Tool Supported Aspectual Predesign

Vladimir A. Shekhovtsov[1], Arkadiy V. Kostanyan[1], Eugene Gritskov[2], Yury Litvinenko[2]

[1]National Technical University
“Kharkiv Polytechnical Institute”
21 Frunze Str., 61002, Kharkiv, Ukraine
shekvl@kpi.kharkov.ua

[2]Kharkiv National University
4 Svobody Sq.
61077, Kharkiv, Ukraine

Abstract: Aspectual predesign is an approach to user-verified mapping of the non-functional requirements to the system into the aspect-oriented design model. It is an extension of the Klagenfurt Conceptual Predesign. In this paper, we present the architecture of modular software tool aiming to support the aspectual predesign workflow. The XML-based Predesign Exchange Format (PEF) for conversion-independent representation of aspectual predesign model (APM) is introduced.

Key words: non-functional requirements, conceptual predesign, aspectual predesign, XML, KCPM, AOSD, APMTool, Eclipse Modeling Framework

1 Introduction

Our paper presents the current results of the research which goal is to allow conversion of the non-functional requirements to the system into an intermediate model residing between requirement analysis and conceptual design and to map this model into the conceptual model. Non-functional requirements [Chu00] are related to such important system characteristics as security and performance. This paper is the follow-up of the previous paper of the authors [SK05]. In that paper, we introduced the *Aspectual Predesign* technique allowing user-verifiable mapping of the non-functional requirements. Being the first paper about the subject, [SK05] touched the implementation issues only briefly. The purpose of this paper is to look at these issues in more detail. Its goal is to outline the directions for the practical implementation of the aspectual predesign workflow and to describe the architecture of its software prototype - *the APMTool.*

The remainder of this work is organized as follows. Section 2 gives brief background information about the Aspectual Predesign approach. Section 3 introduces the APMTool and describes its general architecture. Section 4 contains the detailed description of the core data format of APMTool called Predesign Exchange Format (PEF). A sample of the

APMTool requirements processing is presented in Section 5. In Section 6, our conclusions and the directions for future research in the aspectual predesign field are presented. The home page for the APMTool project is at **http://apmtool.sourceforge.net**.

2 Aspectual Predesign

Aspectual predesign [SK05] is a method that allows non-functional requirements to the system to be converted into the intermediate model (Aspectual Predesign Model, APM) that in turn can be converted into the conceptual model of the system. This model is more general than the conceptual model but more formal than the requirements specification.

Aspectual predesign is based on two well-known software engineering approaches: *Aspect-Oriented Software Development* (AOSD) [Asp05, CB05, JN04] and the *Klagenfurt Conceptual Predesign* [KM02, MK02, FKM03].

AOSD addresses *crosscutting concerns* or *aspects* of the system (goals and principles such as logging, security, performance that are implemented in different parts of the system but have nothing in common with their main functionality). At the requirement analysis phase, aspects correspond to the non-functional requirements to the system. At the design phase, the UML-based and proprietary notations can be used to represent the aspects. At the implementation phase, aspects can be implemented in the program code. The most widely used implementation technique is Aspect-Oriented Programming (AOP) that maintains clear separation between core classes of the system (related to its main functionality) and its aspects. Core classes and aspects are weaved together to form the final application. The AOP introduces several basic constructs that are used in the aspectual predesign:

a) *aspects* (modules that encapsulate crosscutting behavior);
b) *join points* (places where execution of base classes could be intercepted to run the crosscutting code);
c) *pointcuts* (rules that define the set of join points);
d) *advices* (the crosscutting aspect code running at join points defined by pointcuts; at the weaving stage, the code of advice is injected into the code of the base classes at the places defined by pointcut.)

