*Hasso Plattner Institute*
*for Software Systems Engineering*

*Final year bachelor project*
*WS 2005/2006*

*Semantic SOA – Realization of the*
***Adaptive Services Grid***

# Dynamic Supply Chain Scenario for Internet Service Providers

February 28, 2006

Bastian Steinert, Jan Möller, Philipp Sommer,
Sebastian Steinhauer, Stefan Hüttenrauch,
Tobias Queck, Torsten Hahmann

# Tables of Contents

# List of Figures

## List of Listings

# 1 INTRODUCTION

Today's European telecommunication industry is increasingly competitive with many new entrants to the market and a challenging regulatory environment. Along with the ongoing recovery from the technology boom-and-bust, these factors add up to a tough business environment. Price erosion means that providers and operators have realized that they must radically transform the way they do business in order to reduce costs and remain competitive. At the same time, a number of new challenges are emerging, including product innovation, aggressive new market entrants, and the blurring of the boundary between IT and traditional telecommunications. Companies are seeking to grow new business while defending traditional core revenues. The industry suffers from high manpower costs due to a lack of automation, poor time-to-market due to inflexible business processes and poor customer service due to a lack of integrated support systems.

Thus the industry is seeking urgently to reduce IT costs, more than 35% of which are attributable to integration[1]. Furthermore, there is a focus on faster time to market via more flexible business processes and services and a need to reconfigure system components quickly and efficiently in order to satisfy market needs and to provide fully integrated support systems for increasingly sophisticated services.

On the other hand, customers are demanding integrated services, tailored to their specific needs. The market is becoming increasingly federated due both to regulatory pressures and to companies' attempts to catch market opportunities with tailored, bundled services. In this market, the number of Business-to-Business (B2B) relationships between telcos, internet service providers (ISP) and special content and service providers has dramatically increased.

All these factors have led many telcos and ISPs to radically rethink the way they operate. They have realized that the new environment requires tighter yet more flexible management of processes and services.

In this document we present the work in developing a B2B service framework for automated reselling of ISP products based on the "Adaptive Services Grid" (ASG)[2] platform. First we motivate for the B2B solution for the ISP from a business perspective and define a set of requirements, both business and technical requirements. In chapter 3 we switch to a more technical point of view, define a concrete set of Atomic Services and demonstrate their use in exemplary compositions. The main focus of this document lies on a detailed description of the *dynamic supply chain scenario for internet service providers* including technical information of implemented services for scenario realization. Therefore chapter 4 defines the common ontology and the resulting data-types. Finally all services we implemented are described with a formal semantic description, technical details and remarks on implementation.

1 Gartner Group, 2004

2 http://asg-platform.org, funded under the 6FP, EC Contract No. 004617

# 2 Business Perspective

The scenario we present here is an example based on the business-to-business (B2B) wholesale model of an Internet Service Provider (ISP). In our study the ISP specializes on products like domain registration and web hosting, not on providing internet access. To understand requirements for a sophisticated B2B solution a look at the existing Business-to-Customer application can be of great avail. By analyzing the current B2C web shop[1] and its underlying provisioning system, basic functionality required for the B2B model can be identified.

## 2.1 B2C Solution

The present B2C solution based on a web shop allows ordering of domains, emails, and web space. The ISP must provide functionality for domain registration, operating & maintaining DNS information, web hosting configuration, and payment bundled to end-customer products. Selection of domain registration interfaces depends on the specific top-level domain. Assignment of domains with .com or .org endings is governed by ICANN while e.g. national domains are assigned by DENIC in Germany or NICAT in Austria. Registrars accredited by the supervising organizations can register subordinate domains. Web hosting services encapsulate interfaces for web hosting systems. They allow allocation of web space to users while enforcing fine-grained restrictions on data volume, traffic and email configuration.

The goal of developing a B2B solution is the reuse of already available elementary business capabilities. The vision of an enlarged market drives the ISP to shift the existing end-customer-centric application to a more flexible platform that can be used through various front-end solutions operated by resellers. Moreover, service reuse in a flexible environment reduces customer acquisition costs and support costs for the ISP. The underlying internet service provisioning system requires extensions to support a more generalized Business-to-Business approach instead of a restricted B2C application. Currently the complexly interweaved subtasks of product *ordering* in the web shop and *provisioning* of these products through a backend system hinders the reuse of provisioning capabilities through varying order processes.


*Figure 1: B2C Solution*

## 2.2 B2B Solution

Market research shows that there is already a sizable demand for combining domain registration and web hosting services. Especially the association with peregrine products is an interesting market with growth potential. It is anticipated that B2B customers act as resellers that integrate added-value web hosting services (domains, web space, emails) into their existing product portfolio (newspaper subscriptions, broadband internet, community

1 http://www.chillydomains.com

portals). As part of its product bundles, resellers select services offered by the ISP as free add-ons, for bonus programs or as additional features at attractive prices. For example a reseller may order for its customer web space at a special rate or provide them with a domain of their choice when they decide to sign up for a long-term internet access contract. Service provisioning must thus be highly flexible. For the convenience of the resellers, payment services shall be offered as well. Resellers without own billing system or not wanting to deal with payment chooses payment options from the ISP's service pool.

## 2.3 Business Requirements

The wide range of potential reseller necessitates the development of a solution independent of resellers' order processes and its products. The time-consuming definition of static processes should be avoided. In order to instantly add new resellers that can profit from services offered by the ISP, reseller integration costs must be reduced to a minimum. If each joining reseller requires high investments in order to deliver customized products, the costs would probably be covered not adequately by expected revenues.

Generally three kinds of flexibility can be identified as necessary for implementing the described business model:

(BR1) an interface allowing fast and cost-efficient integration of resellers with various background and diverse products; resellers want to offer internet services without massive changes to their own application

(BR2) resellers and/or their customers want to customize products requiring a flexible product management and product composition; minimizing the effort required to define/redefine product bundles

(BR3) a flexible extension mechanism to offer new elementary services quickly to all resellers without having to manually change processes

All three types of flexibility together permit reduced time-to-market – essential for business success. Finally business success depends heavily on the ability to ensure maximum availability and in the consideration of quality of services expressed as non-functional requirements that serve the ISP's and reseller's goals best.

Most problematic is the fact that there is no point in time where all possible processes can be designed beforehand. The number of possible product combinations is exhaustive; static predefined processes cannot cover all combinations within reasonable costs. Dependencies between elementary services will lead to exponentially growing efforts. We derived following technical requirements:

(TR1) separation of order processes from supply chain processes

(TR2) on-demand composition of supply chain processes to cover all possible product combinations and integrate new services automatically

(TR3) adaptive processes for higher availability and consideration of new services at run-time

## 2.4 Advantages of ASG

Service-oriented computing is a paradigm trying to solve the business requirements BR1 and BR3. By providing high-level interfaces that abstract from concrete operations service-oriented architectures aim to loosen coupling (compare to TR1) between components on a

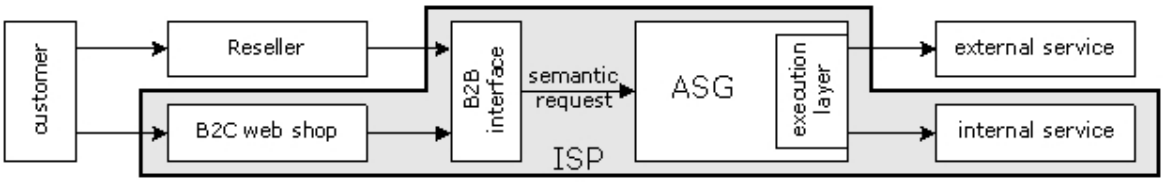


*Figure 2: B2B Solution*

service provider's part from those on the service consumer's side. Services in Service Oriented Architecture (SOA) should be reusable and replaceable; it focuses on the scalability for "Internet-scale provisioning and use of services and the requirement to reduce costs in organization to organization cooperation" [1]. The service consumer must compose services in her applications manually. The reference model for SOA does not demand semantics allowing automated service composition [1]. But SOA is only a first step towards rapid application development [2], other research projects like the Web Service Execution Environment (WSMX)[1] tackle the questions of dynamic selection and semantic web services composition and invocation using ontology mediation.

Formal semantic specifications of services enable the ASG platform to compose supply chain processes. Arbitrary products of the ISP defined by the reseller are expressed through goals in the context of ASG. These goals are used by the platform to compose provisioning processes on-demand. It is for this reason possible to integrate new services dynamically. This resolves TR2 through a very flexible mechanism and is also highly adaptable to changes in the service landscape.

Higher availability is achieved by the renegotiation and replanning feature of ASG. During enactment a monitor detects failures, e.g. service unavailability or violations of service level agreements contracted with atomic services. In such cases, renegotiation of the elementary service with equivalent capability is triggered. If no alternative service fulfill demanded requirements, re-planning takes place in order to adapt the initially planned process to service availability.

## 3 Technical Perspective

In the previous chapters we introduced a business view on the dynamic supply chain for ISPs. We described the business requirements and explained the advantages of ASG. In the following part we will change the point of view to a more technical perspective.

In general the ASG development process consider two major roles. On the one hand the application provider implements an application, consisting of graphical user interface and a set of request for ASG. On the other hand the service provider offers its functionality through Atomic Services. Due to our business scenario the ISP acts in both roles, because it uses ASG as an internal platform for provisioning of its products (see figure 3). The service pool of the ISP mainly contains wrappers to functionality from external service providers, but also Atomic Services with own functionality. The reseller develops its own end customer application, which defines order processes and uses an API of the ISP to create request for the ASG platform.

1 www.wsmx.org

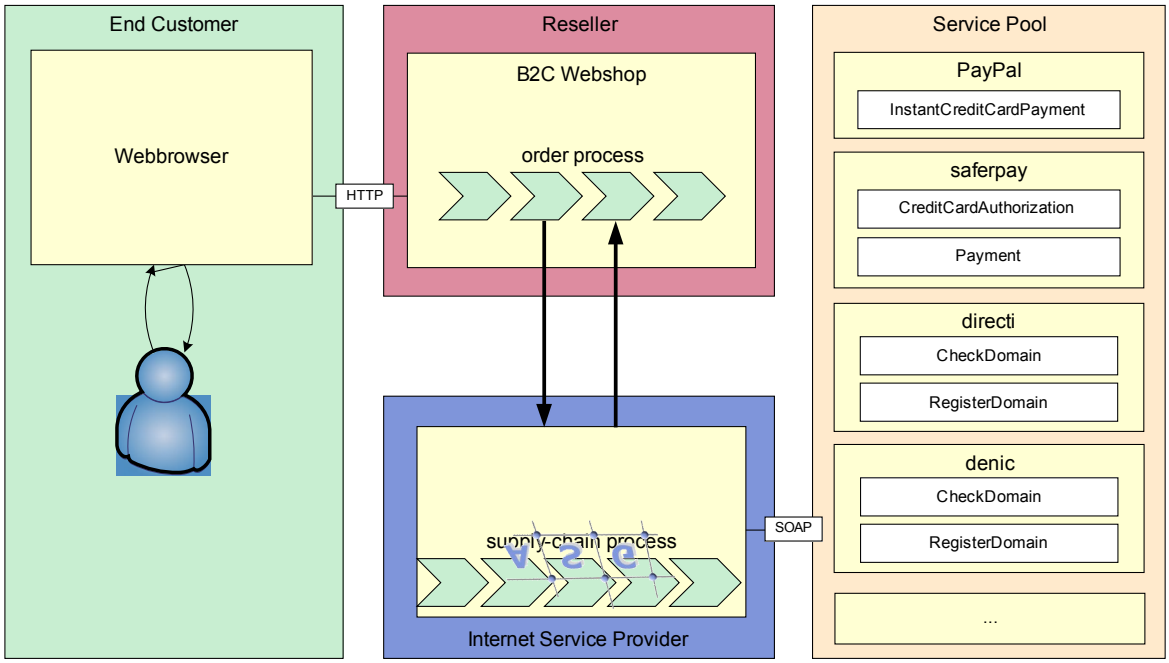


*Figure 3: Stakeholder of the scenario*

## 3.1 Service Landscape

According to the methodology for developing applications based on ASG described in [3] a service landscape can be of great help throughout the whole development process. It evolves over time, beginning with rudimentary capability descriptions. A consistent service landscape enhances knowledge transfer between participants – as it collects knowledge from both domain experts and service engineers. In particular, it proved helpful for the tasks of ontology definition and semantic service definition.

The first step towards a complete service landscape is the identification of services. Most important for a service is the balance between highly reusable functionality and strongly decoupled software components containing inseparable logic. Granularity and composability of individual services must be evaluated according to specific business requirements. The following sections will explain services of three major categories for the dynamic supply chain scenario.

### 3.1.1 Domain Services

In our scenario a category of atomic services are the domain services. There exist different service providers dependant on certain top level domains. The services can be subdivided into three major categories. The group of check services is used to determine whether a particular domain is available. A domain can than be registered using the register services. To complete a domain registration name server services are used to update certain name servers with the name-IP mapping. The input data of the atomic services for registration evolved through analysis of the interfaces provided by domain registrars like Denic or Directi and will be refined in chapter 4 and 5.

