



IST PROJECT 35111

Tightening knowledge sharing in distributed software communities by applying semantic technologies

Project Number:	35111
Project Acronym:	TEAM
Project Title:	Tightening knowledge sharing in distributed software communities by applying semantic technologies
Instrument:	STREP
Thematic Priority:	Information Society Technologies (IST)
Start date of the project:	September 1 st , 2006
Duration:	30 months

D4: The conceptual model and specification of the context observer and history analyzer

Lead contractor:	TUM
Editor(s):	Walid Maalej
Author(s):	Jörn David, Maximilian Kögel, Walid Maalej
Submission date:	March 2007
Dissemination level:	Public

Abstract: This document describes the requirements for the context observer and the history analyzer and presents the conceptual models of these TEAM components. It provides a top-level design as well as a first iteration of the detailed design for the realisation of the functional and non-functional requirements of the context system.

Versioning and Contribution History

Version	Date	Modification reason	Modified by
0.1	01/03/2007	Initial version	Maximilian Kögel
0.3	15/02/2007	Functional & Non-Functional Requirement	Walid Maalej
0.4	27/02/2007	Interpretation model, Analysis model	Walid Maalej, Jörn David
0.5		Proposal 1 and 2: Event vs. State centric models	Walid Maalej, Maximilian Kögel
0.7	05/03/2007	Consolidation of both proposals	Walid Maalej, Maximilian Kögel
0.8	12/03/2007	Reviewed by TEAM members	Maximilian Kögel
0.9	15/03/2007	Reviewed internally	Maximilian Kögel, Walid Maalej
1.0	17/03/2007	Introduction, Diagrams, Finalisation, Executive summary.	Walid Maalej

This document has been produced in the context of the TEAM Project. The TEAM project is part of the European Community's Sixth Framework Program for research and development and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors view.

Table of Contents

EXECUTIVE SUMMARY	5
1 INTRODUCTION.....	6
1.1 PURPOSE OF THE DOCUMENT.....	6
1.2 DOCUMENT AUDIENCE	6
1.3 DOCUMENT STRUCTURE	7
1.4 DOCUMENT CONVENTIONS	7
2 REQUIREMENTS.....	8
2.1 PURPOSE OF THE CONTEXT SYSTEM	8
2.2 SCOPE OF THE CONTEXT SYSTEM	8
2.3 SCENARIOS	9
2.3.1 <i>HELP ON RUNTIME ERROR</i>	9
2.3.2 <i>USING AN EXISTING COMPONENT</i>	9
2.3.3 <i>PROVIDING CONTEXT FOR KNOWLEDGE SEARCH</i>	9
2.4 FUNCTIONAL REQUIREMENTS	10
2.5 NON-FUNCTIONAL REQUIREMENTS.....	11
2.5.1 <i>QUALITY REQUIREMENTS</i>	12
2.5.2 <i>CONSTRAINTS</i>	13
3 HIGH-LEVEL DESIGN	14
3.1 CONCEPTUAL MODEL	14
3.2 SCENARIO FOR USE OF THE CONCEPTUAL MODEL.....	17
3.3 SUBSYSTEM DECOMPOSITION	18
3.3.1 <i>EVENT-SUBSYSTEM</i>	19
3.3.2 <i>CONTEXT-SUBSYSTEM</i>	19
3.3.3 <i>KNOWLEDGE-SUBSYSTEM</i>	19
3.3.4 <i>SENSING-SUBSYSTEM</i>	20
3.3.5 <i>INTERPRETATION-SUBSYSTEM</i>	20
3.3.6 <i>FACADE</i>	20
4 DETAILED DESIGN	22
4.1 THE EVENT-SUBSYSTEM.....	22
4.2 THE CONTEXT- SUBSYSTEM	23
4.3 THE SENSING-SUBSYSTEM.....	24
4.4 THE KNOWLEDGE-SUBSYSTEM.....	25
4.4.1 <i>KNOWLEDGE META MODEL</i>	25
4.5 THE INTERPRETATION-SUBSYSTEM	27
4.5.1 <i>THE SUBSYSTEM MACHINELEARNING</i>	28
5 SUMMARY	35
A REFERENCES.....	36

List of Figures

Figure 1: Conceptual model of the system (UML class diagram).....	14
Figure 2: Context as a tree of PropertySet and Properties (UML object diagram).....	16
Figure 3: Event hierarchy.....	17
Figure 4: Subsystem decomposition (UML package diagram)	18
Figure 5: Interfaces and structure of the event subsystem (UML class diagram)	23
Figure 6: Interfaces and structure of the context subsystem (UML class diagram)	24
Figure 7: Interfaces and structure of the sensing subsystem (UML class diagram)	25
Figure 8: Interfaces and structure of the domain independent knowledge model (UML class diagram)	26
Figure 9: Interfaces and structure of the interpretation subsystem (UML structure diagram).....	28
Figure 10: machineLearning subsystem modelled using the bridge pattern (UML class diagram)	29
Figure 11: Phases of the CRISP data mining process model [CRISPDM 07].....	30
Figure 12: MachineLearning taxonomy (UML class diagram)	31
Figure 13: Communication between different processes or threads, clock synchronisation [Lamport] (UML sequence diagram).....	33
Figure 14: MachineLearnerImpl taxonomy (UML class diagram).....	33
Figure 15: Modelling the clustering using strategy pattern (UML class diagram).....	34

EXECUTIVE SUMMARY

In TEAM we aim at a personalised and context-aware solution for knowledge sharing in distributed software development environments. After presenting a detailed survey on state-of-the-art in personalisation and contextualisation in D2, we introduce a conceptual model and specification of the functional requirements of the context system, initially called context observer and history analyzer in the description of work.

The context system represents a central component of the TEAM system. We distinguish between primary functionalities such as monitoring the developer's behaviours, logging the interaction with TEAM platform, discovering preferences, discovering usage patterns and observing changes and secondary functionalities such as capturing, constructing, storing, synchronizing, analysing, preparing, querying, interpreting and comparing of context. We also introduce the non-functional requirements namely performance, supportability, extensibility, adaptability and flexibility. These requirements serve as a reference when architecture-level decisions need to be reviewed.

In high-level subsystem decomposition we identify five subsystems: event-subsystem, context-subsystem, knowledge-subsystem, sensing-subsystem and interpretation-subsystem. In addition, we use the facade pattern to encapsulate these subsystem services to other TEAM components. In the detailed design we introduce the concepts of target, sensor, properties and context. A target can be instrumented with sensors in order to observe its properties. Thereby, context is an aggregation of certain properties of certain targets. This is very useful to restrict context to only certain properties of a target or several targets. As a consequence, context is not only the aggregation of several properties but also the choice of properties.

Context has been defined in D2 as a special kind of knowledge. We use an entity centric knowledge meta model by the context system to model knowledge. This abstract knowledge representation is instantiated at run time from the different knowledge artefacts, such as ontologies, databases, documents etc.

For us the knowledge discovery process extends data mining capabilities by assessing and interpreting the discovered result according to known measures or categories. Machine learning constitutes a core package of the interpretation subsystem. Machine learning can be split into functionalities and implementers. Functionalities are typically classification, prediction (regression), clustering and recognition. Different types of implementers exist for these functionalities, which can be grouped in the main classes symbolic, connectionistic, data mining and stochastic.

1 INTRODUCTION

Context and context-awareness play a major role in TEAM solution concept for tightening knowledge sharing in distributed software communities. We are convinced about the difficulty and inefficiency of defining and sharing universal knowledge. Thus, in TEAM system we aim at a personalised and context-aware knowledge-sharing model.

The context observer and the history analyzer together represent the TEAM components, which implement the context-awareness, and thus are responsible for delivering all context-related services. Examples of such services are capturing a software developer's behaviour or annotating knowledge with the "convenient" context.

In D2 [Maalej et al. 06], we presented a detailed survey on the state-of-the-art in personalisation and contextualisation. We also defined context as a specific form of knowledge about the user or its environment that can be useful for adapting the application, whether from the functional or non-functional point of views. Additionally in the same report we investigated the potentials of context-awareness in software development and surveyed related work in this field. In the present document, we introduce a conceptual model and specification of the context observer and history analyzer, and thus a conceptual model for the context management in TEAM system.

1.1 Purpose of the document

This document represents a combination of a classical Requirement Analysis Document (RAD) and a System Design Document (SDD). On the one hand, we describe the context system in terms of functional and non-functional requirements. This serves as a first high-level interface specification for the developers of the different TEAM components. On the other hand, we specify in detail the solution concepts of the context observer and the history analyzer. Thus, this document should be understood as a first interface specification between the various TEAM components' developers and serve as a reference when architecture-level decisions need to be reviewed.

It is important to stress that TEAM system as well as the context system will be developed following an iterative and experimental process, cycling through all software engineering activities more than once. Improvements made during the iterations will not only affect the source code, but also all other models of the system, i.e., the requirements, the design models, the test cases and the documentation. These improvements, re-factoring and redesign will be summarized in the corresponding deliverables. For that reason, in this document we present and discuss our main first iteration for the conceptual models of the context system.

1.2 Document Audience

The audience of this document includes the following stakeholder groups:

- The project management.
- The TEAM architecture task force (i.e., developers who participate in the design of the whole TEAM system).
- Developers who design and implement the context system.

- Developers who design other subsystems that use services of the context system.
- Scientists, researchers and designers interested in generic architectural models for context-aware systems.

1.3 Document Structure

We describe the conceptual requirement analysis, system design and first object design following [Bruegge&Dutoit 04]. The authors propose a simplified and understandable template inspired from common IEEE and ISO standards amongst others.

Chapter 2 defines the boundaries of the context system and represents it as a black box in terms of functional and non-functional requirements. This chapter also introduces some major visionary scenarios resulting from intensive discussion with future system users.

Chapter 3 represents an overview of the analysis models resulting from the requirements and the surveyed state of the art systems and technologies. In addition a high-level design is also discussed in this chapter.

Chapter 4 covers, at the authoring time, a detailed interface design of the target system representing a first iteration of the object design.

1.4 Document Conventions

This document was prepared during analysis and design activities. The Unified Modelling Language (UML) [OMG 03] was selected as primary notation for this document, because it provides a range of notations for representing different aspects of a system and has been accepted as a standard notation throughout the industry.

