyncId* which uniquely identifies this payment request, together with
- a *RedirectUrl* containing an specific url located on the PSP server where the initiated payment has to be made.

The second independent workflow in the OnlinePay component handles the incoming payment result from the PSP on the *psp_result?* pin: the payment status is stored or updated and offered to the Koopgoot at the *payment_result!* pin for further processing. Note that multiple status changes can take place for a single payment: a payment status can change from *pending* to *completed*, causing the workflow net connected to the *psp_result?* to run twice.

4.6 PSP

In Figure 4.15 we have modelled the trivial workflows for the PSP subsystem, derived from the specifications supplied by the PSP. These workflows are required to understand the interaction with the Koopgoot and OnlinePay subsystems.

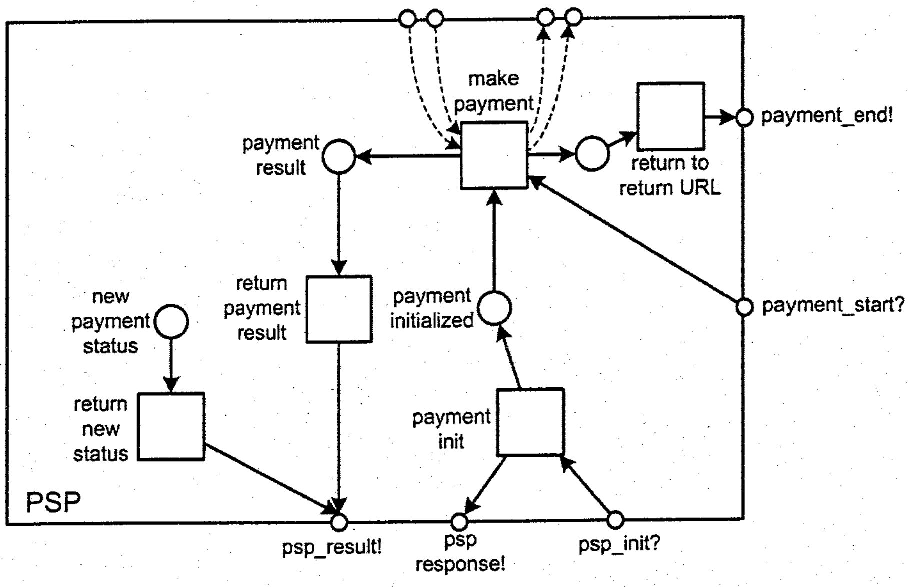


Figure 4.15. PSP

A user enters the PSP subsystem via the *payment_start?* pin and starts making his payment (that has already been initialized). When the payment is finished the user is returned to the Koopgoot and the payment result is sent to the OnlinePay system. We have modelled these two actions in parallel as it is not guaranteed that a payment result is received by the Koopgoot before the user has returned. It is possible that multiple payment result messages are sent by the PSP for a single payment, caused by payment status changes.

4.7 Koopgoot Management

In this paragraph we will present a model of the Koopgoot management functions. To keep the model well-organized we start the model in the *koopgoot management* state. This state can be reached via the *not logged in* state of the Koopgoot when an administrative user logs in. The Koopgoot Management model can be exited via a stop session action performed by the user or by an exception. The start and exit procedure of the management site is equivalent to the one presented in Figure 4.4. Koopgoot with "super place".

In Figure 4.16 the high level model of the Koopgoot management is displayed. In this figure, we have omitted the place labels for the interface places that connect the management subsystem to the Client subsystem as the user interaction arcs are so numerous, describing all of them at this high level would make the model unnecessarily complex. Of course, user interaction will be described in more detail at a lower level. As the figure shows, three subsystems can be started from the *koopgoot management* state.

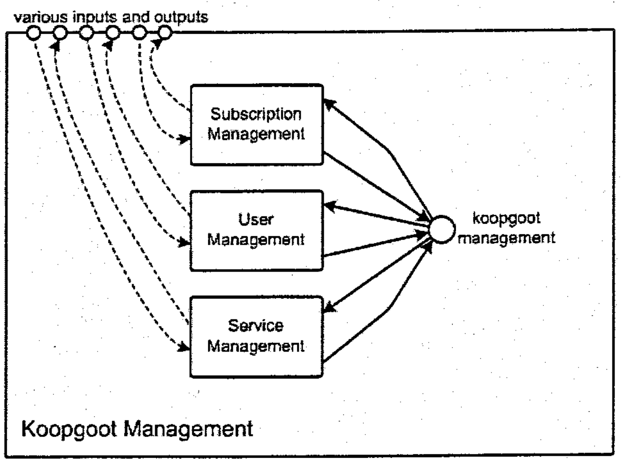


Figure 4.16. Koopgoot Management

Table 4.1 describes the actions that can be performed in the subsystems modeled in Figure 4.16.

Table 4.1. Koopgoot management actions

Service Management
Add services
Edit services
Remove services
User Management
Add users
Edit users
Remove users
Subscription Management
Add subscriptions
Edit subscriptions
Remove subscriptions

Because the models of the three specified subsystems have a similar architecture, we will only describe the model of the Service Management subsystem. To maintain a well-organized model of the Service Management subsystem, we have split up the model into two separate ones, each representing a subset of the complete model. The union of the two models will therefore be a complete model of the Service Management subsystem. The first model, shown in Figure 4.17 contains the numerous user-input and user-output arcs. A second model described in Figure 4.18 uses a "super place" to model the cancel option available to the user.

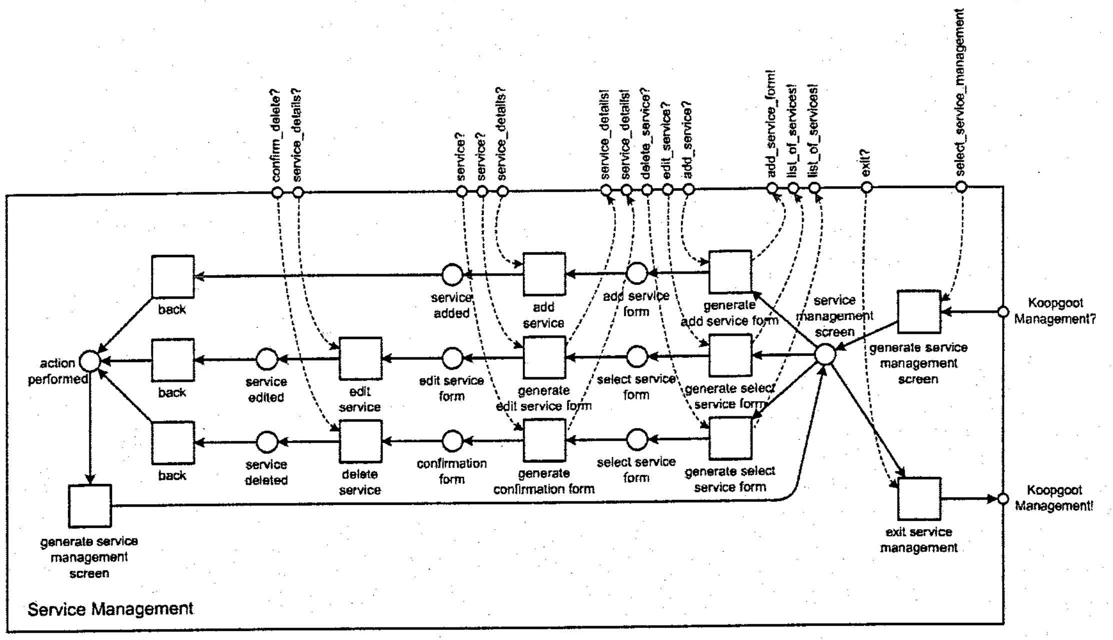


Figure 4.17. Service Management with user interaction

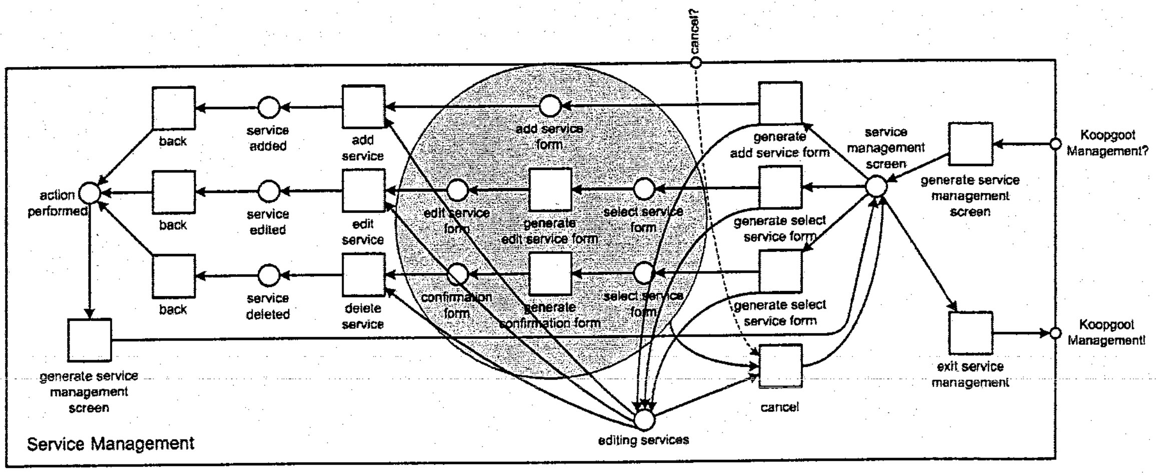


Figure 4.18. Service Management with "super place"

As shown in Figure 4.17, a user enters the Service Management subsystem via the k*oopgoot management?* pin and arrives at the *service management screen*. From here, the user can start adding a service, start editing a service, start deleting a service or exit the service management. In the process of adding, editing or deleting a service, a user can decide to cancel the operation. A cancel action will return the control of the system to the *service management screen* state. This is shown in Figure 4.18.

4.8 Soundness

We place the requirement of 1-soundness (definition A.4 on page 74) on our colored workflow net (definition A.3 on page 74) to eliminate the occurrence of dead-lock and because it is illegal for a system to be ended and at the same time have some tasks enabled.

Soundness of the Koopgoot

We will try to check the soundness of the net by reducing parts of the net to a particular pattern or class of nets. First, we analyse the behaviour of the net presented in Figure 4.2: the net has a begin state that leads to the state *not logged in*. From this state, the *logged in* state can be reached. From both states (*logged in* and *not logged in*) the end state of the net can be reached. All other functionality of the net can be described as loops that begin and end in the *not logged in* state or loops that begin and end in the *logged in* state.

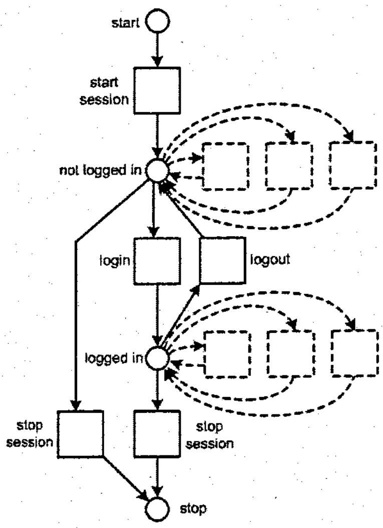


Figure 4.19. Simplified behaviour of the Koopgoot

It is easy to verify that the simplified net in Figure 4.19 is a state-machine workflow-net (definition A.5 on page 74). Since any SMWF is sound (lemma A.1 on page 74) it follows immediately that the net in Figure 4.19 is sound.

To verify that the Koopgoot system is sound, we have to show that the simplified net in Figure 4.19 is a correct reproduction of the Koopgoot model. To do so, we must verify that every outgoing arc from *not logged in* and *logged in* is either part of a loop or an incoming arc of a named transition in Figure 4.19.