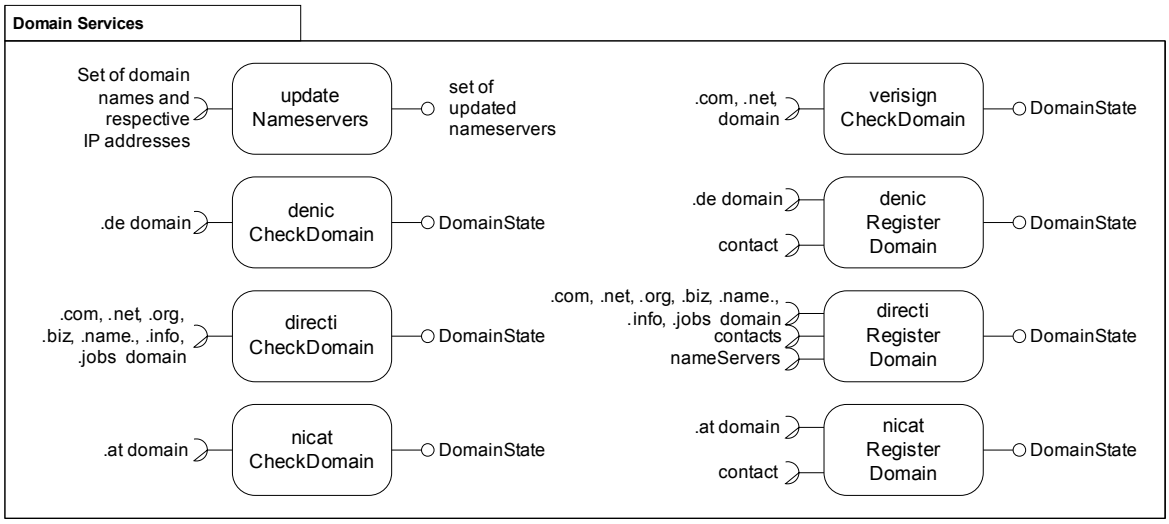


*Figure 4: Service Identification for Domain Services*

### 3.1.2 Payment Services

For dynamic supply chains a essential part is the billing. Therefore we defined a set of services which provide basic functionality to charge common credit cards. Exemplary Paypal and Saferpay are used. Figure 5 shows the difference between these two providers. Saferpay uses two steps to charge a credit card. First it authorizes an amount of money on the credit card, and in a second step a transaction handle is used to finalize the payment. The authorization services can be reused for electronic debiting. Payment with the Paypal-API is done in a single step. Analysing the interfaces of these external service providers in detail, we recognised that Paypal only supports US dollars as a currency. A separate service to covert from other currencies like Euro can be provided. In general payment services are applicable for every scenario where payment is mandatory.

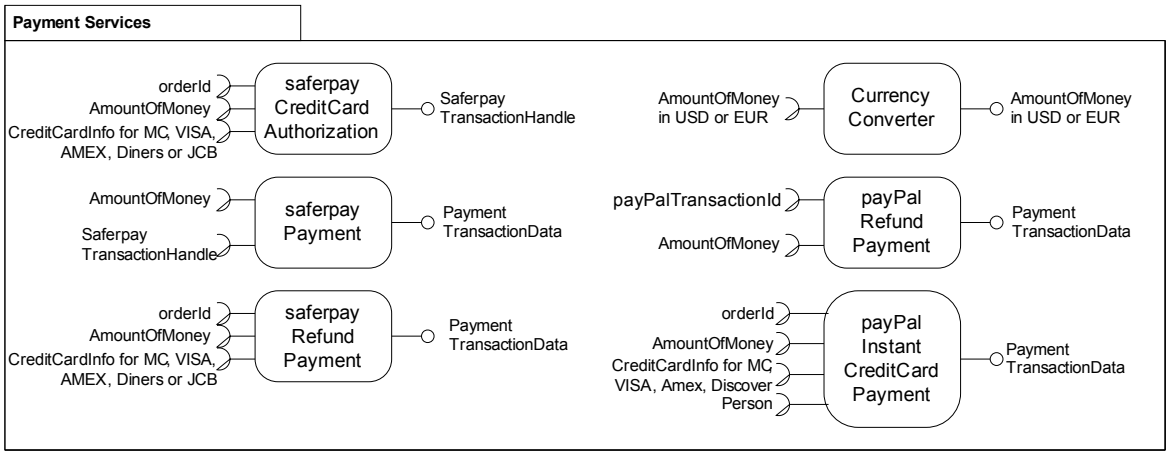


*Figure 5: Service Identification for Payment Services*

### 3.1.3 Web Hosting Services

The last category are the web hosting services. In our scenario these services are provided by the ISP itself. A web hosting account has several features, e.g. world-wide-web, e-mail, ftp, databases, and so forth. Plesk is a server management software offering a comfortable XML-API to administrate and configure server-side software (Apache, Postfix, mySQL). Figure 6 shows a minimum set of services to provide basic web hosting functionality.

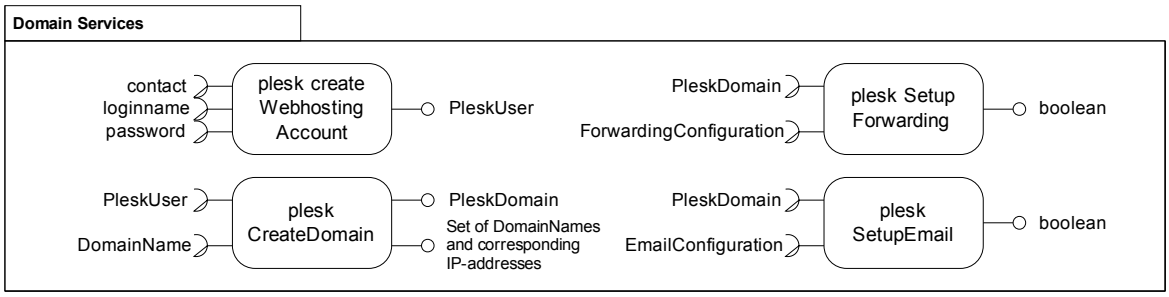


*Figure 6: Service Identification for Web Hosting Services*

## 3.2 Exemplary Service Compositions

An crucial requirement of the dynamic supply chain scenario is the possibility, that the reseller can define arbitrary products and the ASG platform composes appropriate supply chain processes. While developing an application based on ASG, manually created exemplary compositions proved to be very helpful to determine dependencies between single Atomic Services. These dependencies are explained in chapter 5 in detail. Here we just want to clarify the requested tasks and how the composition component of ASG plans possible processes. Further we show simple examples how the features renegotiation and replanning can increase availability.

A request to ASG may looks like: "Register domain lehmann.de for our customer Max Lehmann and provide him with 100MB web space. He wants to pay with his Discover Credit Card". The request contains two main parts specified in a semantic language - Flora in case of the current ASG prototype. The initial state is the customer, his contact details and credit card information. The requested goal state contains the registered domain, provided web hosting and a billed credit card. Figure 7 shows how the Atomic Services defined in the previous section can be combined.

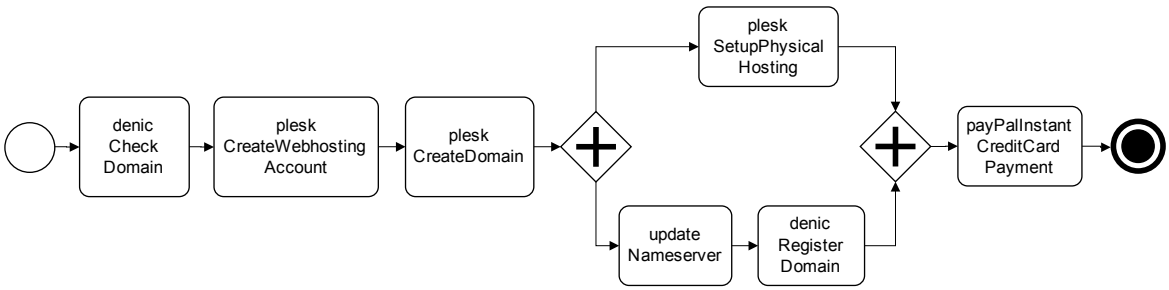


*Figure 7: Exemplary composition*

Alternatively another reseller can define a request just to register a domain and a forwarding of the domain to an already existing web space. In this case the initial state only consists of contact details. The goal state is a registered domain with forwarding. Figure 8 shows the generated process and how renegotiation can compensate a failure while executing the process in the enactment component. The failed service *directiCheckDomain* can be replaced by a service with equivalent capabilities. In our case *verisignCheckDomain*.

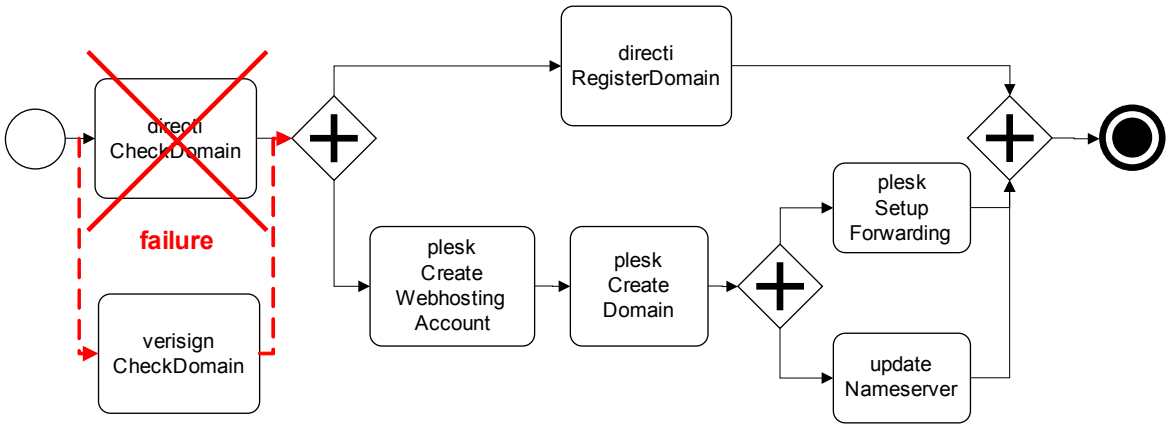


*Figure 8: Exemplary composition with renegotiation*

Payment is a good example for accomplishing dependencies between services. There are two points of view about paying for a product. The customer can pay before or after the supply chain process execution. Hence, the ISP wants to let the reseller choose the time of payment. Thus this dependency cannot be expressed in semantic service specification. A possible solution is to decouple the payment process from the supply chain process.

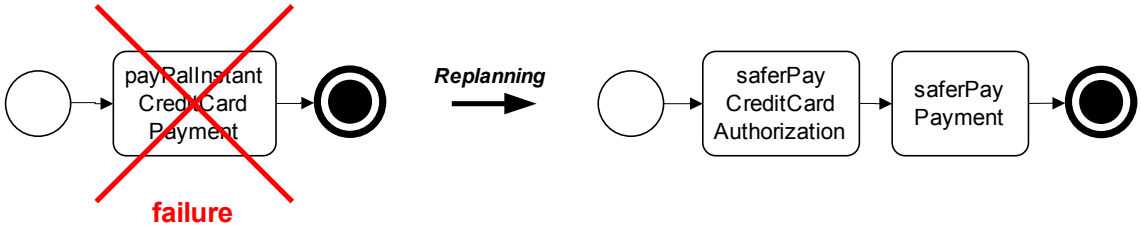


*Figure 9: Exemplary composition with replanning*

Figure 9 shows an example for such a decoupled process. Furthermore in case of unavailability of one payment service replanning is triggered because there is no other single Atomic Service with the same capability.

## 4 Scenario Ontology

For semantic service specification a scenario ontology is required. This ontology can be derived from the domain model containing data types used by the interface definition of Atomic Services. The types are represented by concepts in the domain ontology, developed specifically for the purpose of the dynamic supply chain scenario. In addition the ontology defines a full description of the relations between types. A Flora representation of the domain ontology used for the scenario including all types and relations can be found in the appendix.