The main goal of the Klagenfurt Conceptual Predesign is to implement requirement gathering that could be controlled and verified by the end-user. This goal is achieved with the help of KCPM (*Klagenfurt Conceptual Predesign Model*) – intermediate semantic model that resides between the requirement specification and the conceptual model. This model consists of semantic concepts that are more general than conceptual modeling concepts and could be more easily understood and verified by the end user. These concepts are *thing-type* (generalization of entity/class and attribute), *connection-type* (representing all kinds of relationships among the concepts), *operation* (generalization of the method), and *event*. KCPM could be presented in tabular form (as glossaries) and as the semantic network. After verification, this model could be mapped into the conceptual model. This mapping is performed according to the precisely defined *mapping rules*. The KCPM is supposed to be retrieved from the functional requirement specifications. This can be done manually or with the help of Natural Language Processing (NLP) tools. The latter approach is used in the project NIBA [Nib02]. Aspectual predesign can be seen both as an extension to the Klagenfurt conceptual predesign that allows mapping the non-functional requirements and as an intermediate step of the AOSD residing between aspect-oriented requirements engineering and aspect-oriented modeling (Fig.1).

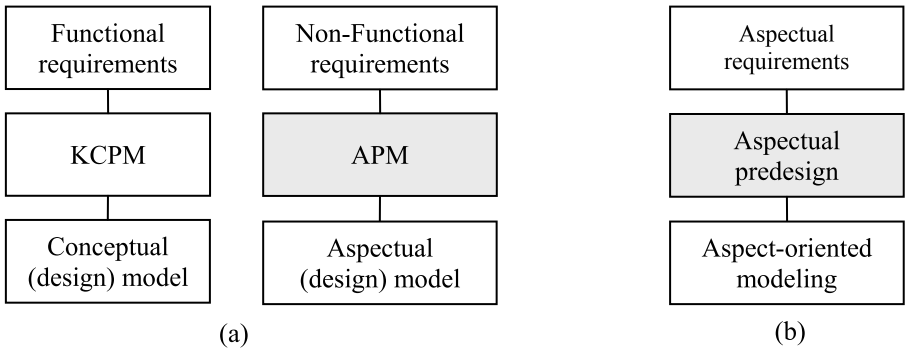


Fig.1. Aspectual predesign as an extension to KCPM (a) and as an intermediate step of AOSD (b)

There are several steps of the aspectual predesign [SK05]:

a) analysis of the natural language requirement specifications and obtaining the non-functional requirements from these specifications (this problem is very complicated and needs additional investigation);
b) user-assisted gathering of the requirements into the entries of the APM schema (this schema is an extension to the KCPM);
c) mapping of the entries of this schema into the aspectual design model (expressed in UML-based or specialized aspectual design notation).

The aspectual predesign model is based on the KCPM. Thing-types are used to represent concerns (aspectual modules), advices (indirectly called operations) are represented via the operation-types; pointcuts (rules that connect advices to some places in code where they are supposed to be called) are represented via the modified connection-types. Aspectual predesign extends the KCPM mapping rule set with additional aspectual mapping rules. The most important rules from this set are the following [SK05]:

a) aspect rule: a thing-type T is mapped into an aspect AT, if the designer in the "classification" column of the thing-type glossary specifies T as "concern".
b) advice rule: an operation O is mapped into an advice ADO if O references the thing-type, which has been previously mapped into an aspect, and O is specified as "auto-called" in the "type" column of the operation glossary.
c) pointcut rule 1: a connection-type C is mapped into pointcut PC if C references the operation that has already been mapped into an advice.

The glossary representation of the APM and the complete set of aspectual mapping rules are described in [SK05] in detail.

3 APMTool architecture

The main requirements for the architecture of the predesign tool are its flexibility and extensibility. To meet these requirements, it is necessary to develop the format for the internal representation of the APM. We call this XML-based format *Predesign Exchange Format* (PEF). It was designed taking into account the glossary representation of the model but is flexible enough to be able to represent other external representations (e.g. the semantic network representation.) All the data exchange inside the system is performed using the PEF. We will describe this format in detail in Section 4. It can also be used as a format describing the internal representation of the KCPM. The APMTool has extensible three-tier architecture (Fig.2) based on the Java platform. We start its description from the middle tier and will then proceed to the data access and presentation tiers.