UML diagrams are specified with their types, which we always put between parentheses in the title of the figure. For readability purposes we will in purpose omit some diagram details such as stereotypes or additional parents.

More over, when we are using the term system without further specification we are then referring to the context system, i.e. the context observer and history analyzer as labelled in the description of work.

2 REQUIREMENTS

In this chapter we present the functional and non-functional requirements of the context system. The TEAM system requirements are addressed by a separate deliverable (D9), in which TEAM system use-cases are described in detail. We first sum up the purpose of the system conceptually described in this document. Then, we specify the scope of the context system in terms of its objects. After that, typical usage scenarios that illustrate the role of the context system are described. In the following section, the services of the system are specified as functional requirements. The non-functional requirements, which serve as a detailed specification of the design goals, are then derived.

2.1 Purpose of the Context System

The context system is responsible for managing the context in the TEAM system. Managing involves observing, capturing, analysing, predicting, creating and delivering context to interested components.

In the description of work, we proposed two components, which are responsible for this context management: the context observer and the history analyzer. From now on, we will use the term context system to refer to the TEAM context components such as context observer, history analyzer and other components.

The context system enables capturing the context of objects such as a software developer's behaviour in his desktop environment (i.e. context elicitation), its analysis (i.e. context processing), as well as triggering the knowledge manipulation mechanism (i.e. knowledge acquisition and delivery).

One major context aspect is the set of *previous* states of the observed objects e.g. the developers. This refers to the time dimension of the context system, mainly introduced as the history analyzer in the description of work.

2.2 Scope of the Context System

The context system is a central component in TEAM, which interacts with all other components. It is thus crucial but at the same time difficult at this stage to define the boundaries of this system. We consider all objects with at least one of the following characteristics as part of the context system:

- Objects that represent an aggregate of the context in memory.
- Objects that represent the context as whole in memory.
- Objects that represent the relationship between the knowledge and the context that annotates this knowledge.
- Objects that detect and instrument the context.
- Objects that interpret and process the context.
- Objects that predict the context.
- Objects that deliver the context to the other TEAM components.

The following objects do not constitute part of the context system:

- Parts of the graphical user interface of TEAM system.
- Parts of the targets that we will observe to define the context (e.g., the IDE).
- The persistent storage of the context-annotated knowledge.

2.3 Scenarios

In this section we describe typical scenarios for using the context system. These scenarios are not complete in any way but should provide a good overview of the intended functionality of the context system component. The following situations represent the most interesting ones discussed with the end-users in the course of the elicitation of the scenario specific context factors (Task T2.4).

2.3.1 *Help on runtime error*

A developer is not initializing an attribute of a class correctly in every constructor, while he calls a method on that attribute later in the code. The context system has observed and analysed the current context. The developer is frequently producing build errors, his code frequently fails tests and he produces many compile errors in the IDE. The system concluded that the developer is immature. The system will trigger the proactive delivery of knowledge to the developer. In this case the developer will be advised to initialize the attribute in the constructor.

In case the developer ignores this advice or he has not been advised to do so because he was considered to be a mature developer, he might experience a `NullPointerException` when using an object of the above-mentioned class. The system will observe this exception and will be able to back trace the `NullPointerException` to the missing initialization.

2.3.2 *Using an existing component*

A developer is using an existing component *x*. The system can automatically detect the use of the component. This might trigger the advice that other components are required for component *x*.

The user is repeatedly searching for keywords associated with the component in his web-browser and also producing different compile and runtime errors. Context system concludes that the user is having difficulties using the above-mentioned component. The Recommender component will then provide the developer with sample code on how to use the component. This sample code was collected earlier as knowledge with its context “component *x*” and “experienced developer” by the System and classified as useful.

2.3.3 *Providing context for knowledge search*

Whenever the developer uses the Knowledge desktop to search for knowledge his queries are automatically enriched by context information. The developer enters a search on how to use SQL. The context system can provide context for this query: the developer is inexperienced; he is developing a Java application and is using Hibernate, a persistency framework. Given this context the knowledge desktop component can now provide the developer with help that is really useful: “There is no need to write SQL, if you use Hibernate”. Additionally it could

provide sample code or tutorials that have been useful for other developers within the same context.

2.4 Functional Requirements

Monitor Developer

The system should monitor the developer's behaviour on his workplace that includes an IDE environment as well as application programs used in the context of the software development task. It is important that the workspace definition is extendible to other monitorable objects.

A developer is working on his computer and the context system is monitoring each action he performs. The focus will be on the action streams inside the IDE. However actions might also include using other applications, like Web Browser, in order to find information useful for his current task.

Due to reluctance of users to provide any feedback, the information related to his software development task should be captured automatically.

Log Interaction

All information about a user's behaviour inside TEAM should be captured in a log file. Examples of such interactions are "the user posts a query", "the user requires a semantic recommendation" or "the user annotate an experience object" etc. This functionality is directly related to the usage of search, recommendation and knowledge desktop subsystems.

Capture Context

Monitoring developer's work does not mean necessarily capturing relevant context information. We argued in D2 [Maalej et al. 06] that context is relative. A piece of knowledge might be a context for another piece of knowledge or might require other knowledge as context information for it. Therefore capturing context implicitly includes selecting the relevant context sources - through enabling and disabling the appropriate sensors - as well as associating context with the subject piece of knowledge.

Construct Context

We have described in details in D2 the different transformations that can be processed on the context as construction mechanisms. Context might be projected, aggregated, filtered etc. (see D2 for more details).

Store Context

The elicited context should be persistently stored in order to enable reuse of experience. Storing context implicitly means representing context as an entity: in-memory object as well as a persistent object.

Synchronise Context

The context system should determine the start and the end of a user's session (the so-called context sessionisation). A user session can be for example a set of actions that is related to resolving a problem.

Analyse Session

Sessions should be analyzed and two types of outputs will be generated:

- Experience as an instantiation of the knowledge model in the metadata repository.

- Working context (including code fragments and problems) as an input for more efficient search.

Discover Preference

The preferences of the developer are discovered while analyzing previous user interactions and personal context in search and recommendation subsystems.

Prepare Context as Knowledge

A problem is observable and the context in which the solution was elaborated is observable as well. The context should be formalized as the experience in which a problem is resolved. Thus the system should differentiate between context and context-aware knowledge.

Query Context

The context system should provide two context-querying services:

- Push: in this case the knowledge or the user query will be annotated with context without any other component explicitly asking for this information (e.g., the recommender system is always interested on the level of expertise of the developer.)
- Pull: in this case the component explicitly asks for the context of a specific knowledge and the context system delivers the context in a predefined interaction language.

Observe Change

Change is one important event type that should be triggered. The context system should observe which changes occur to which properties of which entities.

Discover Patterns of usage

The context system should discover repeatable patterns in order to identify some important characteristics of a user. Therefore, log information should be analyzed from different perspectives and for different periods of time (i.e. set of events).

For this analysis, machine learning and data mining mechanisms should be applied. However the interpretation strategy should be extendible to further mechanisms.

Interpret Captured Information

The interpretation process includes generalising the context, identifying relevant context (e.g. code fragments), identifying/ classifying context (e.g., problems), and associating context to other knowledge. A special case for the interpretation of the captured information is the prediction of the context.

Compare Context with other Peers

Context information will be used in p2p network search, e.g., for the identification of “similar” developers. It is important to be able to compare different context information in different peers.

2.5 Non-functional Requirements

For the categorization of non-functional requirements we follow the FURPS+ model used by the Unified Software Development Process [Jacobson et al. 99]. The FURPS+ model makes a distinction between non-functional requirements that are quality requirements and those that are constraints. For example, while usability, reliability, performance and supportability

are quality requirements, implementation, interface, operations, packaging and legal requirements are constraints, also known as pseudo requirements.

2.5.1 Quality Requirements

Performance

One major requirement is to efficiently represent the monitored user interaction. Moreover, the session analysis and pattern discovery should be done in an efficient way. The context system will perform its activities mainly in the background, in most of the cases without a concrete interaction with the user. The user should not face any performance problems while doing his development tasks or deploying other TEAM functionalities.

Portability

One main requirement of the system is to roll out as much functionality as possible from specific tools. Completely integrating development environment modules or just communicating with them can present ambiguity during the design process. Nevertheless, internal context interpretation and structuring functionalities have to be realized independently from the underlying platforms.

Extensibility

In software engineering and knowledge management domains, which constitute our problem and solution domains respectively, we can observe an increasing number of standards and methodologies, resulting in more and more products and technologies. To deal with highly instable environments, resulting from changes in technical surroundings, the context system has to be extensible. The system should be able not only to support current solutions but also future technologies and standards. Such condition shall not require modification of the existing context system. Additional modules to support new technologies should be easily added.

A particular dimension of extensibility, which is discussed in the description of work, is the “plug-in”-ability. This requirement is closely related to the use of eclipse as a proof of concept platform. Eclipse relies on a plug-in architecture. A plug-in allows itself to be extended in an embodied extension-point. An extension-point is defined by a plug-in that stands in a host role with respect to the extension-point, and may be extended by one or more plug-ins that stand in an extender role with respect to the extension-point. There is a contract associated with each extension-point. The contract puts obligations on both the host and the extender plug-ins.

Adaptability

One major aspect that distinguishes our system from other existent solutions is its ability to adapt to changes of the environments. Adaptability is a common software quality requirement (see [IEEE 90] for a detailed discussion). Particularly, the system has to show high adaptability with regards to modifications and extensions of software engineering domain concepts and structures.

Flexibility

Flexibility is the ability of the system to change its behaviour against various environment interactions. Flexibility – or configurability – is expected from many functions of the system and therefore involves many aspects.

This requirement results from the wide spectrum of project environments that we aim at supporting with TEAM. First surveys and analysis showed that our industrial partners present manifold profiles.

On that account, domain concepts and logics should be interpreted in a flexible way. The system should be able for instance to answer questions like: what are characteristics of the organisations?

Flexibility is also required technically in the interpretation of predefined knowledge. The system should be able for instance to answer questions like: How is an entity related to another? Does a developer e.g. have a task or does the task belong to a developer? Underlying technology issues resulting from monitoring different targets should also be handled in a flexible way.

Moreover, the monitored targets generally offer different language constructs to model similar issues. The system should be able to flexibly interpret all those constructs as well as combine them.