## 4.1 Ontology Concepts

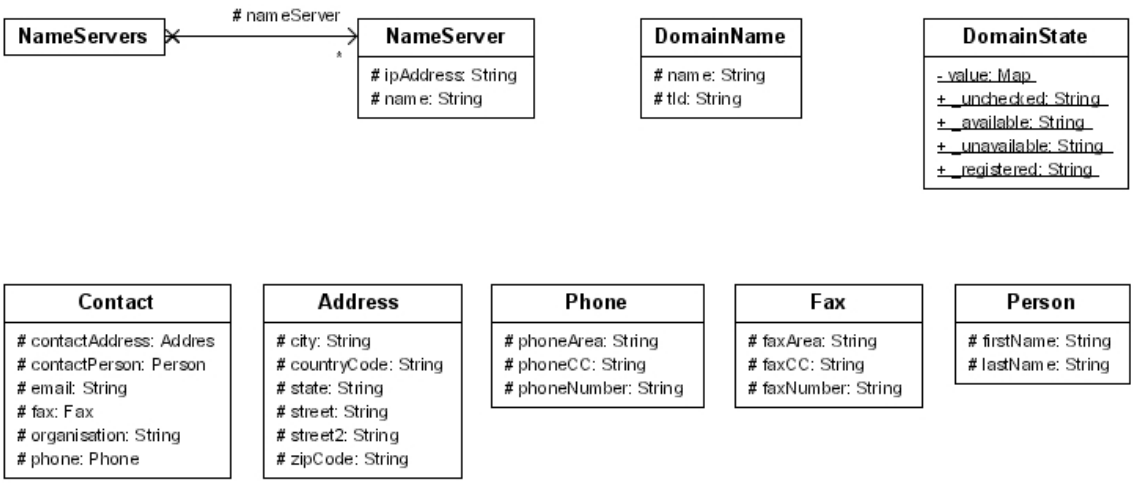


*Figure 10: Concepts for semantic service specifications of Check- and Register Domain Atomic Services*

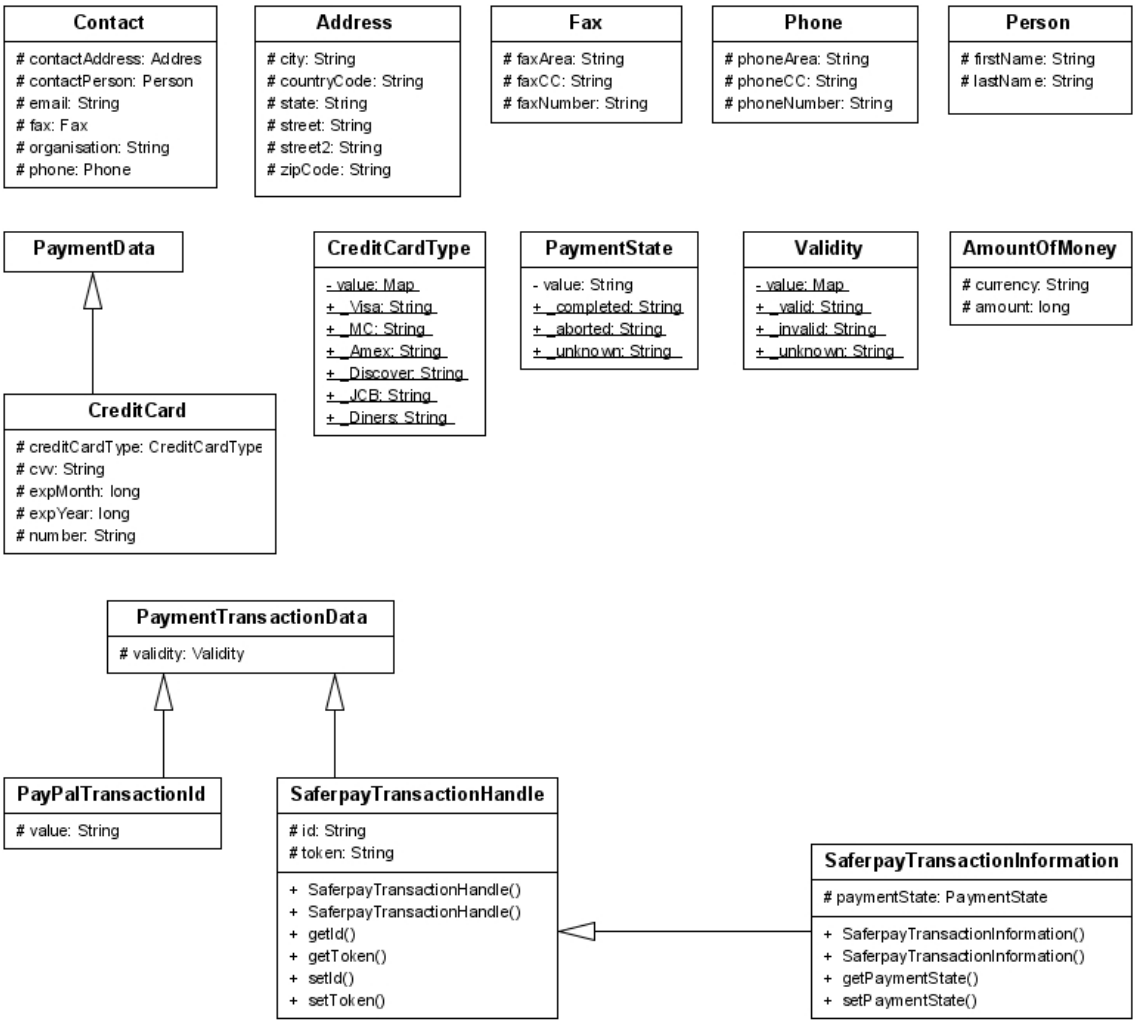


*Figure 11: Concepts for semantic service specifications of Payment Atomic Services*

## 4.2 Relations between ontology concepts

The ontology for the domain of internet service supply chains is more complex than the domain model. In the domain model as seen above only data types are defined. It expresses composition, aggregation, as well as inheritance relations between types. Relations defining semantic associations between types are not represented. Therefore the ontologies contain additional descriptions, specifically for giving services and their results a meaning. For these purposes the following set of relations between ontology concepts has been defined in Flora syntax that is used for semantic description of the atomic services.

```
hasDomainState(domainName, domainState):relation.
```

Attaches an domainState to a domainName object, that indicates the state (“unchecked”, “available”, “unavailable”, “registered”). Domain check services delivers results “available” or “unavailable”, the value of “registered” should be set in a domainState of a successfully registered domainName. Each domainName can be associated only one domainState through this relation.

```
domainNameservers(domainName, nameServers):relation.
```

States that a certain domainName is registered with the given nameServers.

```
creditCardOwner(creditCard, contact):relation.
```

States that a person described through the contact is the legal owner of a creditCard. Each creditCard instance can have only one contact.

```
paymentTransaction(paymentData, paymentTransactionData):relation.
```

The paymentData has been charged through a transaction expressed as paymentTransactionData. Each paymentTransactionData must be associated to exactly one paymentData. However, each paymentData can relate to more than one paymentTransactionData in a sense that a paymentData (i.e. a credit card) has been used for different payments.

```
amountCharged(amountOfMoney, paymentTransactionData):relation.
```

States that the handle expressed as paymentTransactionData has charged an amount of amountOfMoney. Each paymentTransactionData must be associated exactly to one amountOfMoney that has been charged.

The relations paymentTransaction and amountCharged are usually used by payment services in a combination to express a ternary relation between an amountOfMoney, a paymentTransactionData, and a paymentData.

```
authorizedSaferpayAmount(saferpayTransactionHandle, amountOfMoney):relation.
```

The relation is specifically designed for saferpays atomic services. Since the services provided by saferpay split the payment process into an authorization and an actual payment, the payment must know the amountOfMoney a previous authorization has allowed. Each saferpayTransactionHandle must therefore associate exactly one amountOfMoney object.

# 5 Atomic Services

This chapter contains a detailed description of the atomic services, described in section 3.1, that have so far been fully implemented for the *dynamic supply chain scenario for internet service providers*.

Each service description is divided into three parts: a rather informal description of the service functionality with its input- and output-parameters as well as pre- and post-

conditions in a straight forward sense. This short semantic description contains additionally a use case diagram specifying the service by high-level means. The description is intended to provide an overview of the service and how a domain expert without in-depth technical knowledge would describe the desired functionality. Especially the preconditions and effects are far from being sufficient for semantic specifications and are completed in the second part. The second part addresses the formal semantic service specifications in Flora, based on the common ASG ontology. The last part is devoted to the technical service implementation. Its purpose is the description of the actual service interfaces, its implementation (presented as class diagram), and its interaction with external service providers. This chapter optionally contains named "remarks and problems" addressing specific problems of the service implementation.

## 5.1 Directi Check Domain - characteristics

### 5.1.1 Short semantic description

**Purpose:** This service checks via Directi whether a certain domain is available for registration.

- **Parameters/Conditions:**
  - **Input:** DomainName domain
  - **Output:** DomainState
  - **Core precondition:** unchecked domain name (.com, .net, .org, .biz, .name, .info, .jobs)
  - **Optimistic postcondition:** checked domain name
  - (.com, .net, .org, .biz, .name, .info, .jobs)
  - **Errors and Exceptions:** EJBException, AssertionError
- **Scenario Description:** A customer wants to register the domain myFunnyBunny.com. Certainly this can only work if the domain is not already registered. To check the registrations status the user calls his/her ISP which calls Directis *CheckDomainService* to validate whether lehmann.com is available for registration. The service returns available for domains free for registration and unavailable for already registered domains.

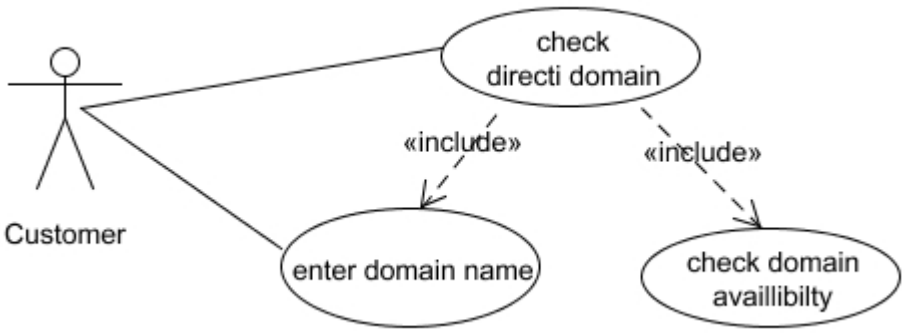


*Figure 12: directi check domain Atomic Service - use case diagram*

### 5.1.2 Formal semantic specification

The formal preconditions contain sets of conditions for each possible top-level domain describing first the optimistic postcondition and second the exceptional postcondition when the domain is not available.

The formal precondition defines the input parameters and additional restrictions to it, here only on the name of the top-level domains. For both cases we have positive effects containing an domainState object that relates to the domainName and has a certain value, which is set to "available" in the optimistic case and to "unavailable" in the exceptional case. The complete formal specification contains pre- and postcondition in a rectified form as well in a string form for discovery purposes.

```
directiCheckDomain:atomicService[
spec -> directiCheckDomainSpec:semanticServiceSpecification[
conditions ->> {
directiCheckComDomainCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"com":string)}
                :reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""com"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"available":string) }:reification,
posEffS  ->     "DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string
],
directiCheckComDomainErrorCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"com":string)}
                :reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""com"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"unavailable":string) }:reification,
posEffS  ->     "DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""unavailable"":string)":string,
isException ->  ${-1}:reification
],
directiCheckNetDomainCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"net":string)}
                :reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""net"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"available":string) }:reification,
posEffS  ->     "DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string
],
directiCheckNetDomainErrorCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"net":string)}
                :reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""net"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"unavailable":string) }:reification,
posEffS  ->     "DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""unavailable"":string)":string,
isException ->  ${-1}:reification
],
```

```
...

}],
grounding ->    directiCheckDomainBridge:serviceGroundingSpecification[
serviceImplRef ->       "3":string,
operationName  ->       "directiCheckDomain":string,
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "DN":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "DS":string]}],
properties -> directiCheckDomainProps::serviceProperties[
serviceName      *=>   directiCheckDomainSNType:enumeration[type   ->   string,   values   ->>
                {"DirectiCheckDomainService":string}],
providerName    *=>   directiCheckDomainPNType:enumeration[type   ->   string,   values   ->>
                {"directi":string}
]]].
```

*Listing 1: Flora specification for directi check domain*

### 5.1.3 TECHNICAL DETAILS

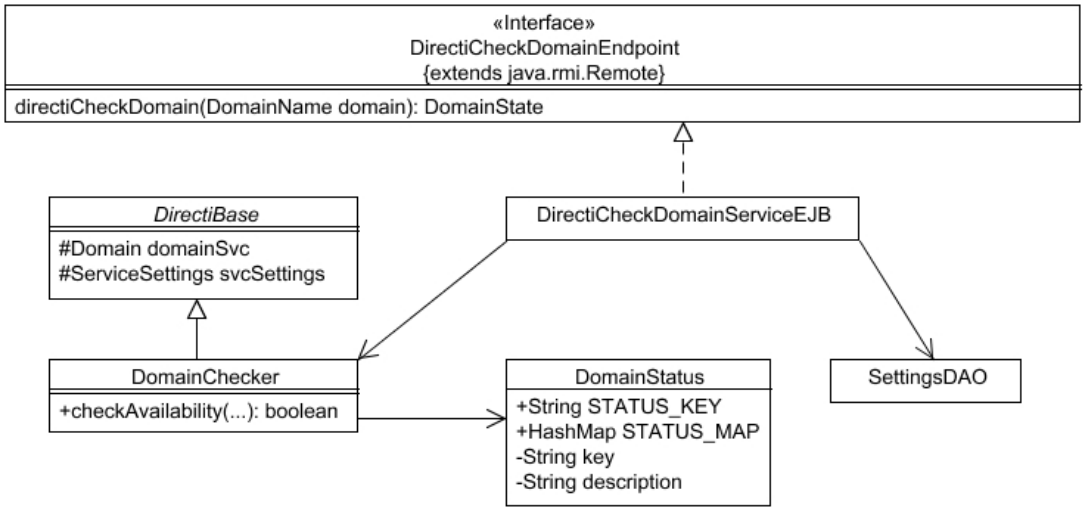


*Figure 13: directi check domain Atomic Service - class diagram*

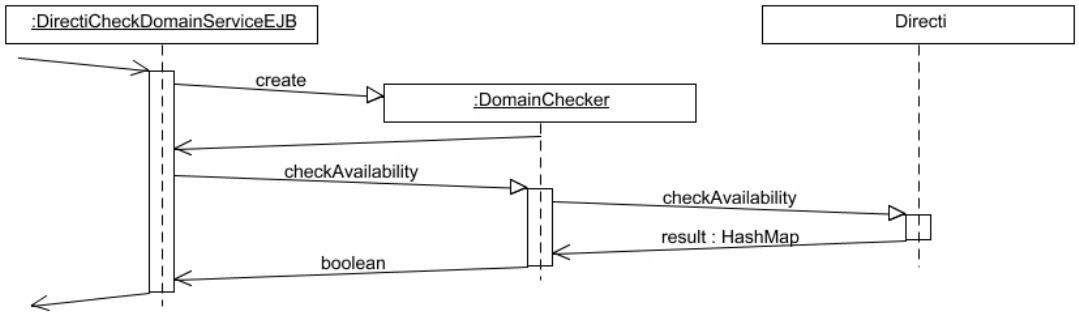


*Figure 14: directi check domain Atomic Services - sequence diagram*

## 5.2 VERISIGN CHECK DOMAIN - CHARACTERISTICS

### 5.2.1 SHORT SEMANTIC DESCRIPTION

**Purpose:** This service checks via Verisign whether a certain domain is available for registration.

- **Parameters/Conditions:**
  - **Input:** DomainName domain
  - **Output:** DomainState
  - **Core precondition:** unchecked domain name (.com, .net)
  - **Optimistic postcondition:** checked domain name (.com, .net)
  - **Errors and Exceptions:** EJBException, AssertionError
- **Scenario Description:** An customer wants to register the domain lehmann.com. Certainly this can only work if the domain is not already registered. To check the registration status the user calls his/her ISP which calls Verisigns *CheckDomainService* to validate whether lehmann.com is available for registration. The service returns available for free domains and unavailable for already registered domains.