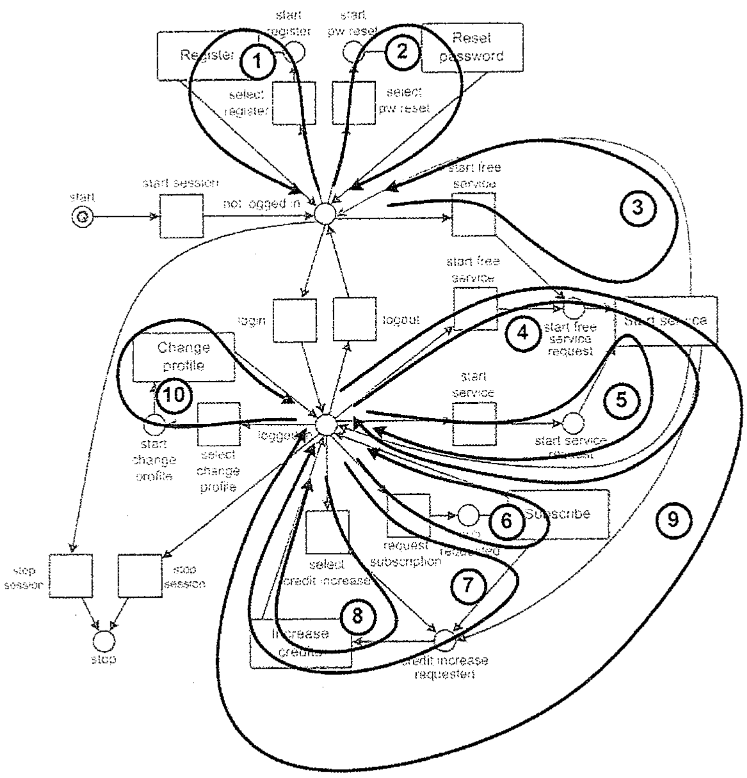


Figure 4.20. Numbered loops

1. It is easy to verify that workflow (1) in Figure 4.20 is a strongly connected state-machine workflow net (definition A.3 on page 74) by looking at the Register component described in Figure 4.6. Since any state-machine is sound (lemma A.1 on page 74) it follows immediately that workflow (1) is sound.
2. Workflow (2) has the same pattern as workflow (1) and is therefore also a strongly connected state-machine and sound.
3. To analyse workflow (3) and its behaviour inside the Start Service component, we describe a flattened model of the workflow:

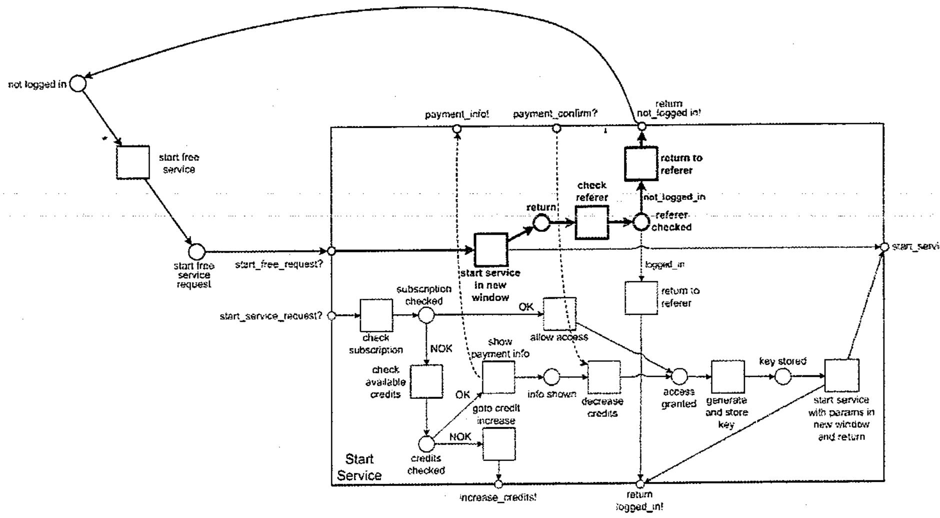


Figure 4.21. Workflow (3)

We can reason that the indicated path in Figure 4.21 is the only possible path because only transition *return to not logged in* can fire from the *referrer checked* state as its precondition *not_logged_in* is the only one that can be satisfied. Transition *Start service in new window* has a second output arc that can be ignored as it directly exits the Koopgoot system. The described path is a strongly connected SMWF and therefore also sound. (lemma A.1)

4. Workflow (4) starts a free service from the *logged in* state and returns to that same state. It has the same pattern as workflow (3): the other precondition (*logged in*) is satisfied after the state *referrer checked* and the Start Service component is exited via the *logged_in* pin. Workflow (4) is thereby also sound.
5. Workflow (5) and workflow (9) describe the two possible paths when a non-free service is started. Workflow (5) returns to the *logged in* state when it exits the Start Service component.

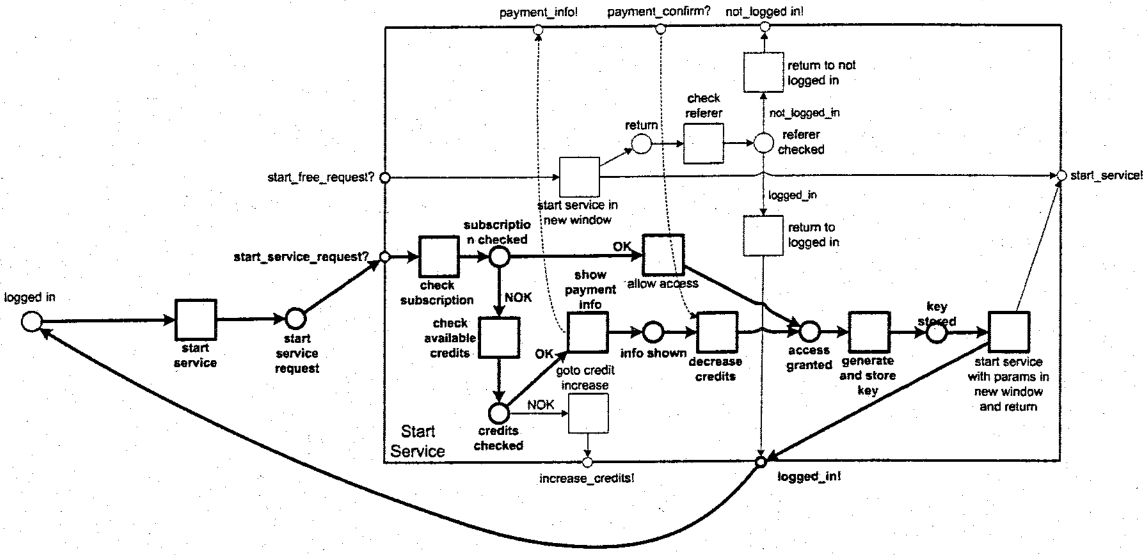


Figure 4.22. Workflow (5)

It's not difficult to see that the path through Start Service is a strongly connected SMWF and thereby sound (lemma A.1). The second outgoing arc of transition *"Start Service with params..."* is ignored for our soundness proof as it exits the Koopgoot-system immediately.

6. Workflow (6) subscribes a user to a service and returns to the *logged in* state when it exits the Subscribe component.

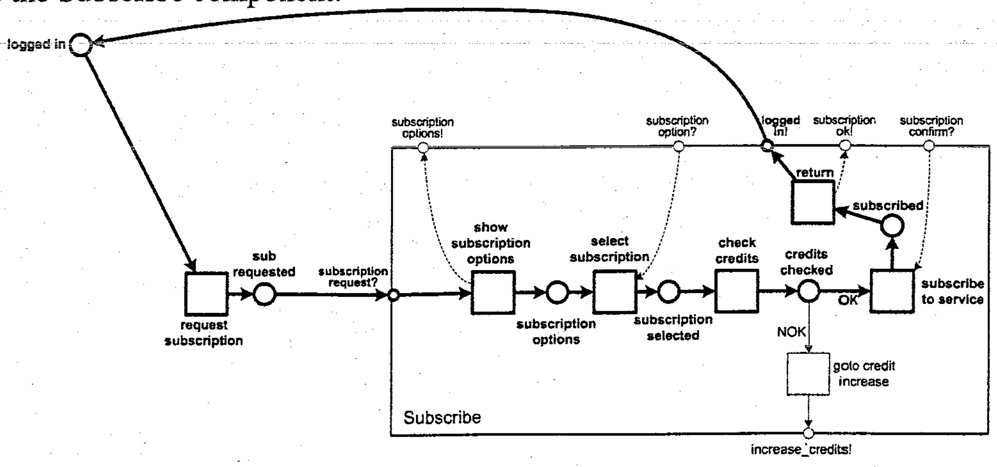


Figure 4.23. Workflow (6)

Workflow (6) is a strongly connected SMWF.

7. Workflow (7) starts a subscription and exits the Subscribe component to enter the Increase Credits component. Subsequently it returns to the *logged in* state.
   It is easy to verify that the path through subscribe is a SMWF (as only the NOK precondition is satisfied in state *credits checked*). To verify that the increase component is also sound, we present a flattened model of the component in Figure 4.24. To come to this flattened model, the communication with the external systems is ignored.

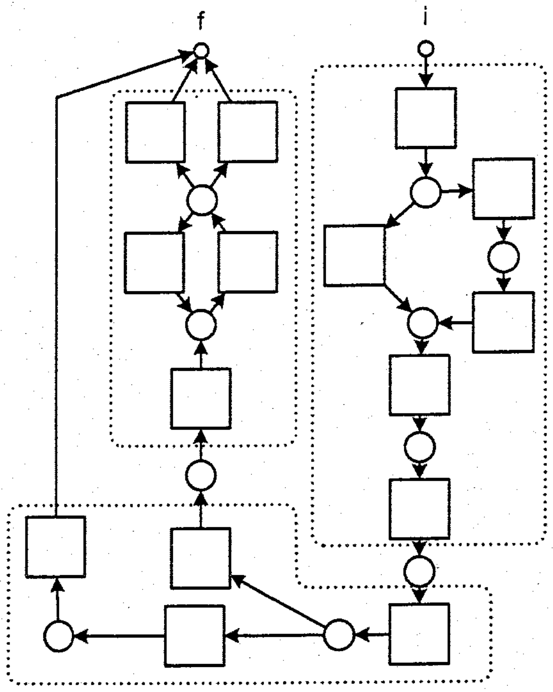


Figure 4.24. Flattened model of increase credits

   Figure 4.24 makes it easy to verify that Increase Credits is also a SMWF.

8. Workflow (8) enters and exits the Increase Credits component. As we have shown for workflow (7) the Increase Credits component is a SMWF. Workflow (8) is therefore also sound.
9. Workflow (9) tries to start a non-free service but exits the Start Service component to enter the Increase Credits component. It can easily be derived from Figure 4.22 that the workflow net through the Start Service component is a SMWF. This workflow is followed by the Increase Credits workflow that is also a SMWF (7).
10. Workflow (10) starts the Change Profile component. This component is a simple state machine workflow net and therefore sound.

4.9 Example scenario I: selecting and starting a non-free service

In this paragraph we will focus on the interaction between “de Koopgoot” and its non-free services. The environment that is described in this paragraph is a subset of the total environment described in paragraph 4.2 and consists of “de Koopgoot” which enables users to select and pay for a service and the services that can be started. A service can be started from “de Koopgoot” when a user has obtained access rights to the service by either paying a single fee required for the service or having a valid subscription for the service. We will therefore start the use-case in state *access_granted* of the Start Service component that is part of the Koopgoot subsystem.