2.5.2 Constraints

Interface requirements

The automatic elicitation of context information requires the ability of the context system to be integrated into other applications where the interaction can be monitored. Above all, the context system should be integrated in the development environment (IDE). The design should support a generic integration mechanism. However, integrating the context system in Eclipse should proof the concept. Therefore, the plug-in extension model provided by Eclipse will be utilised [EclipsePlugins 07]. Eclipse plug-ins offer a flexible model of extensibility, and their abstract architecture for composing systems out of loosely coupled components provides a significant addition to the available repertoire of architectural patterns for software systems.

3 HIGH-LEVEL DESIGN

In this section we describe the conceptual design of the context system. We begin with presenting an overview of the conceptual model, which constitutes the result of the preliminary analysis of the elicited requirements. In the following, we introduce a dynamic model that represents the main behaviours of the system by means of a usage scenario. Finally, we introduce a high level decomposition of the system based on the classification of the identified objects and their common services.

3.1 Conceptual Model

During the analysis we identified objects that make out the context system. In the following we first describe every class. Figure 1 represents an analysis object model that shows classes and their associations.

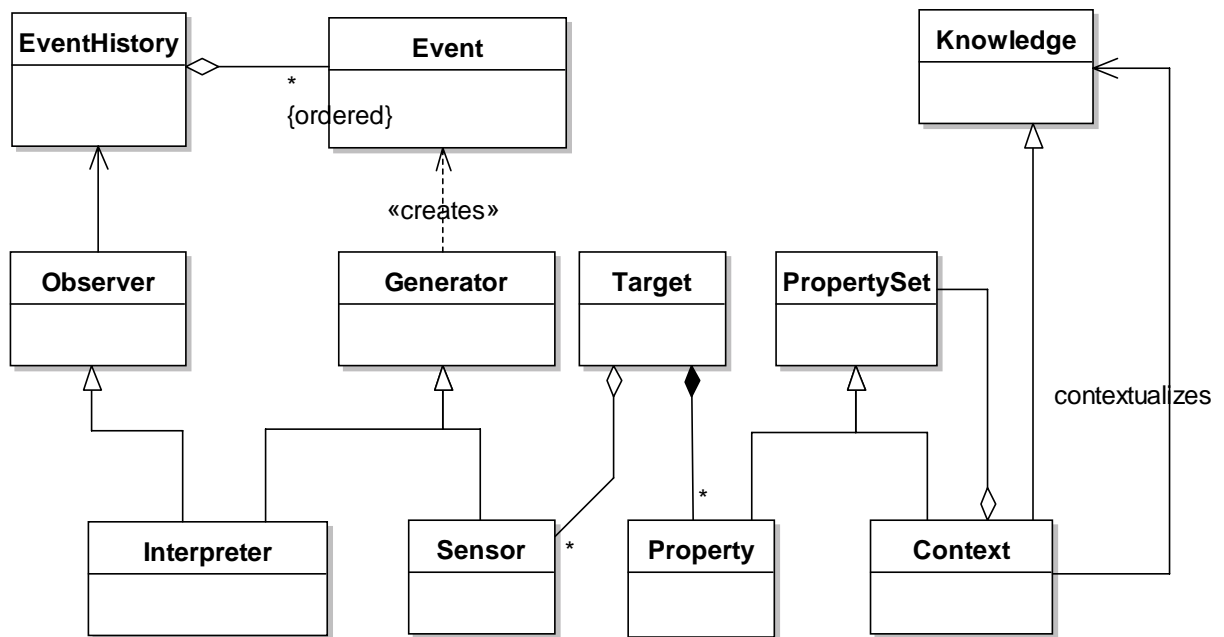


Figure 1: Conceptual model of the system (UML class diagram)

Class: Target

Description: The target is something that we want to observe (e.g. an IDE, a user, the environment etc.). A target constitutes of certain properties (e.g. a property of an IDE is its version or its licence). The values of the target properties represent its state. A target can be instrumented with sensors in order to observe its properties (i.e. its state).

Class: Property

Description: An object of this type represents a property of a target. The value of a property represents part of the targets current state (e.g., property “version” with value “3.18”). Properties can be observed if their target is instrumented with an appropriate sensor.

Class: PropertySet

Description: This class represents a set of properties. PropertySet allows aggregating properties. Moreover PropertySet represents a classification mechanism for properties, that context will make use of.

Class: Knowledge

Description: This class represents a piece of knowledge, e.g. a rule, an experience etc.

Class: Context

Description: Context is a special form of knowledge. Context can contextualize knowledge meaning add context information to knowledge (i.e. annotate knowledge). Context is an aggregation of PropertySets. This implies that context is an aggregation of certain properties of certain targets. This is very useful to restrict context to only certain properties of a target or several targets. An example context could be the aggregation of PropertySets “myInfrastructureContext” “myUserContext” and “myPhysicalContext”. Each of these objects constitutes a bag of concrete Property-objects. The version of the IDE and the licence of the configuration management system might for example comprise the infrastructure context, i.e., the first bag. Figure 2 depicts this example as UML objects. The diagram represents an example instantiation of the class model introduced in Figure 1. In consequence context is not only the assignment of several properties but also the selection of properties.

Class: Generator

Description: A Generator is creating events to notify observers of interesting events. It uses the EventHistory to publish events.

Class: Sensor

Description: A Sensor is a special Generator that is attached to a target and is sensing properties of that target.

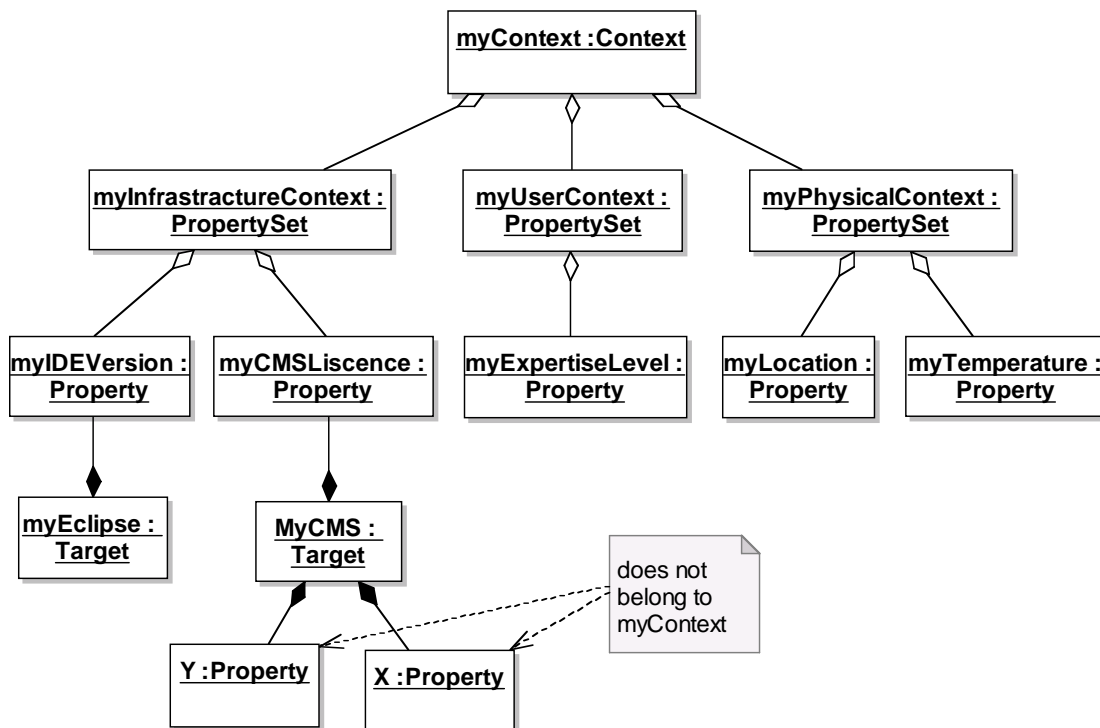


Figure 2: Context as a tree of PropertySet and Properties (UML object diagram)

Class: Event

Description: An Event represents the fact that something has happened. The state of a target might have changed or new knowledge was discovered. An Event has a type and contains data (e.g. about a state change of a target). Also an event is associated with the context, which the event was created in.

Class: EventHistory

Description: The EventHistory contains the ordered sequence of all events that ever occurred. It provides methods to add new events to the history. It also provides methods to query the sequence of events and to register for events that match a specified filter.

Class: Observer

Description: An Observer observes targets. For that purpose, he subscribes for events at the EventHistory by specifying a filter. He is then notified whenever matching events occur.

Class: Interpreter

Description: An Interpreter is an Observer and a Generator, therefore he will observe events, interpret them and create new events. Based on his observation he will

interpret the events, the contained knowledge and context, and come up with new knowledge, which it will then post as a new event.

Superclasses: Observer, Generator

3.2 Scenario for use of the conceptual model

Although the EventHistory itself is just an ordered aggregation of events, with the support of filters we gain the flexibility to realize many different scenarios. This scenario demonstrates how flexible the approach is.

The Eclipse IDE, a target, is instrumented with different sensors that detect the current user interactions with Eclipse. Also the Internet-Browser, another target, is instrumented to detect search keywords the user is entering.

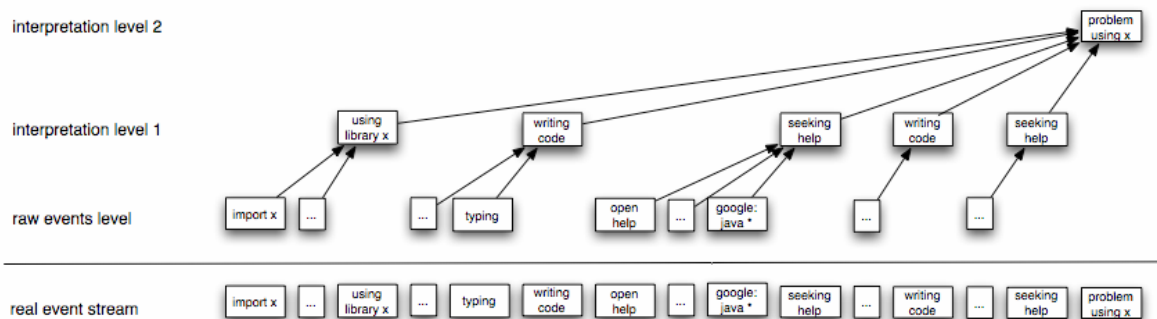


Figure 3: Event hierarchy

In Figure 3 we can see the real event stream as captured by the EventHistory. The first event is initiated by one of the Eclipse Sensors and indicates that the user just imported some class. The sensor publishes the event by notifying the EventHistory of the new event. The Event History will add the event to the sequence of events and notify any subscribed Observers, if the event matches their filter. Observers can subscribe for events at the EventHistory by specifying a filter.