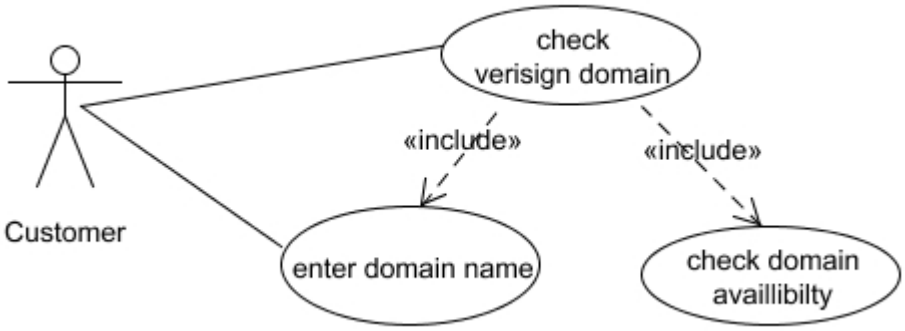


*Figure 15: verisign check domain Atomic Service - use case diagram*

### 5.2.2 FORMAL SEMANTIC SPECIFICATION

The formal semantic specifications of this Atomic Service corresponds to the directi domain check, apart from its restriction to check only .com and .net domains.

```
verisignCheckDomain:atomicService[
spec -> verisignCheckDomainSpec:semanticServiceSpecification[
conditions ->> {
verisignCheckComDomainCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"com":string)}
                :reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""com"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"available":string) }:reification,
posEffS  ->     "DS:domainState,             DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string
],
```

```
verisignCheckComDomainErrorCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"com":string)}
                :reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""com"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"unavailable":string) }:reification,
posEffS  ->     "DS:domainState,             DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""unavailable"":string)":string,
isException ->  ${-1}:reification
],
verisignCheckNetDomainCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"net":string)}
                :reification,
precondS ->     "DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,""net"":string)"
                :string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"available":string) }:reification,
posEffS  ->     "DS:domainState,             DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string
],
verisignCheckNetDomainErrorCond:condition[
precondR ->     ${DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,"net":string)}:reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""net"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"unavailable":string) }:reification,
posEffS  ->     "DS:domainState,             DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""unavailable"":string)":string,
isException ->  ${-1}:reification
]
}],
grounding -> verisignCheckDomainBridge:serviceGroundingSpecification[
serviceImplRef ->       "7":string,
operationName  ->       "verisignCheckDomain":string,
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "DN":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "DS":string]}],
properties -> verisignCheckDomainProps::serviceProperties[
serviceName      *=>  verisignCheckDomainSNType:enumeration[type  ->  string,  values  ->>
                 {"VerisignCheckDomainService":string}],
providerName     *=>verisignCheckDomainPNType:enumeration[type   ->   string,   values   ->>
                 {"verisign":string}
]]].
```

*Listing 2: Flora specification for verisign check domain*

### 5.2.3 Technical details

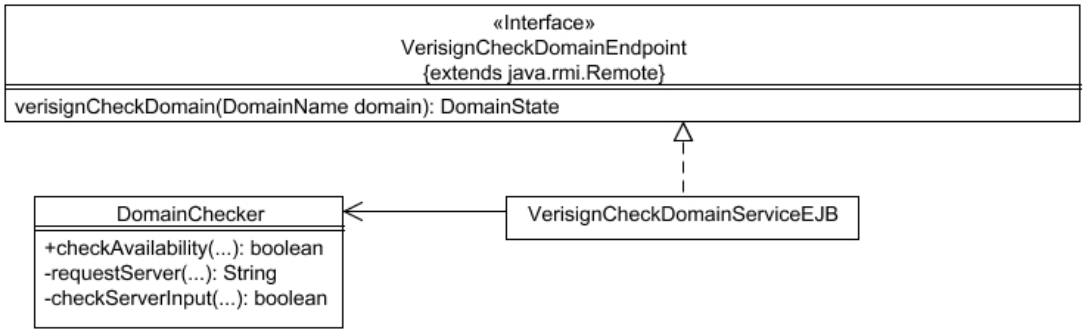


*Figure 16: verisign check domain Atomic Service - class diagram*

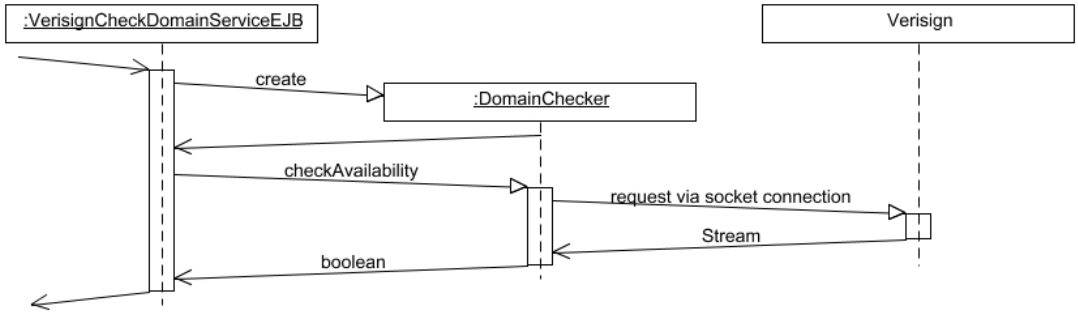


*Figure 17: verisign check domain Atomic Service - sequence diagram*

## 5.3 Denic Check Domain - characteristics

### 5.3.1 Short semantic description

**Purpose:** This service checks via Denic whether a certain domain is available for registration.

- **Parameters/Conditions:**
  - **Input:** DomainName domain
  - **Output:** DomainState
  - **Core precondition:** unchecked domain name (.de)
  - **Optimistic postcondition:** checked domain name (.de)
  - **Errors:** EJBException, AssertionError
- **Scenario Description:** A customer wants to check the domain lehmann.de for availability. To check the registration status the user calls his/her ISP which calls denic's *CheckDomainService* to validate whether lehmann.de is available for registration. The service returns available for free domains and unavailable for already registered domains.

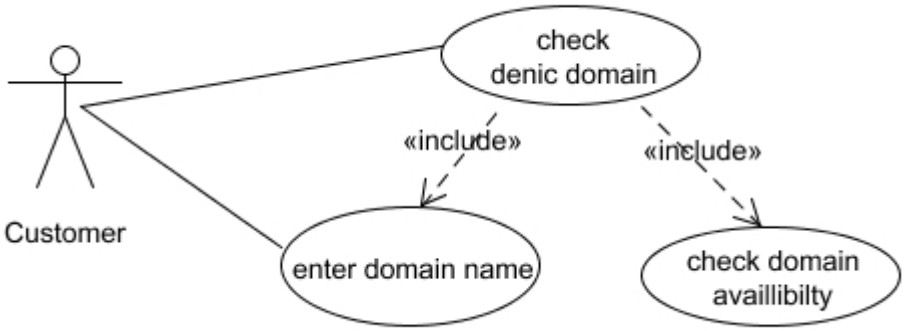


*Figure 18: denic check domain Atomic Service - use case diagram*

### 5.3.2 Formal semantic specification

The formal semantic specifications of this Atomic Service corresponds to the directi domain check. It is specifically designed to check .de domains.

```
denicCheckDomain:atomicService[
spec -> denicCheckDomainSpec:semanticServiceSpecification[
conditions ->> {
denicCheckDeDomainCond:condition[
precondR ->     ${DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,"de":string)}:reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""de"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"available":string) }:reification,
posEffS  ->     "DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string
],
denicCheckDeDomainErrorCond:condition[
precondR ->     ${DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,"de":string)}:reification,
precondS ->     "DN:domainName,         DN:parameter,         DN[tld         ->         TLD],
                hasValue(TLD,""de"":string)":string,
posEffR  ->     ${DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"unavailable":string) }:reification,
posEffS  ->     "DS:domainState,            DS:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""unavailable"":string)":string,
isException ->  ${-1}:reification
]
}],
grounding -> denicCheckDomainBridge:serviceGroundingSpecification[
serviceImplRef ->       "2":string,
operationName  ->       "denicCheckDomain":string,
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "DN":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "DS":string]}],
properties -> denicCheckDomainProps::serviceProperties[
serviceName       *=>   denicCheckDomainSNType:enumeration[type   ->   string,   values   ->>
                {"DenicCheckDomainService":string}],
providerName     *=>denicCheckDomainPNType:enumeration[type    ->    string,    values    ->>
                {"denic":string}
]]].
```

*Listing 3: Flora specification for denic check domain*

### 5.3.3 Technical details

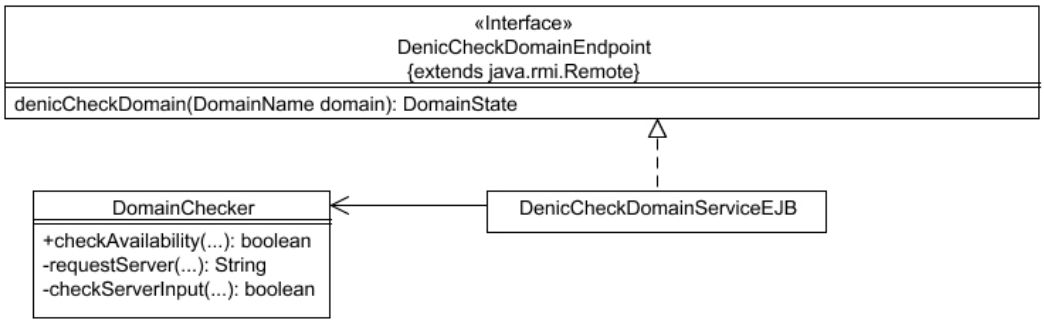


*Figure 19: denic check domain Atomic Service - class diagram*

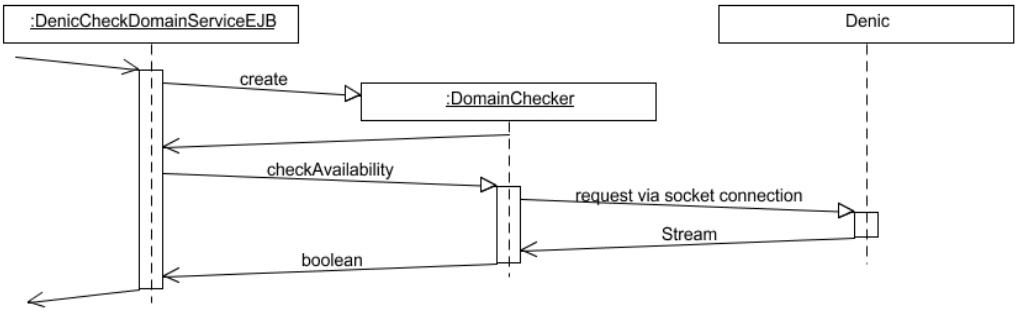


*Figure 20: denic check domain Atomic Service - sequence diagram*

## 5.4 Directi Register Domain - characteristics

### 5.4.1 Short semantic description

**Purpose:** This service registers a certain domain via Directi.

- **Parameters/Conditions:**
  - **Input:** DomainName domainName, Contact customer, Contact admin, NameServers nameServers
  - **Output:** DomainState
  - **Core precondition:** checked domain name
  - (.com, .net, .org, .biz, .name, .info, .jobs)
  - **Optimistic postcondition:** Domain registered
  - **Exceptions:** EJBException
- **Scenario Description:** A customer wants to register the domain lehmann.com. The check for availability is already done therefore the customer enters additional to the checked domain name his/her customer data and voluntarily the data of his/her administrator. Afterwards his/her ISP adds at leased one name servers to complete

the request for Directis *RegisterDomainService*. If the domain is successful registered the service responses *registered* else *unavailable*.