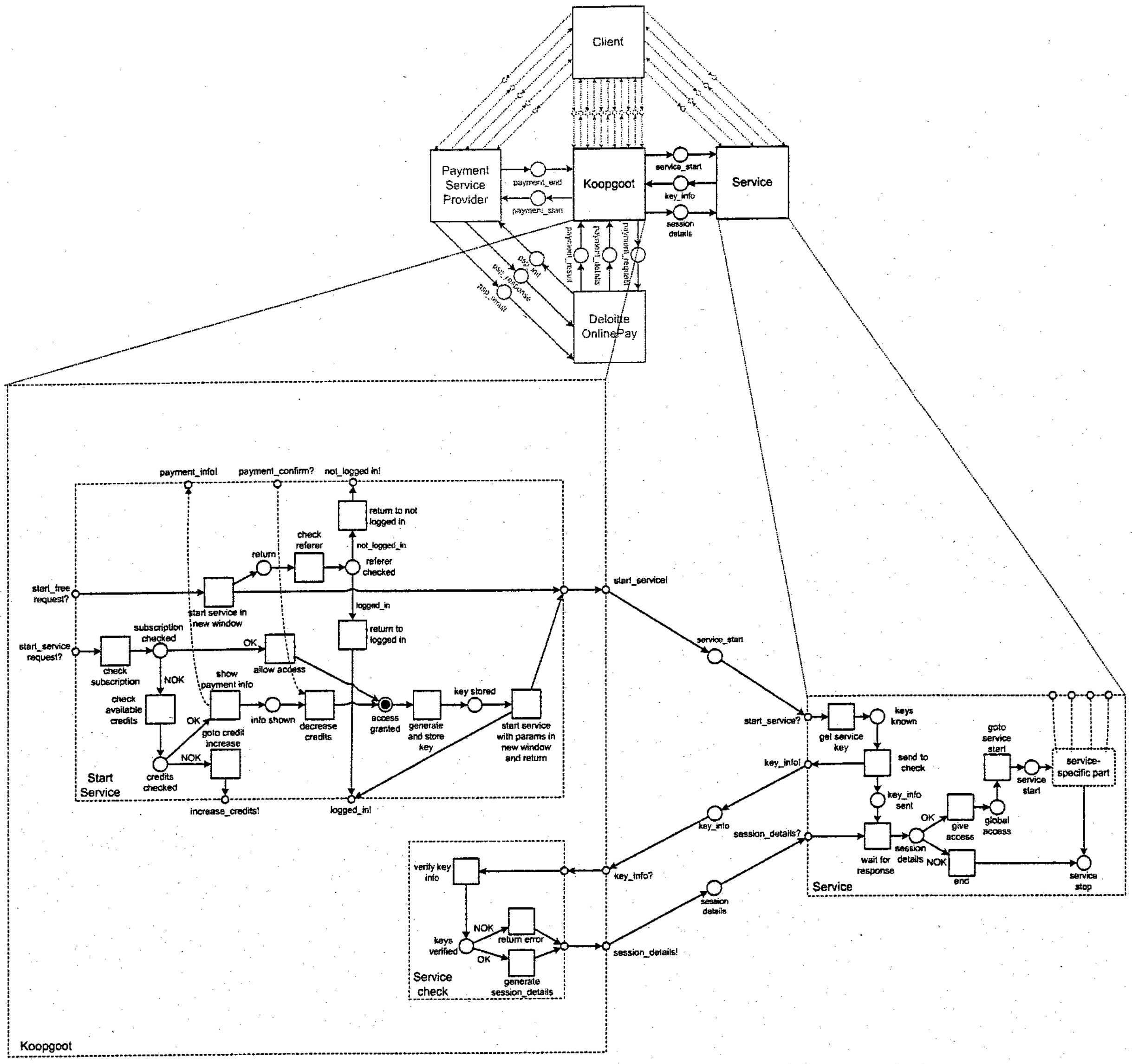


Figure 4.25. Starting a service

Table 4.2 describes the normal scenario for this use-case: a service is started with valid parameters. The parameters are successfully checked and service access is granted to the user.

Table 4.2. Scenario - Starting a service

Transition	*System*	
generate and store key	Koopgoot, Start Service	*Generate a random key for the current session and store it in the Koopgoot database ServiceAccess table, together with other session-specific data like the sessionId, serviceId and timestamp.*
start service with params..	Koopgoot, Start Service	*Redirect to service with parameters sessionId and sessionKey*
get service key	Service	*Retrieve unique service key stored in service*
send to check	Service	*Send serviceKey (retrieved), sessionKey (from parameters) and serviceId (from parameters) to Koopgoot*
verify key info	Koopgoot, Service Check	*Check the validity of the received keys by querying the Koopgoot database*
generate session details	Koopgoot, Service Check	*If the keys are valid, a set of session details (userId, email, address etcetera) is generated to return to the service.*
wait for response	Service	*Receive and process the session details*
give access	Service	*If the session details are valid, set global service variables to allow access to the service*
go to service start	Service	*Go to the welcome page to start using the service*

4.10 Example scenario II: making an online payment

In this paragraph we will focus on the interaction between "de Koopgoot" system, the OnlinePay system and the PSP (Payment Service Provider) system. The environment that is described in this paragraph is a subset of the total environment described in paragraph 4.2. We will start the normal scenario of this use-case with a token that arrives at the Increase Credits component.

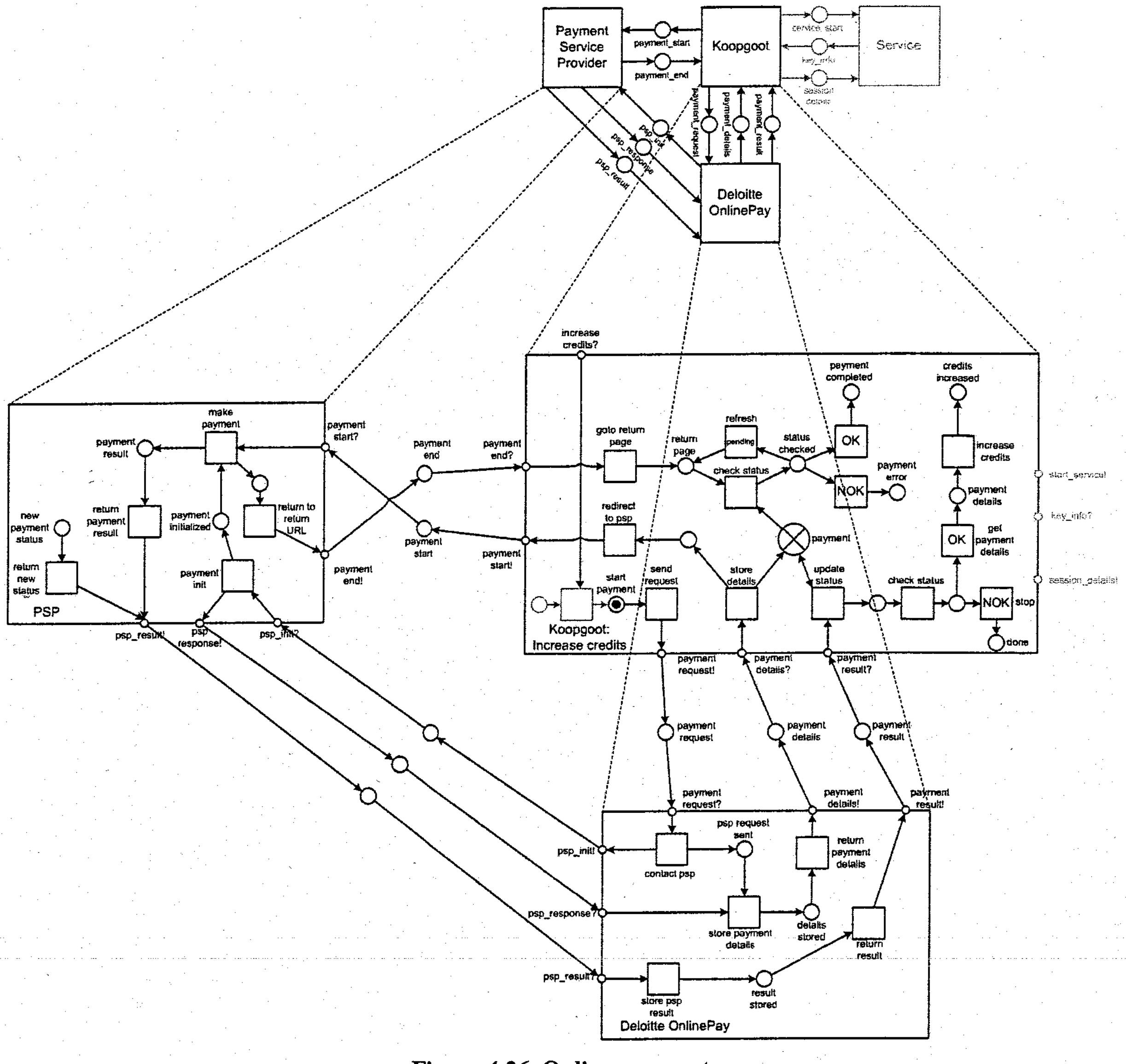


Figure 4.26. Online payment

Table 4.2 describes the normal scenario for this use-case: a credit increase request is accepted after which the Koopgoot requests a payment at OnlinePay. OnlinePay informs PSP about the payment coming up and sends a response to the Koopgoot, containing an unique ID that is associated with the payment and a URL the user should be redirected to. Subsequently the Koopgoot stores the payment information and redirects the user to the PSP. At the PSP the user performs the actual payment. As a result, the PSP informs OnlinePay about the payment results and redirects the user to the Koopgoot. OnlinePay stores the results and sends them to the Koopgoot. The Koopgoot stores those results, increases the amount of credits of the user and as a result the payment is completed successfully.

Table 4.3. Scenario – Online payment

Transition	*System*	
send request	Koopgoot, Increase Credits	*Send payment request to OnlinePay*
contact PSP	OnlinePay	*Contact the PSP to initiate a new payment*
payment init	PSP	*Initialize payment and return new payment details (syncId, payment URL) to OnlinePay*
store payment details	OnlinePay	*Store returned payment details*
return payment details	OnlinePay	*Return payment details (syncId, payment URL) to Koopgoot.*
check details	Koopgoot, Increase Credits	*Receive and check OnlinePay response. If the response is empty, the PSP is unavailable.*
store details	Koopgoot, Increase Credits	*Update Payment table with syncId*
redirect to PSP	Koopgoot, Increase Credits	*Redirect user to PSP payment URL (received from OnlinePay)*
make payment	PSP	*User payment*
(*) return payment result	PSP	*Return payment results (inlusive syncId) to OnlinePay*
store PSP result	OnlinePay	*Store returned payment results*
return result	Koopgoot, Increase Credits	*Return payment results (syncId) to Koopgoot.*
update status	Koopgoot, Increase Credits	*Update Payment table with new payment results*
check status	Koopgoot, Increase Credits	*Checks if the payment is completed*
get payment details	Koopgoot, Increase Credits	*Retrieve the payment details from the database*
increase credits	Koopgoot, Increase Credits	*Increases the amount of credits from the user*
in parallel with (*) return to return URL	PSP	*Redirect user to Koopgoot return URL (provided by Koopgoot)*
goto return page	Koopgoot, Increase Credits	*Redirects the user to the return page of the Koopgoot*
check status	Koopgoot, Increase Credits	*Checks if the credit increase is completed*
OK	Koopgoot, Increase Credits	*The online payment is completed*

4.11 Class Diagram

We use an adapted version of the class diagram used in [CHS2003] to model the relevant Koopgoot classes.

