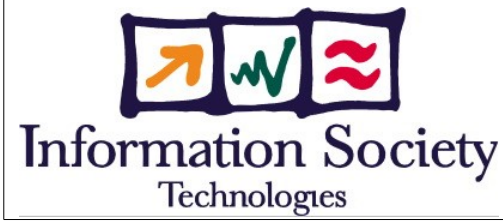
Digital Business Ecosystem

Contract n° 507953

Workpackage 6: Self Organisation

Deliverable D6.7:

Implementation of Distributed Intelligence System



Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953

Project Acronym: DBE

Title: Digital Business Ecosystem

Deliverable N°: 6.7

Due dates: 12/2006

Delivery Date: 12/2006

Short Description: This document outlines the task for the implementation of the Distributed Intelligence System (DIS). The DIS provides the implementation of distributed intelligence for the Evolutionary Environment (EvE) and Habitat network within the Digital Business Ecosystem (DBE) project. From the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2], it was derived that the DIS is required to perform two major tasks, service matching and targeted migration. The design and implementation of the DIS are outlined in this document.

Author: Intel Ireland Ltd.

Partners contributed: Intel Ireland Ltd.

Made available to: Public

Versioning

Version	Date	Author, Organisation
D6.7 (0.1)	07/11/2006	David McKitterick, Intel Ireland
D6.7 (0.2)	04/12/2006	David McKitterick, Intel Ireland
D6.7 (0.3)	14/12/2006	David McKitterick, Intel Ireland

Quality check:

1st Internal Reviewer : Gerard Briscoe, HWU

2nd Internal Reviewer: Claudius Masuch, LSE

TABLE OF CONTENTS

Table of Figures

Executive Summary

This document describes the only deliverable for task C53, the Implementation of the Distributed Intelligence System (DIS). This software deliverable provides a design and implementation that is consistent with the high level requirements and design as outlined in the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2]. The implementation of the DIS is a sub-component of the Evolutionary Environment (EvE) implementation. The EvE is a layer within the Digital Business Ecosystem (DBE) architecture which provides for a digital ecosystem with self-organising complex behaviour. The architectural requirements of the EvE are outlined the 'Evolutionary Environment Architecture Requirements' internal project document [1] and updated in the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2] . The relevant requirements of the EvE with respect to the DIS implementation are summarised in this document.

The objective of the DIS implementation is to provide an intelligent software component that can be distributed throughout the EvE network and be used to perform a pattern recognition process on service descriptions to determine similarity between services. The pattern recognition process is implemented with the use of neural networks. Once two services are found to be similar, then the service representations of these services, called EveServices, are cloned and migrated to targeted Habitat nodes within the EvE network. In addition to the implementation of this distributed intelligence component, other features are provided by the DIS, such as service description pre-processing, DIS core functionality, a multi-threaded matching framework and targeted migration. The DIS software can be found on the EvEnet SourceForge project [3].

1. Introduction

The Distributed Intelligence System (DIS) provides the implementation of the distributed intelligence for the Evolutionary Environment (EvE) and Habitat network within the Digital Business Ecosystem (DBE) project. From the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2], it was derived that the DIS is required to perform two major tasks. The first task is to compare sets of service descriptions of EveServices by using an appropriate matching process. The selected matching mechanism for the DIS is the use of neural networks, as proposed by the high level design in deliverable D6.6. The second task is to trigger the targeted migration of EveServices following successful matches of these EveServices and using their usage history to distinguish optimal migration destination habitats.

Chapter 2 provides some background information on the EvE and how distributed intelligence was intended to be integrated as part of the Execution Environment (ExE) architecture. It outlines some of the core components within the EvE such as the Habitat service and the service representations called EveServices. In particular, the specific role of distributed intelligence is explained. Other interesting aspects of the DIS are introduced such as neural networks and targeted migration.

The architectural requirements and design of the DIS are outlined in Chapter 3. Firstly the relevant aspects of the high level design specification document [2] are summarised to provide an introduction to the requirements to be fulfilled by the DIS implementation. The design of the DIS is then presented with a focus on the main internal components of the DIS. These include the Business Modeling Language (BML) Preprocessor, the matching framework and the embedded Neural Network (NN).

Chapter 4 describes the implementation of the DIS with a detailed look at the class structure within each component. The relationship between the Habitat service implementation and the DIS are explained by outlining the integration points between both implementations. These descriptions are supported by class and sequence diagrams.

2. Evolutionary Environment and Distributed Intelligence

This chapter describes that part of the DBE architecture called the EvE. This environment is explained with a focus on the components within it that have a relationship with distributed intelligence and specifically the DIS.

The Evolutionary Environment

The Evolutionary Environment or EvE is a layer within the DBE architecture which provides for a digital ecosystem with self-organising complex behaviour. This behaviour is inspired by those of biological ecosystems [1]. The EvE is defined by the Digital Ecosystem model [2] which is based on Mobile Agent Systems, Distributed Evolutionary Computing (DEC) and ecosystems theory. This model was introduced into the DBE computing architecture to integrate the objectives of the project's science domain with the rest of the computing architectural components. The architectural requirements of the EvE are defined within the 'Evolutionary Environment Architecture Requirements' internal project document [1] and updated in the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2] .

The EvE consists of interconnected Habitats, similar to the concept of Habitats in a biological ecosystem, with each Habitat representing an SME or business user. From a logical perspective, the Habitat is where evolutionary optimisation and selection processes can take place [3]. A Habitat contains a service pool which is similar to an agent pool from mobile agent systems where a set of agents are registered to that agent pool. This service pool is called the EveService-Pool and contains EveServices. Habitats and EveServices will be discussed in more detail in the next section.

The EvE architecture also provides for an intelligence model for performing the evolutionary processes within Habitats. This intelligence comes in the form of genetic algorithms and distributed intelligence. The generic algorithms optimise user requests by finding the best service solution to fit that request. Distributed intelligence performs direct or targeted migration of EveServices from EveService-Pool to

EveService-Pool. This distributed intelligence is provided by the DIS, which will be discussed in more detail in the following sections and chapters.

Habitat Networks and EveServices

The interconnected Habitats which make up the EvE are collectively known as the Habitat Network. This network is built on top of the ExE's peer-to-peer (P2P) network. The Habitat network does not implement any network features but uses the underlying P2P infrastructure provided by the ExE. Each Habitat is a node in the Habitat network, where this node represents an individual SME or business user in the DBE network. The Habitat is responsible for migrating EveServices, managing EveService-Pools and maintaining usage and migration histories. Each Habitat has a unique identifier, called a Habitat ID, which enables the Habitat to be located within the P2P network.

An EveService is a pointer to a DBE service description which is known as a Service Manifest (SM). EveServices are modeled as mobile agents within a Multi-Agent System (MAS) [2]. They can be migrated and duplicated throughout the Habitat network. EveServices are created within a Habitat following the deployment of a DBE service. Each Habitat hosts an EveService-Pool which contains all the EveServices related to that Habitat. The migration of EveServices through interconnected Habitats is performed by the EvE intelligence. The targeted migration is controlled and driven by the DIS.

Habitat clusters can be formed within the Habitat network. These clusters represent a community of Habitats with related characteristics and services. Clusters are formed by successful migration of EveServices between Habitats, therefore creating and strengthening the connections between them.

Distributed Intelligence

Distributed intelligence [2] is an important aspect of the EvE model although it is complementary to the evolutionary optimisation processes performed by the genetic algorithms. The main objective of distributed intelligence within the EvE is to target and migrate EveServices directly to particular Habitats. Targeted migrations will occur based on similarities between the service descriptions of EveServices. The

targeted habitats are decided from the usage history of the related EveServices. This migration of EveServices will significantly increase the size of EveService-Pools, therefore improving the speed of the succession process of Habitat clusters. Distributed intelligence is a proactive method of enabling the evolutionary aspect of digital ecosystems.

The DIS provides the implementation of the distributed intelligence for the EvE and Habitat network. The role of the DIS is defined in the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2]. As outlined in the document, the DIS is responsible to provide the features of service matching and targeted migration. To compare sets of service descriptions of EveServices, an appropriate matching process is required. The selected matching process for the DIS is the use of neural networks, as proposed by the high level design in deliverable D6.6. In addition to EveService matching, the DIS is designed to trigger the targeted migration of these EveServices following successful matches of EveServices and using the usage history of these EveServices to distinguish optimal migration destination habitats. These two areas are described in more detail in the following two sections.

Neural Networks

Neural networks, or artificial neural networks, are computational algorithms that process an attempt to mimic biological neural networks. A neural network normally consists of numerous interconnected neurons, where each neuron is an individual cell that process small amounts of information before triggering other neurons to continue the process. The human brain is a classical example of a biological neural network [4].

Neural networks are not capable of solving all types of problems but they are suitable for solving problems like classifying groups, series prediction, data mining and in particular pattern recognition [4]. This pattern recognition capability of neural networks can be used to determine similarities between textual descriptions. This is why neural networks were proposed in the high level design specification for the intelligent matching mechanism of the DIS.

The neural network approach for the DIS provides for the implementation of individual intelligence components which are to be embedded within EveServices. With EveServices being distributed throughout the Habitat network, this then provides for a distributed intelligence architecture. As proposed by the high level design specification, NN components will be created for each EveService and trained with the single service description of that EveService.

Targeted Migration

Natural migration of EveServices occurs after service deployment. A 'Home Habitat' will migrate a new EveService to each of its connected Habitats. The propagation of these migrations depends on the probabilities associated with the Habitat connections [2]. Targeted migration of EveServices is more proactive in nature. This occurs when a new EveService is compared with all the existing EveServices within a Habitat's EveService-Pool. If EveServices are found to be similar then both EveServices are migrated to connected and unconnected Habitats where the other EveService was used. When migration occurs between unconnected Habitats, this accelerates the adaptive nature of the Habitat network more so than with just natural migration.

The selection of destination Habitats is based on the usage history of the EveService. Each EveService has an associated migration and usage history (MUH). MUH is a record of the migration and usage of an EveService through out its existence within the EvE.

3. Architectural Requirements and Design

This chapter describes the architectural requirements and design of the DIS. To begin the architectural requirements, as specified by the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2], will be summarised. The design of DIS is then outlined in the second section.

Architectural Requirements

The high-level design specification [2] of the DIS describes the structural and behavioural requirements of the EvE and the DIS. The structural requirements were divided into four sub-sections: EveService and Intelligence, EveService-Pool and Inter-EveSerice Interaction, Evolving Population, and Habitat Service and Targeted-Migration. The sub-section on EveService and Intelligence will be summarised, due to its relevance to the requirements of the DIS. The other sub-sections do not detail any specific requirements for the DIS design, therefore they will not be summarised here.