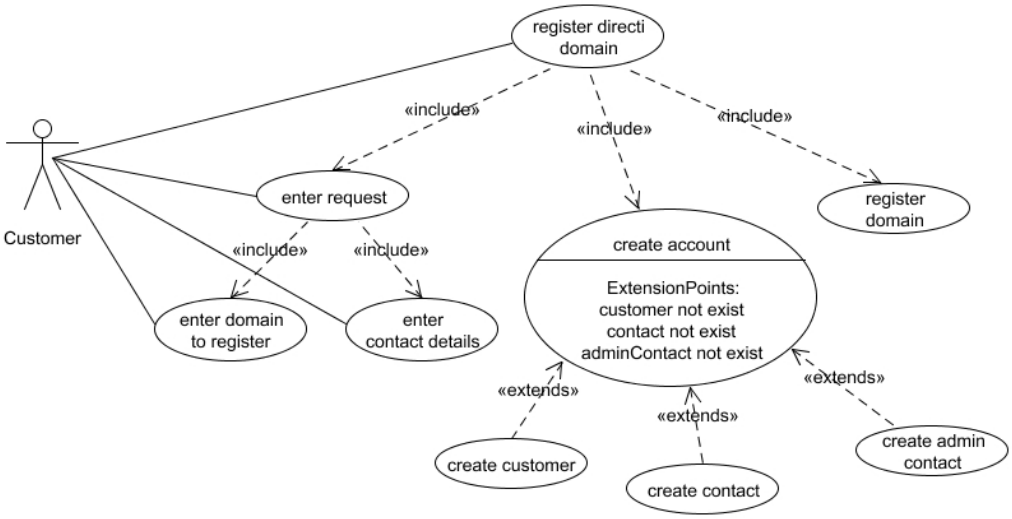


*Figure 21: directi register domain Atomic Service- use case diagram*

### 5.4.2 Formal semantic specification

For each possible top-level domain the service offers two conditions: one for the regular case that the desired domainName can be registered and one for the exceptional case that the domain is not available at the moment. To protect the external service provider from unnecessary high unsuccessful requests, the ordered domain must have been checked before. In both cases – the successful and unsuccessful – the service returns a domainState object with the appropriate value set to "registered" or "unavailable".

```
directiCheckDomain:atomicService[
spec -> directiCheckDomainSpec:semanticServiceSpecification[
conditions ->> {
directiRegisterDomain:atomicService[
spec -> directiRegisterDomainSpec:semanticServiceSpecification[
conditions ->> {
directiRegisterComDomainCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"com":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,"available":string)}:reification,
precondS ->     "DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,""com"":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string,
posEffR  ->     ${DS2:domainState,          DS2:parameter,          hasDomainState(DN,DS2),
                hasValue(DS2,"registered":string) }:reification,
posEffS  ->     "DS2:domainState,          DS2:parameter,          hasDomainState(DN,DS2),
                hasValue(DS2,""registered"":string)":string],
directiRegisterComDomainErrorCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"com":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,"available":string)}:reification,
precondS ->     "DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,""com"":string),
```

```
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string,
posEffR  ->     ${DS2:domainState,           DS2:parameter,           hasDomainState(DN,DS2),
                hasValue(DS2,"unavailable":string) }:reification,
posEffS  ->     "DS2:domainState,           DS2:parameter,           hasDomainState(DN,DS2),
                hasValue(DS2,""unavailable"":string)":string,
isException ->  ${-1}:reification],
directiRegisterNetDomainCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"net":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,"available":string)}:reification,
precondS ->     "DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,""net"":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string,
posEffR  ->     ${DS2:domainState,           DS2:parameter,           hasDomainState(DN,DS2),
                hasValue(DS2,"registered":string) }:reification,
posEffS  ->     "DS2:domainState,           DS2:parameter,           hasDomainState(DN,DS2),
                hasValue(DS2,""registered"":string)":string],
directiRegisterNetDomainErrorCond:condition[
precondR ->     ${DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,"net":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,"available":string)}:reification,
precondS ->     "DN:domainName, DN:parameter, DN[tld -> TLD], hasValue(TLD,""net"":string),
                ADMINC:contact, ADMINC:parameter, DH:contact, DH: parameter, NS:nameServers,
                NS:parameter,     DS:domainState,     DS:parameter,     hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string,
posEffR  ->     ${DS2:domainState,           DS2:parameter,           hasDomainState(DN,DS2),
                hasValue(DS2,"unavailable":string) }:reification,
posEffS  ->     "DS2:domainState,           DS2:parameter,           hasDomainState(DN,DS2),
                hasValue(DS2,""unavailable"":string)":string,
isException ->  ${-1}:reification],

...

}],
grounding->     directiRegisterDomainBridge:serviceGroundingSpecification[
serviceImplRef ->       "6":string,
operationName  ->       "directiRegisterDomain":string,
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "DN":string],
                        _#:oSP[ord -> 2, str -> "DH":string],
                        _#:oSP[ord -> 3, str -> "ADMINC":string],
                        _#:oSP[ord -> 4, str -> "NS":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "DS":string]}],
properties -> directiRegisterDomainProps::serviceProperties[
serviceName     *=>  directiRegisterDomainSNType:enumeration[type  ->  string,  values  ->>
                {"directiRegisterDomainService":string}],
providerName   *=>  directiRegisterDomainPNType:enumeration[type  ->  string,  values  ->>
                {"directi":string}]]].
```

*Listing 4: Flora specification for directi register domain*

### 5.4.3 TECHNICAL DETAILS

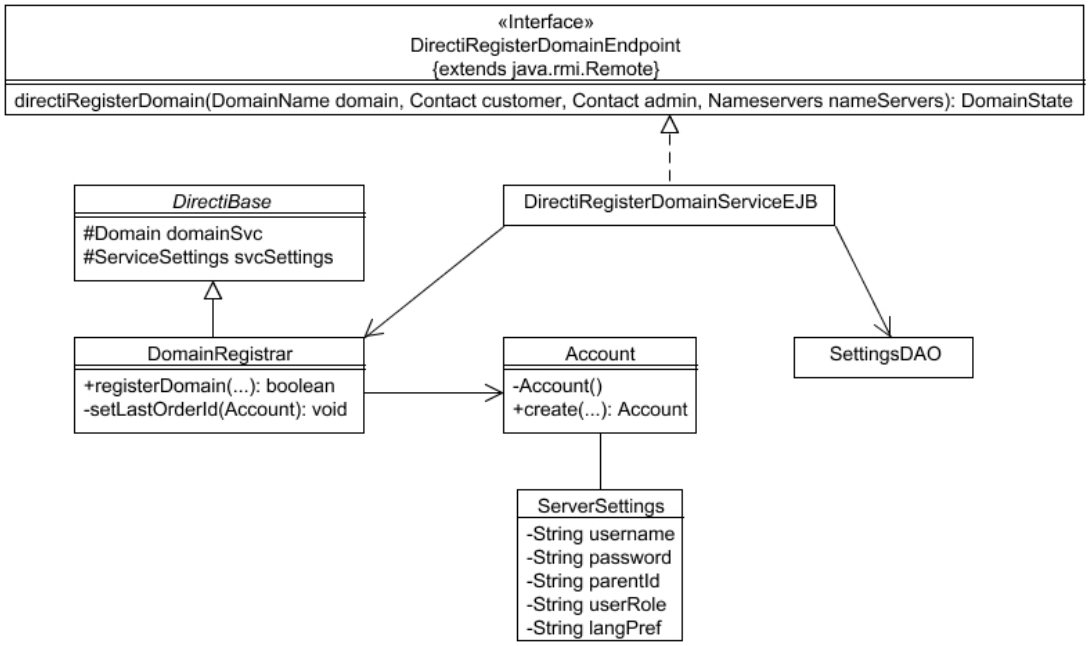


*Figure 22: directi register domain Atomic Service - class diagram*

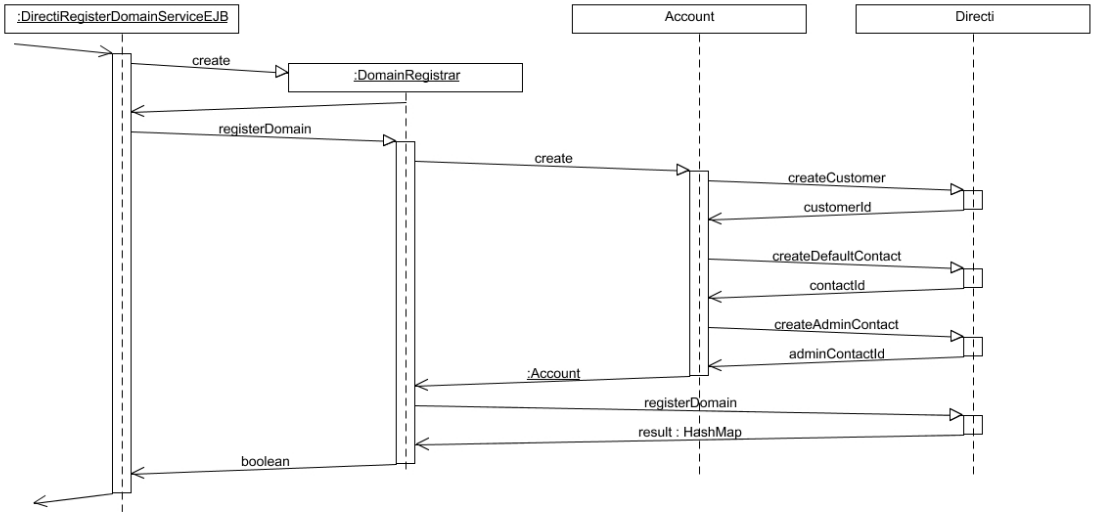


*Figure 23: directi register domain Atomic Service - sequence diagram*

## 5.5 Saferpay Credit Card Authorization - characteristics

### 5.5.1 Short semantic description

**Purpose:** This service authorizes a creditcard payment using the Saferpay Creditcard Authorization Interface (CIA). It can authorize payments with credit cards of type Visa, MasterCard, Amex, Diners and JCB.

- **Parameters/Conditions:**
  - **Input:** String invoiceNumber, AmountOfMoney amount, CreditCard creditCard
  - **Output:** SaferpayTransactionHandle
  - **Core precondition:** -
  - **Optimistic postcondition:** An "authorized" payment to be used to charge the creditcard later
- **Scenario Description:** A customer acquires a fee required service. The service provider accepts only Saferpay payment. Thus he/she has to pay for the received service using Saferpays *PaymentService*. But before he/she can do this he/she has to get a creditcard authorization from Saferpays *CreditCardAuthorizationService*. To get this authorization the customer has to assign his/her creditcard details and the amount of money he/she has to pay. If the payment is authorized by Saferpay the service returns a valid TransactionHandle which has to be used for payment later else if he/she is not authorized an invalid TransactionHandle is responded.

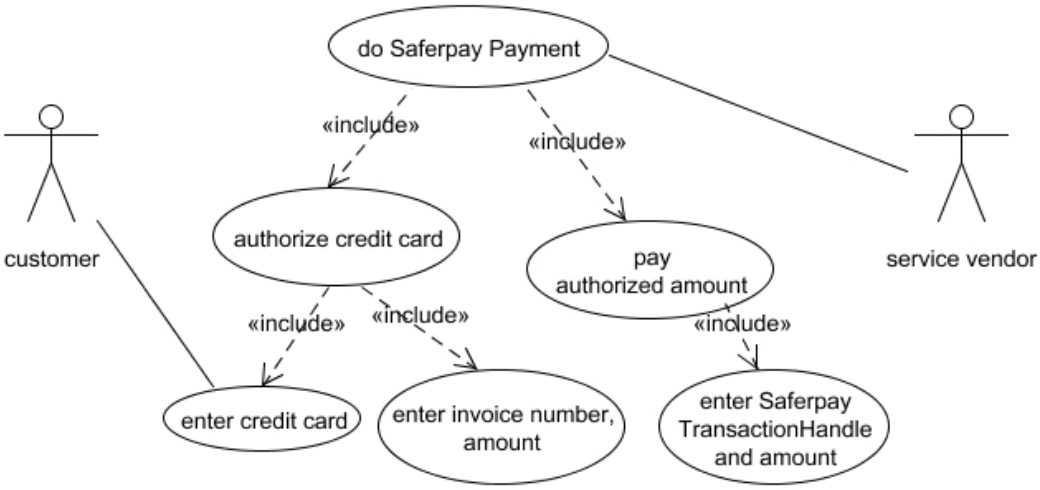


*Figure 24: saferpay Atomic Services - use case diagram*

### 5.5.2 Formal semantic specification

The saferpay credit card authorization restricts the amountOfMoney (to non-negative values and a reasonable maximal amount) that cannot be expressed otherwise than in the semantic service specification.