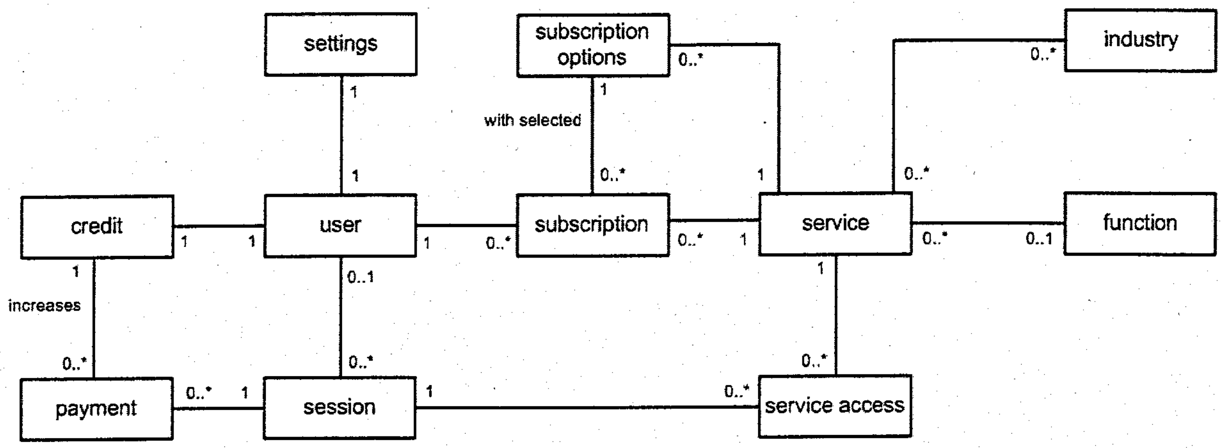


Figure 4.27. Class model of the Koopgoot

Each visit of a user to the Koopgoot involves a session. A user can make payments in a session to increase his Koopgoot-credits. Credits can be used to subscribe to a service (by selecting a subscription option that defines the period and price) and to enable single usage of a service. Access control to a service in a specific session is controlled by the service access class. A service can belong to industries and to a global function group.

5 Technical architecture

5.1 Introduction

This chapter will present an overview of the technical architecture of the Koopgoot environment. We will describe the physical components of the system and the software techniques that have been used to realize them. Subsequently, we will map the logical components to the physical ones, to show the link between the logical model and the architectural design [CHS2003].

5.2 Technical Requirements

The technical architecture has to meet the following non-function requirements:

- Performance – the system's performance must be at such a level that users experience no noticeable delays when using the system;
- Scalability – the system or components of the system must be modifiable to fit a problem area;
- Security – the system must be secured to protect all sensitive information;
- Maintainability – the software system or component must be easy to modify to correct faults, improve performance, or other attributes, or adapt to a changed environment [IEEE1990].
- Standardization – the system architecture must fit in with existing Deloitte IT-standards.

5.3 The three tier structure of the 'Koopgoot'.

To meet the technical requirements, we will use the three tier architecture[3]. As the name indicates, this architecture divides a system into three tiers: the presentation tier, the logical tier and the data tier. The presentation tier presents the system to the outside world and communicates with the outside world. The logical tier will provide functions, such as calculations and communications with other systems in the same environment. The logical tier serves the presentation tier and interacts with the data tier, where the actual database(s), belonging to the systems are located.

The environment model presented in chapter four can now be mapped onto the three tier model. The environment contains the following actors:

- Users. The users of the system will use an internet browser and will communicate with the Koopgoot presentation tier, which is the only tier accessible from the internet.
- Deloitte OnlinePay. This system that handles the PSP communication will communicate with the logical tier of the Koopgoot.
- Payment Service Provider. When a Koopgoot user wants to make a payment, the Koopgoot will have to announce the payment to the PSP before redirecting the user to the PSP site. When the user has completed the payment, the PSP will report the

[3] For detailed information on three tier architectures see [Eckerson1995].

payment result to the Koopgoot and redirect the user back to the Koopgoot. For announcing payments to the PSP and retrieving notifications from the PSP, the Koopgoot will use Deloitte OnlinePay. To redirect users from the Koopgoot to the PSP and visa versa, no direct communication between the Koopgoot and the PSP is needed. (In fact, the Koopgoot sends a redirect command to the user)

- Services. When a user has been redirected from the Koopgoot to a service, the service will have to determine if the user has the correct rights to use it. Because the services are not necessarily hosted in the same environment as the 'Koopgoot', they will have to communicate with the Koopgoot presentation tier.

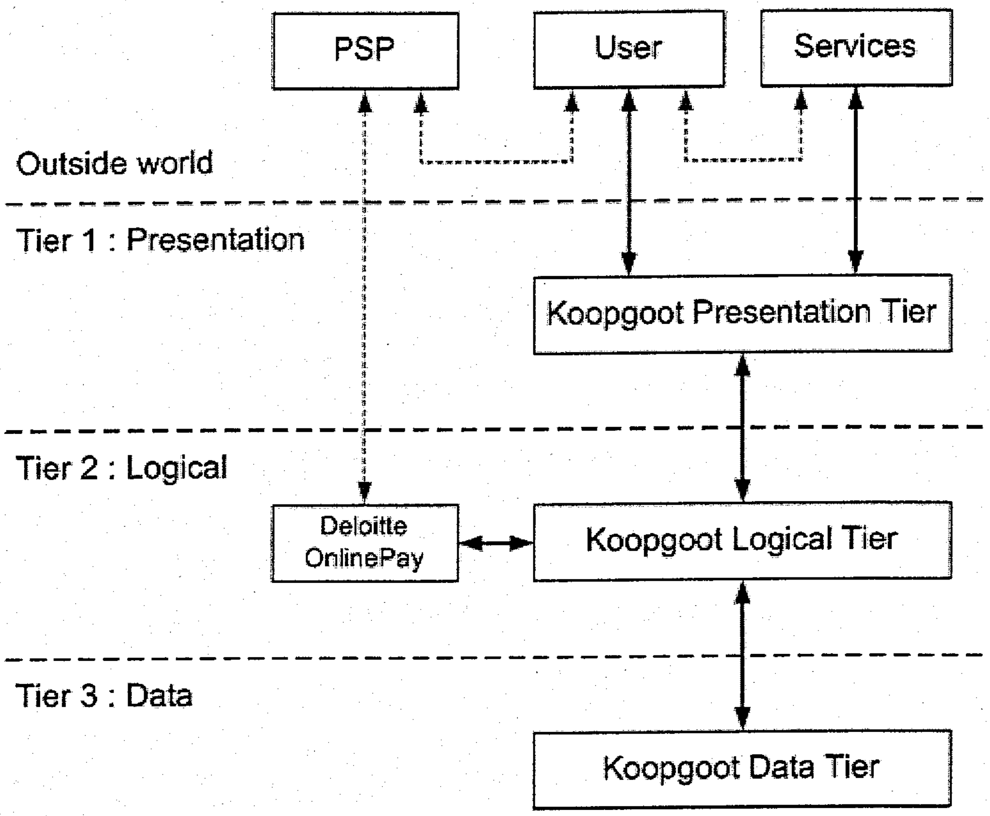


Figure 5.1. Three tier architecture of the Koopgoot

This scheme summarizes the communication between the Koopgoot and the other parties. The dotted lines indicate communication lines that are not important to the technical architecture of the Koopgoot. The other lines however have an influence on what kinds of techniques to use. To decide what techniques to use on which tier, each tier will have to be treaded distinctly:

- The presentation tier.
  The presentation tier communicates with the logical tier, the users and the services. For the communication between the user and the Koopgoot logical tier, a web application is needed that presents web pages to the user and deals with the user input. A web application can only communicate with users and not with services, so besides the web application, a component is required to communicate with the services. Because the services should not be restricted to one operating system, we have chosen to use a XML web service to communicate between the services and the Koopgoot logical tier. XML web services are discussed in paragraph 5.8.

- The logical tier.
  The logical tier serves the logical tier, interacts with the data tier and communicates with (the logical tier of) Deloitte OnlinePay. Because we did not want to be restricted to one communication channel and we did want to be able to use object oriented references, we have chosen to use .NET Remoting to serve the web application of the presentation tier and to communicate with Deloitte OnlinePay. .NET Remoting is discussed in paragraph 5.6. Because it is relatively easy to put a XML web service on top of another XML web service, we have chosen to use another XML web service for the communication with the XML web service of the presentation tier.
- The data tier.
  To maintain the system's data, a SQL Server database will be used.

After walking through all three tiers we come to the following model of the technical architecture:

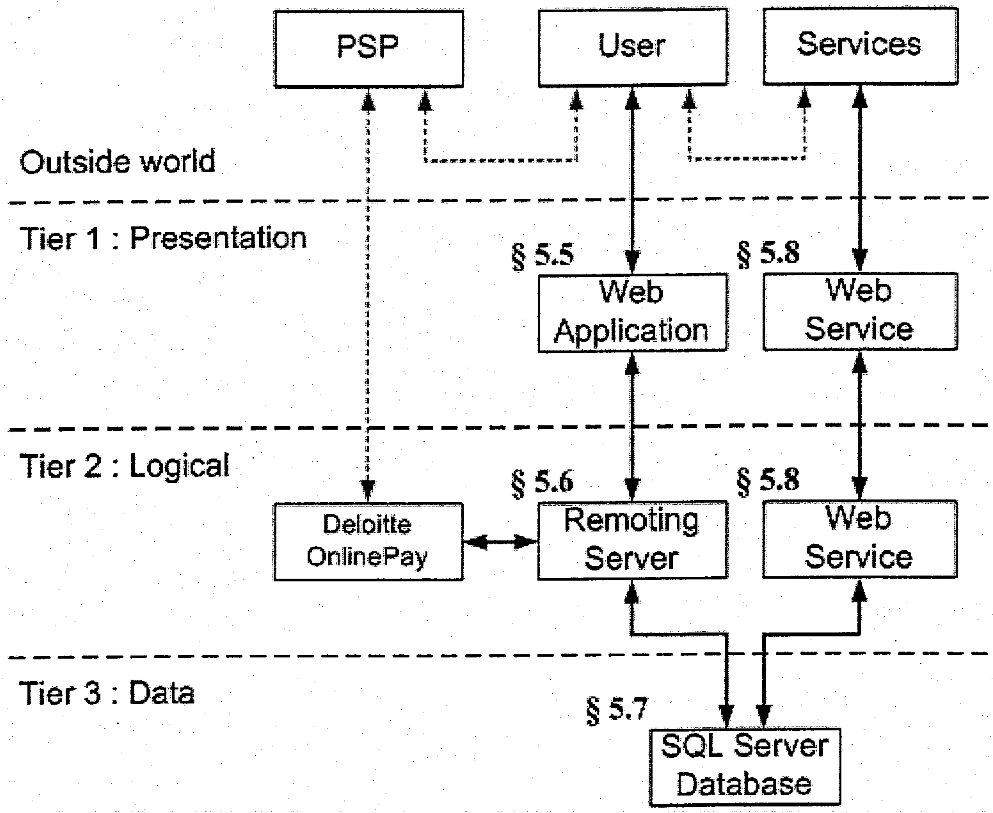


Figure 5.2. Technical architecture of the Koopgoot

The technical components are covered in the paragraphs indicated in Figure 5.2. Before covering the technical components, we will map the logical components presented in chapter four onto the technical architecture that is displayed in Figure 5.2.

5.4 Mapping the logical onto the technical components

Table 5.1. Mapping the normal logical components

	Web Application										Remoting Server								Web Services
Technical Components / Logical Components	Activation.aspx	EditProfile.aspx	IncreaseCredits.aspx	Login.apsx	Logout.aspx	Register.aspx	Resetpassword.aspx	StartService.aspx	Subscribe.aspx		DAAccountRqeuest.cs	DACredit.cs	DAPayment.cs	DAService.cs	DASession.cs	DASubscription.cs	DAUser.cs		ServiceCheck.asmx
Change profile		+															+		
Increase Credits			+									+	+						
Login				+											+		+		
Logout					+										+				
Register	+					+					+	+					+		
Reset password							+										+		
Start service								+				+		+	+	+			
Service check																			+
Subscribe									+			+		+		+			

Table 5.1 shows the mapping of the normal logical components onto the technical components. To keep the table well-organized, we have not shown the SQL database in this table and we have considered both web services as one part. All logical components contain database interaction and most of them interact with more than one table. The Service check is the only logical component which functionality is located at the (XML) Web Services. The Web Application and the Remoting Server are divided into sub-components that relate to the logical components. The sub-components of the Web Application are described in Paragraph 5.5 .