An EveService is a pointer to the service description of a DBE service. In addition to the service description, the EveService also contains other attributes, such as, a SMID (Sevice Manifest unique Identifier), a home Habitat-ID, a MUH, a failure counter, an escape counter, an intelligence component and a DIS migration counter.

The intelligence component of an EveService is required to be provided by the DIS implementation. This component will include the NN for pattern recognition between the BML descriptions of services within an EveService-Pool. It is required that the size of the input layer or the number if input neurons for the embedded NN is to be proportional to the BML description of the EveService. A single hidden layer is sufficient and usually larger than the input layer, as described in the high-level design deliverable. The output layer should consist of a single neuron with a range of 0 to 1. The output neuron will be used to determine similarity, where a threshold value will be used to decide whether a match is successful or not. It is stated that the greatest challenge of the DIS implementation will be the pre-processing of the BML descriptions to be used with the neural networks.

The behavioural requirements for the DIS outlined in the previously mentioned document can be summarised as follows. The DIS is triggered when an EveService is deployed to an EveService-Pool. The intelligence component, i.e. the NN, of that EveService is then to be initialised by the DIS. Following this initialisation step the DIS is activated and the matching process begins. If any successful matches are found then the DIS will trigger a targeted migration of a copy of the newly deployed EveService. This will lead to additional matching processes at the new targeted EveService-Pool, and potentially more targeted migration.

Design of the DIS

This section will outline the design of the DIS implementation which is based on the architectural requirements as defined in the high-level design specification deliverable. This design is described in the following sub-sections from a component viewpoint.

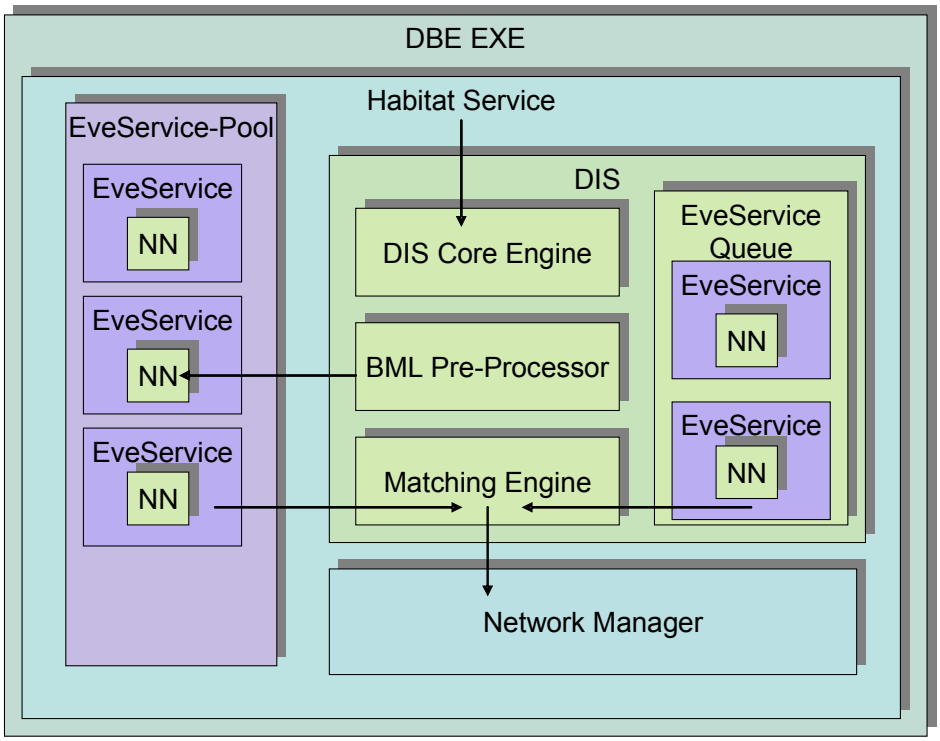


Figure 1: DIS Components and Associated Habitat Components

The design of the DIS is divided into four logical components: DIS Core (Core Engine and EveService Queue), DIS Matching, BML Pre-processor, and Embedded NN. These components will be described in the following sub-sections.

DIS Core

This component includes the core parts of the DIS implementation such as the DIS core engine and an EveService queue. The core engine is the central component which brings together all the features of the DIS. Once the core engine is initialised then all the other DIS components can be activated in conjunction with external EvE components. The core engine uses the EveService queue to provide EveServices to an instance of a matching engine. The core engine will create new instances of a matching engine by using a matching engine factory. The EveService queue provides a queuing system for newly added EveServices to the EveService-Pool. EveServices are removed from the queue on a first-in-first-out basis and forwarded for DIS processing.

DIS Matching

The DIS Matching component includes a framework for creating and integrating various matching engines for the DIS. This allows the DIS to be independent of a particular pattern recognition mechanism, such as neural networks. This design aspect was included because from the architectural requirements it was only suggested that using embedded neural networks would be a good option for the DIS. It was possible that another matching approach may have needed to be introduced at a later stage.

The matching framework provides a generic interface to a matching engine which all implementing engines will inherit. A factory is also provided for creating specific matching engines. The role of the matching engine is to analyse a given EveService and compare its BML description with that of the BML description from all of the other EveServices present in the local EveService-Pool. In the case of the NN matching engine, the embedded NN of the incoming EveService is used to process the pre-processed BML data of the existing EveServices contained within the EveService-Pool. The pre-processed data is provided by the BML Pre-processor which will be discussed in more detail in the next section. Once the matching process is complete and if the result is a successful match then the targeted migration process is initiated. Otherwise, the matching engine will stop. All matching engines are created as individual threads, and are started once the matching process is requested by the DIS core. Once the matching process is completed for the incoming

EveService and all targeted migrations, if any, have concluded then the matching engine thread stops. The maximum number of active matching threads can be configured to enable the most optimal resource processing which may vary depending of the frequency of deployed EveServices to a particular EveService-Pool.

The targeted migration occurs after a successful match. The matching engine will use the usage history of the both EveServices to determine the destination Habitats where the EveServices should be migrated to. The matching engine uses the Habitat service's network manager component to perform these migrations.

BML Pre-Processor

The role of the BML Pre-Processor component is to process a BML description into a format that can be used by the embedded intelligence. Specifically, it will provide the input data required for the embedded NN. The design of the BML Pre-Processor is derived from the appendix document, 'BMLv1 Pre-Processing For Neural Network Matching', which is contained in the high-level design specification deliverable [2]. In earlier revisions of the high-level deign of the DIS, it was assummed that the DIS and EvE would be based on BML version 2 (SBVR), but during the implementation of the DIS it was realised that this would not be supported in time for a successful DIS implementation. It was therefore agreed that BMLv1 would be supported by the DIS and EvE. The pre-processing algorithm was redesigned to support BMLv1 and is based on the assumption that the matching mechanism uses a NN implementation.

The use of BMLv1 means that both the BML model and BML data are required as input data. Both the BML model and BML data parts are contained within the SM of a DBE service. Given that EveServices are also represented by SMs then the BML parts could be extracted from the SM. BMLv1 has many sections including Semantic Service Language (SSL), Business Organisation and Business Process. The only relevant part to the DIS is Business Organisation. This includes elements such as Service and Product. These and their respective BML data parts will provide the required business terms that describe a service.

For pattern recognition an NN implementation usually requires a set of input data for training and for matching. The way this data is processed by the NN will be explained in more detail in the next section. For the explanation of the pre-processing, suffice to

say a set of input data in the form of numbers is required. These numbers could be in binary or ASCII, depending on the implementation of the NN.

The first step in the BML preprocessing algorithm is to extract the relevant business terms from the BML model and data descriptions. These descriptions are elements within a SM. As discussed earlier, the important parts of the BML model and data descriptions for the DIS is the Service and Product sub-elements within the BML Organisation element. The definition of these elements are declared within the BML model and the respective element values are contained within the BML data description. A customised XML parser is required to extract the terms as the BML model and data parts are contained separately within a SM and with a different XML structure. The extracted terms are then returned from the parser to the preprocessor as a set of strings.

The second step of preprocessing is to take the set of extracted terms and alphabetically order the terms. Also other processes are required to simplify the ordered set of terms, such as converting all uppercase letters to lowercase letters, and removing any spaces and irrelevant characters (e.g. commas).

The third step in the algorithm requires the standardisation of the length of all business terms. The design document suggests using the average term length or median term length (MTL) for Business English. This MTL for Business English is six characters while for English, the MTL is just under nine characters. All terms that do not fit the desired MTL will either have to be shortened by removing surplus characters or increased by adding a padding character. The padding character should be a neutral encoding value so as to not indirectly affect the outcome of the matching process.

The fourth step is to convert the set of terms into a format that the NN will be able to interpret. The BML preprocessing algorithm [2] proposes that the ordered business terms should be encoded into a binary format. As NN implementations require a set of numbers then this binary approach fits the requirements for preprocessed data. ASCII encoding, which provides 8-bit encoding per character, was selected for the encoding strategy. After some initial tests with the chosen NN engine, it was discovered that the format of the set of input data was of the Java type double, i.e. real or integer numbers. Given that the ASCII encoding of characters would provide an

integer representation for each character then it was decided that it was no longer necessary to perform the further step of converting the numbers to binary. ASCII integers would be used as the representation format. Initially it was considered that each term would be used as an input, but this was to be proven impractical as BML terms are open to constant change and it did not make sense to associate a numeric value with each term. Therefore, each character in each term would be used as an input value within the set of inputs for the NN.

Embedded Neural Network

The distributed part of the DIS consists of mainly the embedded NN within an EveService. This encapsulated intelligence is initiated when the EveService is first created and is therefore distributed throughout the Habitat network as the EveService is migrated to connected Habitats. The purpose of the embedded NN is to provide pattern recognition of BML descriptions between the hosting EveService and other EveServices within an EveService-Pool. As stated in the previous section, the NN requires a set of input data. This input data is required to be a numerical representation of the BML descriptions from a SM. The BML Preprocessor converts each character from each extracted business term into a numerical format.

The design of the NN consists of three layers, an input layer, a hidden layer and an output layer. The input layer, which represents the number of input neurons, is proportional to the set of characters from the extracted business terms. A single hidden layer was included, as suggested by the high level design document [2], although additional hidden layers could be added depending on the desired functionality of the NN. The hidden layer is usually significantly larger than the input layer. Therefore, it was decided to have the hidden layer equal to 1.5 times the size of the input layer. The output layer consists of a single neuron, which will vary between 0 and 1 depending on the success of matching the two sets of data. The NN executes a sigmoid transfer function to train and decide pattern similarities. The links between the input layer and hidden layer neurons are defined by weights where these weights are randomly initialised and then trained to the real numbers that provide the desired output [2]. Figure 2 shows the structure of the embedded NN as described above.