The service accepts an individual set of three conditions (optimistic case, case where invalid credit card data is detected, and all other fault cases) for each credit card type (Visa, Mastercard, American Express, Diners, JCB). The effects associates a *Validity* object with the credit card data, marking it as "valid", "invalid" or "unknown". The optimistic case additionally associates the amountOfMoney with the saferpay-TransactionHandle, thus stating that this amount has been authorized by saferpay with the returned saferpayTransactionHandle.

```
saferpayCreditCardAuthorization:atomicService[
spec -> saferpayCreditCardAuthorizationSpec: semanticServiceSpecification[
conditions ->> {
saferpayVisaAuthorizationCond:condition[
precondR ->     ${I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,"Visa":string)}:reification,
precondS ->     "I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,""Visa"":string)":string,
posEffR  ->     ${STH:saferpayTransactionHandle, STH:parameter, STH[validity -> PV],
                hasValue(PV,"valid":string), paymentTransaction(CC,STH), STH[token ->
                TOKEN], hasValue(TOKEN,"(unused)"),
                authorizedSaferpayAmount(STH,AM)}:reification,
posEffS  ->     "STH:saferpayTransactionHandle, STH:parameter, STH[validity -> PV],
                hasValue(PV,""valid"":string), paymentTransaction(CC,STH), STH[token ->
                TOKEN], hasValue(TOKEN,""(unused)""),
                authorizedSaferpayAmount(STH,AM)":string
],
saferpayVisaAuthorizationInvalidCond:condition[
precondR ->     ${I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,"Visa":string)}:reification,
precondS ->     "I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,""Visa"":string)":string,
posEffR  ->     ${STH:saferpayTransactionHandle, STH:parameter, STH[validity -> PV],
                hasValue(PV,"invalid":string), paymentTransaction(CC,STH)}:reification,
posEffS  ->     "STH:saferpayTransactionHandle, STH:parameter, STH[validity -> PV],
                hasValue(PV,""invalid"":string), paymentTransaction(CC,STH)":string,
isException ->  ${-1}:reification
],
saferpayVisaAuthorizationErrorCond:condition[
precondR ->     ${I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,"Visa":string)}:reification,
precondS ->     "I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,""Visa"":string)":string,
posEffR  ->     ${STH:saferpayTransactionHandle, STH:parameter, STH[validity -> PV],
                hasValue(PV,"unknown":string), paymentTransaction(CC,STH)}:reification,
posEffS  ->     "STH:saferpayTransactionHandle, STH:parameter, STH[validity -> PV],
                hasValue(PV,""unknown"":string), paymentTransaction(CC,STH)":string,
isException ->  ${-2}:reification
],

...

}],
grounding -> saferpayCreditCardAuthorizationBridge: serviceGroundingSpecification[
serviceImplRef ->       "10":string,
operationName  ->       "authorizeSaferpayCreditCard":string,
```

```
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "I":string],
                         _#:oSP[ord -> 2, str -> "AM":string],
                         _#:oSP[ord -> 3, str -> "CC":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "STH":string]}],
properties -> saferpayCreditCardAuthorizationProps::serviceProperties[
serviceName   *=> saferpayCreditCardAuthorizationSNType:enumeration[type -> string, values
                ->> {"saferpayCreditCardAuthorizationService":string}],
providerName *=>saferpayCreditCardAuthorizationPNType:enumeration[type -> string, values ->>
                {"saferpay":string}]]].
```

*Listing 5: Flora specification for saferpay credit card authorization*

### 5.5.3 TECHNICAL DETAILS

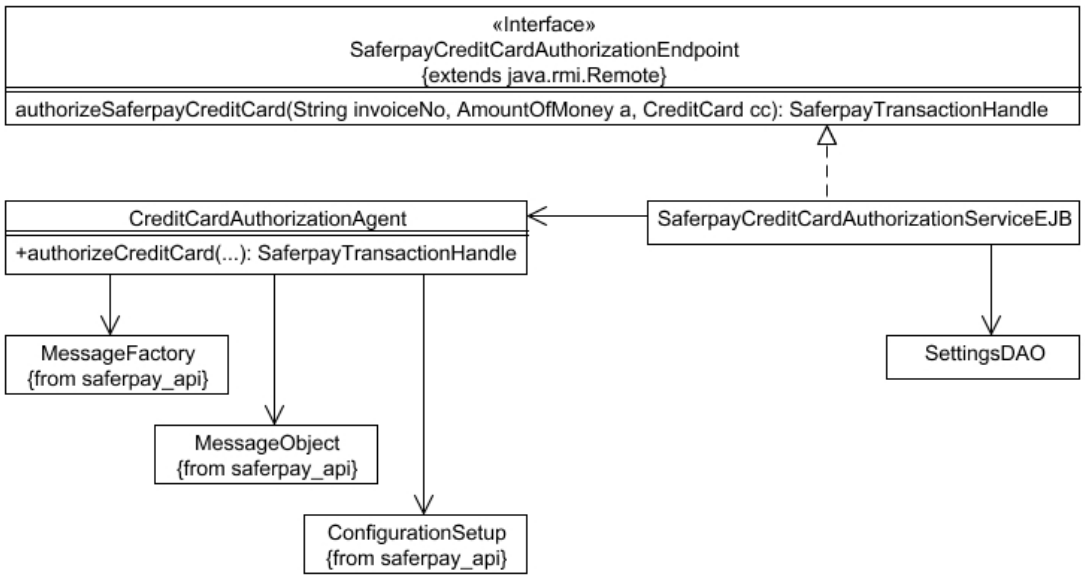


*Figure 25: saferpay creditcard authorization Atomic Service - class diagram*

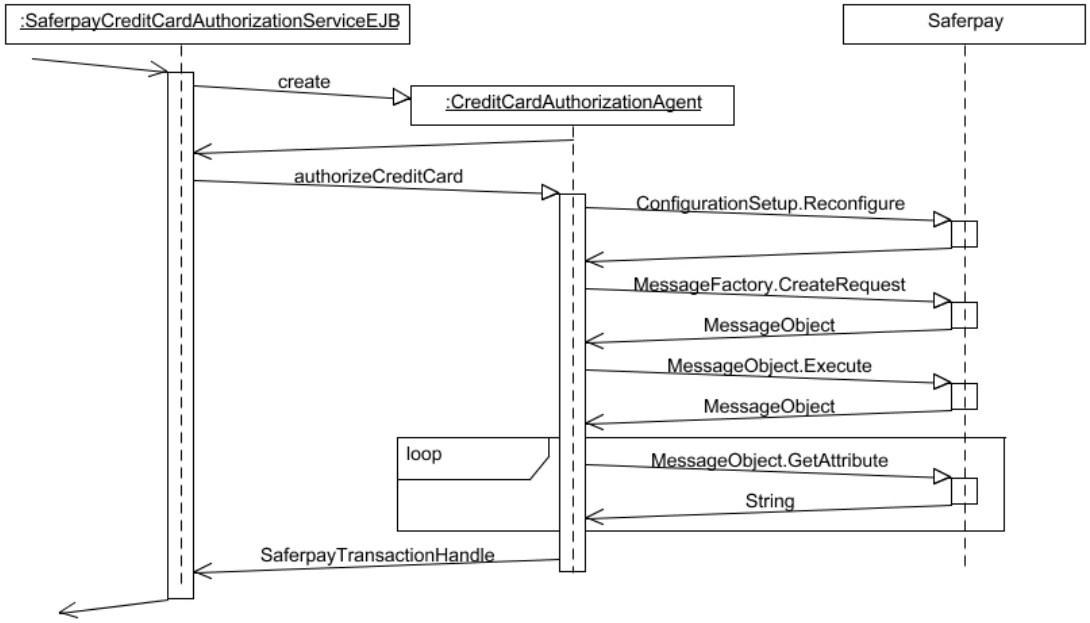


*Figure 26: saferpay creditcard authorization Atomic Service - sequence diagram*

### 5.5.4 Problems and remarks

Saferpay's payment services have to be accessed using a special API. To provide a secure information transfer from our application server to Saferpay's gateway this API uses secure socket layer. The implementation made by saferpay is based on implementations provided together with SUN's JRE. Since IBM's WebSphere is based upon IBM's J9 VM it provides its own implementation of certain security and encryption algorithms which leaded to several interoperability problems. For that reason working with SUN's we needed to change the application server to JBoss. Also the services we developed are not compliant to the EJB 2.1 specification. For CreditCardAuthorization and Payment the API does some file IO to save generated keys.

## 5.6 Saferpay Payment - characteristics

### 5.6.1 Short semantic description

**Purpose:** Charge a previously authorized credit card.

- **Parameters/Conditions:**
  - **Input:** SaferpayTransactionHandle transHandle, AmountOfMoney amount
  - **Output:** SaferpayTransactionInformation
  - **Core precondition:** An "authorized" payment to be used to charge associated the credit card
  - **Optimistic postcondition:** credit card charged
- **Scenario Description:** A customer acquires a fee required service. The service provider accepts only Saferpay payment. The customer already got a creditcard authorization from Saferpay's *CreditCardAuthorizationService*. Thus he/she has to pay for the received service using Saferpay's *PaymentService*. To pay an invoice the customer has to send his/her valid TransactionHandle and the amount of money

he/she has to complete payment. If the payment is successful the service returns information about the transaction. If not it failed and *payment aborted* is responded.

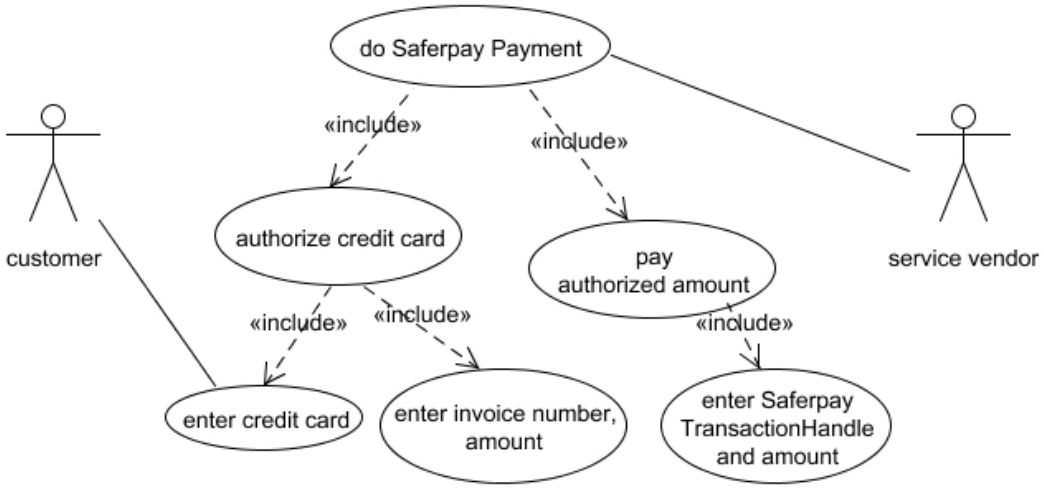


*Figure 27: saferpay Atomic Services - use case diagram*

### 5.6.2 Formal semantic specification

This service requires that a payment has been authorized before with saferpay. It needs the saferpayTransactionHandle of the authorization. The handle must reference a valid payment data, expressed through the value "valid" of the validity object contained in the handle. The service is restricted to a maximum of 115% of the authorized amount, with the currency identical to the one previously authorized.

```
saferpayPayment:atomicService[
spec -> saferpayPaymentSpec:semanticServiceSpecification[
conditions ->> {
saferpayPaymentCond:condition[
precondR  ->  ${STH:saferpayTransactionHandle,  STH:parameter,  STH[validity  ->  PV],
              hasValue(PV,"valid":string),  authorizedSaferpayAmount(STH,AMA),  AMA[amount
              ->  AMAV], AMA[currency -> AMAC], AM:amountOfMoney, AM:parameter, AM[amount
              -> AMV], AM[currency -> AMC], AMC:=:AMAC,  AMV =< 1.15*AMAV, STH[token ->
              TOKEN], hasValue(TOKEN,"(unused)")}:reification,
precondS  ->   "SSTH:saferpayTransactionHandle,  STH:parameter,  STH[validity  ->  PV],
              hasValue(PV,""valid"":string), authorizedSaferpayAmount(STH,AMA), AMA[amount
              ->  AMAV], AMA[currency -> AMAC], AM:amountOfMoney, AM:parameter, AM[amount
              -> AMV], AM[currency -> AMC], AMC:=:AMAC,  AMV =< 1.15*AMAV, STH[token ->
              TOKEN], hasValue(TOKEN,"(unused)")":string,
posEffR  ->  ${STI:saferpayTransactionInformation, STI:parameter, STI[paymentState -> PS],
              hasValue(PS,"completed"), amountCharged(AM,STI)}:reification,
posEffS  ->   "STI:saferpayTransactionInformation, STI:parameter, STI[paymentState -> PS],
              hasValue(PS,""completed""), amountCharged(AM,STI)":string
],
saferpayPaymentErrorCond:condition[
precondR  ->  ${STH:saferpayTransactionHandle,  STH:parameter,  STH[validity  ->  PV],
              hasValue(PV,"valid":string),  authorizedSaferpayAmount(STH,AMA),  AMA[amount
              ->  AMAV], AMA[currency -> AMAC], AM:amountOfMoney, AM:parameter, AM[amount
              -> AMV], AM[currency -> AMC], AMC:=:AMAC,  AMV =< 1.15*AMAV, STH[token ->
              TOKEN], hasValue(TOKEN,"(unused)")}:reification,
precondS  ->   "SSTH:saferpayTransactionHandle,  STH:parameter,  STH[validity  ->  PV],
```

```
                hasValue(PV,""valid"":string), authorizedSaferpayAmount(STH,AMA), AMA[amount
                ->  AMAV], AMA[currency -> AMAC], AM:amountOfMoney, AM:parameter, AM[amount
                -> AMV], AM[currency -> AMC], AMC:=:AMAC,  AMV =< 1.15*AMAV, STH[token ->
                TOKEN], hasValue(TOKEN,"(unused)")":string,
posEffR   -> ${STI:saferpayTransactionInformation, STI:parameter, STI[paymentState -> PS],
                hasValue(PS,"aborted")}:refication,
posEffS   ->  "STI:saferpayTransactionInformation, STI:parameter, STI[paymentState -> PS],
                hasValue(PS,""aborted"")":string,
isException -> ${-1}:reification
]
}],

grounding -> saferpayPaymentBridge:serviceGroundingSpecification[
serviceImplRef -> "11":string,
operationName  -> "doSaferpayPayment":string,
inParamSeq  ->>        {_#:oSP[ord -> 1, str -> "AM":string],
                       _#:oSP[ord -> 2, str -> "STH":string]},
outParamSeq ->> {_#:oSP[ord -> 1, str -> "STH":string]}],
properties -> saferpayPaymentProps::serviceProperties[
serviceName       *=>   saferpayPaymentSNType:enumeration[type   ->   string,   values   ->>
                {"saferpayPaymentService":string}],
providerName    *=>   saferpayPaymentPNType:enumeration[type   ->   string,   values   ->>
                {"saferpay":string}]]].
```