Interpreters are Observers and Generator, because they observe events, interpret them and possibly initiate new events to announce the result of their interpretation.

We assume that an interpreter has subscribed for all events related to imports and usage of classes in Eclipse. After he observed the “import x” and other events as shown in Figure 3, he determines that a certain library is being used and announces his result by initiating a new event “using library x”. Other interpreters determine, that the user is “writing code” or “seeking help and also initiate events accordingly. While these interpreters only used raw events provided by sensors that are not interpreters, there will also be interpreters that only use events produced by other interpreters.

We imagine that another interpreter has subscribed for all user activities in Eclipse. He observed that the user is “using library x” and is in a loop of writing code and seeking help containing the keyword x. He concludes that the user is experiencing a problem with using library x and announces his result as a new event.

The process of observing and then announcing new knowledge as events can be repeated over and over again and will lead to more abstract and useful knowledge over time.

At some point of time the context system might decide that a certain piece of knowledge is useful and add it to the knowledge database, where it is publicly available.

3.3 Subsystem Decomposition

In this section we introduce the initial decomposition of the context system and describe the general services and boundaries of each subsystem. First, based on the functional requirements and the analysis models of last section, we identified five subsystems: event, context, knowledge, interpretation and sensing subsystem. In order to increase cohesion and decrease coupling of these components, we further introduced the facade and the domain subsystems. Domain is inspired from the Model-View-Controller architectural style, where the logical concepts are bounded together. The facade represents the view in the pattern since the context system does not include any user interfaces. The subsystems are presented in detail in the following sections. Figure 4 shows an overview of the subsystem decomposition the dependencies and also presents the gateway of the context system to the metadata repository of TEAM.

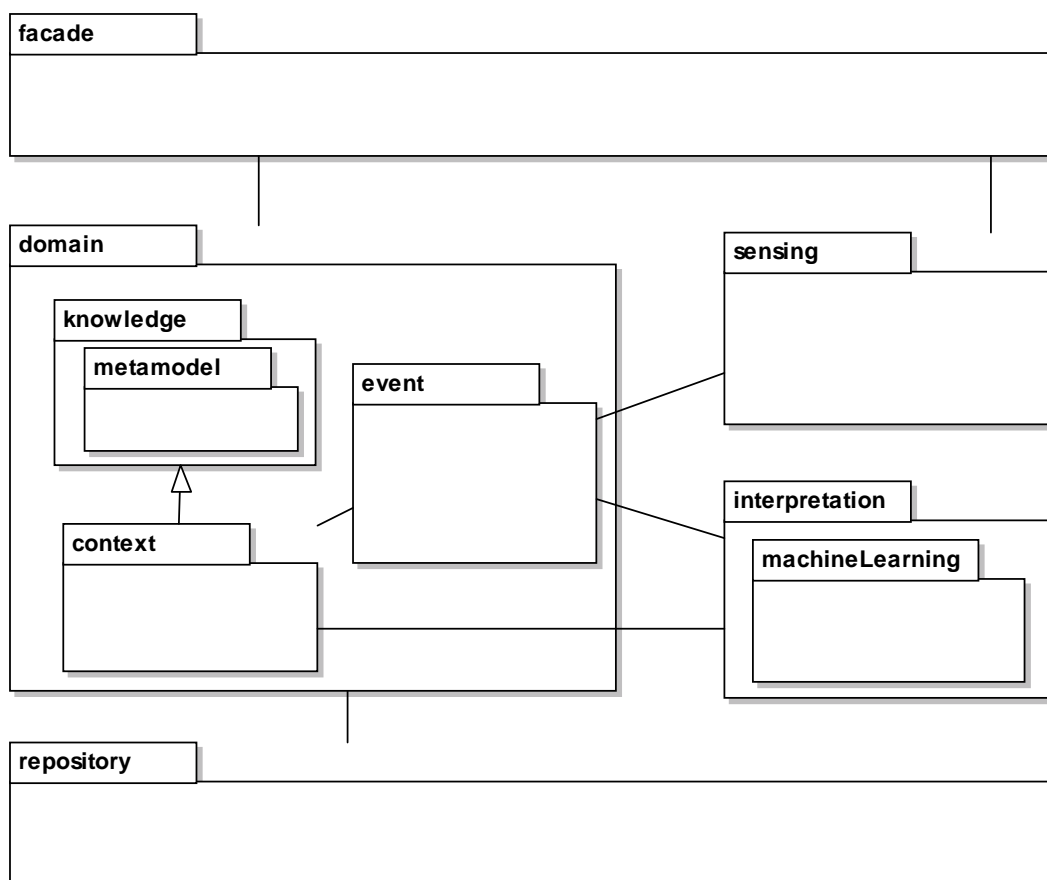


Figure 4: Subsystem decomposition (UML package diagram)

3.3.1 *Event-Subsystem*

The domain.event Subsystem is responsible for representing, managing and delivering different kind of events. The event subsystem also defines the event taxonomy.

- Services:
 - Define events.
 - Publish events.
 - Subscribe for events.
 - Query events.
 - Categorize events.
 - Serialize events.

3.3.2 *Context-Subsystem*

The domain.context subsystem is responsible for the context elicitation, definition and construction. The delivery of context to other components is one important service the system must provide.

- Services:
 - Represent context.
 - Capture context.
 - Query context.
 - Construct context.
 - Deliver context quality.

3.3.3 *Knowledge-Subsystem*

The representation of knowledge in the runtime environment of the TEAM system is represented by domain.knowledge. The metamodel for knowledge is represented by the package domain.knowledge.metamodel. At a first glance, this system does not belong to the context system. Since we define context as a special form of knowledge, it is important for the context system to represent knowledge and annotate it with context. Moreover, none of the other TEAM components will be responsible for representing the in-memory knowledge centrally. Thus, we introduce this package as a representation of the knowledge. Nevertheless, the interfaces of this subsystem should be used centrally by other components of TEAM system. We expect therefore some adjustments of our proposal when the conceptual models of further TEAM components are ready.

- Services:
 - Represent knowledge.
 - Manipulate the knowledge structure.
 - Manipulate the knowledge content.
 - Focus on a knowledge piece.

- Structure knowledge.

3.3.4 Sensing-Subsystem

The subsystem sensing is responsible for providing sensors in order to sense context information. It also takes care of configuring sensors according to the current context. Targets which we instrument and sense their context-relevant properties, are represented and managed by this subsystem as well.

- Services:
 - Define targets.
 - Manage targets.
 - Provide sensors for various targets.
 - Configure sensors according to the current context.

3.3.5 Interpretation-Subsystem

The subsystem interpretation is the analyser of the captured information and the generator of potentially useful knowledge. Captured information primarily consists of events. This subsystem delivers an output of the analysis that might be a context definition or a new knowledge. The package `interpretation.machineLearning` represents interpretation interfaces and algorithms based on machine learning mechanisms.

- Services:
 - Interpret events.
 - Trigger interactive help.
 - Create context.
 - Predict context.
 - Annotate knowledge with context.
 - Synchronize context.
 - Discover user preferences.
 - Prepare context as knowledge.
 - Discover patterns of usage.

3.3.6 Facade

Subsystem decomposition reduces the complexity of the presented solution by minimizing coupling among subsystems. The facade design pattern [Gamma et al. 94] allows us to further reduce dependencies between classes by encapsulating the context system with a simple, unified interface. Thus the subsystem facade allows an easy access to all services of the system. The main functionality of facade is to delegate the requests from outside the system to the appropriate subsystem.

- Services:
 - Query context (push and pull).

- Subscribe for context changes.
- Define context.
- Define targets and its properties.
- Specify knowledge to be annotated by context.

4 DETAILED DESIGN

In this section, we divide the classes presented in the analysis model into different packages and present additional classes to detail the model when necessary. Furthermore, attributes and methods are added to the existing classes. The diagrams for each package also show classes related to the package that are not part of the package for readability reasons.

4.1 The Event-Subsystem

The event package consists of the following classes:

- Event.
- EventHistory.
- Observer.
- Generator.
- Filter.

The classes Event, EventHistory, Observer and Generator have already been described and now only have additional attributes and methods. EventHistory allows Observers to subscribe for events specified by a filter using the subscribe method and to query the history on events by using the query method. The Event has an attribute eventType to determine its type. Observer supports a notify method, to allow EventHistory to notify it in case of appropriate events. Obviously we are making use of a modified version of the observer pattern [Gamma et al. 94]. The main reason behind this design decision is to maintain consistency across the states of one Generator and many subscribers. We added one additional association class Filter, which is described in the following.

Class: Filter

Description: A filter specifies the events an Observer is interested in. A filter may represent an interpretation level of events. An interpreter can interpret events that are initiated by some low level sensor, such as GPS, and those interpreters create new knowledge in the form of an event in turn. These events can then be interpreted again. This creates a hierarchy of different levels of interpretation, where the knowledge in each level is more abstract and sophisticated. A filter is not limited to this purpose but can be used to implement it.