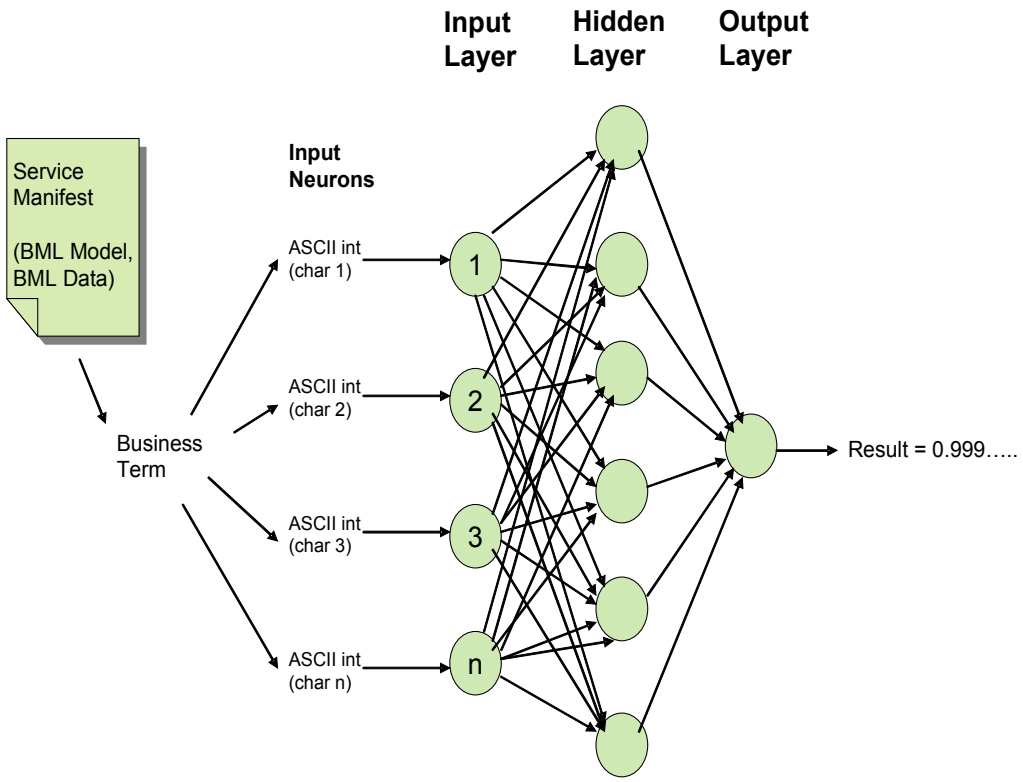


Figure 2: Neural Network Structure for Embedded Intelligence

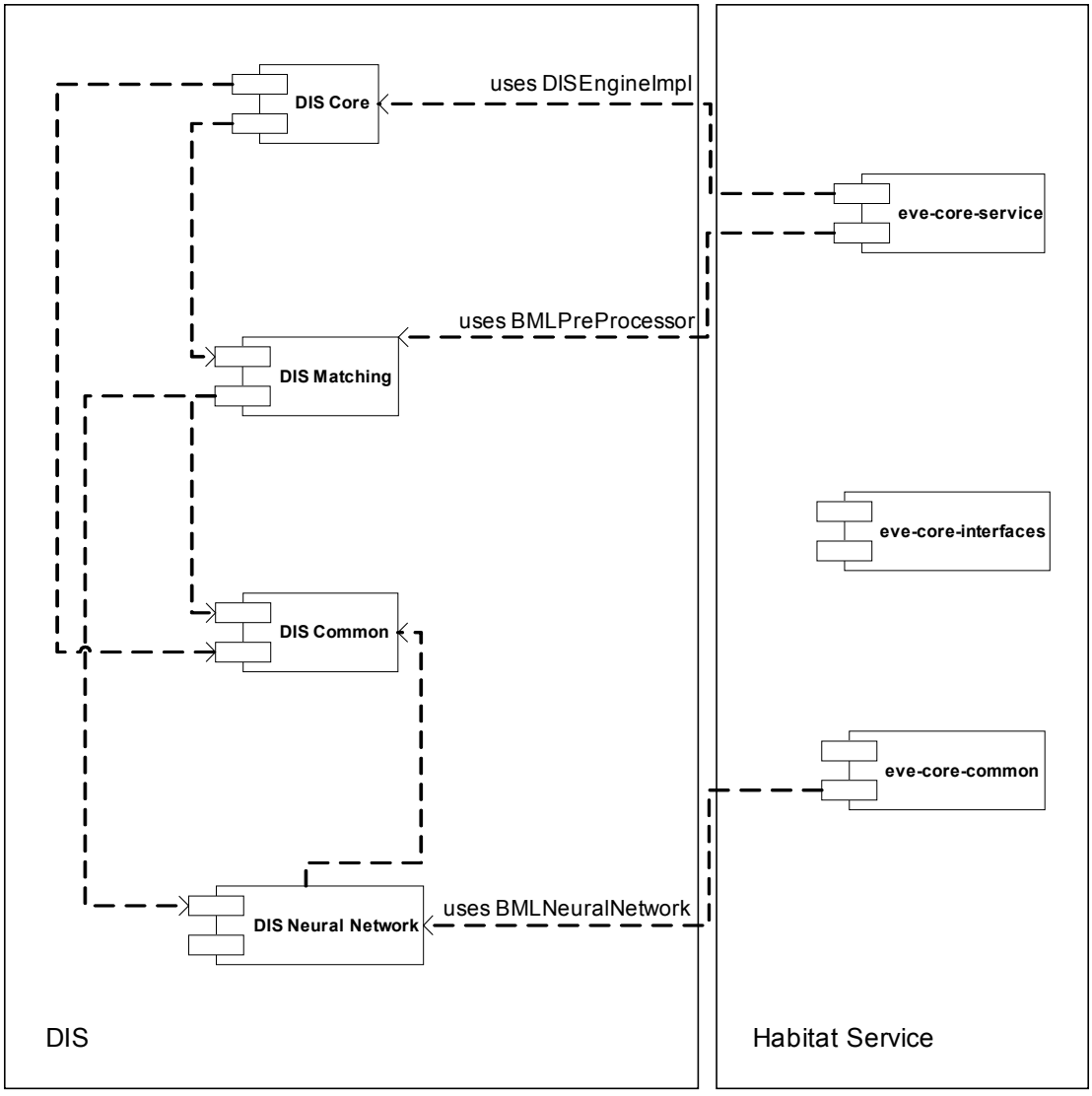
The NN is designed for two operations: to be trained with the input data from the hosting EveService and to compare a set of data from another EveService. The training process is executed when the NN is created. The preprocessed data from the hosting EveService is used as the initial set of input neurons. This is called the training data. The desired output value must also be provided with the training data. This value will be 1.0, as this is the desired output when the same data is inputed into the NN. After some initial tests with the chosen NN engine, it was discovered that additional training data was required to properly train the NN. This was because the NN was only able to distinguish similar sets of data but it was not able to recognise dissimilar data sets. Therefore, the NN was designed to include a training factor which would use the initial training data and modify the inputs and output incrementally with this factor constant. Following this, a full set of training data was available with multiple sets of input data and their corresponding output results.

The second operation of the NN is to compare a new set of input data with its trained intelligence and provide a resultant output. This output is taken from the output layer of one neuron, and is within the range of 0 and 1. A result close to 1 would demonstrate a close similarity to the hosting EveService and a result close to 0 would demonstrate a large difference between the input data set and that which was used to

train the NN initially. A threshold value is required to define whether a match was successful or not. As suggested in the high level design document [2], a threshold value of 0.9 was initially choosen. The implementation, however, allows for this threshold value to be configurable if it required to increase or decrease it depending on desired performance of the matching process.

4. Implementation and Integration

This chapter will describe the implementation of the DIS and its integration with the EvE implementation, in particular the Habitat service. The DIS has been implemented into four separate components, which are DIS Common, DIS Core, DIS Matching and DIS Neural Network. The implementation of each component will be described in detail in addition to the integration between these components and the Habitat service implementation. Figure 3 shows the relationships between DIS components and others within the EvE implementation.



Note: DIS Core, Matching and Common all have a dependency on Eve Core Interfaces but it is not shown in the diagram

Figure 3: DIS Component Diagram

DIS Common

This component provides a set of interfaces and common utility classes to be used by other DIS components and external implementations. As shown in Figure 4, DIS Core and DIS Common Class Diagram, DIS common currently consists of two interfaces, `DISEngine` and `MatchingEngine`, and one utility class, `DISException`. The interface, `DISEngine`, represents public interface of the DIS engine and has two methods. The `notify` method notifies the DIS engine that a new EveService has been added to the EveService-Pool. This method will be invoked by the Habitat service implementation. The method, `matchingSessionComplete`, is called by the matching engine when its matching process has ceased and the thread is finished. The `DISEngine` interface throws a `DISException` on both methods. The `DISException` class extends the `java.lang.Exception` class to provide a specific throwable exception for the DIS components. The second interface, `MatchingEngine`, represents the public interface for the matching engine. This interface has one method, `startEngine`, which takes the parameters of `EveService`, `ServicePool` (the implementation class for the EveService-Pool), `NetworkManager` and `DISEngine`.

DIS Core

The DIS Core component mainly provides the implementation of the `DISEngine` interface. This implementation is contained in the `DISEngineImpl` class, which implements the `DISEngine` interface. The implementation of the `notify` method adds the supplied `EveService` object to an instance of the `EveServiceQueue` class. This class extends the `java.util.Vector` class to provide a queue for all `EveService` objects to be processed by the DIS. When the `getNextEveService` method is called the `EveService` object will be returned to the engine and removed from the queue.

The `notify` method also calls a private method, `checkQforMatching`. This method is called both by the `notify` method, when a new EveServices arrives, and by the `matchingSessionComplete` method, when a matching engine thread finishes. The execution of this method checks to see if there are any EveServices

waiting in the queue, then if the queue is not empty or if the number of active matching threads is not greater than the specified maximum amount, it will start a new matching engine thread. The maximum number of active matching threads can be configured and the default is set to 1. This configuration value can be adjusted if the speed of DIS to match EveServices becomes more important than the overall processing limitations of an ExE node.

