K-ANNOTATIONS
An Approach for Conceptual Knowledge Implementation using Metadata Annotations

Eduardo S. E. Castro, Mara Abel and R. Tom Price
Instituto de Informática, UFRGS, Av. Bento Gonçalves, 9500, Bloco IV, Campus do Vale, Porto Alegre, Brazil

Keywords: Knowledge System, Conceptual Model Implementation, Metadata Annotations for Knowledge Interpretation, Aspect Oriented Programming.

Abstract: A number of Knowledge Engineering methodologies have been proposed during the last decades. These methodologies use different languages for knowledge modelling. As most of these languages are based on logic, knowledge models defined using theses languages cannot be easily converted to the Object-Oriented (OO) paradigm. This brings a relevant problem to the development phase of KS projects: several complex knowledge systems are developed using OO languages. So, even if the conceptual model can be modelled using the logical paradigm, it is important to provide a standard knowledge representation with the OO paradigm. This paper introduces the k-annotations, an approach for conceptual knowledge implementation using metadata annotations and the aspect oriented paradigm. The proposed approach allows the development of the conceptual model using the OO paradigm and it establishes a standard path to implement this model. The main goal of the approach is to provide ways to reuse both the knowledge design and related programming code of the model based on a single model representation.

1 INTRODUCTION

Several Knowledge Engineering methodologies have been proposed during the last decades for building knowledge systems (KS). For instance, VITAL (Meseguer & Preece, 1995), MIKE (Angele et al., 1998), CommonKADS (Schreiber et al., 2000), XP.K (Knublauch, 2002), RapidOWL (Auer, 2006) and KM-IRIS (Chalmeta & Grangel, 2008). Almost all those methodologies have focused on the modelling phase of the project, specifically, in the knowledge model elaboration. This model is composed by three components according to Schreiber et al. (2000):

- Conceptual Component: describes the static information/knowledge structure (concepts, attributes, relations, rules and axioms) related to the application domain;
- Task Component: defines the strategies used by the system (on the conceptual component) to solve problems;
- Inferential Component: defines the basic reasoning steps used to complete a task.

These methodologies use different languages for knowledge modelling. For instance, AI-based languages (e.g.: Ontolingua, CML) and ontology markup languages (eg: RDF, OWL). As most of these languages are based on logic, knowledge models defined using theses languages cannot be easily converted to the Object-Oriented (OO) paradigm. This brings a relevant problem to the development phase of KS projects: several complex knowledge systems are developed using OO languages. So, even if the conceptual model can be modelled using the logical paradigm, it is important to provide a standard knowledge representation with the OO paradigm, because many projects use Java and C# as languages for KS development.

However, in spite of all the methodological efforts, there is not yet a methodology that offers an extensive approach composed by guidelines and tools for the representation of the conceptual model using the OO paradigm. Currently each KS project uses an *ad-hoc* solution for the implementation in OO of the conceptual model. For instance, an *ad-hoc* solution in use by a project complex KS for underground oil reserves evaluation (Castro et al., 2008) defines the following mapping between the conceptual model and its implementation using the OO paradigm:

Castro E., Abel M. and Price R. (2009).
K-ANNOTATIONS - An Approach for Conceptual Knowledge Implementation using Metadata Annotations.
In *Proceedings of the 11th International Conference on Enterprise Information Systems - Artificial Intelligence and Decision Support Systems*, pages 67-73
DOI: 10.5220/0001863100670073

- Concepts map to classes;
- Concept attributes map to class properties;
- Facets (for constraint definition), Axioms and Rules map to external classes (validator design pattern (Fowler, 1997)) or to methods of the conceptual classes.

In consequence, not only the aforementioned ad-hoc approach but also several KS projects face the following problems:

- The ad-hoc solution design can cope with the current system requisites, but, it may become quickly inadequate because of the frequent changes in system requirements due to knowledge evolution;
- In most of cases, the *ad-hoc* solution generates components that merge the conceptual model with other requirements. In the aforementioned example, it is hard to distinguish methods that implement facets from methods that implement other requirements. This ambiguity of method role turns much harder to understand the code goals and structure and to perform testing and maintenance. Also, it is almost impossible to extract documentation about the conceptual model from the code and it is hard to maintain the documentation of the conceptual model coherent with its implementation;
- The model implementation is well-know only by the team of programmers of the project. This increases this risk of maintenance problems. This is usually worsened because of high staff turnover.