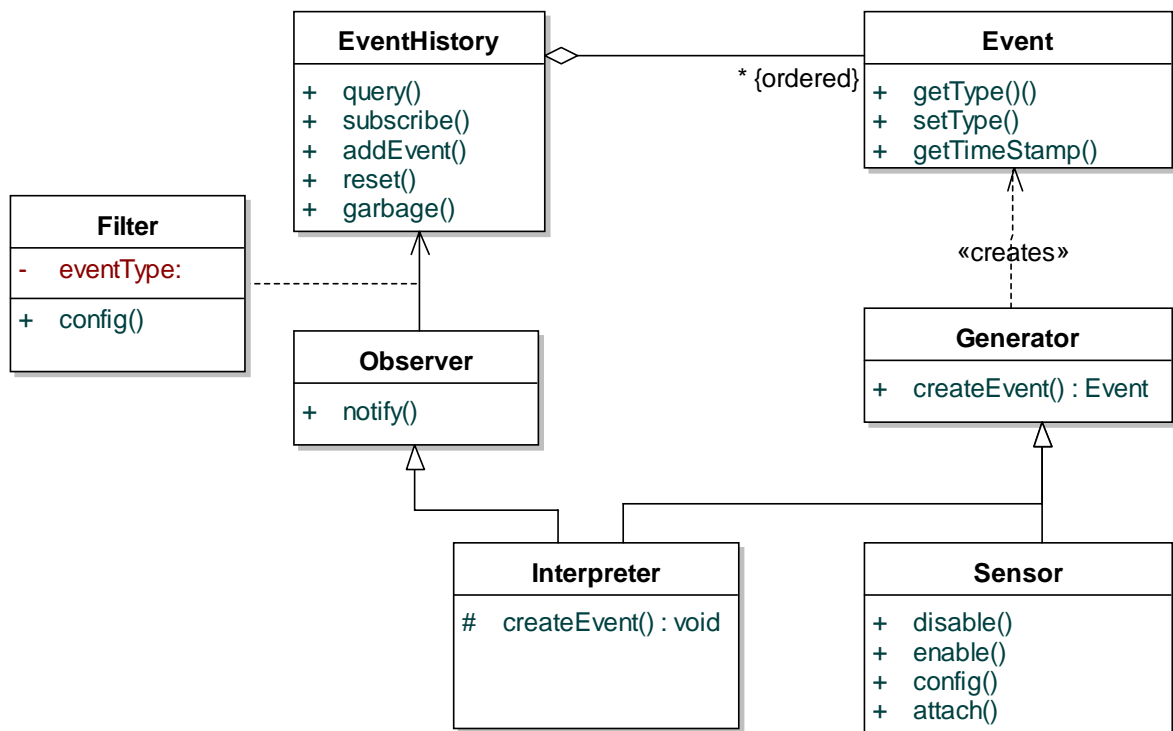


Figure 5: Interfaces and structure of the event subsystem (UML class diagram)

4.2 The Context- Subsystem

The context package is composed of the following classes:

- Context.
- ContextManager.

Context provides methods to transform, project or aggregate its values. The interface `getQuality()` provides an evaluation mechanism for the context quality, which we discussed in detail in D2. Additionally there is the ContextManager class.

Class: ContextManager

Description: The ContextManager is a controller responsible for providing and maintaining the current context. The context manager is an observer because he will listen to events to determine changes in the context and decide for the current context, in other words which properties are part of the current context. Also the context manager is a sensor because he observes the context and creates events when the context changes.

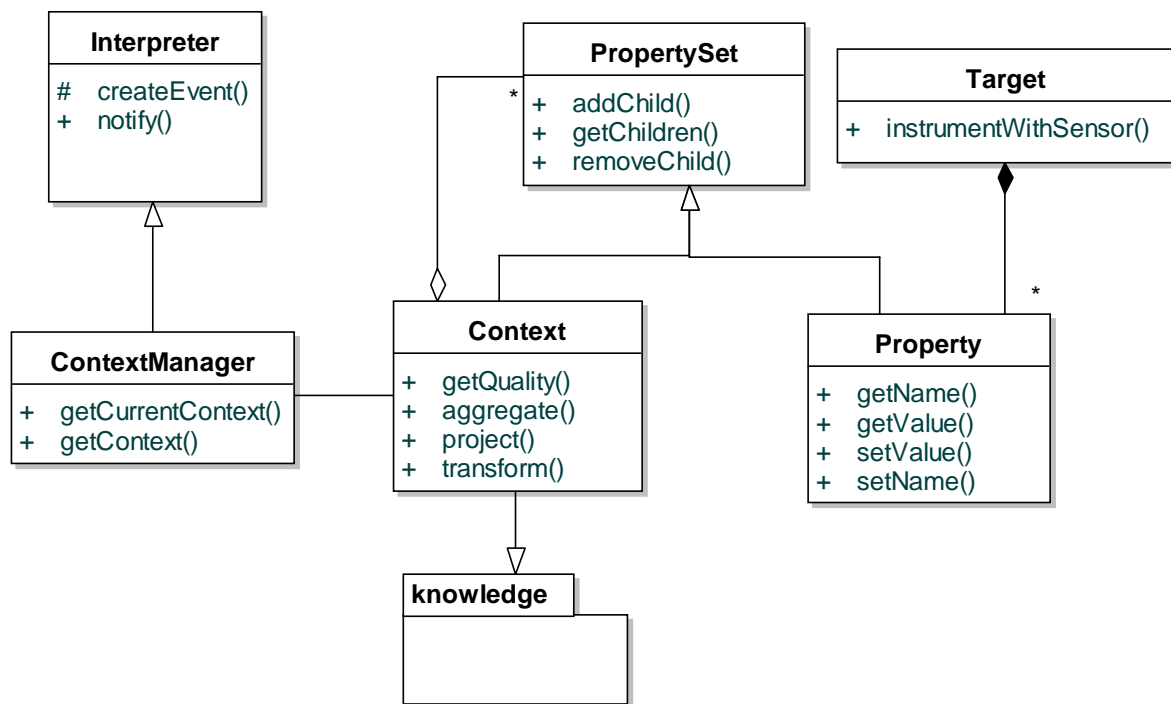


Figure 6: Interfaces and structure of the context subsystem (UML class diagram)

4.3 The Sensing-Subsystem

The package sensing consists of the classes:

- Sensor.
- Target.
- Property.
- SensorManager.

Classes Sensor, Target and Property have already been introduced. Sensor now supports the methods `enable()` and `disable()` to switch a sensor on and off. A Target provides the `instrumentWithSensor` to allow for adding sensors to a target. And finally property has the attribute `name` to uniquely identify a property and `value` to store the properties value. The only additional class, `SensorManager`, is presented below.

Class: `SensorManager`

Description: The `SensorManager` is responsible for configuring (switch on or off) the sensors of a target according to the current context.

SuperClasses: `Interpreter`

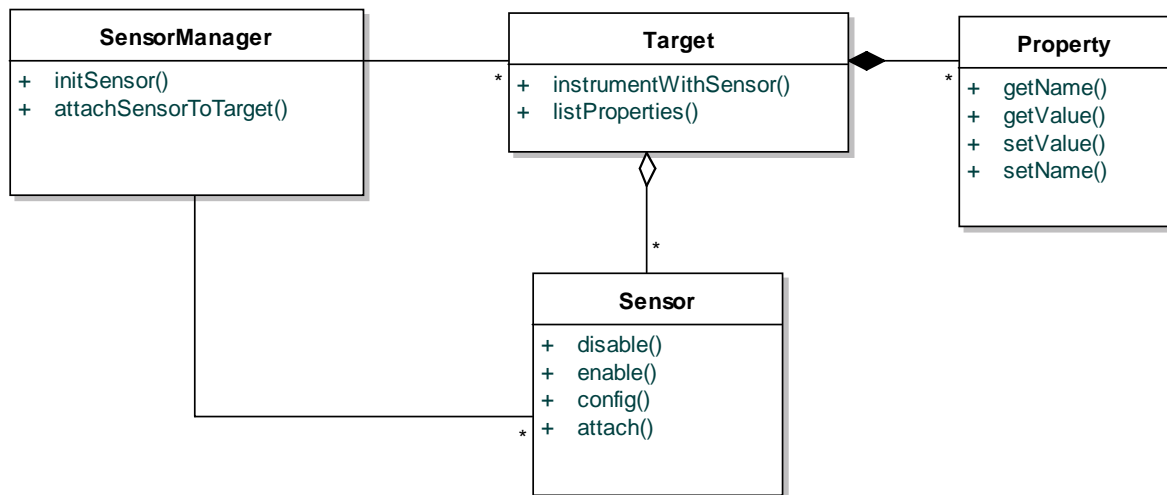


Figure 7: Interfaces and structure of the sensing subsystem (UML class diagram)

4.4 The Knowledge-Subsystem

The following classes are part of the knowledge package:

- Knowledge.
- KnowledgeBase.
- KnowledgeManager.

Knowledge was already described earlier and now provides a method contextualize to add context to it. We will present knowledge meta model that will define knowledge in more detail in Section 4.4.1.

Additional classes are KnowledgeBase and KnowledgeManager:

Class: KnowledgeManager

Description: The KnowledgeManager decides also based on the current context whether knowledge represented by events he observes should be included in the knowledge base. If it should be included he will add the knowledge and its context to the knowledge base.

SuperClasses: Interpreter

Class: KnowledgeBase

Description: The KnowledgeBase is an entity where knowledge is stored and is available for everyone. It is very different from the History, because the History only stores local events, which are not yet processed to such an extent to represent interesting or valuable knowledge that should be publicly available

4.4.1 Knowledge Meta Model

The model represented on Figure 8 depicts an entity centric representation of domain knowledge adopted from [Maalej 05]. This abstract representation will be instantiated at run time from the different knowledge artefacts, such as ontologies describing the software

engineering domain or a bug database representing bug descriptions in a concrete software development project. In fact this is a meta-description of the well-known Entity Relationship Diagram (ERD). One can imagine the domain ontology with its manifold business models and data as a huge fishing net, where the different domain concepts represent the nodes while the wires are the associations and the relationships that connect them. At run time we are interested in one entity in the domain, and require an entity centric consideration of this knowledge. This would then be like taking the net from the node that corresponds to the search-entity and cutting around the third line. In the following paragraphs we briefly describe the main interfaces of these different components.

Class: Property

Description: This component provides three abstract interfaces, which are:

- `getName()` for getting the name of the property which is required to identify it;
- `getLabel()` for getting the label of the property which will be needed for constructing and representing the graphical user interface; and
- `isMultipleValued()` for checking if the property can be multi-valued or not. A multi-value property is a property that can represent many values, e.g. the property visited-lecture of a student.