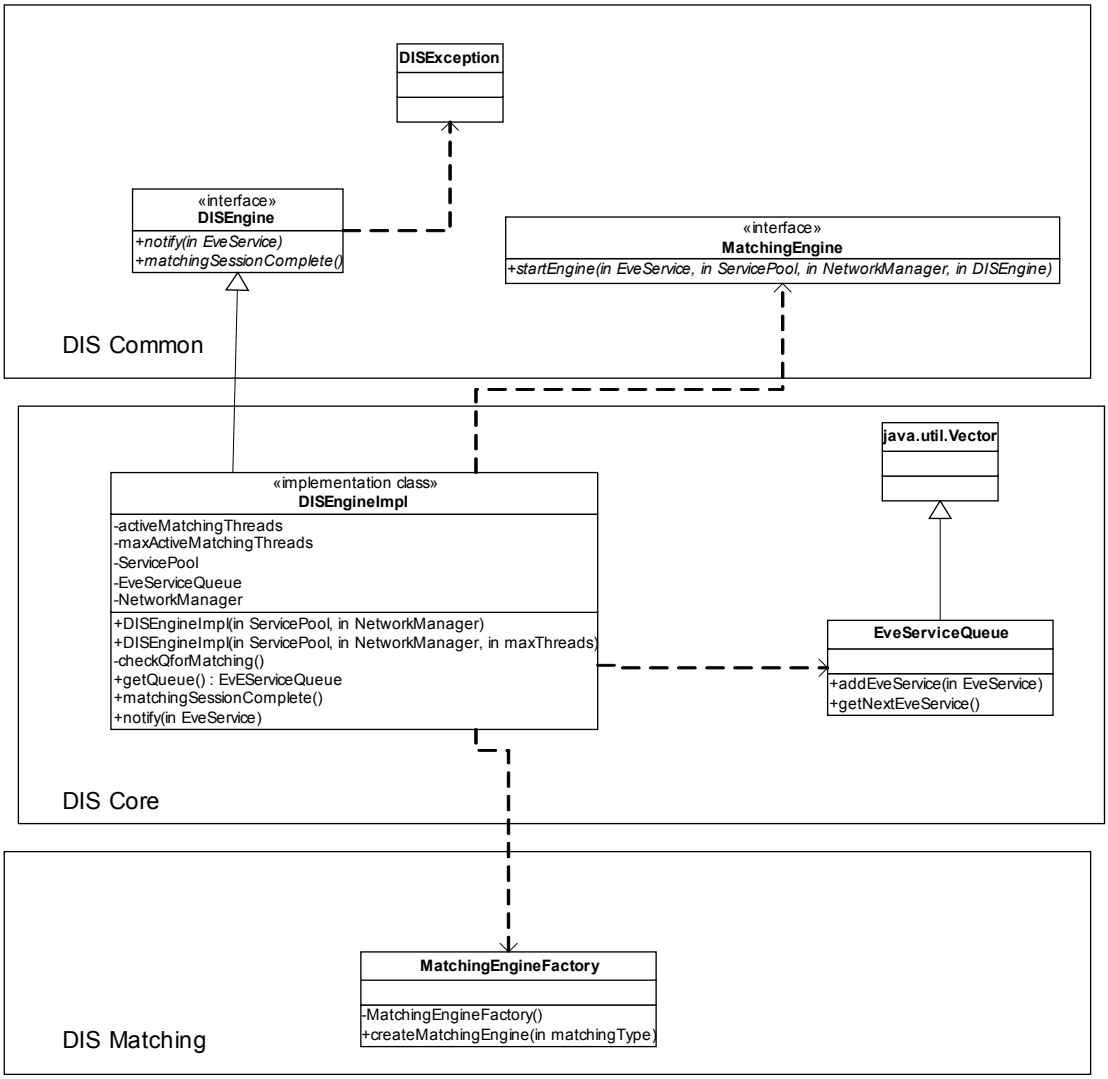


Figure 4: DIS Core and DIS Common Class Diagram

To start a new matching thread the DIS engine uses the matching engine framework or DIS Matching component to create new instances of a matching engine. The `MatchingEngineFactory` class provides a method to create a new instance of the `MatchingEngine` interface. The engine does not need to know what type of matching engine is created therefore it is independent of any matching engine implementations. This provides a pluggable architecture where matching engines can be substituted at runtime depending on performance requirements.

DIS Matching

The DIS Matching component includes a multi-threaded matching engine framework, for creating new instances of matching engine types, and the BML preprocessing implementation, for extracting and converting BML business terms to numeric value sets. Figure 5 shows the classes contained within this component.

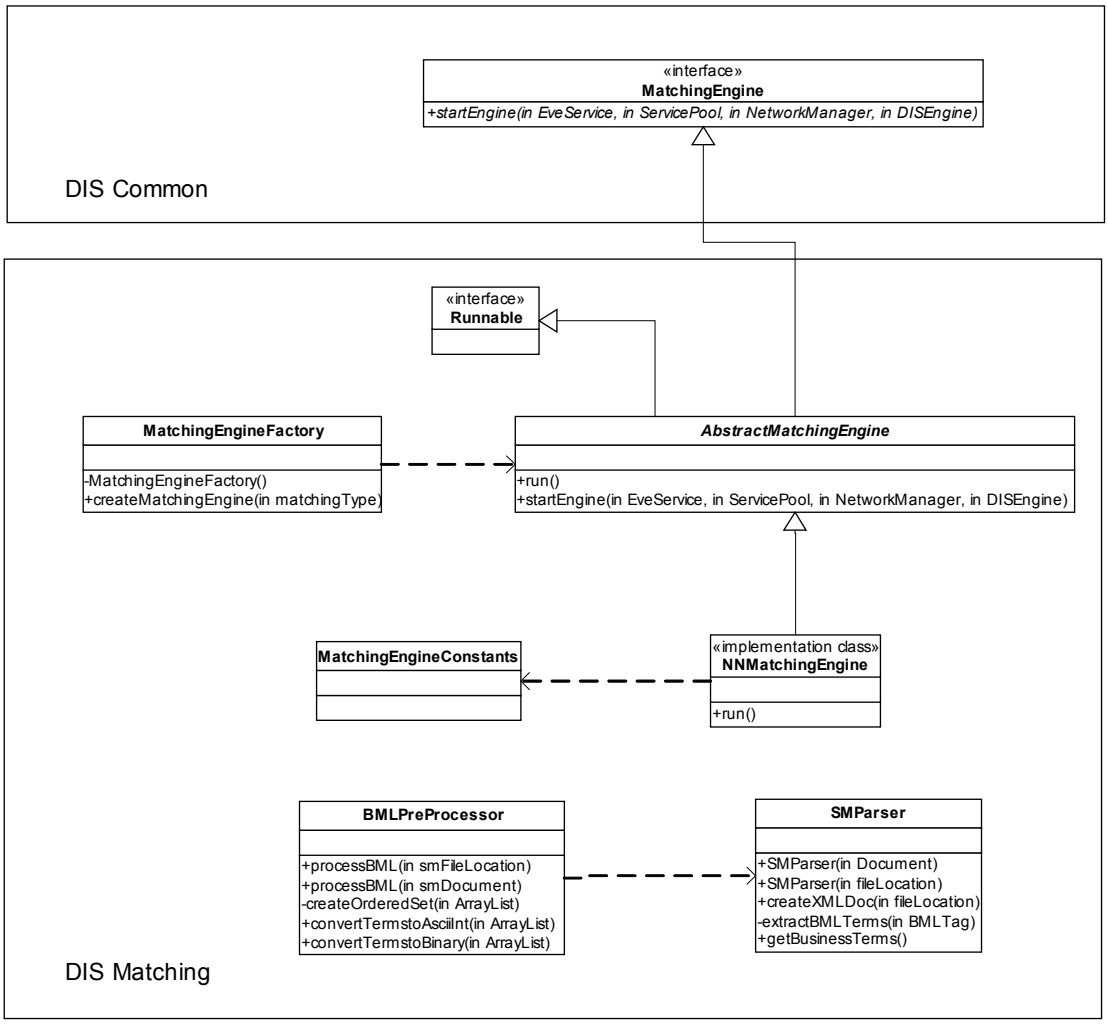


Figure 5: DIS Matching Class Diagram

The matching engine framework consists of a factory class, `MatchingEngineFactory`, which provides a static method to creating instances of `MatchingEngine` interface, as described previously in the DIS Common section. When this method is called the factory class will load a configuration property which specifies the matching engine class type and then create an instance of the class. The default matching type as implemented in this task in a NN engine. Other matching engine types could be implemented in the future if it makes more sense and these new engines can be easily integrated into the DIS because of the

design of this pluggable matching engine framework.

All implemented matching engines must extend the `AbstractMatchingEngine` class. This is an abstract class which implements the `MatchingEngine` and `Runnable` interfaces. It provides an implementation for the `startEngine` method which is declared as final so it cannot be overridden by any sub class. This method creates an instance of the `Thread` class and calls its `start` method, therefore starting the thread of the sub class. The sub class, which is the implementation class of the matching engine, just needs to provide an implementation for the run method. This will be called once the thread has been started. As the default implementation class, the `NNMatchingEngine` class provides an implementation of the matching engine which uses the NN mechanism for matching EveServices. Within the run method of the `NNMatchingEngine` class it executes the matching process using the embedded NN from the newly added EveService with all other EveServices present in the EveService-Pool. If a successful match is found, then it uses the `NetworkManager` class, from the Habitat service, to migrate both EveServices to other Habitats depending on the usage history from both matching EveServices.

Another aspect of the DIS Matching component is the BML preprocesing implementation. This is mainly provided by the `BMLPreProcessor` class. This class implements the BML preprocessing algorithm as described in Chapter 3. An instance of it is created by the Habitat service implementation when a new EveService is created. Here the accompanied SM with the EveService is included when invoking the `processBML` method. A SM parser, provided by the `SMParser` class, is uses to parse the SM for relevant BML model and data elements and then to extract the business terms which are returned to the BML preprocessor. The private method, `createOrderedSet`, performs the next step in the preprocessing algorithm to arrange the data set in alphabetical order while also removing any spaces or commas within the business terms. The methods `convertTermstoAsciiInt` or `convertTermstoBinary` are then used to maintain the MTL of each term and then convert their characters to a numerical value. By default, the ASCII Integer conversion is used but the binary conversion was also implemented. The preprocessor class returns an array of double values which represent each character in the extracted business terms. The ASCII integer values of lowercase letters range from 97 to 122

while the values of numbers characters range from 48 to 57. From some initial tests with the NN implementation it was discovered that because the difference between the ASCII values of letters was small, the NN provided unpredictable answers. To widen the difference between the input data values, each integer value was squared, therefore giving a wider range of values.

As the preprocessing task is required before an EveService can be matched, then after the initial preprocessing is complete, the processed data is stored within the EveService. This preprocessed data can be extracted by the matching engine to compare it against another EveService. In the case of the NN matching engine, the preprocessed data is feed into the embedded NN of the other EveService to distinguish whether a match is successful.

DIS Neural Network

This component mainly consist of the implementation of a NN for the DIS. This implementation is provided in the `BMLNeuralNetwork` class. The methods of this class are shown in Figure 6. This class is designed to be embedded within an `EveService` object which can be cloned and migrated around the Habitat network. Due to the distributed characteristic required by this object it implements the `java.io.Serializable` interface. This enables for the object to be serialised when distributed between Habitat nodes.