Table 5.2. Mapping the logical management components

	Web Application				Remoting Server						
Technical Components / Logical Components	UserMan.aspx	ServiceMan.aspx	SubscriptionMan.aspx		DAAccountRqeuest.cs	DACredit.cs	DAPayment.cs	DAService.cs	DASession.cs	DASubscription.cs	DAUser.cs
User Management	+					+					+
Service Management		+						+			
Subscription Management			+					+		+	+

In Table 5.2 the logical management components are mapped onto the technical components. For the same reason as before, the SQL database is not shown in the table. The Web Server is also not shown in the table because no management functionality is situated in it. Since the management functionality relates to other sub-components of the Web Application other components are shown than in Table 5.1. Despite some of the sub-components of the Remoting Server do not relate to any management function, we have displayed the same sub-components as in Table 5.1. This is done to present an overview that shows whether the sub-components of the Remoting Server host management functionality or not.

5.5 Web application

The goal of the web application is to present the requested pages to the user, using the input from the user and the remoting server.

5.5.1 Technology

The technology used to realize the web application is the ASP.NET. Typically ASPX pages contain HTML elements, server side codes and client side codes. When a user request an APSX page, the server retrieves it from the disk and then sends it to the ASPX engine for further processing. The ASPX Engine compiles the server side code and generates the page class file. It then instantiates the class file and executes the instructions to develop the response object. During the execution stage, the system follows the programmatic instructions (in the server-side code) to process the data submitted by the user. Finally, the server transmits the response object to the client. In short, the major steps in processing a request for an ASPX page are as follows:

1. The server receives a request for a desired ASPX page.
2. The ASP.NET Engine compiles the page and generates the page class. If the class had already been loaded, it simply provides a thread to the running class instead of regenerating the class. During compilation, it may require other code classes, such as code-behind classes and component classes. These are assembled during this step.
3. The ASP.NET instantiates the class, performs necessary processing, and it generates the Response object.
4. The Web server then sends the Response object to the client.

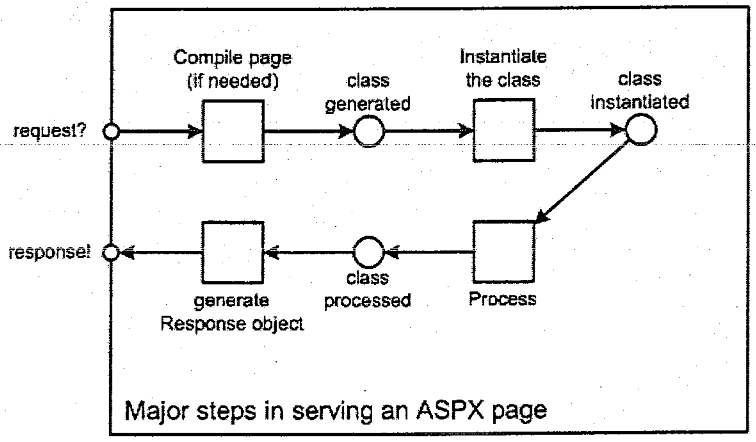


Figure 5.3. ASPX technology

The structure of an ASPX file is similar to HTML pages. Besides normal HTML tags and contents, ASPX files contain ASPX extensions. The beginning of an ASPX file generally contains links to related files, separated between tags starting with "`<%@`" and ending with "`%>`". The rest of the HTML code contain some parts of code which will be executed at the server before the HTML page is send to the user's browser. These parts are also separated between tags, starting with "`<%`" and ending with "`%>`".

The class file that is associated with an ASPX file contains the actions that are performed. A part of these actions is the "`Page_Load`", which describes the actions that are performed before the page is presented to the user. Other parts describe the actions that are executed when a user performs certain actions.

5.5.2 Webpages of the Koopgoot

The Koopgoot exist of 32 normal ASPX pages and 24 administrative ASPX pages. All these ASPX pages have their own class files. Furthermore, there are also 4 ASCX files, which describe custom controls. When certain features like a menu are used at more places its useful to use a custom control, so that this often used code is located at one place. A list of the more important pages and their functions is shown below:

Table 5.3. Koopgoot webpages

Page name	*Functionality*
Activation	The user account is created based on the registration form. After that the user is redirected to *Log in.*
Edit Profile	The user can edit his profile and submit the changes. The user can also start changing his password, his secret question or his e-mail address.
Increase Credits	The user can enter the amount that he wishes to increase his credits with. When this amount is submitted by the user, the user is redirected to the PSP.
Log in	When a user is not logged in he can log in, register, contact or go to a service list. When a user is logged in he can see a list of subscribed services and his amount of credits left. He can directly start a subscribed service, contact go to a service list, go to the profile page or go to the credit increase page. To the last three things will be referred as 'general' things.
Log out	The user is logged out and redirected to *Log in.*
Register	Fill in the registration form and submit it.
Reset Password	When the user is logged in he can reset his password, by entering his current password once, entering his new password twice and confirming the action. When a user is not logged in, he can reset his password, by entering the answer of the secret question, entering his new password twice and confirming the action.
Start Service	If the service is free or the user is subscribed to the service, the service starts immediately in a new window. If the service is not free and a user is not subscribed to it, the costs for single usage of the services and the user's current amount of credits displayed. If the user has enough credits left he can chose

	between paying for usage of the service, after which the service starts in a new window, and canceling the start of the service. If the user's amount of credits is insufficient he is redirected to the *Increase Credits* page on which the amount of credits needed to pay for the service is already filled in.
Subscribe	Here all subscription options to the selected service are presented to the user. The user can select one of those options or cancel subscribing to the service. When the user has selected a subscription option, after which the costs for the selected subscription and the user's current amount of credits displayed. From here the user can subscribe to the service, cancel to subscription or be redirected to the *Increase Credits* page.
User Management	The user can chose between adding a new user, editing a current user or deleting a current user.
Service Management	The user can chose between adding a new service, editing a service or deleting a service.
Subscription Management	The user can chose between adding a new subscription, editing a subscription or deleting a subscription.

5.6 Remoting Server

The goal of the remoting server is to serve the web application, using the database, modifying the database and communicating with the remoting server of Deloitte OnlinePay.

5.6.1 Technology

.NET Remoting is a technology that allows objects to be used remotely over a network. It is comparable it with XML Web Services, which are explained in Paragraph 5.8. However, unlike Web Services, the communication is not cross-platform as both the client and the server must be implemented in .NET. The main advantage of .NET Remoting over XML Web Services is performance. .NET Remoting can be configured to transfer data using a binary format, while XML Web Services are restricted to XML and SOAP communication. In terms of performance, .NET Remoting provides the fastest communication using the TCP channel and the binary formatter.

In .NET remoting, the remote object is implemented in a class that derives from *System.MarshalByRefObject*. The *MarshalByRefObject* class provides the core foundation for enabling remote access of objects across application domains. A remote object is confined to the application domain where it is created. In .NET remoting, a client doesn't call the methods directly; instead a proxy object is used to invoke methods on the remote object. Every public method that we define in the remote object class is available to be called from clients.

The remoting infrastructure allows two distinct types of remote objects to be created:

1. Client-activated objects - A client-activated object is a server-side object whose creation and destruction is controlled by the client application. An instance of the remote object is created when the client calls the *new* operator on the server object.

2. Server-activated objects - A server-activated object's lifetime is managed by the remote server. The server-activated objects are created when the client invokes a method on the proxy. There are two types of server activated objects:
    1. Single call - Single-call objects handle a single request coming from a client. When the client calls a method on a single call object, the object constructs itself, performs the action the method calls for, and destroys the object. No state is held between calls, and each call is called on a new object instance.
    2. Singleton - Singleton objects are stateful objects, meaning that they can be used to retain state across multiple method calls. An instance of a singleton object serves multiple clients, allowing those to share data among themselves.

.NET remoting objects are objects and can be treated as such. As a result, you can use object references to remoted objects.

5.6.2 Implementation

5.6.3 Object oriented class model of the classes at the remoting server

The class model of the remoting server is shown in Figure 5.4. The model correspondents to a large extend to the class diagram design presented in paragraph 4.11. All classes inherit from a new class "DABasis" that contains basic functionality needed.