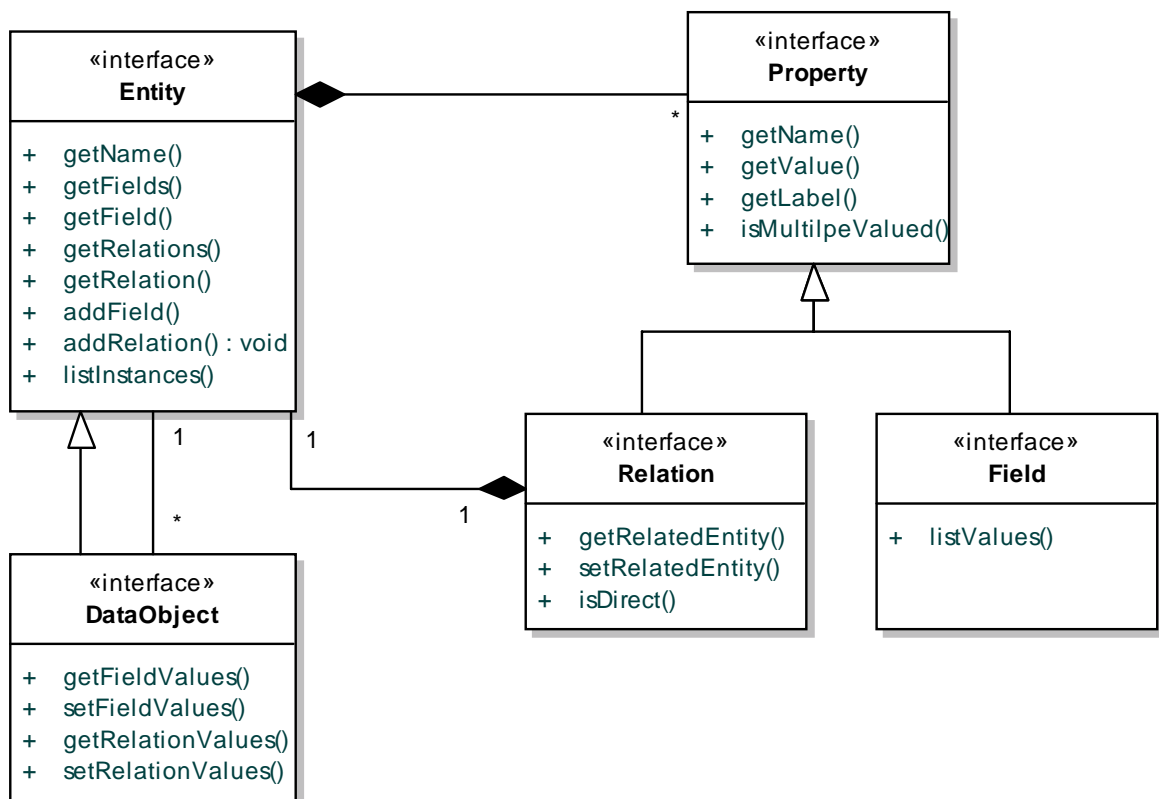


Figure 8: Interfaces and structure of the domain independent knowledge model (UML class diagram)

Class: Field

Description: This component models standard datatypes as integer, date, string etc. It involves the interface `listValues()` that delivers a list of all the values of the Field. In case the field is single-valued the list includes only one value.

Class: Relation

Description: Every Relation is associated with one Entity, and therefore it offers the interfaces to get and set this related entity-object. The interface `isDirect()` should check if the relation is defined in the direct way from the origin entity to the target entity or the in the reverse direction.

Class: Entity

Description: This central system component provides interfaces to list its fields and its relations as well as getting one of them based on its name.

Class: DataObject

Description: Each DataObject inherits the Entity interfaces so that it can reproduce meta information about itself. Thereto, instances of DataObject must know their concrete entity, and thus the second association between the two classes [Gamma et al. 94]. DataObject also presents interfaces to set the values of its properties.

To recapitulate, the data structure hierarchy discussed above represents the container APIs, which will be filled differently with the data at the instantiation time.

4.5 The Interpretation-Subsystem

The package interpretation consists of the following:

- Interpreter
- MachineLearning

An Interpreter provides the methods `notify()` and `createEvents()` as required or implemented by its super classes.

Machine learning is a very important part of the system. The package `machineLearning` as well as the class `MachineLearning` will be covered in detail in the following subsections.

Three interpreters, i.e. `SensorManager`, `KnowledgeManager` and `ContextManager`, do not belong to the interpretation package because they serve a special purpose.

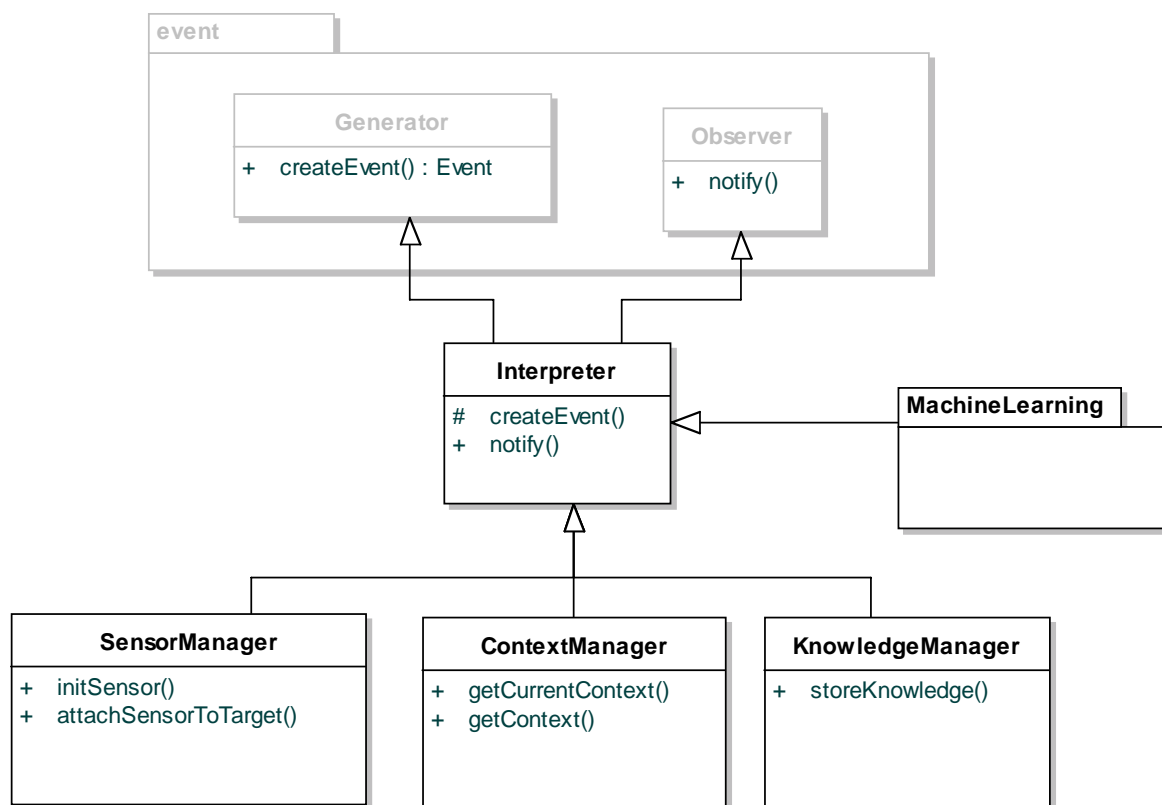


Figure 9: Interfaces and structure of the interpretation subsystem (UML structure diagram)

4.5.1 The Subsystem machineLearning

Machine learning can be split into functionalities and implementers. Functionalities are typically classification, prediction (regression), clustering, recognition etc. There are many implementers for each of these and a few of them are capable to support multiple functionalities. For example, a neural network is an implementer that can serve several purposes like classification, prediction, recognition and grammar learning. To ensure the independency of the functionalities from the used implementers, and to enhance the extendibility of the system with future functionalities and implementers, we make use of the Bridge Pattern (see [Gamma et al. 94]). Figure 10 depicts this design decision showing some important functionalities and implementers. A bridge is constructed between MachineLearning and MachineLernerImpl. MachineLearning plays the role of the Abstraction in the pattern. This enables decoupling the functionalities' interfaces from their implementations, which enables the substitution of implementation at runtime.

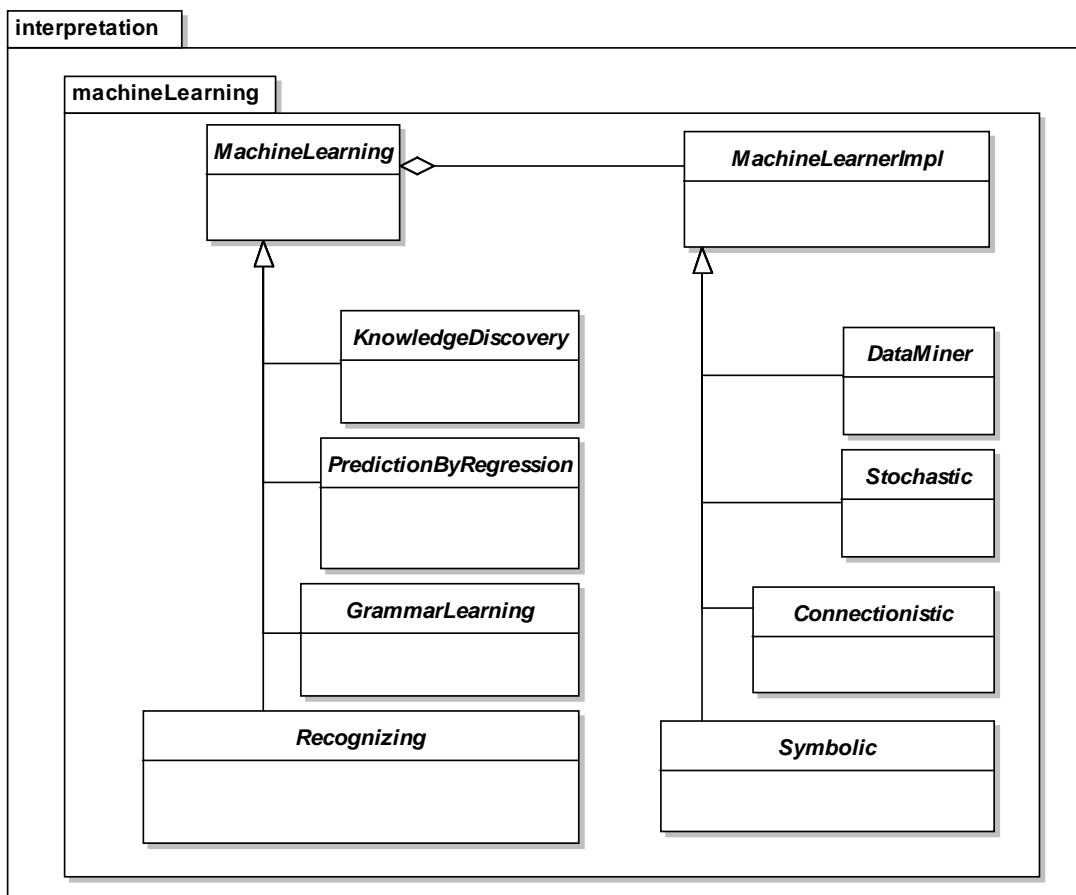


Figure 10: machineLearning subsystem modelled using the bridge pattern (UML class diagram)

The knowledge discovery process extends data mining capabilities by assessing and interpreting the discovered result according to known measures or categories. This semi-automatic process consists of phases like defining analysis goals, data selection, data clean up, data reduction, choosing a representation model, actual data analysis and interpretation.