The `BMLNeuralNetwork` class also implements the `NeuralNetListener` interface. This interface is provided by the Joone [5] project. Joone is an open source project that provides a NN framework for creating, training and testing artificial neural networks. This framework has a modular architecture of pluggable components with features for persistence, multithreading, serialization and parameterisation. The Joone engine is implemented in Java and provides an API that developers can implement, train and execute NNs. This engine was chosen as the NN engine for the DIS. The main class used from the Joone framework is the `NeuralNet` class. This object represents a container of a NN where the NN can be managed or persisted for further use [6]. Using the `NeuralNet` object, the NN can easily be transported to remote machines and executed by simple Java applications. The `NeuralNetListener` provides methods to enable an application to listen for

events triggered within the NN. This is why the `BMLNeuralNetwork` class implements the `NeuralNetListener` interface. The NeuralNet object is created within the `BMLNeuralNetwork` class, which then listens for events from the `NeuralNet`.

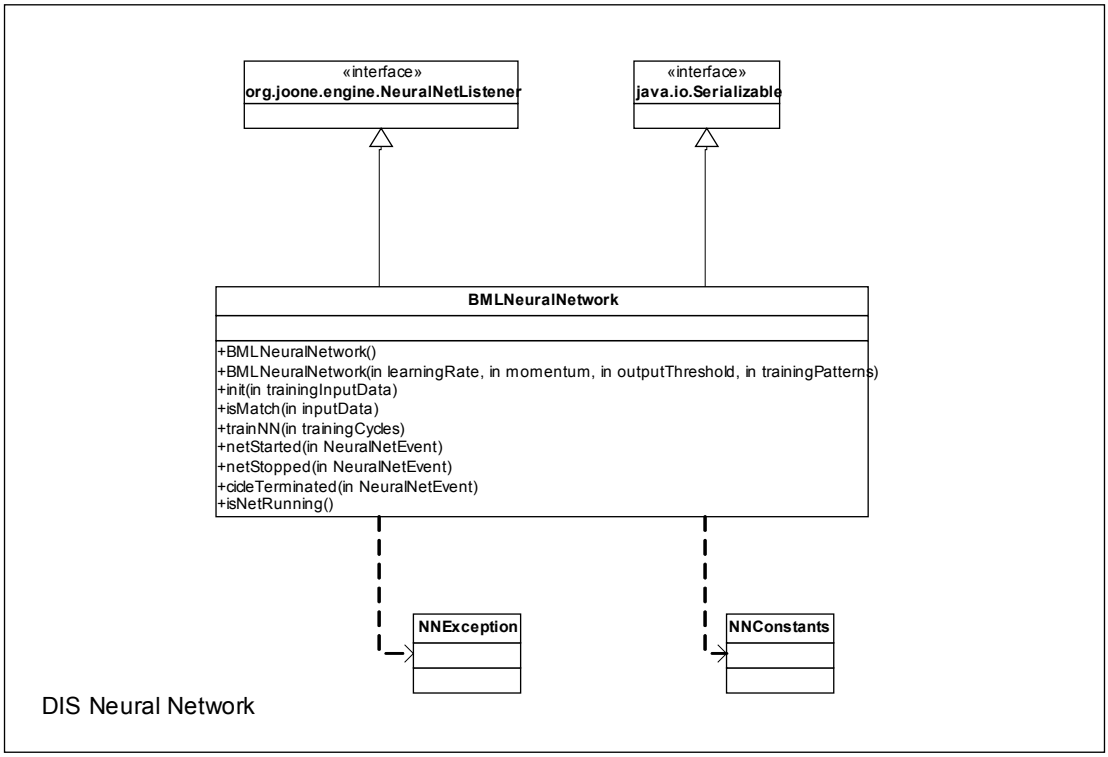


Figure 6: DIS Neural Network Class Diagram

The `BMLNeuralNetwork` class has three important methods: `init`, `trainNN` and `isMatch`. The `init` method is first called to create the NN structure and initialise the `NeuralNet` object. It takes the array of training data as an input parameter. The three layers of the NN structure are created with the help of the Joone framework. The input layer is created with a `LinearLayer` class. The linear layer is a simple layer where the input data is transferred to the output by applying a linear transform. Is it commonly used as a buffer, sending unmodified data to the hidden layers [6]. The second layer in the DIS NN structure is the only hidden layer. This is created with the `SigmoidLayer` class. The sigmoid layer applies a sigmoid transfer function to the input data used to build the hidden layers of NN with non-linear elements [6]. The output layer also is created with the `SigmoidLayer` class. These layers are then added to the initialised `NeuralNet` object in addition to a `TeachingSynapse` object. This class is used during the training process to teach the NN to apply a desired output to a particular input. The `trainNN` method sets the number of training

cycles and patterns, and then starts the `NeuralNet` object. Once the training of the NN is complete the `netStopped` event method is called. The `isMatch` method is used during the matching process, where it takes a set of preprocessed input data from a different EveService and runs this data through the previously trained NN. The input data is first adapted to fit the size to the training data, which is required by the Joone engine. The `NeuralNet` object is started again and the resultant output is then extracted. This method returns a `boolean` true or false depending whether the return output value, which is one neuron, is greater or equal to the output threshold value. This threshold value is configurable through a properties file.

Habitat Integration

The DIS components are highly integrated with core components from the Habitat service implementation. Although the internals of the DIS are mostly independent of the Habitat service, all external interaction of the DIS is executed with Habitat service. The first integration point between the Habitat and DIS, is when a new EveService is created. Here the BML description of this EveService must be preprocessed before it is added to the EveService-Pool. The DIS provides the BML preprocessing features to convert the business description into numerical values. The second integration point occurs when the a new NN is created and trained. This is initiated by the Habitat service which then embeds the trained NN into the EveService. Figure 7 shows a sequence diagram of the previous two integration steps.

The next set of integration points happen just before an EveService is added to EveService-Pool. The Habitat service activates the DIS engine and notifies it that a new EveService is available for processing. This EveService is then added to the DIS EveService queue where it waits to be processed by a matching engine thread. The matching engine extracts the embedded NN from a new EveService to perform the pattern recognition. If a successful match is found, then the matching engine will use the `NetworkManager` class, provided by the Habitat implementation, to migrate the EveServices to other Habitats based on both on their usage histories. Figure 8 shows a sequence diagram of the DIS activation and matching process.