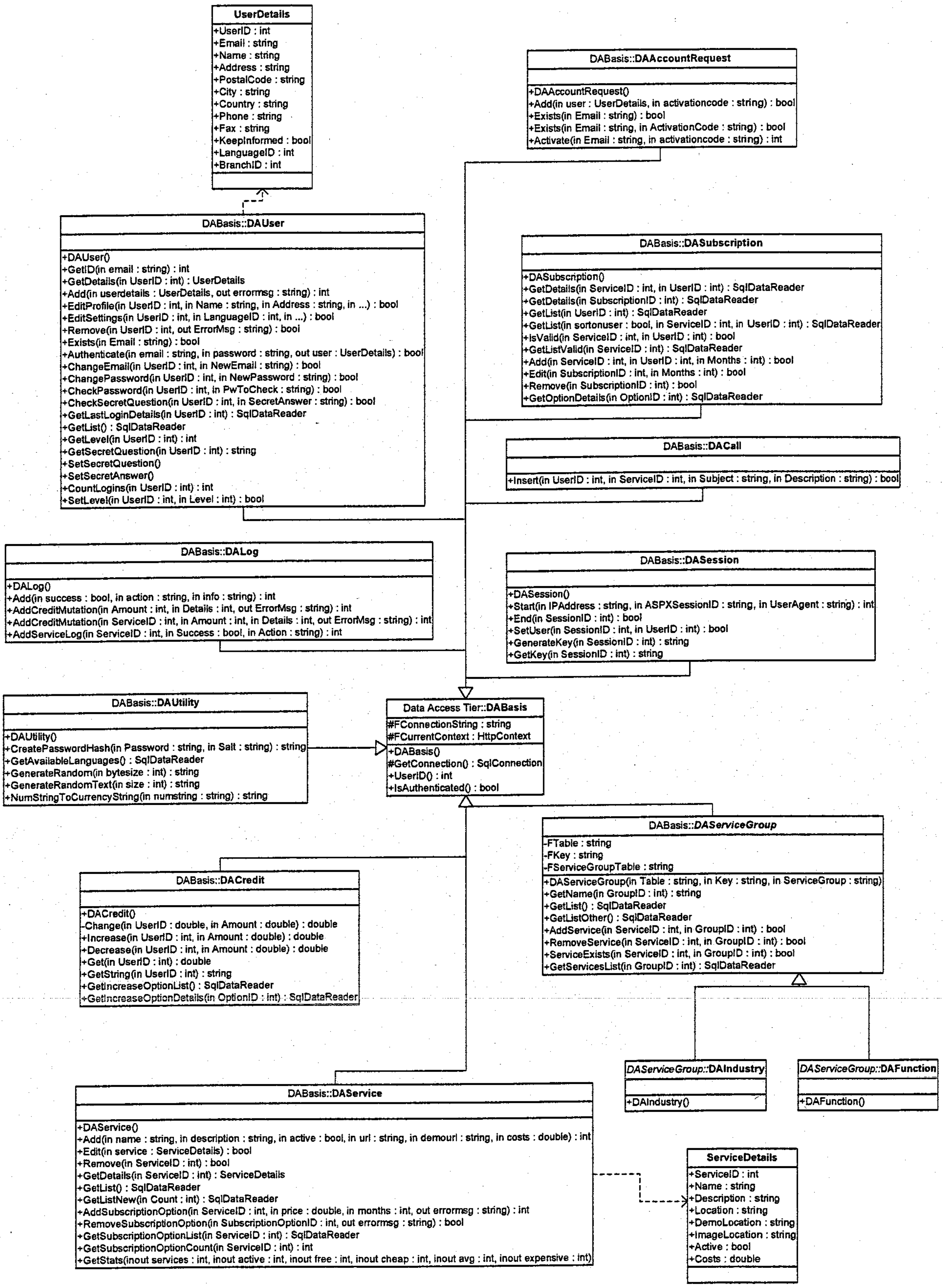

UserDetails
+UserID : int
+Email : string
+Name : string
+Address : string
+PostalCode : string
+City : string
+Country : string
+Phone : string
+Fax : string
+KeepInformed : bool
+LanguageID : int
+BranchID : int
DABasis::DAAccountRequest
+DAAccountRequest()
+Add(in user : UserDetails, in activationcode : string) : bool
+Exists(in Email : string) : bool
+Exists(in Email : string, in ActivationCode : string) : bool
+Activate(in Email : string, in activationcode : string) : int
DABasis::DAUser
+DAUser()
+GetID(in email : string) : int
+GetDetails(in UserID : int) : UserDetails
+Add(in userdetails : UserDetails, out errormsg : string) : int
+EditProfile(in UserID : int, in Name : string, in Address : string, in ...) : bool
+EditSettings(in UserID : int, in LanguageID : int, in ...) : bool
+Remove(in UserID : int, out ErrorMsg : string) : bool
+Exists(in Email : string) : bool
+Authenticate(in email : string, in password : string, out user : UserDetails) : bool
+ChangeEmail(in UserID : int, in NewEmail : string) : bool
+ChangePassword(in UserID : int, in NewPassword : string) : bool
+CheckPassword(in UserID : int, in PwToCheck : string) : bool
+CheckSecretQuestion(in UserID : int, in SecretAnswer : string) : bool
+GetLastLoginDetails(in UserID : int) : SqlDataReader
+GetList() : SqlDataReader
+GetLevel(in UserID : int) : int
+GetSecretQuestion(in UserID : int) : string
+SetSecretQuestion()
+SetSecretAnswer()
+CountLogins(in UserID : int) : int
+SetLevel(in UserID : int, in Level : int) : bool
DABasis::DASubscription
+DASubscription()
+GetDetails(in ServiceID : int, in UserID : int) : SqlDataReader
+GetDetails(in SubscriptionID : int) : SqlDataReader
+GetList(in UserID : int) : SqlDataReader
+GetList(in sortonuser : bool, in ServiceID : int, in UserID : int) : SqlDataReader
+IsValid(in ServiceID : int, in UserID : int) : bool
+GetListValid(in ServiceID : int) : SqlDataReader
+Add(in ServiceID : int, in UserID : int, in Months : int) : bool
+Edit(in SubscriptionID : int, in Months : int) : bool
+Remove(in SubscriptionID : int) : bool
+GetOptionDetails(in OptionID : int) : SqlDataReader
DABasis::DACall
+Insert(in UserID : int, in ServiceID : int, in Subject : string, in Description : string) : bool
DABasis::DALog
+DALog()
+Add(in success : bool, in action : string, in info : string) : int
+AddCreditMutation(in Amount : int, in Details : int, out ErrorMsg : string) : int
+AddCreditMutation(in ServiceID : int, in Amount : int, in Details : int, out ErrorMsg : string) : int
+AddServiceLog(in ServiceID : int, in Success : bool, in Action : string) : int
DABasis::DASession
+DASession()
+Start(in IPAddress : string, in ASPXSessionID : string, in UserAgent : string) : int
+End(in SessionID : int) : bool
+SetUser(in SessionID : int, in UserID : int) : bool
+GenerateKey(in SessionID : int) : string
+GetKey(in SessionID : int) : string
DABasis::DAUtility
+DAUtility()
+CreatePasswordHash(in Password : string, in Salt : string) : string
+GetAvailableLanguages() : SqlDataReader
+GenerateRandom(in bytesize : int) : string
+GenerateRandomText(in size : int) : string
+NumStringToCurrencyString(in numstring : string) : string
Data Access Tier::DABasis
#FConnectionString : string
#FCurrentContext : HttpContext
+DABasis()
#GetConnection() : SqlConnection
+UserID() : int
+IsAuthenticated() : bool
DABasis::DAServiceGroup
-FTable : string
-FKey : string
-FServiceGroupTable : string
+DAServiceGroup(in Table : string, in Key : string, in ServiceGroup : string)
+GetName(in GroupID : int) : string
+GetList() : SqlDataReader
+GetListOther() : SqlDataReader
+AddService(in ServiceID : int, in GroupID : int) : bool
+RemoveService(in ServiceID : int, in GroupID : int) : bool
+ServiceExists(in ServiceID : int, in GroupID : int) : bool
+GetServicesList(in GroupID : int) : SqlDataReader
DABasis::DACredit
+DACredit()
-Change(in UserID : double, in Amount : double) : double
+Increase(in UserID : int, in Amount : double) : double
+Decrease(in UserID : int, in Amount : double) : double
+Get(in UserID : int) : double
+GetString(in UserID : int) : string
+GetIncreaseOptionList() : SqlDataReader
+GetIncreaseOptionDetails(in OptionID : int) : SqlDataReader
DAServiceGroup::DAIndustry
+DAIndustry()
DAServiceGroup::DAFunction
+DAFunction()
DABasis::DAService
+DAService()
+Add(in name : string, in description : string, in active : bool, in url : string, in demourl : string, in costs : double) : int
+Edit(in service : ServiceDetails) : bool
+Remove(in ServiceID : int) : bool
+GetDetails(in ServiceID : int) : ServiceDetails
+GetList() : SqlDataReader
+GetListNew(in Count : int) : SqlDataReader
+AddSubscriptionOption(in ServiceID : int, in price : double, in months : int, out errormsg : string) : int
+RemoveSubscriptionOption(in SubscriptionOptionID : int, out errormsg : string) : bool
+GetSubscriptionOptionList(in ServiceID : int) : SqlDataReader
+GetSubscriptionOptionCount(in ServiceID : int) : int
+GetStats(inout services : int, inout active : int, inout free : int, inout cheap : int, inout avg : int, inout expensive : int)
ServiceDetails
+ServiceID : int
+Name : string
+Description : string
+Location : string
+DemoLocation : string
+ImageLocation : string
+Active : bool
+Costs : double

Figure 5.4. Remoting server class model

5.7 SQL Server

The goal of the SQL Server is to manage all Koopgoot related data and to present this data to the remoting server and the web service. The remoting server and the web service can be seen as users from the database. A database, called KoopgootDB runs on the SQL Server. Besides data tables, the database also exists of views on the database and stored procedures. In this paragraph stored procedures, which are powerful methods that can be executed on a SQL Server database are treated. We will also show the database model of the Koopgoot.

5.7.1 Stored Procedures

Stored Procedures are methods that can be executed on a SQL Server database. By putting pieces of SQL code into a stored procedure, reusable code can be created. When you want to update the code or modify it for use in another application, all you have to do is make the change in one place. That means more maintainable code and less time trying to track down problems. The code in stored procedures is written in SQL Server's dialect of SQL, called Transact SQL (T-SQL). Stored procedures are as flexible as many programming languages thanks to T-SQL's flow control, parameters, and output functionality.

One way to look at stored procedures is to categorize them as either precompiled queries or as parameterized queries. Normal SQL statements that are executed are parsed before SQL Server can execute it. However, by creating a stored procedure, queries will run more quickly because SQL Server determines the execution plan at the time the stored procedure is created and stores that information along with the procedure.

Stored procedures allow you to do much more than store precompiled versions of static queries. Parameters can be added to a stored procedure, which results in more flexibility. Just like the parameters of a method in a normal programming language, the parameters can then be manipulated within the stored procedure.

Besides retrieving data, stored procedures are often used to manipulate data. In fact, in certain organizations, programmers do not have direct access to database tables. Instead, they do all of their interacting with the database through stored procedures. Because the normal update command is not allowed, the chance of programming errors damaging the database decreases.

5.7.2 Example - "Activation" stored procedure

In the Koopgoot, all data manipulation is done by stored procedures. To demonstrate the use of the stored procedures, we have presented an example of a stored procedure that is used in the Koopgoot.

When a new user registers to the Koopgoot he receives an activation mail. The link that activates his account actually calls the stored procedure, called Activation. This stored procedure manipulates four tables. These tables are shown below.

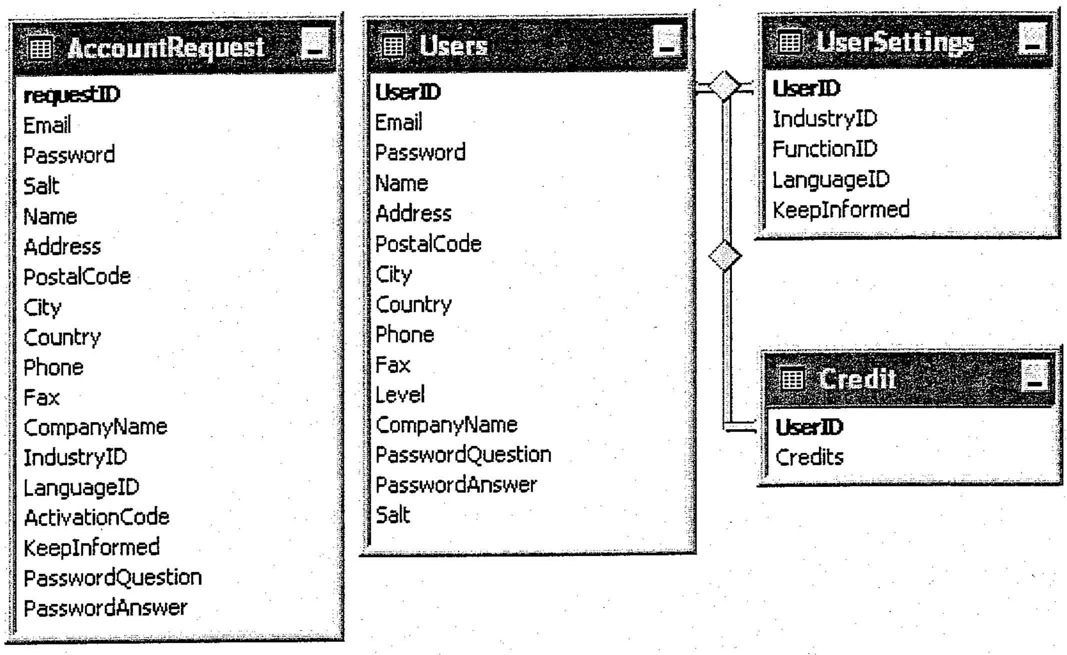


Figure 5.5. The datatables of the Activation stored procedure

This is actually the create script that creates the stored procedure, called Activation. All variables in a stored procedure are indicated by the "@" sign. When the stored procedure is called, two parameters are supplied, the *@Email*, containing the users e-mail address and a corresponding *@Activationcode*. The output parameter is *@userID*, which is a unique number corresponding to the newly created user.

Just like in a normal programming language, two variables are declared: *LanguageID* and *IndustryID*. The next line contains the statement *BEGIN TRANSACTION*. All actions between this line and *TRANSMIT TRANSACTION* will be performed 'in one transaction', meaning that during this transaction no other activities take place at the database. If one action in this transaction results causes an error, all other actions will be undone.