For the sub process data mining there are already several life cycle models, for example the CRISP-DM process model depicted in Figure 11.

On the other hand there is the classical machine learning branch that fulfils tasks like prediction (enabled by error regression), character recognition or grammar incorporation by automatically learning a certain domain model.

For the TEAM system it will be important to realize event, sequence learning by a concrete implementer on the right side of the depicted Bridge Pattern (see Figure 10) in order to provide context prediction, for example.

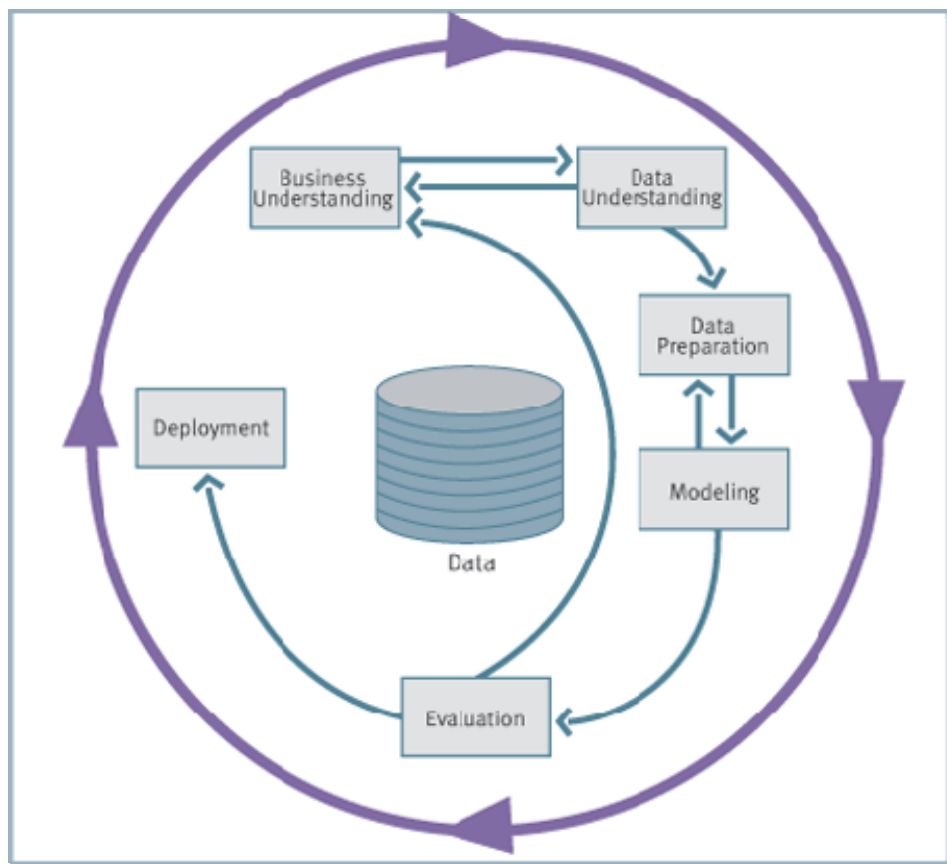


Figure 11: Phases of the CRISP data mining process model [CRISPDM 07]

4.5.1.1 Functionalities

Figure 12 shows a taxonomy for the existing machine learning functionalities. The following paragraphs describe the corresponding classes.

Class: KnowledgeDiscovery

Description: Super class for all classical data mining tasks, which are normally executed in batch runs on fixed datasets like user histories or access statistics. Knowledge discovery is the semi-automatic process of extracting unknown and potentially useful knowledge from databases.

Many data mining methods support incremental mining in order to efficiently adapt the results to updated datasets.

Class: DataMining

Description: Usually data mining is understood as the extraction of knowledge from raw data. This is the core functionality of this class. Interpretation of the results is not comprised by data mining itself, since it belongs to the entire knowledge discovery process.

Class: Clustering

Description: Grouping similar objects (similarity measure, context similarity) into groups or clusters. Objects in one cluster are more similar to each other than objects from different clusters.

Class: Classification

Description: Classifies unknown objects according to predefined classes or categories. For building a classifier training with known data objects is necessary, which are labelled according to their true class.

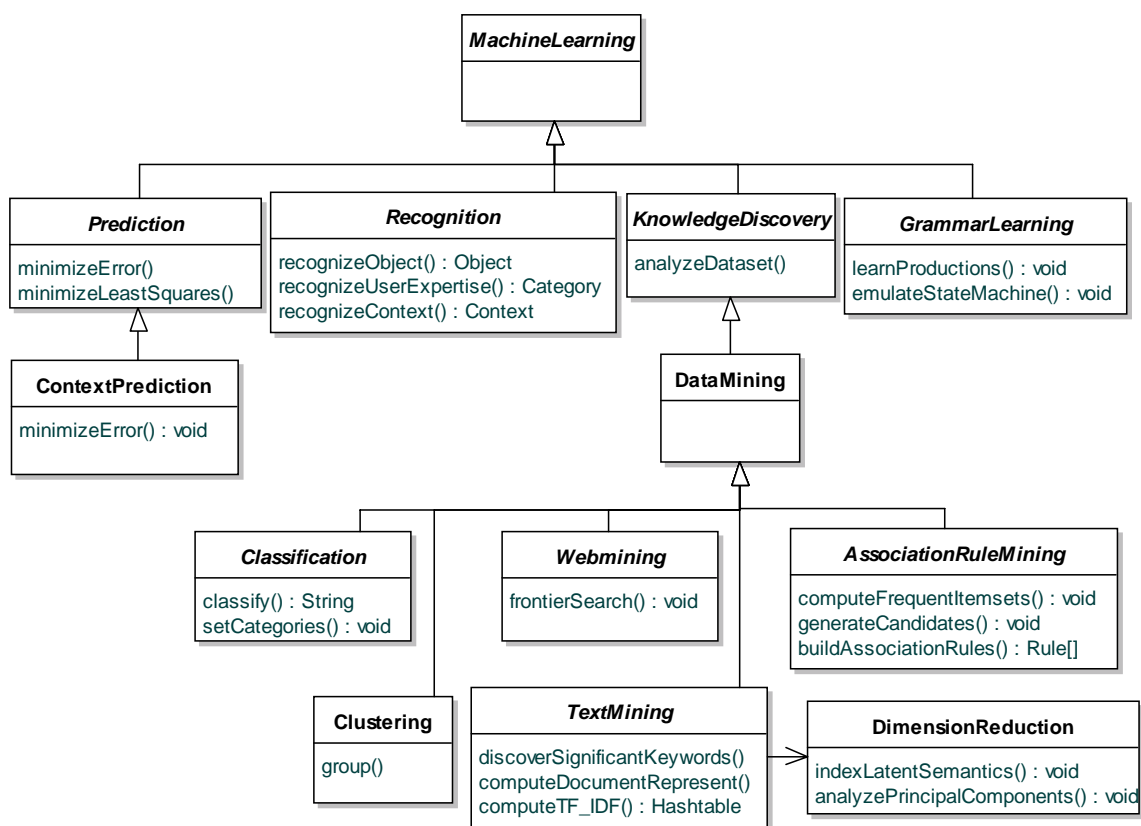


Figure 12: MachineLearning taxonomy (UML class diagram)

Class: AssociationRuleMining

Description: Classical method to analyze basket of goods or shopping cart in order to determine the items that are bought together frequently (associated items). This kind of mining can be used in TEAM to automatically derive behaviour rules for the rule base.

Class: DimensionReduction

Description: Generic purpose class for dimension reduction of arbitrary data sets consisting of high-dimensional feature vectors (e.g. sparse vectors from text mining,

document representation), which can be transformed to a more lower dimensional representation with little information loss.

Class: GrammarLearning

Description: Learning of existing state machines with their transitions or learning of grammar productions. In case of realization by a (implement) neural network can be used in order to accept or reject user behaviour in a smooth and fuzzy way.

Class: Prediction

Description: Prediction exploits knowledge learned from history in order to determine future states. Often prediction is related to dynamical systems that evolve over time and allow learning from their past behaviour to predict their behaviour in the short-, middle- or long-term future.

Subject of prediction can be random variables or time series, whose realizations are predicted for future points of time. An example is gene function prediction in bioinformatics or the anticipation of future user contexts.

Class: ContextPrediction

Description: Forecast of categorical, ordinal and metric values learned from history. An example would be the network traffic or the check-in/check-out rate at a future point of time.

Class: Recognition

Description: User behaviour or level of usage and domain expertise can be determined in terms of dynamically changing categories, which can be automatically found by clustering, for example.

The operation *recognizeContext* takes a sequence of events (e.g., a serialized context aggregation) that have to be ordered with respect to a logical time. If a certain sequence can be recognized as known behaviour, the respective category is returned.

For logical ordering compare the sequence diagram on Figure 13.