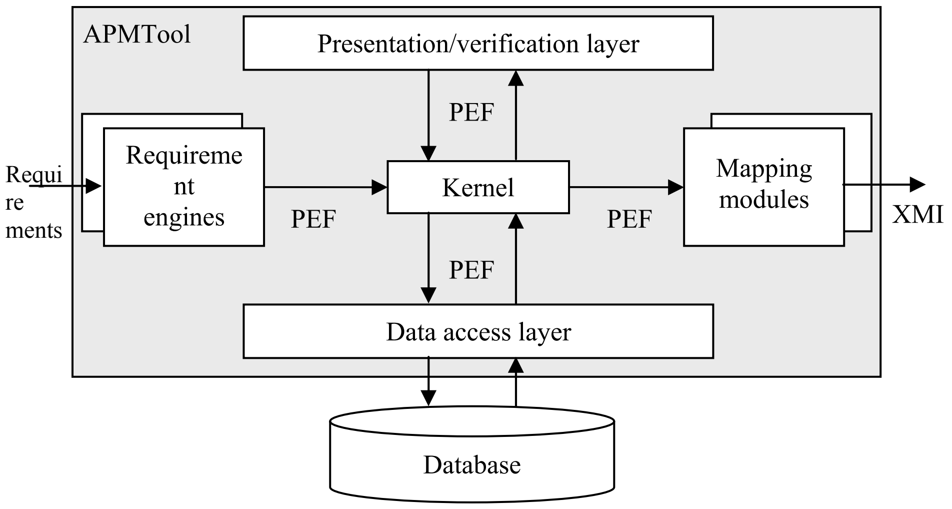


Fig.2. APMTool architecture

System kernel. The APMTool kernel is responsible for transferring the information between other parts of the system and for coordinating their work. The implementation of the kernel is based on the Spring lightweight container [Joh05]. The main kernel component is a PEF-processing engine that parses PEF structures into core objects trees. All core system components correspond to the instances of the APM artifacts such as thing-types, connection-types, and operation-types. These components are plain Java objects, their integration into the system infrastructure is supported with Spring extensibility mechanisms based on the dependency injection pattern [Fow04]. With this pattern applied, the inter-component dependencies are specified in the separate configuration file so it is not necessary to change any code to add the support for the new kind of object or to alter the associations among the objects.

The code for the PEF processing engine and for the core components is supposed to be partially automatically generated from the PEF specification using the Eclipse Modeling Framework (EMF). This framework [Bud03] allows developers of the software systems to switch between different modeling notions for their design needs. It supports the general metamodel called Ecore that can be converted into different models. Among these models are UML models, annotated Java code, and XML schema. The EMF allows the developer to start working on a model from one format (e.g. XML Schema), and later convert it to other representations. The EMF Generator framework allows generating the corresponding implementation classes from any kind of model.

At first, the XML schema of PEF is mapped into the EMF Ecore model. After that, the generation of the code for the parser, the validator, and the core components is performed using the EMF Generator framework. The kernel delivers events for the different parts of the system. For example, obtaining new requirements from the requirements engine fires an event forcing the presentation layer to update the view and the data access layer - to start the database transaction.

Requirements engines. These engines are responsible for the requirements specifications processing. It can be performed via the natural language processing techniques similar to those developed for NIBA project [Nib02], or via other approaches. All such engines must present the results of their analysis in the PEF format. These results will be transferred to the kernel for further processing. The system can contain several requirements engines implementing different approaches to the aspectual requirement analysis. For now, we are working on an interface with requirements engines along with a Java library for creating PEF documents that can be used in their code.

Mapping modules. The APMTool has to support different AOSD design notations such as aSideML [Cha04], Theme/UML [CB05, CW05], AODM [SHU02, SHU04], and to allow the user to add support for the new notations. The reason for this requirement is relative instability of the AOSD design standards. Currently there are no official standard for the AOSD design notation and it would be unwise to tie the implementation to one particular notation.

Mapping modules implement the mapping rules for the particular aspectual design notations. The mapping module obtains the PEF data as its input and applies the mapping rules for the corresponding design notation to this data. As a result, it performs the conversion into the format specific for this notation (in most cases, the target format is supposed to be XMI, but proprietary formats can also be used). We do not expect from the modules to add any diagram-specific information to the XMI data, because this information is layout-specific and is very difficult to generate. The APMTool supports different implementations of the mapping modules. If PEF-to-XMI mapping is not very complicated, the module can consist of the XSLT conversion. Another approach is to create the design models directly in code using specialized APIs suitable for this purpose. For example, EMF and EMF-based Eclipse UML2 Framework [Ecl05] can be used in this case.