*Listing 6: Flora specification for saferpay payment*

### 5.6.3 TECHNICAL DETAILS

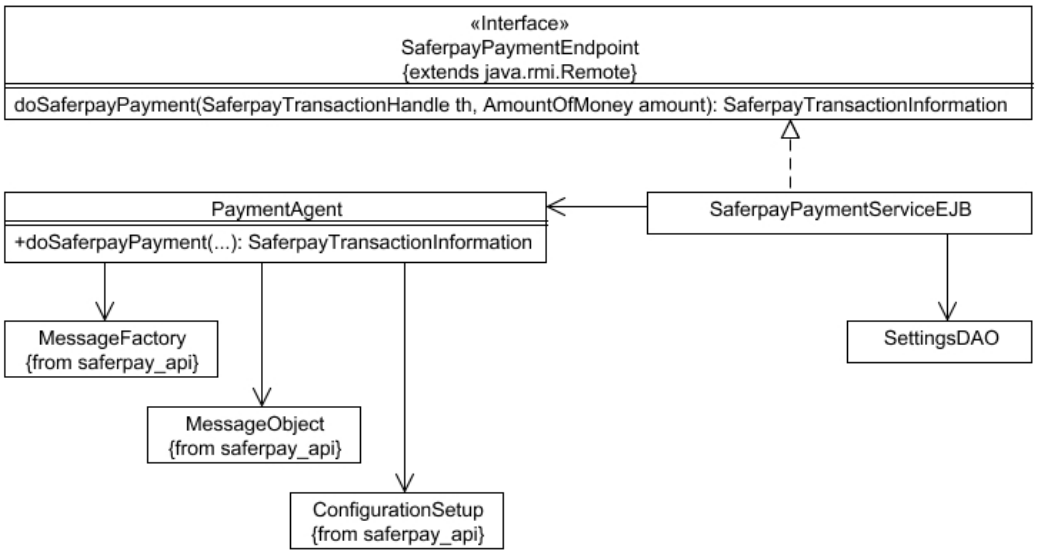


*Figure 28: saferpay payment Atomic Service - class diagram*

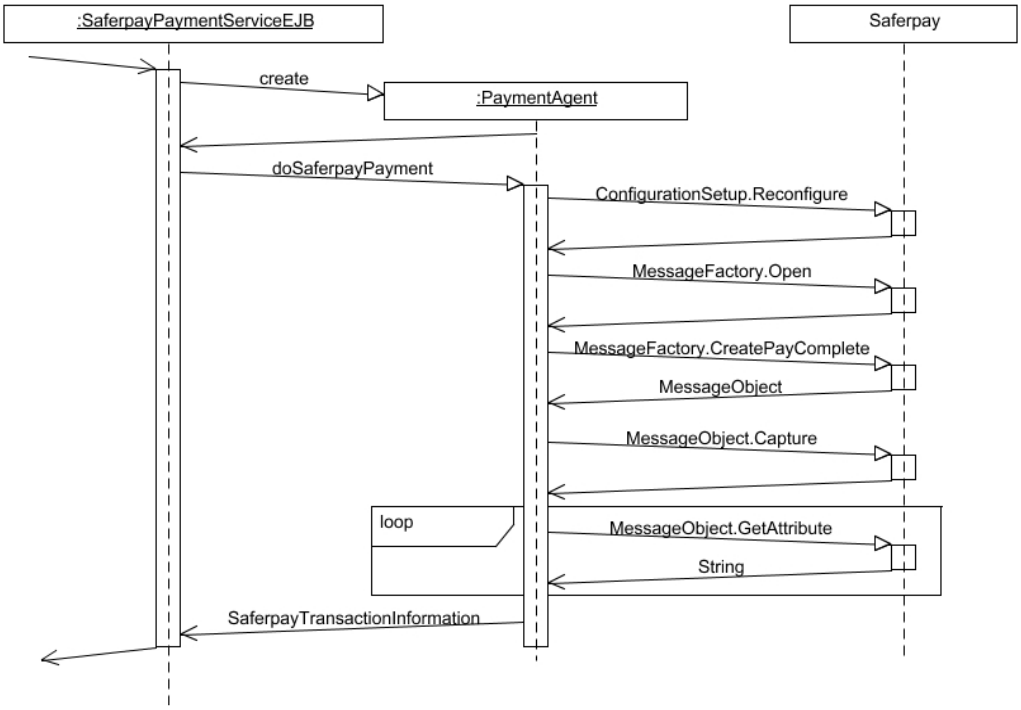


*Figure 29: saferpay payment Atomic Service - sequence diagram*

### 5.6.4 Problems and remarks

Saferpay's payment services have to be accessed using a special API. To provide a secure information transfer from our application server to Saferpay's gateway this API uses secure socket layer. The implementation made by saferpay is based on implementations provided together with SUN's JRE. Since IBM's WebSphere is based upon IBM's J9 VM it provides its own implementation of certain security and encryption algorithms which leaded to several interoperability problems. For that reason working with SUN's we needed to change the application server to JBoss. Also the services we developed are not compliant to the EJB 2.1 specification. For CreditCardAuthorization and Payment the API does some file IO to save generated keys.

## 5.7 Paypal direct Payment – characteristics

### 5.7.1 Short semantic description

**Purpose:** Uses the Paypal web service API (DirectPaymentAPI) to process a credit card payment immediately. The service supports Visa, MasterCard, Amex and Discover.

- **Parameters/Conditions:**
  - **Input:** String invoiceNumber, AmountOfMoney amount, CreditCard card, Contact contact
  - **Output:** PayPalTransactionId
  - **Core precondition:** -

- **Optimistic postcondition:** the given creditCard is charged the stated amount of money

- **Scenario Description:** A customer acquires a fee required service. To pay the invoice the customer has to assign his/her creditcard and contact details to the service vendor. The service vendor adds the amount and an invoice number and triggers the payment process.

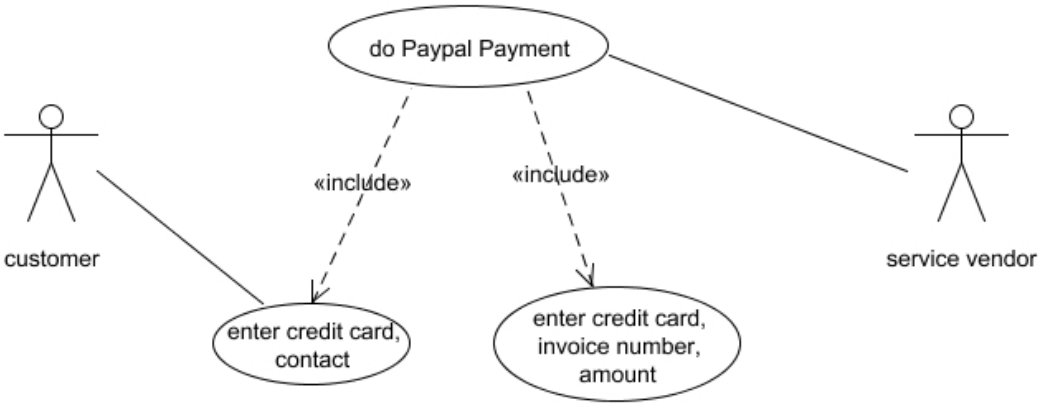


*Figure 30: paypal direct payment Atomic Service - use case diagram*

### 5.7.2 Formal semantic specification

This service does not require any previous authorization. Only all of the input parameters must be given, including the users (payers) IP address for tracing credit card fraud. For the given credit card, the holder must contain the name and address of the credit card owner to fulfil the request.

The amount to be charged must be greater than zero and within reasonable upper bounds (10000 USD as limited by Paypal). This service can only process US-dollar, therefore the currency must be set to "USD".

When finished, the service returns a payPalTransactionId, no matter whether the payment was successful or not. In case of success, the associated *Validity* object has a value of "valid", else it will be set to "invalid". Only when payment was successful, the association amountCharged will express that the credit card connected with the payPalTransactionId has been charged the amount specified in the precondition.

```
payPalInstantCreditCardPayment:atomicService[
spec -> payPalInstantCreditCardPaymentSpec: semanticServiceSpecification[
conditions ->> {
payPalInstantVisaPaymentCond:condition[
precondR ->     ${I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, AM[currency ->CY], hasValue(CY,"USD":string),
                CC:creditCard, CC:parameter, CC[type -> TYPE], hasValue(TYPE,"Visa":string),
                C:contact, C:parameter, creditCardOwner(CC, C)}:reification,
precondS ->     "I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A], A>0, A<1000000, AM[currency ->CY], hasValue(CY,""USD"":string),
                CC:creditCard, CC:parameter, CC[type -> TYPE],
                hasValue(TYPE,""Visa"":string), C:contact, C:parameter, creditCardOwner(CC,
                C)":string,
```

```
posEffR  ->     ${PPT:payPalTransactionId,     PPT:parameter,     PPT[validity     ->     VAL],
                hasValue(VAL,"valid":string),                          amountCharged(AM,PPT),
                paymentTransaction(CC,PPT)}:reification,
posEffS  ->     "PPT:payPalTransactionId,     PPT:parameter,     PPT[validity     ->     VAL],
                hasValue(VAL,""valid"":string),                        amountCharged(AM,PPT),
                paymentTransaction(CC,PPT)":string
],
payPalInstantVisaInvalidDataErrorCond:condition[
precondR ->     ${I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A],   A>0,   A<1000000,   AM[currency   ->CY],   hasValue(CY,"USD":string),
                CC:creditCard, CC:parameter, CC[type -> TYPE], hasValue(TYPE,"Visa":string),
                C:contact, C:parameter, creditCardOwner(CC, C)}:reification,
precondS ->     "I:invoiceNumber, I:parameter, AM:amountOfMoney, AM:parameter, AM[amount ->
                A],   A>0,   A<1000000,   AM[currency   ->CY],   hasValue(CY,""USD"":string),
                CC:creditCard,         CC:parameter,         CC[type         ->         TYPE],
                hasValue(TYPE,""Visa"":string), C:contact, C:parameter, creditCardOwner(CC,
                C)":string,
posEffR  ->     ${PPT:payPalTransactionId,     PPT:parameter,     PPT[validity     ->     VAL],
                hasValue(VAL,"invalid":string), paymentTransaction(CC,PPT)}:reification,
posEffS  ->     "PPT:payPalTransactionId,     PPT:parameter,     PPT[validity     ->     VAL],
                hasValue(VAL,""invalid"":string), paymentTransaction(CC,PPT)":string,
isException ->  ${-2}:reification
],

...

}],
grounding -> payPalInstantCreditCardPaymentBridge: serviceGroundingSpecification[
serviceImplRef ->       "9":string,
operationName  ->       "doDirectPayment":string,
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "I":string],
                        _#:oSP[ord -> 2, str -> "AM":string],
                        _#:oSP[ord -> 3, str -> "CC":string],
                        _#:oSP[ord -> 4, str -> "C":string],
                        _#:oSP[ord -> 5, str -> "IP":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "PPT":string]}],
properties -> payPalInstantCreditCardPaymentProps::serviceProperties[
serviceName  *=> payPalInstantCreditCardPaymentSNType:enumeration[type -> string, values ->>
                {"payPalInstantCreditCardPaymentService":string}],
providerName *=> payPalInstantCreditCardPaymentPNType:enumeration[type -> string, values ->>
                {"payPal":string}]]].
```