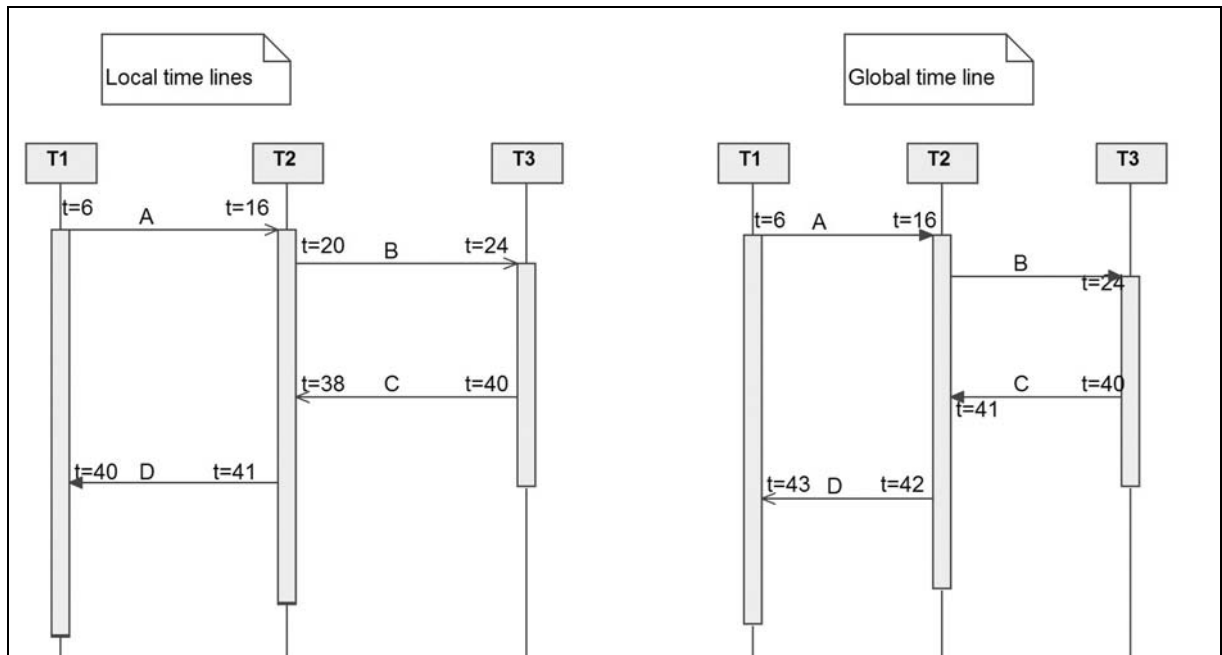


Figure 13: Communication between different processes or threads, clock synchronisation [Lamport] (UML sequence diagram)

4.5.1.2 Implementers

In Figure 14 we present a taxonomy for different implementers of the introduced functionalities. All these implementers must realize the interface `MachineLearnerImpl`. During further development iterations, new implementers might be added and used easily, by extending the upper class `MachineLearnerImpl`.

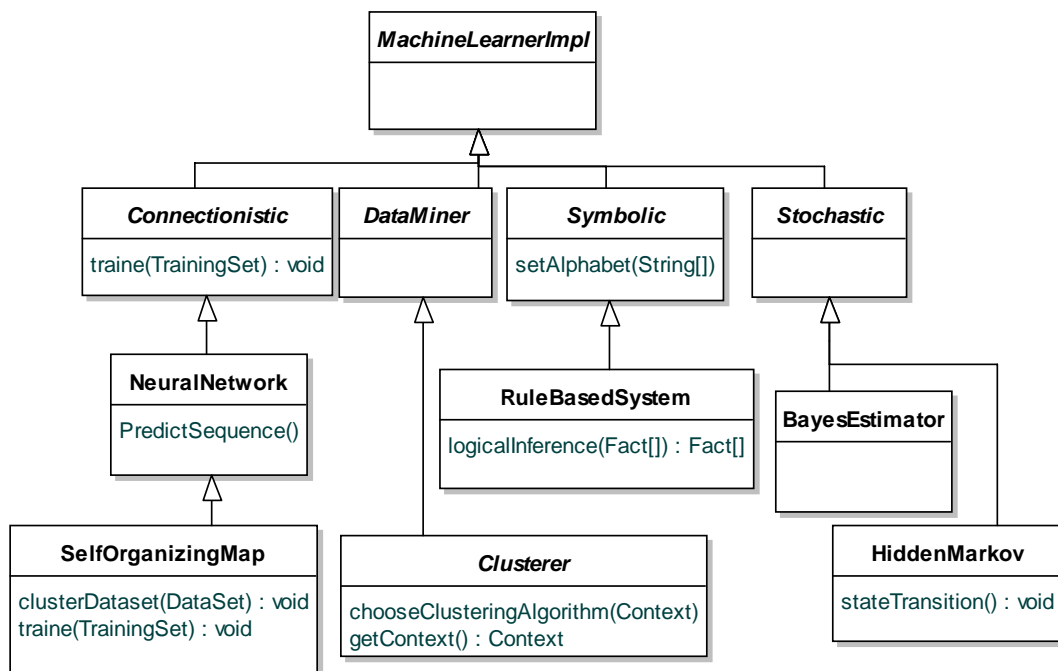


Figure 14: MachineLearnerImpl taxonomy (UML class diagram)

In general there exist different types of implementers for the above-mentioned functionalities, which can be grouped in the main classes *Symbolic*, *Connectionistic*, *DataMiner* and *Stochastic*.

Symbolic or rule-based schemes like those used in expert systems benefit from their strict systematics and high interpretability. Symbols are arbitrary, but can be composed to meaningful symbolic structures like formal grammars. Their rules are exploited by expert systems in form of $A \Rightarrow B$ implications, for example.

On the other side, the connectionistic paradigm has the powerful concept of artificial neural networks at its disposal, which are universally applicable to every problem that can semantically be represented by a mapping $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$. Main benefits of neural networks are their inherent context representation and generalization capabilities.

DataMiner means the implementation of data mining functionality, which can be realized by a standard k-means clusterer, for example. Figure 15 shows different clustering algorithms invoked by the Strategy Pattern [Gamma et al. 94].

The fourth class consisting of stochastic methods includes such as Bayesian estimators for parameter/random variable prediction or models for stochastic state transitions and random processes like *Hidden Markov Models*.

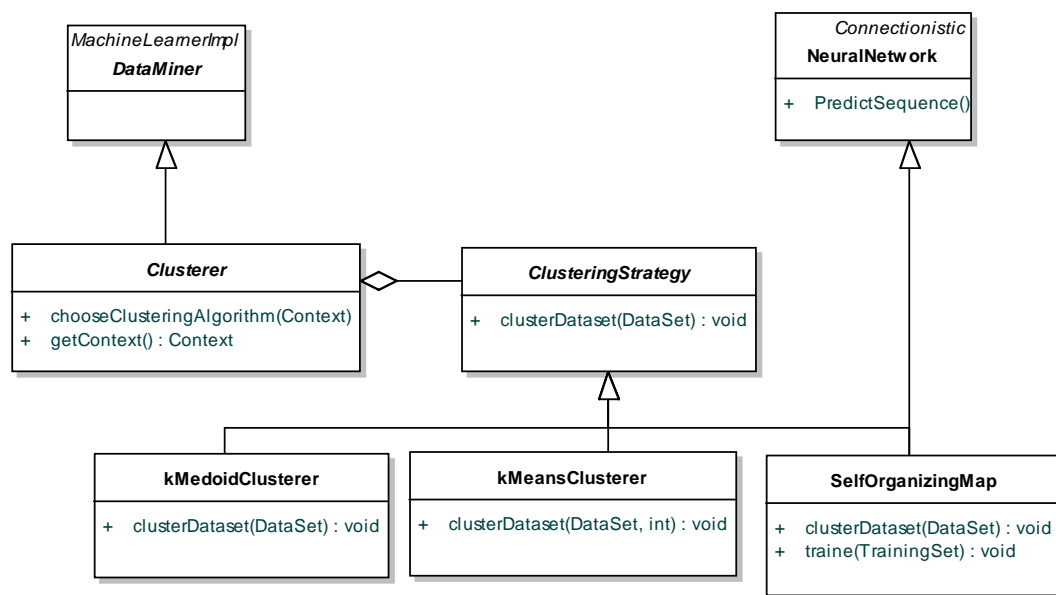


Figure 15: Modelling the clustering using strategy pattern (UML class diagram)

5 SUMMARY

In this document we introduced the conceptual model and specification of the functional requirements of the context system as result of a first iteration of requirements engineering and design activities.

As functional requirements we identified monitoring the developer's behaviours, logging the interaction with TEAM platform, discovering preferences, discovering usage patterns and observing changes. Additional functionalities include capturing, constructing, storing, synchronizing, analysing, preparing, querying, interpreting and comparing context. We also elicited non-functional requirements namely performance, supportability, extensibility, adaptability and flexibility. These requirements represent decision criteria for design and implementation issues in the current as well as future iterations.

We decomposed the context system into five subsystems: event-subsystem, context-subsystem, knowledge-subsystem, sensing-subsystem and interpretation-subsystem.

We model context as an aggregation of selected properties of targets. These targets are instrumented by sensors that measure the values of the selected properties.

Machine learning is the most important package in the interpretation subsystem. Various implementers can realize different functionalities of machine learning. Functionalities are typically classification, prediction (regression), clustering and recognition. Implementers can be grouped in the main classes symbolic, connectionistic, data miner and stochastic.

The presented model will be explored and detailed by the implementation of a prototype in the next iteration. The high extensibility, adaptability and portability of our model allows for an iterative implementation of vertical slices of the system. In the next iteration we plan to focus on one target and its sensors and one aspect of machine learning for interpretation.

A REFERENCES

- [Bruegge&Dutoit 04] Brügge, B. & Dutoit, A.H. (2004), *Object-Oriented Software Engineering: Using UML, Patterns and Java*, Prentice Hall International.
- [OMG 03] Object Management Group (2003), *Unified Modelling Language UML® Resource Page*, <http://www.uml.org/>.
- [Eriksson 00] Eriksson, H. & Penker, M. (2000), *Business Modeling with UML - Business Pattern at Work*, Wiley New York.
- [Lamport] Lamport L., *Time, Clocks, and the Ordering of Events in a Distributed System*, Massachusetts Computer Associates, Inc.
- [CRISPDM 07] Cross Industry Standard Process for Data Mining
- [EclipsePlugin] http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
- [Jacobson et al. 99] I. Jacobson, G.Booch, &J.Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999
- [Gamma et al. 99] Gamma et al., *Design Patterns*, Addison-Wesley 1994.
- [Maalej et al. 06] Maalej W., Happel H.-J., Ntioudis S., David J., D2: Report describing state-of-the-art regarding personalisation and contextualisation in software engineering, IST PROJECT 35111, December 2006. http://www.team-project.eu/documents/TEAM_deliverable_D2.pdf
- [Maalej 05] Maalej W. *Domain Independent Generation and Management of User Queries in Semantic Web Applications*, Technische Universität München, October 2006.