The first SQL query in the transition inserts a record to the *Users* table. The values for the various tables are retrieved from the *Accountrequest* table and the unique *userID* corresponding to the newly created user is placed in the output variable. The following INSERT query inserts a record in the *Credit* table using the newly retieved *@userID*. Subsequently a SELECT query fills the two local variables. These variables are used in the next INSERT query, where a record is inserted in the *UserSettings* table. The last query in the transition deletes the 'accountrequest' from the *AccountRequest* table.

```
CREATE PROCEDURE dbo.Activering

      (
            @ActivationCode nvarchar(255),
            @Email nvarchar(255),
            @userID int OUTPUT
      )

AS
      DECLARE @IndustryID INT
      DECLARE @LanguageID INT

      BEGIN TRANSACTION

      INSERT INTO Users ( [Email], [Password], [Name], Address, PostalCode,
City, Country, Phone, Fax, CompanyName, PasswordQuestion, PasswordAnswer,
Salt)
      (
      SELECT  [Email],  [Password],  [Name],  Address,  PostalCode,  City,
Country, Phone, Fax, CompanyName, PasswordQuestion, PasswordAnswer, Salt
      FROM AccountRequest
      WHERE ( ActivationCode = @ActivationCode ) AND ( [Email] = @Email )
      )
      SET @userID =  SCOPE_IDENTITY()

      INSERT INTO Credit ( UserID, Credits )
      VALUES ( @userID , 0 )

      SELECT @IndustryID = IndustryID, @LanguageID = LanguageID
      FROM AccountRequest
      WHERE ( ActivationCode = @ActivationCode ) AND ( [Email] = @Email )

      INSERT  INTO  UserSettings  (  UserID,  IndustryID,  LanguageID,
KeepInformed )
      VALUES ( @userID , @IndustryID, @LanguageID, 0 )

      DELETE AccountRequest
      WHERE ( ActivationCode = @ActivationCode ) AND ( [Email] = @Email )

      COMMIT TRANSACTION
      RETURN
GO
```

Figure 5.6. The create script of the Activation stored procedure

5.8 Web Services

To enable user validation for the services we have used XML Web Services. Because services contact the web services for user validation purposes, a second goal for the web services is to register these contacting services as started.

5.8.1 Technology

XML Web services can be seen as methods that are offered over the web. XML Web services are built on the common infrastructure of the HTTP protocol and SOAP formatting, which uses XML. These are public standards, and can be used with current Web infrastructures without worrying about additional proxy or firewall issues. XML Web services are accessible using URLs, HTTP, and XML. The advantages of using XML Web services are ease of use, inbuilt security, and its use of standard protocols which allow operation between computers running different operating systems communicating across the internet.

Web services are a stateless programming model, which means each incoming request is handled independently. In addition, each time a client invokes an ASP.NET Web service, a new object is created to service the request. The object is destroyed after the method call completes.

XML Web services are implemented in classes that derive from the *System.Web.Services.WebService* class. The *WebMethod* attribute is used to expose methods as Web services, so they can be invoked by sending HTTP requests using SOAP. The XML Web service receives the SOAP request, executes the method, and sends the results in the form of a SOAP response to the client.

5.8.2 Implementation

To validate their users, services have to communicate to the Web service which is situated at the presentation tier. The file that is called from the services is *check.asmx*. This file contains only one rule:

```
<%@ WebService Language="c#" Codebehind="Check.asmx.cs" Class="ServiceCheck.Check" %>
```

Figure 5.7. The content of check.asmx

As you can see it only contains a link to the code and it defines the programming language used in the code. The code behind the *Check.asmx* file is located in the *Check.asmx.cs* file, which is shown in Figure 5.8.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace ServiceCheck
{
        [WebService(Namespace="KoopgootPublicServiceCheck")]
        public class Check : System.Web.Services.WebService
        {
                public Check()
                {// This call is required by the ASP.NET Web Services Designer
                        InitializeComponent();
                }

                ⊞ #region Component Designer generated code

                [WebMethod]
                public bool SessionIsValid(int SessionID, string SessionKey, string ServiceKey)
                {
                        return   PublicKoopgootFunctions.SessionIsValid(SessionID,   SessionKey,
ServiceKey);
                }

                [WebMethod]
                public   SessionDetails   GetSessionDetails(int   SessionID,   string   SessionKey,
string ServiceKey)
                {
                        return  PublicKoopgootFunctions.GetSessionDetails(SessionID,  SessionKey,
ServiceKey);
                }