*Listing 7: Flora specification for paypal direct payment*

### 5.7.3 Technical details

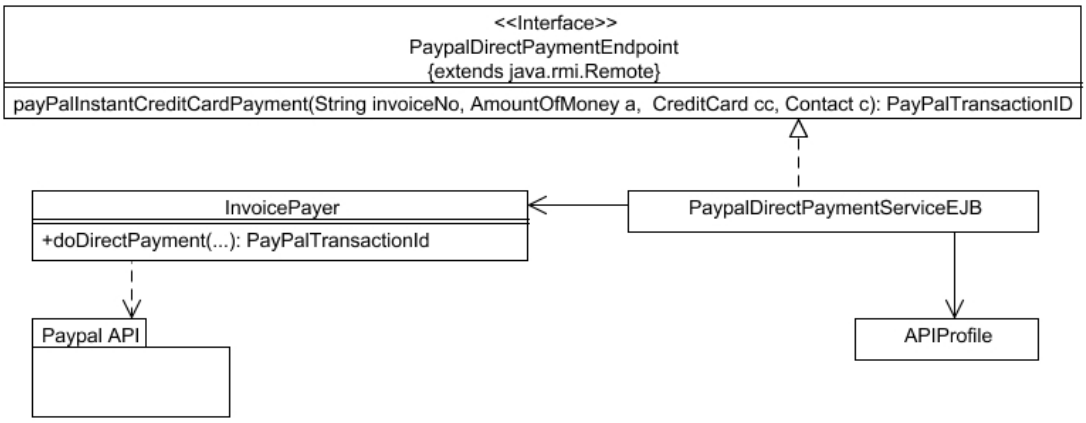


*Figure 31: paypal direct payment Atomic Service - class diagram*

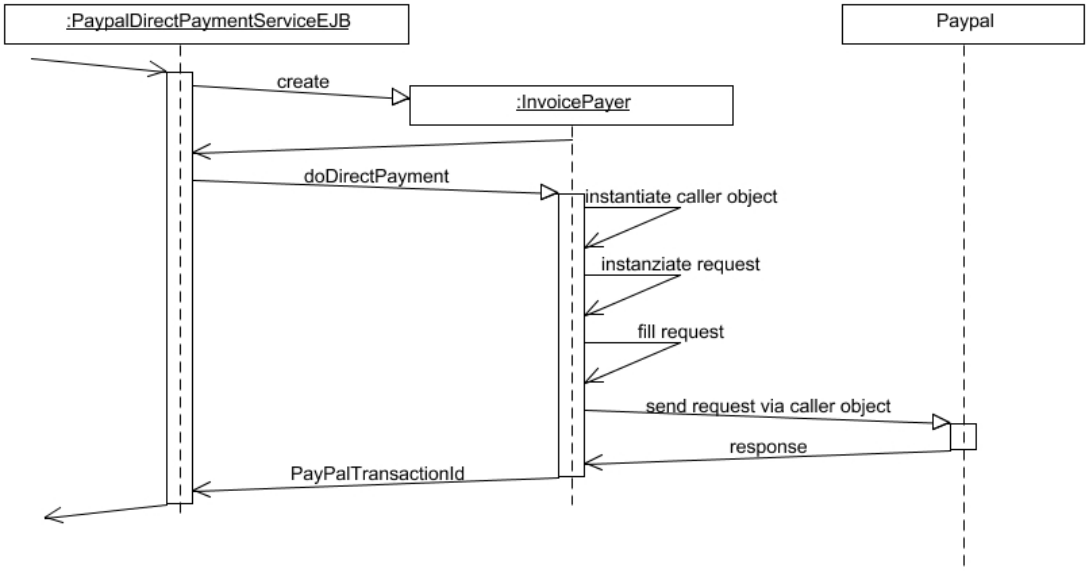


*Figure 32: paypal direct payment Atomic Service - sequence diagram*

### 5.7.4 Problems and remarks

Paypal provides an API - the most close configuration supported is the following:

- Sun Solaris 9.0
- Axis 1.2
- JDK 1.4.2 or later
- WSDL 1.1 Standard

Even with hard trying we failed on running the Paypal service using the given API running on IBMs JVM. We learned that IBMs Crypto packages does not support several key sizes,

among others the key size demanded from paypal. Based upon this problem we tried to deploy the corresponding classes developed from SUN to IBMs J9, which was not possible in the given time frame. We also failed on running the paypal API without the included Axis package.

As an additional problem we figured out, that the API forces the programmer to pass a path of a key file to the API. This file is opened by the API itself. Unfortunately there is no functionality provided which gave us the possibility to avoid this file IO.

Due to that reason we developed a non-ejb object and deployed this on a JBoss application server running on Java 1.4.2. Sadly as a consequence the result does not fit the atomic service definition, which demands Atomic Services to be EJBs with web service interface. Based on this issue a possible workaround would be to develop an additional level on indirection - a secondary proxy instance. A similar sample can be found in 5.8.

## 5.8 Update Local Nameserver - characteristics

### 5.8.1 Short semantic description

**Purpose:** Registers a passed domain name using a tcp-server which assigns an IP from it's pool. *This assignment is done by editing /etc/hosts on the machine a special tcp-servers runs on, for test purposes.*

- **Parameters/Conditions:**
  - **Input:** domain to register at the local nameserver
  - **Output:** Nameservers which contains all nameservers (nameserver) updated by the service.
  - **Core precondition:** -
  - **Optimistic postcondition:** the assigned IP-Address can be looked up using the returned nameservers
- **Scenario Description:** To assign lehmann.com to an IP-Address, the service can be called with the domain parameter lehmann.com. After a correct execution of the service lehmann.com will be resolvable on the machine which runs the tcp-server. After successful execution the service returns an array of updated nameserves which only contains the IP of the machine which runs the tcp-server.

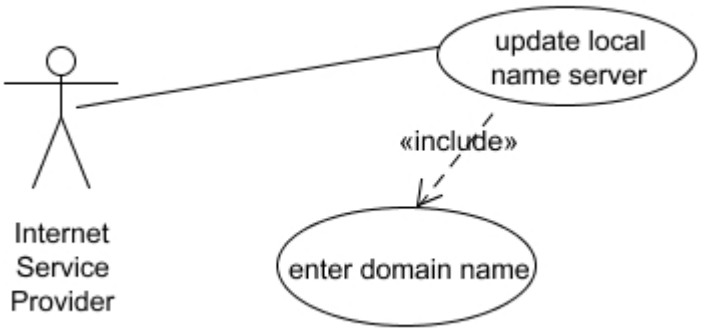


*Figure 33: update local nameserver Atomic Service - use case diagram*

### 5.8.2 Formal semantic specification

This service can be applied in two cases. First, if a domain is available and will afterwards be registered (presumed that all data is correct and a service for domain registration is executable). Second, if the domain has been registered through the internet service provider and will be afterwards inserted into local nameservers (can be one or more). Both cases have the effect that the returned set of nameServers lists the given domainName.

```
updateLocalNameserver:atomicService[
spec            updateLocalNameserverSpec:semanticServiceSpecification[
conditions ->> {
updateLocalNameserver1Cond:condition[
precondR ->     ${DN:domainName,            DN:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"available":string)}:reification,
precondS ->     "DN:domainName,             DN:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""available"":string)":string,
posEffR  ->     ${NS:nameServers, NS:parameter, domainNameservers(DN,NS)}:reification,
posEffS  ->     "NS:nameServers, NS:parameter, domainNameservers(DN,NS)":string],
updateLocalNameserver2Cond:condition[
precondR ->     ${DN:domainName,            DN:parameter,            hasDomainState(DN,DS),
                hasValue(DS,"registered":string)}:reification,
precondS ->     "DN:domainName,             DN:parameter,            hasDomainState(DN,DS),
                hasValue(DS,""registered"":string)":string,
posEffR  ->     ${NS:nameServers, NS:parameter, domainNameservers(DN,NS)}:reification,
posEffS  ->     "NS:nameServers, NS:parameter, domainNameservers(DN,NS)":string
]
}],
grounding  ->   updateLocalNameserverBridge:serviceGroundingSpecification[
serviceImplRef ->       "8":string,
operationName  ->       "updateLocalNameServer":string,
inParamSeq  ->>         {_#:oSP[ord -> 1, str -> "DN":string]},
outParamSeq ->>         {_#:oSP[ord -> 1, str -> "NS":string]}],
properties -> updateLocalNameserverProps::serviceProperties[
serviceName     *=>  updateLocalNameserverSNType:enumeration[type  ->  string,  values  ->>
                {"updateLocalNameserverService":string}],
providerName   *=>  updateLocalNameserverPNType:enumeration[type  ->  string,  values  ->>
                {"local":string}]]].
```

*Listing 8: Flora specification for update local nameserver*

### 5.8.3 Technical details

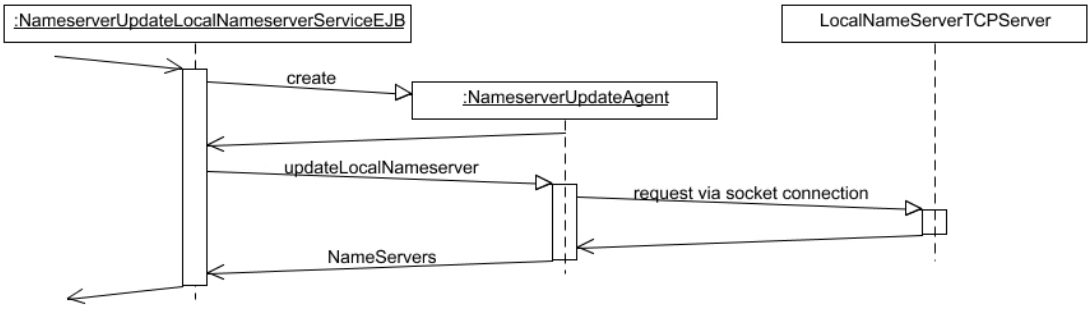


*Figure 35: update local nameserver Atomic Service - sequence diagram*

*Figure 34: update local nameserver Atomic Service - class diagram*

### 5.8.4 Problems and remarks

Because of the EJB Spec which denies file I/O we added an additional level of indirection. Based upon the additional level of indirection we gained the possibility to increase the level of distribution. The added tcp-server listens on a socket and adds a new line to /etc/hosts if it receives an incoming string. This string is used as domain name. An IP will be taken from a local pool (for test purposes only one IP is used). The Bean itself just opens an outgoing tcp-connection and sends a string containing the domain name to register to the service described above. The developed atomic service fully fits into the atomic services specification; we developed an EJB with web service interface. The DNS-lookup cache has to be configured using the networkaddress.cache.ttl and networkaddress.cache.negative.ttl properties changeable in the security/java.security file.

## 6 Appendix – Scenario Ontology in Flora

oth cases have the effect that the returned set of nameServers lists the given domainName.

```
// ********************************
// * Filename: personsOntology.flr *
// ********************************

?- flAdd 'asgOntology'.

// ============================================
// Definition of domain specific classes/records
// ============================================
person:record[firstName *=> string,
                      lastName *=> string].
phone:record[phoneCC *=> string,
                      phoneArea *=> string,
                      phoneNumber *=> string].
fax:record[phoneCC *=> string,
```

```
                        phoneArea *=> string,
                        phoneNumber *=> string].
address:record[city *=> string,
                        zipCode *=> string,
                        state *=> string,
                        countryCode *=> string,
                        street *=> string,
                        street2 *=> string].
contact:record[contactPerson *=> person,
                        organisation *=> string,
                        email *=> string,
                        phoneNumber *=> phone,
                        faxNumber *=> fax,
                        contactAddress *=> address].
customer:contact.
admin:contact.

// =====================================================
// Definition of domain specific relationships and rules
// =====================================================
hasAddress(person, address):relation.
```

*Listing 9: Flora ontology for persons*

cases have the effect that the returned set of nameServers lists the given domainName.

sdljslkdgjsdlgsdflgkdfgd

oth cases have the effect that the returned set of nameServers lists the given domainName.

```
// ******************************
// * Filename: domainOntology.flr *
// ******************************

?- flAdd 'asgOntology'.
?- flAdd 'personsOntology'.

// ============================================
// Definition of domain specific classes/records
// ============================================
domainName:record[name *=> string,
                        tld *=> string].
domainState:enumeration[type -> string,
                        values ->> {"unchecked":string, "available":string,
                        "registered":string, "unavailable":string}].
nameServer:record[name *=> string,
                        ipAddress *=> ipAddress].
nameServers:bag[type -> nameServer].
ipAddress::string.
ipAddresses:bag[type -> ipAddress].

// =====================================================
// Definition of domain specific relationships and rules
// =====================================================
hasDomainState(domainName, domainState):relation.
domainNameservers(domainName, nameServers):relation.
```

*Listing 10: Flora ontology for domains*

```
// ********************************
// * Filename: paymentOntology.flr *
// ********************************

?- flAdd 'asgOntology'.
?- flAdd 'personsOntology'.

// ==============================================
// Definition of domain specific classes/records
// ==============================================
invoiceNumber::string.
paymentData::record[].
creditCard::paymentData[number *=> string,
                        type *=> creditCardType,
                        expMonth *=> ordinal,
                        expYear *=> ordinal,
                        cvv *=> string].
amountOfMoney::record[currency *=> string,
                        amount *=> ordinal].
validity:enumeration[type -> string,
                        values ->>{"valid":string, "invalid":string, "unknown":string}].
paymentTransactionData::record[validity *=> validity].
payPalTransactionId::paymentTransactionData[value *=> string].
saferpayTransactionHandle::paymentTransactionData[id *=> string, token *=> string].
saferpayTransactionInformation::paymentTransactionData[authorizationCode *=> string,
                         authorizationResultMessage *=> string,
                         date *=> string,
                         time *=> string,
                         providerName *=> string,
                         contractNumber *=> string,
                         paymentState *=> paymentState].
paymentState:enumeration[type -> string,
                        values ->> {"completed":string, "aborted":string,
                        "unknown":string}].
creditCardType:enumeration[type -> string,
                        values ->> {"Visa":string, "MC":string, "Amex":string,
                        "Diners":string, "JCB":string, "Discover":string}].

// ======================================================
// Definition of domain specific relationships and rules
// ======================================================
creditCardOwner(creditCard, contact):relation.
paymentTransaction(paymentData, paymentTransactionData):relation.
amountCharged(amountOfMoney, paymentTransactionData):relation.
authorizedSaferpayAmount(saferpayTransactionHandle, amountOfMoney):relation.
```

*Listing 11: Flora ontology for payment*

## References

[1] MacKenzie, C., Laskey, K., McCabe, F., Brown, P., Metz, R., Reference model for service oriented architectures. Working draft; OASIS, 09. September 2005.

[2] Laures, G., Jank K., D6.V-1: Reference Architecture: Requirements, Current Efforts and Design; The ASG Project, Europe 2005.

[3] Hahmann, T., Möller, J., Sommer, P., Peissl, B., Wahler, A., An Adaptive solution for internet services' supply chains; Semantics 2005, Vienna, 24. November 2005.