Data access layer. The APMTool data access layer consists of two base components. The APMWriter component converts PEF-representation into the relational form and stores it into the database; the APMReader component obtains the data from the database and performs reverse conversion. In the current version of the system, both these components are supposed to use pure JDBC, as a result, APMTool will be able to use any DBMS with JDBC drivers.

The system architecture does not rely on DBMS-specific code (stored procedures, triggers etc.), the database is supposed to be used mostly for storing the data. The requirements for DBMS are not strict because we do not currently expect to use APMTool under heavy load or with large amount of data. APMTool currently uses MySQL 5 for storing the data.

Presentation/verification layer. This layer allows the user (business analyst or system administrator) to look at the APM artifacts in the user-friendly format and to perform the verification of the model. This layer obtains the PEF-formatted APM data from the kernel, parses and displays it, interacts with the user in the process of its verification, and sends the user-verified data back to the kernel.

We do not tie the presentation/verification layer to one particular implementation; the only requirement for its code is the PEF support. In fact, the system can use several such implementations at the same time. In the current version of the system, standalone AWT-based interface is implemented (the samples of this interface are shown below on Fig.3-5), it is planned to add web-based interface in future, in this case XSLT can be used to map PEF into XHTML.

4 Predesign Exchange Format

The design of the format for internal representation of the APM was performed with the following goals in mind:

a) **completeness:** it has to describe the APM completely (including the verification information and the information allowing working with several models at the same time;)
b) **extensibility:** it should be possible to add support for the new APM artifacts easily;
c) **self-maintainability:** for example, one should be able to apply integrity checks to the data in this format using only the format description.

To meet these goals, we introduced XML-based Predesign Exchange Format (PEF) defined by means of XML Schema. In this section, we describe the general structure of the PEF document and the presentation of thing-types and connection-types. Other parts of the format will be discussed briefly. The complete schema of the PEF is available online at **http://apmtool.sourceforge.net/pef/pef.xsd**.

General structure of the PEF document. The top-level structure of the PEF document is shown below. All the PEF structures are declared in their own namespace "pef".

```
<xs:schema xmlns:pef="http://www.kpi.kharkov.ua/pef" … >
<xs:element name="predesign_model">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="model_info" type="pef:ModelInfo"/>
      <xs:element name="requirements_library" type="pef:ReqLibrary"/>
      <xs:element name="thingtype_list" type="pef:ThingTypeList"/>
      <xs:element name="opertype_list" type="pef:OperationTypeList"/>
      <xs:element name="conntype_list" type="pef:ConnectionTypeList"/>
    <xs:sequence>
  </xs:complexType>
</xs:element>
```

General model information. The `<model_info>` element contains the general information about the aspectual predesign model (its unique identifier and description). This information allows working with different models at the same time because it is used to distinguish between the models loaded into the tool.

Requirements library. The `<requirement_library>` element contains the library of the requirements specifications. This library contains the specification texts characterized with unique identifiers and titles. Every test contains the list of specification sentences. The sentences identifiers are unique throughout the entire library (they will be referred to from the specifications of APM artifacts).

Thing-type format. The `<thingtype_list>` element contains the list of the thing types represented via `<thingtype>` elements:

```
<xs:complexType name="ThingTypeList">
  <xs:sequence>
    <xs:element name="thingtype" type="pef:ThingType"
                maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

The thing-type specification uses the simple types for its unique identifier (string-based) and for the type classification (enumeration with possible values "thing-type", "concern" etc.). The `<requirement_source>` element contains the list of the identifiers of requirements sentences from the requirements library (such lists are contained in other artifact specifications as well.)

```
<xs:complexType name="ThingType">
  <xs:sequence>
    <xs:element name="thingtype_id" type="pef:ThingTypeID"/>
    <xs:element name="thingtype_name" type="xs:string"/>
    <xs:element name="classification"
                type="pef:ThingTypeClassification"/>
    <xs:element name="requirement_source" type="pef:ReqSourceType"/> ...
  </xs:sequence>
</xs:complexType>
```

Operation-type format. The `<opertype_list>` element contains the list of the operation-types declared similarly to the list of the thing-types. The operation-type declaration (`<opertype>` element) contains the unique identifier, the enumeration-based classification element (with possible values “manual”, “auto-called” etc.), the reference to the ID of executing thing-type (the `<executing>` element), and the requirements list.

Connection-type format. The `<conntype_list>` element contains the list of the connection-types (represented via `<conntype>` elements). Inside the connection-type declaration along with connection ID, connection name, and other elements there are declaration of the two `<perspective>` elements representing the sides of the connection.