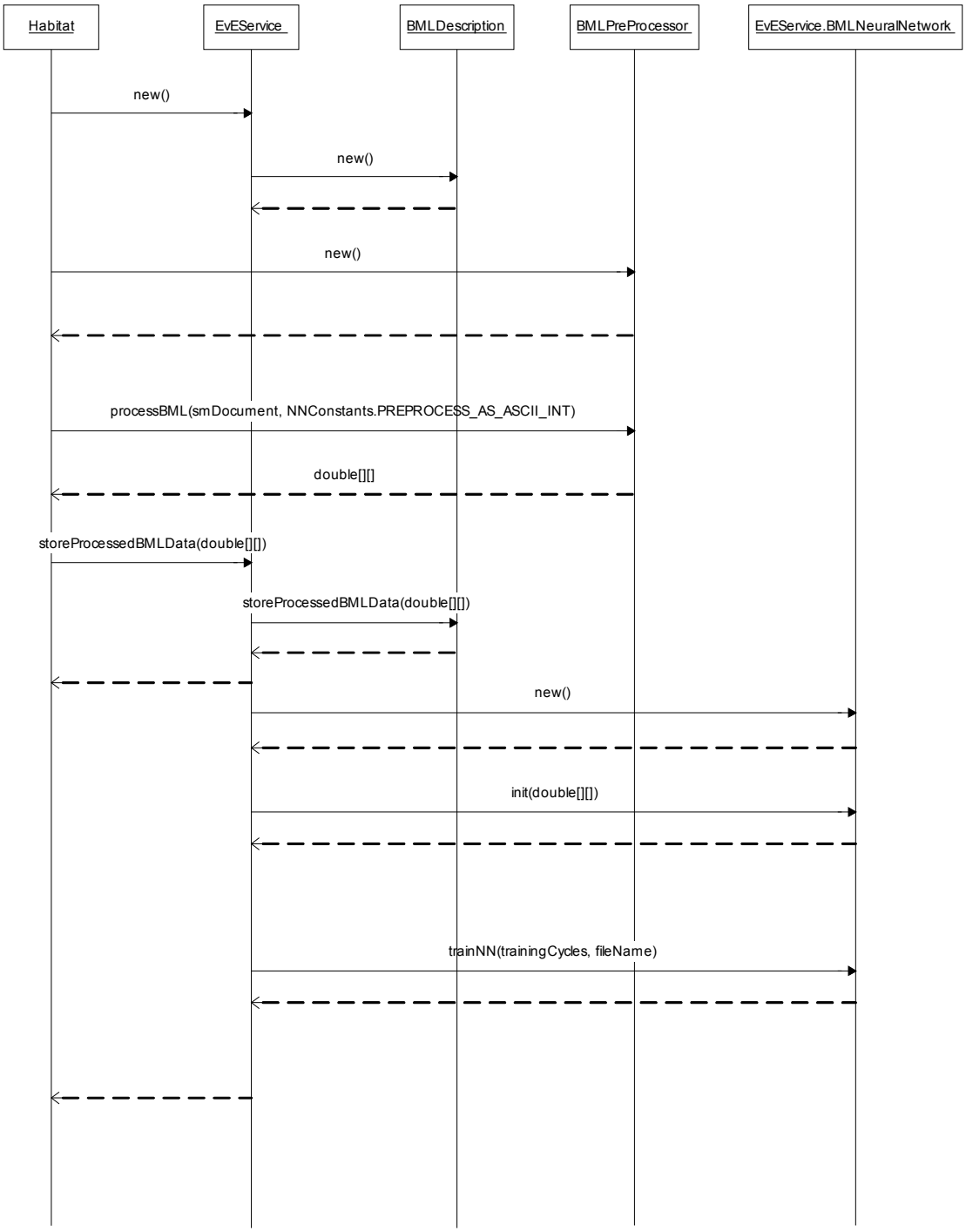


Figure 7: Sequence Diagram for DIS Preprocessing and NN training

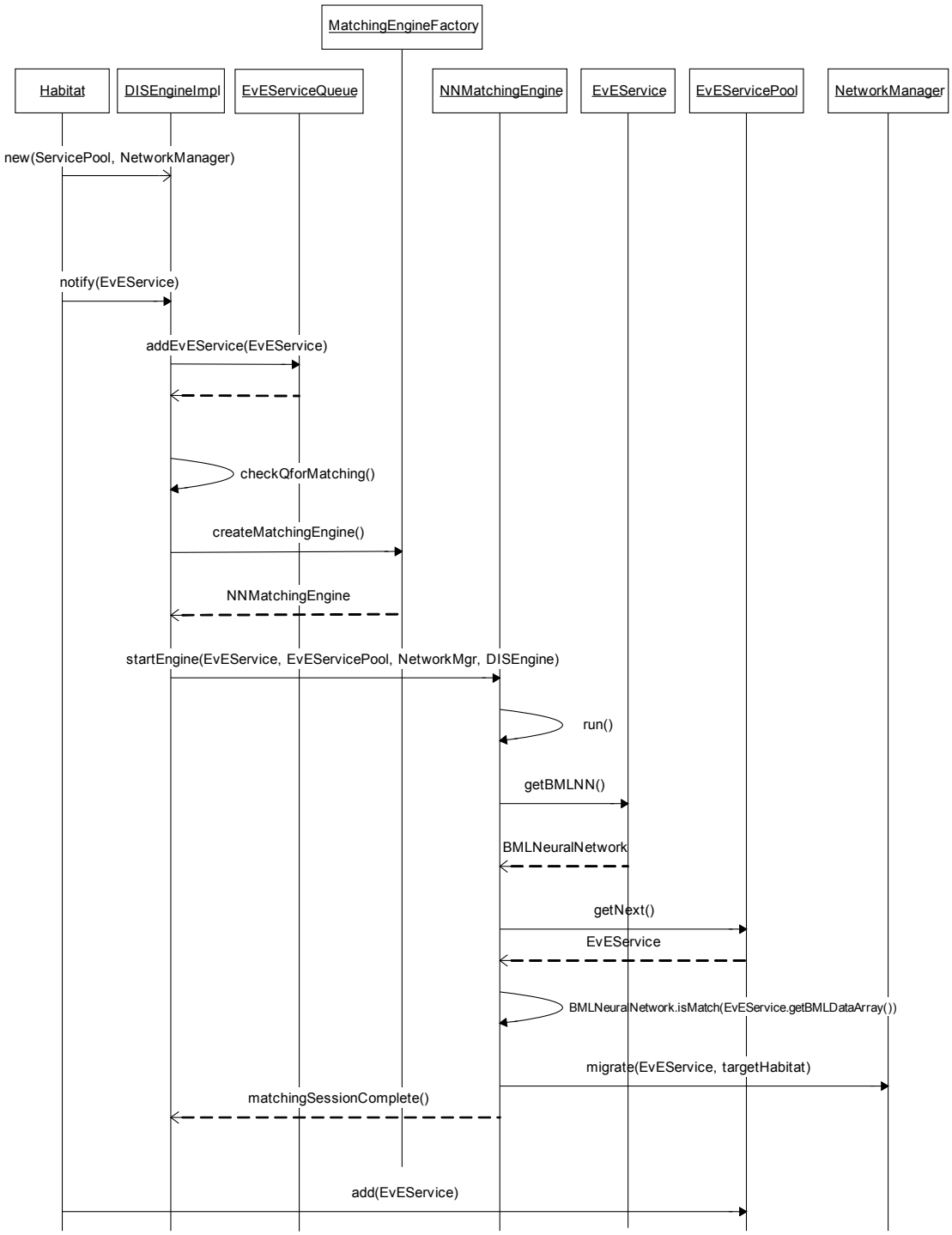


Figure 8: Sequence Diagram of DIS Activation and Matching Process

5. Conclusion and Future Work

This document described the implementation of the DIS beginning from the architectural requirements to the component design and then followed by the implementation specifics of each internal component. The architectural requirements were taken from the 'High-Level Design Specification of the Distributed Intelligence System' deliverable, D6.6 [2]. These requirements, which included a high level design, enabled the creation of a component design for the DIS. Initially, the DIS was designed to support the preprocessing and matching of SBVR (BML version 2.0) descriptions, but due to project adjustments the general EvE implementation was moved to support BML version 1.0. This change required some extra work to modify the preprocessing implementation but due to the loosely coupled design of the DIS the remaining components were unaffected.

The adopted approach for service matching was provided by an implementation of a neural network. This implementation of the NN component involved the integration and testing of an open source Java based NN engine, called Joone [5]. A large part of the implementation task was required for the development of the BML preprocessing algorithm and the embedded NN component. Although, NNs were chosen as the matching mechanism for the DIS, a matching engine framework was design and developed to enable the future integration of alternative matching processes if required by the DIS.

Future work could include the continuation of the SBVR preprocessing and the integration of this business modeling language within the EvE and the DBE as a whole. The matching mechanism provided by the NN engine could be analysed to devise a better design for improved performance in real world networks. Also, other matching engines could be implemented and easily integrated into the DIS matching component.

6. Glossary

BML Business Modelling Language

DBE Digital Business Ecosystem

DEC Distributed Evolutionary Computing

DIS Distributed Intelligence System

EvE Evolutionary Environment

ExE Execution Environment

MAS Multi-Agent Systems

MTL Median Term Length

MUH Migration and Usage History

NN Neural Network

P2P Peer-to-Peer

SBVR Semantics of Business Vocabulary and Business Rules

SDL Service Description Language

SM Service Manifest

SME Small and Medium Enterprises

SMID Service Manifest Identifier

SSL Semantic Service Language

7. References

1. Gerard Briscoe. Evolutionary Environment Architecture Requirements. Internal Document, April 2005.
2. Gerard Brisoce and Philippe De Wilde. D6.6 High-Level Design Specification of the Distributed Intelligence System, December 2005.
3. EvEnet. http://evenet.sourceforge.net
4. Heaton Research. Understanding Neural Networks, November 2005, http://www.heatonresearch.com/articles/1/page2.html
5. Joone Project. http://www.jooneworld.com/
6. Paolo Marrone. The Joone Complete Guide, February 2005

Appendix A: Source Code Java Doc

Package Summary		Page
org.dbe.eve.dis.common		33
org.dbe.eve.dis.core		37
org.dbe.eve.dis.matching		43
org.dbe.eve.dis.matching.nn		51
org.dbe.eve.dis.matching.nn.util		55
org.dbe.eve.dis.nn		57
org.dbe.eve.dis.nn.util		65

Package org.dbe.eve.dis.common

Interface Summary		Page
DISEngine	DISEngine is the Interface of the implementation of the DIS engine.	33
MatchingEngine	This is the interface of the implementation of the DIS matching engine.	36

Exception Summary		Page
DISException	This class implements a DIS core exception.	35

Interface DISEngine

org.dbe.eve.dis.common

All Known Implementing Classes:

DISEngineImpl

---

public interface **DISEngine**

DISEngine is the Interface of the implementation of the DIS engine. This interface is to be used by the HabitatService to notify the DIS engine.

Author:

David McKitterick, Intel Ireland

Method Summary	Page

void	**matchingSessionComplete**() This method is called by the matching engine when the matching process is complete.	*34*
void	**notify**(org.dbe.eve.EvEService service) This method notifies the DIS engine that a eve service has been added to the service pool.	*34*

Method Detail

notify

public void **notify**(org.dbe.eve.EvEService service)
throws DISException

This method notifies the DIS engine that a eve service has been added to the service pool.

Parameters:

`service` - the eve service recently added to the service pool.

Throws:

DISException

Exception

See Also:

`org.dbe.eve.EvEService`

---

matchingSessionComplete

public void **matchingSessionComplete**()
throws DISException

This method is called by the matching engine when the matching process is complete.

Throws:

DISException

Exception

Class DISException

org.dbe.eve.dis.common

```
java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
        └ org.dbe.eve.dis.common.DISException
```

All Implemented Interfaces:

Serializable

---

public class **DISException**

extends Exception

This class implements a DIS core exception.

Author:

David McKitterick, Intel Ireland

Constructor Summary	*Page*
DISException() Constructor 1	*35*
DISException(String message) Constructor 2	*35*

Constructor Detail

DISException

```
public DISException()
```

Constructor 1

DISException

```
public DISException(String message)
```

Constructor 2

Interface MatchingEngine

org.dbe.eve.dis.common

All Known Implementing Classes:

AbstractMatchingEngine, NNMatchingEngine

---

public interface **MatchingEngine**

This is the interface of the implementation of the DIS matching engine. All matching engine must implement this interface.

Author:

David McKitterick, Intel Ireland

Method Summary		Page
void	**startEngine**(org.dbe.eve.EvEService service, org.dbe.eve.ServicePool pool, org.dbe.eve.net.NetworkManager nwtMgr, DISEngine dis) Starts a new thread of the matching engine.	36

Method Detail

startEngine

public void **startEngine**(org.dbe.eve.EvEService service,
org.dbe.eve.ServicePool pool,
org.dbe.eve.net.NetworkManager nwtMgr,
DISEngine dis)

Starts a new thread of the matching engine.

Parameters:

`service` - the eve service to be matched

`pool` - a service pool

`nwtMgr` - a network manager

`dis` - a DIS engine

Package org.dbe.eve.dis.core

Class Summary		Page
DISCoreConstants	This class provides constants for the matching engine classes.	37
DISEngineImpl	DISEngineImpl is the implementation class of the DIS engine.	38
EvEServiceQueue	The EvEServiceQueue class implements an EvE service queue object which extends the Vector class.	40

Class DISCoreConstants

org.dbe.eve.dis.core

```
java.lang.Object
  └ org.dbe.eve.dis.core.DISCoreConstants
```

---

final public class **DISCoreConstants**
extends Object

This class provides constants for the matching engine classes.

Author:

David McKitterick, Intel Ireland

Field Summary		Page
static String	MAX_MATCHING_THREAD_PROP	38
static String	PROP_FILENAME	37

Constructor Summary	Page
DISCoreConstants()	38

Field Detail

PROP_FILENAME

```
public static String PROP_FILENAME
```

MAX_MATCHING_THREAD_PROP

```
public static String MAX_MATCHING_THREAD_PROP
```

Constructor Detail

DISCoreConstants

```
public DISCoreConstants()
```

Class DISEngineImpl

org.dbe.eve.dis.core

```
java.lang.Object
  └ org.dbe.eve.dis.core.DISEngineImpl
```

All Implemented Interfaces:

DISEngine

---

public class **DISEngineImpl**
extends Object
implements DISEngine

DISEngineImpl is the implementation class of the DIS engine. This class implements the DISEngine interface. This class is to be used by the HabitatService to start and notify the DIS engine.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page
DISEngineImpl(org.dbe.eve.ServicePool aPool, org.dbe.eve.net.NetworkManager aNwtMgr) Constructor.	39

Method Summary		Page
Properties	getDISProperties() Returns the DIS Properties object.	40

final EvEServiceQueue	**getQueue**() Returns an instance of the eve service queue.	*40*
final void	**matchingSessionComplete**() This method is called by the matching engine when the matching process is complete.	*40*
final void	**notify**(org.dbe.eve.EvEService newService) This method notifies the DIS engine that a eve service has been added to the service pool.	*39*

Methods inherited from interface org.dbe.eve.dis.common.DISEngine
matchingSessionComplete, notify

Constructor Detail

DISEngineImpl

```
public DISEngineImpl(org.dbe.eve.ServicePool aPool,
                     org.dbe.eve.net.NetworkManager aNwtMgr)
```

Constructor. Creates a new eve service queue.