        }
}
```

Figure 5.8. The code behind the web service

At first some classes providing basic .NET methods are included by the using statements. Also the *System.Web.Services.WebService* class is included, where all classes that implement web services derive from. There are two methods in this file, which are exposed as web services:

1. *SessionIsValid* - This method only returns whether the user which is visiting the service should be granted access to the service. To invoke this method, three parameters are required, which are an integer called *sessionID*, a string called *SessionKey* and a string called *ServiceKey*. This function only returns a Boolean value to the service indicating whether the user should be granted access to the service or not. To obtain this result another method from another class is envoked.
2. *GetSessionDetails* – This method returns more details about the user that is accessing the service. Off course on of these details states whether the user should be granted access to the service or not. Other details vary from the name of the user to the *IPadress* of the user and the *ASPXSessionID*. These details can for example be used by the services to make present the services more personnel to the user.

The class *PublicKoopgootFunctions*, which contains the methods *GetSessionDetails* and *SessionIsValid* consists of three main parts. One parts describes the class *SessionDetails*. Another part is the method *SessionIsValid*. This method simply invokes the *GetSessionDetails* method and returns is *ValidSession* property, which is a part of the results from the *GetSessionDetails* method. The third part is the *GetSessionDetails* method itself, which actually communicates with the XML Web service of the logical tier, to return the session details.

The XML Web service of the logical tier is partially a copy of the XML Web service of the presentation tier. The main difference is in the *GetSessionDetails* method that is situated in one of the class files of the XML Web service of the logical tier. This method uses the same input and output variables, but the main difference is in retrieving the results. Where the XML Web service of the presentation tier uses the XML Web service of the logical tier to retrieve the results, the XML Web service of the logical tier uses the SQL Server Database to retrieve the results. Apart from retrieving the results the XML Web service of the logical tier also invokes a stored procedure of the SQL Server Database that registers that the user has accessed the service.

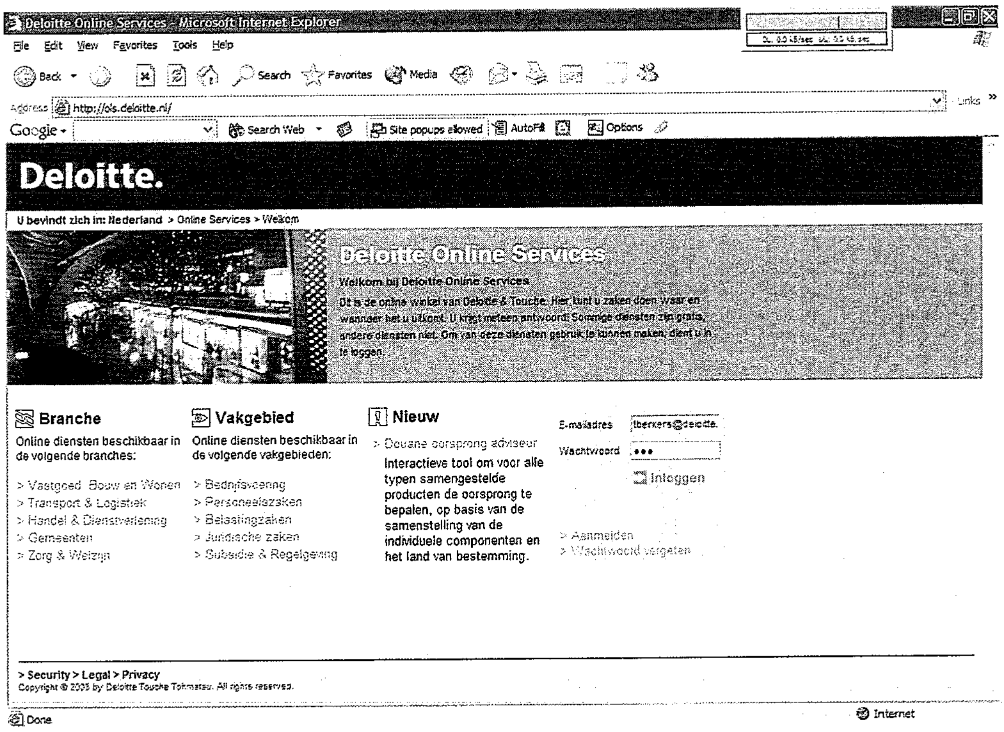
6 Conclusions and Recommendations

6.1 Conclusions and Recommendations

In chapter one we set the follow objectives:

1. The design and implementation of a technical architecture that is needed to realize a new distribution channel of online decision support systems.
2. The design and implementation of one or more decision support systems for this environment.

The first objective has been realized. The technical architecture is in place and it is our belief that all functional requirements set in chapter three, paragraph four and technical requirements set in chapter five paragraph two have been met. In chapter four we have studied one important aspect of the system: soundness. Naturally, soundness is not the only important property of a system but it is an import requirement to place on a system, as explained in chapter four.


Deloitte Online Services - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://ols.deloitte.nl/
Links
Google Search Web Site popups allowed AutoFill Options
Deloitte.

U bevindt zich in: Nederland > Online Services > Welkom

Deloitte Online Services

Welkom bij Deloitte Online Services

Dit is de online winkel van Deloitte & Touche. Hier kunt u zaken doen waar en wanneer het u uitkomt. U krijgt meteen antwoord. Sommige diensten zijn gratis, andere diensten niet. Om van deze diensten gebruik te kunnen maken, dient u in te loggen.

Branche

Online diensten beschikbaar in de volgende branches:

> Vastgoed, Bouw en Wonen
> Transport & Logistiek
> Handel & Dienstverlening
> Gemeenten
> Zorg & Welzijn

Vakgebied

Online diensten beschikbaar in de volgende vakgebieden:

> Bedrijfsvoering
> Personeelszaken
> Belastingzaken
> Juridische zaken
> Subsidie & Regelgeving

Nieuw

> Douane oorsprong adviseur

Interactieve tool om voor alle typen samengestelde producten de oorsprong te bepalen, op basis van de samenstelling van de individuele componenten en het land van bestemming.

E-mailadres: lberkers@deloitte.
Wachtwoord: •••
Inloggen

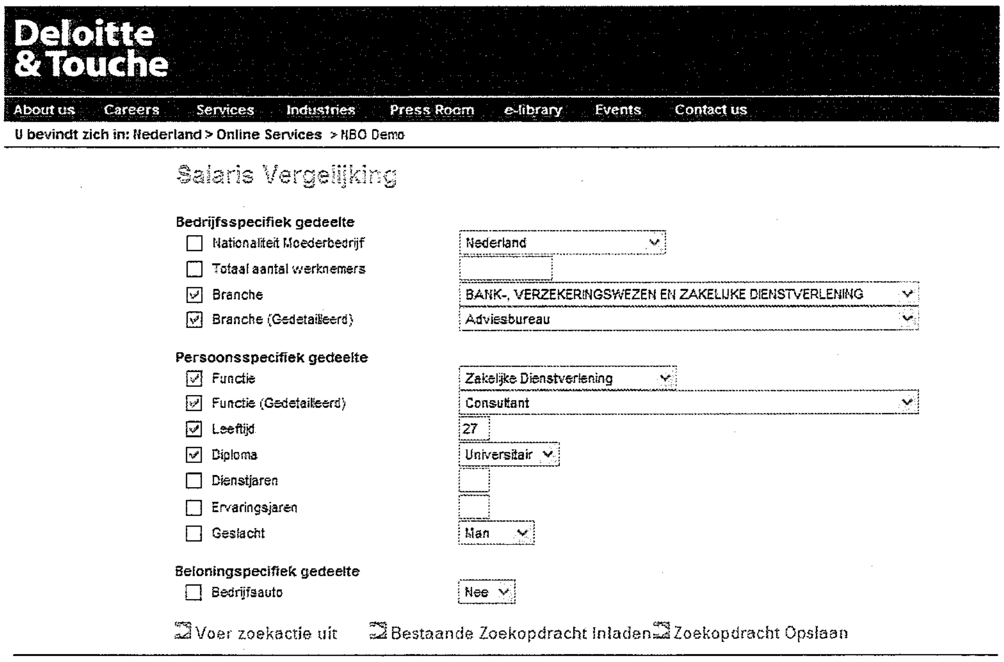
> Aanmelden
> Wachtwoord vergeten

> Security > Legal > Privacy
Copyright © 2005 by Deloitte Touche Tohmatsu. All rights reserved.

Figure 6.1. End result of the Koopgoot - welcome page

The environment is now ready for extensive testing by a representative group of target users. The test can bring potential problems or flaws to the surface, which can be eliminated or added in future versions of the Koopgoot. This would not mean the logical and technical architecture have been inadequate: the development of an information system is an elaborate process that requires recurrent user feedback to improve the system.

The second objective –to design and implement one or more decision support systems according to the techniques described in chapter two- as proof of concept for the environment has also been realized. We have realized an online service that is designed using the technique of case based reasoning as discussed in chapter two, paragraph three. The service provides salary indication and uses a database of job functions that is exclusively available to Deloitte. The user is interested in the details of a specific function and enters all relevant information about it. The system calculates and returns a list of matching functions. The user can choose a specific function from the list and look at it in detail.


Deloitte & Touche
About us Careers Services Industries Press Room e-library Events Contact us

U bevindt zich in: Nederland > Online Services > NBO Demo

Salaris Vergelijking

Bedrijfsspecifiek gedeelte

☐ Nationaliteit Moederbedrijf	Nederland
☐ Totaal aantal werknemers	
☑ Branche	BANK-, VERZEKERINGSWEZEN EN ZAKELIJKE DIENSTVERLENING
☑ Branche (Gedetailleerd)	Adviesbureau

Persoonsspecifiek gedeelte

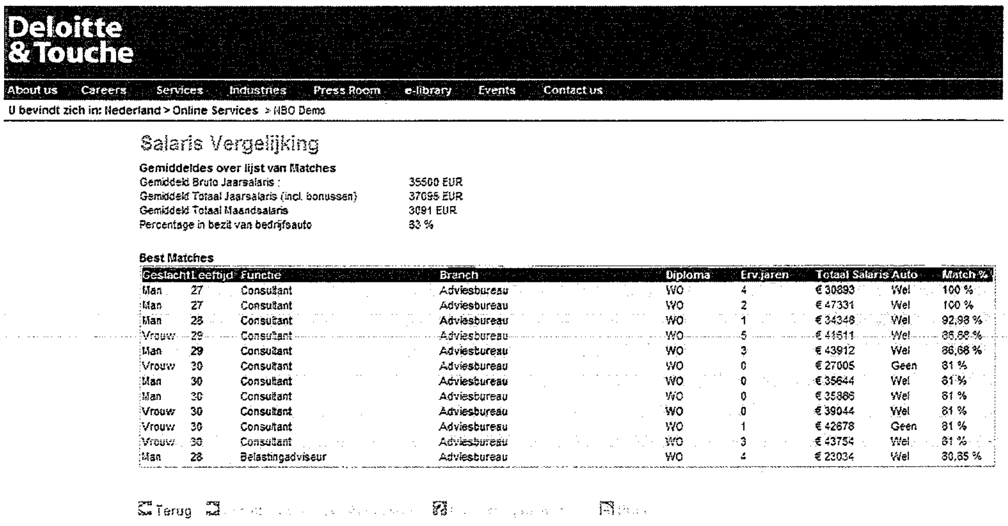
☑ Functie	Zakelijke Dienstverlening
☑ Functie (Gedetailleerd)	Consultant
☑ Leeftijd	27
☑ Diploma	Universitair
☐ Dienstjaren	
☐ Ervaringsjaren	
☐ Geslacht	Man

Beloningsspecifiek gedeelte

☐ Bedrijfsauto	Nee

Voer zoekactie uit Bestaande Zoekopdracht Inladen Zoekopdracht Opslaan

Figure 6.2. Case based reasoning example I


Deloitte & Touche
About us Careers Services Industries Press Room e-library Events Contact us

U bevindt zich in: Nederland > Online Services > NBO Demo

Salaris Vergelijking

Gemiddeldes over lijst van Matches

Gemiddeld Bruto Jaarsalaris :	35500 EUR
Gemiddeld Totaal Jaarsalaris (incl. bonussen)	37095 EUR
Gemiddeld Totaal Maandsalaris	3091 EUR
Percentage in bezit van bedrijfsauto	33 %

Best Matches

Geslacht	Leeftijd	Functie	Branch	Diploma	Erv.jaren	Totaal Salaris	Auto	Match %
Man	27	Consultant	Adviesbureau	WO	4	€ 30893	Wel	100 %
Man	27	Consultant	Adviesbureau	WO	2	€ 47331	Wel	100 %
Man	28	Consultant	Adviesbureau	WO	1	€ 34348	Wel	92,98 %
Vrouw	29	Consultant	Adviesbureau	WO	5	€ 41611	Wel	86,68 %
Man	29	Consultant	Adviesbureau	WO	3	€ 43912	Wel	86,68 %
Vrouw	30	Consultant	Adviesbureau	WO	0	€ 27005	Geen	81 %
Man	30	Consultant	Adviesbureau	WO	0	€ 35644	Wel	81 %
Man	30	Consultant	Adviesbureau	WO	0	€ 35886	Wel	81 %
Vrouw	30	Consultant	Adviesbureau	WO	0	€ 39044	Wel	81 %
Vrouw	30	Consultant	Adviesbureau	WO	1	€ 42678	Geen	81 %
Vrouw	30	Consultant	Adviesbureau	WO	3	€ 43754	Wel	81 %
Man	28	Belastingadviseur	Adviesbureau	WO	4	€ 23034	Wel	80,85 %

Terug

Figure 6.3. Case based reasoning example II

Besides the case based reasoning tool, a number of other services is already active in the Koopgoot:

Table 6.1. Some currently active services in the Koopgoot and their technology

Technology	*Service*
benchmarking tools	generic benchmarking tool, used for: local authority benchmark, taxi benchmark, consumer business benchmark among others
case based reasoning tools	salary indication tool, tool can be adapted for use with other databases
queuing models	performance analysis for business processes
decision trees	various Wisdom[4] applications
value calculators	various HRM tools for pension advice, golden handshake calculators among others

The services currently active in the Koopgoot show that:

- the intelligence of traditional services can be re-used for new online decision support systems;
- the environment enables developers to fully develop decision support systems in a short period of time by providing a framework that lays down service functionality and criteria as described in chapter two and three.

[4] Wisdom™ is a software tool to develop and consult knowledge based systems, developed by Deloitte.

Literature

[AD1998]	Albert A. Angehrn and Soumitra Dutta, Case-Base Decision Support. CACM 41(5), 1998, p157-165
[AHT2002]	Aalst, W. van der, Hee, K. van, Toorn, R. van der: Component-based Software Architectures: a Framework Based on Inheritance of Behavior. Science of Computer Programming, Vol. 42 (2002) 129–171
[CHS2003]	Chaudron, M., Hee, K. van, Use Cases as Workflows. Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings. p.88-103
[Dijk2002]	Dijk, A. van, Online Services based on Waiting Line Models, Deloitte & Touche, first draft
[Eckerson1995]	Eckerson, Wayne W. "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." Open Information Systems 10, 1 (January 1995): 3(20)
[EGSMW1998]	Eppen, Gary D., Gould, F.J., Schmidt, Charles P., Moore, Jeffrey H., Weatherford, Larry R., "Introductory management science", Queuing, p.573-604
[GV2003]	Girault, C., Valk, R., Petri Nets for Systems Engineering – A Guide to Modeling, Verification and Applications, Springer, 2003, chapters 9-11, 25.
[Hee1994]	Hee, K. van: Information Systems Engineering: A Formal Approach. Cambridge University Press (1994)
[HSVW2003]	Hee, K. van, Sidorova, N., Voorhoeve, M., Woude, J. van der, Architecture of Information Systems using the theory of Petri nets, Lecure notes 2M310, Department of Computing Science, Technische Universiteit Eindhoven, 2003
[HW1996]	Holsapple, C.W., Whinston, A.B., Decision Support Systems: A knowledge-based approach, ITP New York, 1996
[IEEE1990]	Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990.
[Kleinrock1975]	Kleinrock, Leonard, Queueing systems, Wiley-Interscience, London, 1975
[Kolodner1991]	Kolodner, J.L. Improving human decision making through case-based decision aiding. AI Mag. (Summer 1991), p52-68
[Kolodner1993]	Kolodner, J.L. Case-based Reasoning. Morgan Kaufman, 1993
[MFG2003]	Mora, M., Forgionne, G.A., Gupta J.N.D., Decision Making Support Systems – Achievements and Challenges for the New Decade, IGP, 2003
[Mors1993]	Mors, N.P.M., Beslissingstabellen, Lansa, 1993, Chapter 1.
[SKS1997]	Silberschatz, A., Korth, H.F., Sudarshan, S., Database System Concepts, McGraw-Hill International Editions, 1997, chapter 1-3
[Wets1998]	Wets, G., Decision tables in knowledge-based systems : adding knowledge discovery and fuzzy concepts to the decision table formalism, 1998, Thesis TU Eindhoven, p.1-2, 46-47

Appendix

A. Some definitions and Lemma's for Petri nets used

Definitions and lemma's taken from [HSVW2003]:

Definition A.1 (path). *Let $N = \langle P, T, F\rangle$ be a Petri net, and let $n_1, n_k \in$* (P ∪ T). *An* undirected path *C from a node n_1 to a node n_k, is a sequence $\langle n_1, n_2,\ldots, n_k\rangle$, where $n_j \in (P \cup T)$, for $j = 1,\ldots, k$, such that for every i with $1 \leq i \leq k$, we have either $(n_i, n_{i+1}) \in F$ or $(n_i+1, n_i) \in F$. The path is* directed *if $(n_i, n_{i+1}) \in F$ for all suitable i.*

Definition A.2 (state machine) . *Let $N = \langle P, T, F\rangle$ be a Petri net. N is a* state machine *(SM) iff*

$$\forall t \in T : \left|{}^{\bullet}t\right| \leq 1 \wedge \left|t^{\bullet}\right| \leq 1.$$

State machines are equivalent to finite automata.

Definition A.3 (workflow net). *A Petri net is a* WF-net (Workflow net) *if and only if:*

- *N has two special places: i and f. Place i is an initial place: ${}^{\bullet}i = \varnothing$ and f is a final place: $f^{\bullet} = \varnothing$.*
- *If we add a* closing transition *t to N that connects place f with i (i.e., ${}^{\bullet}i = \{f\}$ and $t^{\bullet} = \{i\}$), then the resulting Petri net is strongly connected.*

Definition A.4 (soundness). *A WF net is k-*sound *iff for every marking M reachable from marking i^k, there exists a firing sequence leading from marking M to marking f^k. Formally:*

$$\forall M : (i^k \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} f^k)$$

A WF-net is sound *iff for every natural k, it is k-sound.*

Definition A.5 (SMWF). *N is a State Machine Workflow net (SMWF) iff N is a Workflow net and a state machine.*

Lemma A.1. *Any SMWF is a sound workflow net.*

Proof. For the proof of this lemma we refer to [HSVW2003, p.12].