```
<xs:element name="perspective" minOccurs="2" maxOccurs="2">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="perspective_no">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[AB]"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:choice>
        <xs:element name="involved_thingtype" type="pef:did"/>
        <xs:element name="involved_opertype" type="pef:oid"/>
      </xs:choice>
      <xs:element name="perspective_name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Each perspective is characterized by side code (`<perspective_no>` element with possible values “A” and “B”), its name and either involved thing-type or involved operation-type. Both variants are the references to the corresponding artifact specifications.

Verification support. Every artifact specification in PEF is supplemented with the verification information structured as the list of verification elements. The verification element currently contains the name of the verifier, the time of verification, and the optional comments. In future, we plan to extend this information (using the separate experts/verifiers directory.)

Integrity checking. PEF-structured documents are self-maintainable with respect to the integrity checking. This self-maintainability is implemented via the `<key>` and `<keyref>` constructs of the XML Schema. The `<key>` constructs mark the unique identifiers of the artifacts (thing-type ID etc.), the `<keyref>` constructs mark the references to other artifacts:

```
<!-- thing-type unique constraint -->
<xs:key name="thingtype_key">
  <xs:selector xpath="pef:thingtype_list/pef:thingtype"/>
  <xs:field xpath="pef:thingtype_id"/>
</xs:key>
```

```
<!-- the reference to this constraint from the connection-type -->
<xs:keyref name="involved_thingtype_ref" refer="pef:thingtype_key">
  <xs:selector xpath="pef:conntype_list/pef:conntype"/>
  <xs:field xpath="pef:involved_thingtype"/>
</xs:keyref>
```

These declarations guarantee that the verification errors supplementing with meaningful error messages will be received from the XML parser if the PEF data contains

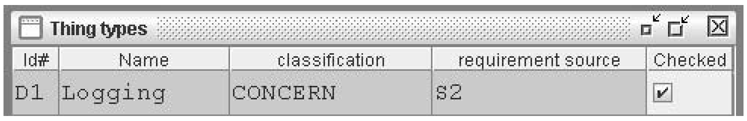
a) the `<thingtype>` elements with duplicate `<thingtype_id>` identifiers;
b) the `<conntype>` elements with `<involved_thingtype>` elements referring the non-existing thing-types.

5 An Example

We start with the simple example introduced in [SK05].

"The banking system deals with accounts and customers. All the operations with account must be logged into the file. When the customers attempt to withdraw the money from their accounts, it is necessary for them to supply a password."

Let us look at the fragments of the PEF output from the requirements engine (it would be also the output from the presentation/verification layer if the APM glossaries were filled manually.) On Fig.3-5, we present the APMTool AWT-based rendering of the different APM glossaries and the corresponding PEF code. The thing-type glossary will contain two concerns: *Logging* and *Security*. On Fig.3, the fragments of this glossary and the PEF data representing the *Logging* concern are shown.



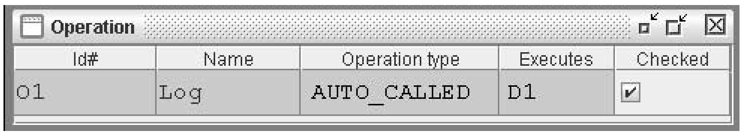
Thing types

Id#	Name	classification	requirement source	Checked
D1	Logging	CONCERN	S2	✓

```
<thingtype>
  <thingtype_id>D1</thingtype_id>
  <thingtype_name>Logging</thingtype_name>
  <classification>concern</classification>
  <requirement_source>
    <source><source_id>S2</source_id></source>
  </requirement_source>
</thingtype>
```

Fig.3. Part of thing-type glossary and the PEF data for the example specification

The mapping of such glossary fragment into the set of general aspect-oriented concepts, according to [SK05] can be performed by applying the aspect rule to map *Logging* (D1) thing-type into an aspect. The operation-type glossary will contain the *log* and *ask for password* operations corresponding to the advices. On Fig.4, the fragments of this glossary and the PEF data representing the logging operation are shown.