Method Detail

notify

public final void **notify**(org.dbe.eve.EvEService newService)
throws DISException

This method notifies the DIS engine that a eve service has been added to the service pool.

Specified by:

notify in interface DISEngine

Parameters:

newService - the EveService recently added to the service pool.

Throws:

DISException

Exception

See Also:

org.dbe.eve.EvEService

matchingSessionComplete

public final void **matchingSessionComplete**()

throws DISException

This method is called by the matching engine when the matching process is complete.

Specified by:

matchingSessionComplete in interface DISEngine

Throws:

DISException

Exception

---

getQueue

public final EvEServiceQueue **getQueue**()

Returns an instance of the eve service queue.

Returns:

EvEServiceQueue

---

getDISProperties

public Properties **getDISProperties**()

Returns the DIS Properties object.

Returns:

Properties

Class EvEServiceQueue

org.dbe.eve.dis.core

java.lang.Object
  └ java.util.AbstractCollection
    └ java.util.AbstractList
      └ java.util.Vector
        └ **org.dbe.eve.dis.core.EvEServiceQueue**

All Implemented Interfaces:

Cloneable, Collection, Iterable, List, RandomAccess, Serializable

---

public class **EvEServiceQueue**
extends Vector

The EvEServiceQueue class implements an EvE service queue object which extends the Vector class.

Author:

David McKitterick, Intel Ireland

See Also:

`Vector`

Constructor Summary	Page
EvEServiceQueue()	41

Method Summary		Page
final boolean	**addEvEService**(org.dbe.eve.EvEService newService) This method adds a new eve service to the eve service queue.	41
final org.dbe.eve.EvEService	**getNextEvEService**() This method returns the first eve service object in the eve service queue and removes this object from the vector queue.	42

Constructor Detail

EvEServiceQueue

```
public EvEServiceQueue()
```

Method Detail

addEvEService

```
public final boolean addEvEService(org.dbe.eve.EvEService newService)
```

This method adds a new eve service to the eve service queue.

Parameters:

`newService` - a new eve service

---

getNextEvEService

```
public final org.dbe.eve.EvEService getNextEvEService()
```

This method returns the first eve service object in the eve service queue and removes this object from the vector queue.

Returns:

EvEService

Package org.dbe.eve.dis.matching

Class Summary		Page
AbstractMatchingEngine	The AbstractMatchingEngine class implements the MatchingEngine and Runnable interfaces.	43
MatchingEngineConstants	This class provides constants for the matching engine classes.	45
MatchingEngineFactory	The MatchingEngineFactory class implements a factory for creating instances of the AbstractMatchingEngine class.	47
SMParser	The SMParser class for parsing a service manifest, as a Document object or xml file, and extract the business terms from the BML model and data elements.	48

Class AbstractMatchingEngine

org.dbe.eve.dis.matching

```
java.lang.Object
  └ org.dbe.eve.dis.matching.AbstractMatchingEngine
```

All Implemented Interfaces:

MatchingEngine, Runnable

Direct Known Subclasses:

NNMatchingEngine

---

abstract public class **AbstractMatchingEngine**
extends Object
implements MatchingEngine, Runnable

The AbstractMatchingEngine class implements the MatchingEngine and Runnable interfaces. This class provides an implementation for the startEngine() method which cannot be overridden by its sub classes.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page

AbstractMatchingEngine()	*44*

Method Summary		Page
final void	**startEngine**(org.dbe.eve.EvEService service, org.dbe.eve.ServicePool aPool, org.dbe.eve.net.NetworkManager nwtMgr, DISEngine dis) This method provides an implementation for the inherited startEngine() method.	*44*

Methods inherited from interface org.dbe.eve.dis.common.MatchingEngine
startEngine

Constructor Detail

AbstractMatchingEngine

```
public AbstractMatchingEngine()
```

Method Detail

startEngine

public final void **startEngine**(org.dbe.eve.EvEService service,
org.dbe.eve.ServicePool aPool,
org.dbe.eve.net.NetworkManager nwtMgr,
DISEngine dis)

This method provides an implementation for the inherited startEngine() method. Creates a new thread with this runnable instance. Sets the thread priority to minimum.

Specified by:

startEngine in interface MatchingEngine

Parameters:

`service` - the eve sesrvice to be matched

`aPool` - a service pool

`nwtMgr` - a network manager

`dis` - a DIS engine

Class MatchingEngineConstants

org.dbe.eve.dis.matching

```
java.lang.Object
  └ org.dbe.eve.dis.matching.MatchingEngineConstants
```

---

final public class **MatchingEngineConstants**
extends Object

This class provides constants for the matching engine classes.

Author:

David McKitterick, Intel Ireland

Field Summary		Page
static final String	BML_ATTRIBUTE_TAG	46
static final String	BML_DATA_OBJECT	47
static final String	BML_DATA_TAG	46
static final String	BML_DATA_VALUE	47
static final String	BML_PRODUCT_TAG	46
static final String	BML_SERVICE_TAG	46
static final String	BML_TAG	46
static final int	DEFAULT_TERM_LENGTH	47
static final String	MATCHING_TYPE_PROP	45
static final String	NN_MATCHING_TYPE	46
static final char	PADDING_CHAR	47
static final String	PREPROCESS_AS_ASCII_INT	46
static final String	PREPROCESS_AS_BINARY	46
static final String	PREPROCESS_TYPE_PROP	46

Constructor Summary	Page
MatchingEngineConstants()	47

Field Detail

MATCHING_TYPE_PROP

```
public static final String MATCHING_TYPE_PROP
```

NN_MATCHING_TYPE

```
public static final String NN_MATCHING_TYPE
```

PREPROCESS_TYPE_PROP

```
public static final String PREPROCESS_TYPE_PROP
```

PREPROCESS_AS_BINARY

```
public static final String PREPROCESS_AS_BINARY
```

PREPROCESS_AS_ASCII_INT

```
public static final String PREPROCESS_AS_ASCII_INT
```

BML_TAG

```
public static final String BML_TAG
```

BML_DATA_TAG

```
public static final String BML_DATA_TAG
```

BML_SERVICE_TAG

```
public static final String BML_SERVICE_TAG
```

BML_PRODUCT_TAG

```
public static final String BML_PRODUCT_TAG
```

BML_ATTRIBUTE_TAG

```
public static final String BML_ATTRIBUTE_TAG
```

BML_DATA_OBJECT

```
public static final String BML_DATA_OBJECT
```

BML_DATA_VALUE

```
public static final String BML_DATA_VALUE
```

DEFAULT_TERM_LENGTH

```
public static final int DEFAULT_TERM_LENGTH
```

PADDING_CHAR

```
public static final char PADDING_CHAR
```

Constructor Detail

MatchingEngineConstants

```
public MatchingEngineConstants()
```

Class MatchingEngineFactory

org.dbe.eve.dis.matching

```
java.lang.Object
  └ org.dbe.eve.dis.matching.MatchingEngineFactory
```

---

final public class **MatchingEngineFactory**
extends Object

The MatchingEngineFactory class implements a factory for creating instances of the AbstractMatchingEngine class.

Author:

David McKitterick, Intel Ireland

Method Summary		Page
static AbstractMatchingEngine	**createMatchingEngine**(Properties disProperties) Returns an instance of the AbstractMatchingEngine class.	48

Method Detail

createMatchingEngine

public static AbstractMatchingEngine **createMatchingEngine**(Properties disProperties) throws DISException

Returns an instance of the AbstractMatchingEngine class. Requires a matching type to decide which engine to type to create. This is taken from the DIS properties settings. The default matching engine is the Neural Network matching engine.

Parameters:

`disProperties` - a Properties object which contains all dis properties

Returns:

AbstractMatchingEngine

Throws:

DISException

Class SMParser

org.dbe.eve.dis.matching

```
java.lang.Object
  └ org.dbe.eve.dis.matching.SMParser
```

---

public class **SMParser**
extends Object

The SMParser class for parsing a service manifest, as a Document object or xml file, and extract the business terms from the BML model and data elements.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page

SMParser(String smFile) Constructor 2.	*49*
SMParser(org.w3c.dom.Document sm) Constructor 1.	*49*

Method Summary		Page
org.w3c.dom.Document	**createXMLDoc**(String xmlFile) Reads in a file containing XML and creates a DOM of it's content.	*50*
ArrayList	**getBusinessTerms**() Returns an ArrayList object of the extracted BML model and data terms.	*49*

Constructor Detail

SMParser

```
public SMParser(org.w3c.dom.Document sm)
```

Constructor 1. Takes a Document object as a parameter

SMParser

```
public SMParser(String smFile)
```

Constructor 2. Takes a file name as a parameter

Method Detail

getBusinessTerms

```
public ArrayList getBusinessTerms()
```

Returns an ArrayList object of the extracted BML model and data terms.

Returns:

ArrayList

---

createXMLDoc

```
public org.w3c.dom.Document createXMLDoc(String xmlFile)
                                    throws FactoryConfigurationError
```

Reads in a file containing XML and creates a DOM of it's content.

Returns:

the xml DOM

Throws:

`FactoryConfigurationError`

Package org.dbe.eve.dis.matching.nn

Class Summary		Page
BMLPreProcessor	The BMLPreProcessor class provides methods for the pre processing of BML service descriptions for use in a Neural Network matching engine.	51
NNMatchingEngine	The NNMatchingEngine class implements a DIS matching engine and extends the AbstractMatchingEngine class.	53

Class BMLPreProcessor

org.dbe.eve.dis.matching.nn

```
java.lang.Object
  └ org.dbe.eve.dis.matching.nn.BMLPreProcessor
```

---

public class **BMLPreProcessor**
extends Object

The BMLPreProcessor class provides methods for the pre processing of BML service descriptions for use in a Neural Network matching engine. It implements an algorithm to convert BML service descriptions to ASCII int or binary values.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page
BMLPreProcessor() Constructor.	52

Method Summary		Page
double[][]	processBML(String smFile, Properties disProperties) This method is called to process the BML model and data parts of SM.	52
double[][]	processBML(org.w3c.dom.Document serviceManifest, Properties disProperties) This method is called to process the BML model and data parts of SM.	52

Constructor Detail

BMLPreProcessor

```
public BMLPreProcessor()
```

Constructor.