Operation

Id#	Name	Operation type	Executes	Checked
O1	Log	AUTO_CALLED	D1	✓

```
<opertype>
  <opertype_id>O1</opertype_id>
  <opertype_name>Log</opertype_name>
  <opertype_type>auto-called</opertype_type>
  <executing>D1</executing> ...
</opertype>
```

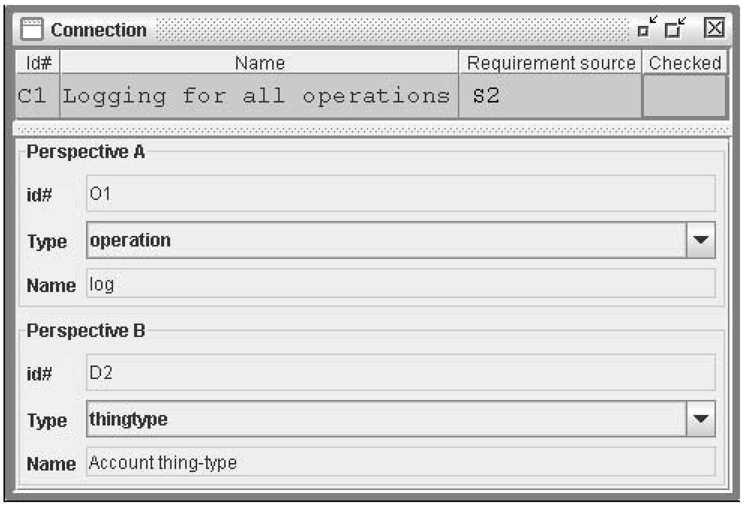
Fig.4. Operation-type glossary and the PEF data for the example specification

To perform the mapping of this glossary fragment, it is necessary to apply the advice rule to map *log* (O1) operation into an advice.The connection-type glossary contains items representing the logging pointcut defined on all operations of the Account thing-type, and the password-asking pointcut connecting the logging advice to all the operations of the Account thing-type. On Fig.5, only logging pointcut is shown.To perform the mapping of this glossary fragment, it is necessary to apply the pointcut rule to map C1 connection-type into a pointcut.The complete PEF data for this example together with more complicated examples can be obtained online at **http://apmtool.sourceforge.net/pef/examples/**.

6 Conclusions and future work

In this paper, we outlined the architecture of the Java-based software tool implementing the aspectual predesign workflow. Selecting XML-based PEF as the core format for its architecture allowed us to make an important step towards the flexibility and extensibility of the tool and to simplify the handling of the core objects and the presentation/verification layer with the Eclipse Modeling Framework. Separating requirements processing and mapping modules from the system kernel allows us to support different requirement analysis techniques and different aspect-oriented design notations. Currently, main research in aspectual predesign field is concerned with collecting the non-functional requirements from the natural language specifications (i.e. the activities that correspond to the first phases of the NIBA project like those described in [Nib02].)

The aim of this research is to design flexible architecture for recognition of language constructs specific to non-functional requirements in the specification documents. After recognition, these structures will be converted into the PEF documents that can be later verified by the infrastructure experts. The core of this architecture is supposed to be language-independent allowing the external language modules to be plugged in. Some results are already obtained and will be described in forthcoming papers, but much work still needs to be done. This problem is hard to solve, because user specifications of the non-functional requirements is one of the most imprecisely defined parts in the natural language specifications. We need to develop approaches that allow us to gather these requirements in a way that supports user validation. It is difficult because the level of experience of the user that is supposed to perform such validation is quite high by


Connection

Id#	Name	Requirement source	Checked
C1	Logging for all operations	S2	

Perspective A

id# 01

Type operation

Name log

Perspective B

id# D2

Type thingtype

Name Account thing-type

```
<conntype>
  <conntype_id>C1</conntype_id>
  <conntype_name>Logging for all operations</conntype_name>
  <perspective>
    <perspective_no>A</perspective_no>
    <involved_opertype>O1</involved_opertype>
    <perspective_name>log</perspective_name>
  </perspective>
  <perspective>
    <perspective_no>B</perspective_no>
    <involved_thingtype>D2</involved_thingtype>
    <perspective_name>Account thing-type</perspective_name>
  </perspective> ...
</conntype>
```

Fig.5. Connection-type glossary and the PEF data for the example specification

default. Usually, the business users do not understand these requirements, only system administrators and architects can work with them. Therefore, we need to define formal specification of such aspects as security, logging, performance etc. in the format that can be explained to the business user. To be able to do so, we need to perform thorough investigation of business-related facets of these properties based on real system requirements specifications. To start with, we are going to perform separate analysis of business-related facets of the several important non-functional properties, such as security and performance. The results of this analysis will be the source of the further generalization. On the later stages, we consider merging our investigations with the research of the NIBA group.

We see two main directions of this merging. The first direction aims to extend the KCPM allowing the whole spectrum of requirements (both functional and non-functional ones) to be collected and mapped into the whole spectrum of conceptual models (both object-oriented and aspectual ones) using the single workflow. The second direction aims to add language-independency to the requirements elicitation steps of the NIBA workflow (currently this workflow only supports the German language requirements specifications.) The ultimate goal is to come with the universal framework that is able to obtain various kinds of requirements from the specifications in various natural languages and to map them into various kinds of conceptual models.

Bibliography

[Asp05] Aspect-Oriented Software Development, Addison-Wesley, 2005.

[Bud03] Budinsky, F.; Steinberg, D.; Merks, E. et al. Eclipse Modeling Framework: A Developer's Guide. Addison-Wesley, 2003.

[CB05] Clarke, S.; Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach, Addison-Wesley, 2005.

[Cha04] Chavez, C.: A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004.

[Chu00] Chung, L.; Nixon, B.; Yu, E.; Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Boston: Kluwer Academic Publishers, 2000.

[CW05] Clarke, S.; Walker, R.J.: Generic Aspect-Oriented Design with Theme/UML. In: Aspect-Oriented Software Development, Addison-Wesley, 2005, pp. 425-458.

[Ecl05] Eclipse UML2 Project. URL: http://www.eclipse.org/uml2

[FKM03] Fliedl, G.; Kop, Ch.; Mayr, H.C.: From Scenarios to KCPM Dynamic Schemas: Aspects of Automatic Mapping. In: (Düsterhöft, A.; Thalheim, B. Eds.): Natural Language Processing and Information Systems - NLDB'2003. Lecture Notes in Informatics P-29, GI-Edition, 2003, pp. 91-105.

[Fow04] Fowler, M. Inversion of Control Containers and the Dependency Injection pattern, 2004. URL: http://martinfowler.com/articles/injection.html

[JN04] Jacobson, I.; Ng, P-W. Aspect-Oriented Software Development with Use Cases. Addison-Wesley, 2004.

[Joh05] Johnson, R.; Hoeller, J.; Arendsen, A.S. et al. Professional Java Development with the Spring Framework. Wiley, 2005.

[KM02] Kop, Ch.; Mayr, H.C.: Mapping Functional Requirements: From Natural Language to Conceptual Schemata. In: Proc. International Conference SEA 2002, Cambridge, USA, Nov. 4-6, 2002, pp. 82-87.

[MK02] Mayr, H.C.; Kop, Ch.: A User Centered Approach to Requirements Modeling. In: M.Glinz, G. Müller-Luschnat (eds.): Proc. Modellierung 2002. Lecture Notes in Informatics P-12 (LNI), GI-Edition, 2002, pp.75-86.

[Nib02] Niba, L.C.: The NIBA workflow: From textual requirements specifications to UML-schemata. In: Proc. ICSSEA'2002, Paris, December 2002.

[SHU02] Stein, D.; Hanenberg, S.; Unland, R.: An UML-based Aspect-Oriented Design Notation for AspectJ. In: Proceedings of AOSD 2002 International Conference, pp. 106-112.

[SHU04] Stein, D.; Hanenberg, S.; Unland, R.: Modeling Pointcuts. In: Early Aspects 2004: Workshop at International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, 2004. pp. 107-113.

[SK05] Shekhovtsov, V.A.; Kostanyan, A.V.: Aspectual Predesign. In: Kaschek, R.; Mayr, H.C.; Liddle, S. (eds.): Information Systems Technology and its Applications - ISTA'2005. Lecture Notes in Informatics P-63, GI-Edition, 2005, pp. 216-226. URL: http://apmtool.sourceforge.net/apm/apmpaper.pdf