Method Detail

processBML

public double[][] **processBML**(String smFile,
Properties disProperties)
throws DISException

This method is called to process the BML model and data parts of SM. A double[][] object is returned which contains the numerical values which represent the preprocessed data.

Parameters:

`smFile` - the file name and location of the SM file

`disProperties` - a Properties object which contains all dis properties

Returns:

double[][]

Throws:

DISException

---

processBML

public double[][] **processBML**(org.w3c.dom.Document serviceManifest,
Properties disProperties)
throws DISException

This method is called to process the BML model and data parts of SM. A double[][] object is returned which contains the numerical values which represent the preprocessed data.

Parameters:

`disProperties` - a Properties object which contains all dis properties

Returns:

double[][]

Throws:

DISException

Class NNMatchingEngine

org.dbe.eve.dis.matching.nn

```
java.lang.Object
  └ org.dbe.eve.dis.matching.AbstractMatchingEngine
      └ org.dbe.eve.dis.matching.nn.NNMatchingEngine
```

All Implemented Interfaces:

MatchingEngine, Runnable

---

public class **NNMatchingEngine**
extends AbstractMatchingEngine

The NNMatchingEngine class implements a DIS matching engine and extends the AbstractMatchingEngine class.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page
NNMatchingEngine()	53

Method Summary		Page
void	**run**() This method contains the matching engine thread implementation.	54

Methods inherited from class org.dbe.eve.dis.matching.AbstractMatchingEngine
startEngine

Methods inherited from interface org.dbe.eve.dis.common.MatchingEngine
startEngine

Constructor Detail

NNMatchingEngine

```
public NNMatchingEngine()
```

Method Detail

run

```
public void run()
```

This method contains the matching engine thread implementation.

Specified by:

`run` in interface `Runnable`

Package org.dbe.eve.dis.matching.nn.util

Class Summary		Page
NNMatchingTestApp	Test Application for testing the matching process.	55

Class NNMatchingTestApp

org.dbe.eve.dis.matching.nn.util

```
java.lang.Object
  └ org.dbe.eve.dis.matching.nn.util.NNMatchingTestApp
```

---

public class **NNMatchingTestApp**
extends Object

Test Application for testing the matching process.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page
NNMatchingTestApp()	55

Method Summary		Page
static void	main(String[] args) Main method.	55

Constructor Detail

NNMatchingTestApp

```
public NNMatchingTestApp()
```

Method Detail

main

```
public static void main(String[] args)
```

Main method.

Package org.dbe.eve.dis.nn

Class Summary		Page
BMLNeuralNetwork	This class provides a neural network for BML descriptions.	57
NNConstants	This class provides constants for the DIS NN classes.	62

Class BMLNeuralNetwork

org.dbe.eve.dis.nn

```
java.lang.Object
  └ org.dbe.eve.dis.nn.BMLNeuralNetwork
```

All Implemented Interfaces:

EventListener, org.joone.engine.NeuralNetListener, Serializable

---

public class **BMLNeuralNetwork**
extends Object
implements org.joone.engine.NeuralNetListener, Serializable

This class provides a neural network for BML descriptions. The size of the input layer needs to be directly proportional to the BML description, so one neurone for each binary/ascii number. The hidden layer should be 1.5 times the size of the input layer. The output layer should be a single neurone. A default learning rate of 0.5 and momentum of 0.7 is used, but these are configurable with the properties file.

Author:

David McKitterick, Intel Ireland

Field Summary		Page
org.apache.log4j.Logger	logger logger	58

Constructor Summary	Page
BMLNeuralNetwork(Properties disProperties) Constructor.	59

Method Summary		Page
void	cicleTerminated(org.joone.engine.NeuralNetEvent arg0)	61
void	errorChanged(org.joone.engine.NeuralNetEvent arg0)	61

void	**init**(double[][] trainingInputData) This method is for initialising the NeuralNet by creating its layer structure.	*59*
boolean	**isMatch**(double[][] inputData) This method performs the matching process with a input data set against a trained NN.	*60*
boolean	**isNetRunning**() Returns 'true' is NN is still running	*60*
void	**loadNeuralNet**(String filename) Loads a saved NN from a file.	*61*
void	**netStarted**(org.joone.engine.NeuralNetEvent arg0)	*61*
void	**netStopped**(org.joone.engine.NeuralNetEvent arg0) Saves NN to file if option is on.	*61*
void	**netStoppedError**(org.joone.engine.NeuralNetEvent arg0, String arg1)	*62*
boolean	**saveNeuralnet**(String filename) Saves the NN to a file.	*60*
void	**setNNFileName**(String fileName) Sets the file name of the saved NN.	*60*
void	**trainNN**() This method starts the training process.	*59*
void	**trainNN**(String fileName) This method starts the training process and saves the trained NN to a file when the training has finished.	*59*

Methods inherited from interface org.joone.engine.NeuralNetListener
cicleTerminated, errorChanged, netStarted, netStopped, netStoppedError

Field Detail

logger

```
public org.apache.log4j.Logger logger
```

logger

Constructor Detail

BMLNeuralNetwork

public **BMLNeuralNetwork**(Properties disProperties)
throws NNException

Constructor.

Throws:

NNException

Method Detail

init

public void **init**(double[][] trainingInputData)
throws NNException

This method is for initialising the NeuralNet by creating its layer structure.

Parameters:

`trainingInputData` - the input data to be trained

Throws:

NNException

---

trainNN

public void **trainNN**()
throws NNException

This method starts the training process.

Throws:

NNException

---

trainNN

public void **trainNN**(String fileName)
throws NNException

This method starts the training process and saves the trained NN to a file when the training has finished.

Throws:

NNException

---

isMatch

public boolean **isMatch**(double[][] inputData)
throws NNException

This method performs the matching process with a input data set against a trained NN. It returns 'true' if the match is successful.

Parameters:

`inputData` - input data to be matched

Returns:

boolean

Throws:

NNException

---

isNetRunning

```
public boolean isNetRunning()
```

Returns 'true' is NN is still running

Returns:

boolean

---

setNNFileName

```
public void setNNFileName(String fileName)
```

Sets the file name of the saved NN.

---

saveNeuralnet

```
public boolean saveNeuralnet(String filename)
```

Saves the NN to a file. Returns 'true' if NN was successfully saved to a file.

Returns:

boolean

---

loadNeuralNet

```
public void loadNeuralNet(String filename)
```

Loads a saved NN from a file.

---

netStarted

```
public void netStarted(org.joone.engine.NeuralNetEvent arg0)
```

Specified by:

`netStarted` in interface `org.joone.engine.NeuralNetListener`

See Also:

`org.joone.engine.NeuralNetListener`

---

cicleTerminated

```
public void cicleTerminated(org.joone.engine.NeuralNetEvent arg0)
```

Specified by:

`cicleTerminated` in interface `org.joone.engine.NeuralNetListener`

See Also:

`org.joone.engine.NeuralNetListener`

---

netStopped

```
public void netStopped(org.joone.engine.NeuralNetEvent arg0)
```

Saves NN to file if option is on.

Specified by:

`netStopped` in interface `org.joone.engine.NeuralNetListener`

See Also:

`org.joone.engine.NeuralNetListener`

---

errorChanged

```
public void errorChanged(org.joone.engine.NeuralNetEvent arg0)
```

Specified by:

`errorChanged` in interface `org.joone.engine.NeuralNetListener`

See Also:

`org.joone.engine.NeuralNetListener`

---

netStoppedError

```
public void netStoppedError(org.joone.engine.NeuralNetEvent arg0,
                            String arg1)
```

Specified by:

`netStoppedError` in interface `org.joone.engine.NeuralNetListener`

See Also:

`org.joone.engine.NeuralNetListener`

Class NNConstants

org.dbe.eve.dis.nn

```
java.lang.Object
  └ org.dbe.eve.dis.nn.NNConstants
```

---

public class **NNConstants**
extends Object

This class provides constants for the DIS NN classes.

Author:

David McKitterick, Intel Ireland

Field Summary		Page
static final double	ASCII_NORM_MAX	64
static final double	ASCII_NORM_MIN	64
static final int	DEFAULT_CYCLES	64
static final double	DEFAULT_LEARNING_RATE	63
static final double	DEFAULT_MOMENTUM	63
static final int	DEFAULT_PATTERNS	64
static final double	DEFAULT_THRESHOLD	63
static final double	DEFAULT_TRAINING_FACTOR	64
static final String	LEARNING_RATE_PROP	63
static final String	MOMENTUM_PROP	63
static final String	OUTPUT_THRESHOLD_PROP	63
static final String	TRAINING_CYCLES_PROP	64
static final String	TRAINING_FACTOR_PROP	63

static final String	**TRAINING_PATTERNS_PROP**	64

Constructor Summary	Page
NNConstants()	64

Field Detail

LEARNING_RATE_PROP

public static final String **LEARNING_RATE_PROP**

DEFAULT_LEARNING_RATE

public static final double **DEFAULT_LEARNING_RATE**

MOMENTUM_PROP

public static final String **MOMENTUM_PROP**

DEFAULT_MOMENTUM

public static final double **DEFAULT_MOMENTUM**

OUTPUT_THRESHOLD_PROP

public static final String **OUTPUT_THRESHOLD_PROP**

DEFAULT_THRESHOLD

public static final double **DEFAULT_THRESHOLD**

TRAINING_FACTOR_PROP

public static final String **TRAINING_FACTOR_PROP**

DEFAULT_TRAINING_FACTOR

```
public static final double DEFAULT_TRAINING_FACTOR
```

TRAINING_PATTERNS_PROP

```
public static final String TRAINING_PATTERNS_PROP
```

DEFAULT_PATTERNS

```
public static final int DEFAULT_PATTERNS
```

TRAINING_CYCLES_PROP

```
public static final String TRAINING_CYCLES_PROP
```

DEFAULT_CYCLES

```
public static final int DEFAULT_CYCLES
```

ASCII_NORM_MIN

```
public static final double ASCII_NORM_MIN
```

ASCII_NORM_MAX

```
public static final double ASCII_NORM_MAX
```

Constructor Detail

NNConstants

```
public NNConstants()
```

Package org.dbe.eve.dis.nn.util

Exception Summary		Page
NNException	This class implements a DIS NN exception.	65

Class NNException

org.dbe.eve.dis.nn.util

```
java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Exception
        └ org.dbe.eve.dis.nn.util.NNException
```

All Implemented Interfaces:

Serializable

---

public class **NNException**

extends Exception

This class implements a DIS NN exception.

Author:

David McKitterick, Intel Ireland

Constructor Summary	Page
NNException() Constrictor 1	65
NNException(String message) Constructor 2	66

Constructor Detail

NNException

```
public NNException()
```

Constructor 1

NNException

```
public NNException(String message)
```

Constructor 2