The proposed approach allows the development of the conceptual model using the OO paradigm and it establishes a standard path to implement this model. The main goal of the approach is to provide ways to reuse both the knowledge design and related programming code of the model based in a single model representation.

The approach is based on a set of metadata annotations integrated with the aspect-oriented paradigm (AOP). According to Stephens (2004) metadata is traditionally defined as 'information about information'. In the proposed approach, annotations are used to distinguish conceptual model elements from the rest of the code, and these annotations are stored within the metadata components of the interpreters. In the case of Java and C#, metadata is information about the elements of a class., for instance, constructors, methods and properties.

Widely used OO languages provide annotations for metadata definition, so the proposed approach uses this resource to add metadata. Annotations are used to clearly and visually distinguish conceptual model elements in the software code, marking them as such. As these annotations are identifying knowledge elements, they are here called K-Annotations (KA). Concepts, attributes, facets, axioms and rules are defined using k-annotations. Each annotation must be processed by the KA-Processor tool that generates the correspondent code for each annotation. Section 3.1 defines the set of k-annotations proposed in this article and the KA-Processor that generates code based on the annotations.

Integrated to the k-annotations, the Aspect Oriented Paradigm is used to avoid the proliferation of auxiliary properties and methods in the class that implements the conceptual model. For instance, in the *ad-hoc* solution, if a facet that inhibits null values is used four times, the developer must implement four times the correspondent value comparison. The code for each check is manually inserted for each use, so the system is more susceptible to bugs. An aspect is defined (Kiczales et al., 2001) as a software entity that captures a transversal functionality in relation of an application. Regarding this definition, k-annotations identifies aspects which must be managed by the tool that generates code.

The management is performed by a tool that injects the correspondent code for each annotation. So, the developers do not need anymore to insert repetitively the code, thus avoiding the phenomenon of dispersion and reducing the risk of bugs and the cost of maintenance. The activation of the code generated for each annotation is automatically configured by the annotation processor tool. It is not necessary anymore for the developer to define when it must be called. Also, a code library is part of this approach, so, the code generated by the annotation processor tool can be reduced using calls to the library.

Regarding the above propositions, several KS projects can use the proposed notation and interpreting tool for developing the conceptual model, avoiding *ad-hoc* solutions for every project.

In section 2, related works are discussed, in particular XP.K that includes KBeans (Knublauch, 2002), a proposition of an OO implementation for the conceptual model. In section 3, the central concepts of the proposed annotations and interpreting tool are briefly described, and the set of annotations to be used, with their roles, are defined. Also, in section 3, the description of a tool for code generation and a tool for monitoring the

maintenance of the knowledge body is presented. In section 4, a case study using the proposed tools is presented. In section 5, conclusions and future work are presented.

2 RELATED WORK

Among the methodologies for KS development, XP.K is distinguished because it proposes the OO paradigm as a starting point. It proposes an OO solution called KBeans for implementing the conceptual model. Section 2.1 is uses XP.K (Knublauch, 2002) as source base.

2.1 KBeans

KBeans is based on JavaBeans and code conventions for defining transparently the semantics of the conceptual model elements. Semantic transparency can be reduced to formal information about the model elements and their relations (Knublauch, 2002). KBeans proposes the use of code conventions to provide metadata information related to the model elements and their relations. It uses reflection to process each metadata and to generate the correspondent code.

KBeans maps concepts to classes and attributes to class properties. Also, it provides a pre-defined catalog of facets for constraint checking. Each facet is defined on a class using a code convention related to properties and methods. For instance, to define that a class property named *age* can not be less than 0, another class property named a*geMinValue* must be defined as 0. However, some disadvantages arise when using only this sort of code conventions to provide metadata:

- The use of code conventions creates ambiguities related to method and property roles. Properties and methods that define facets can be confused with properties and methods related to the application domain. For instance, a property named *ageMin* can be defined as a property of a concept, but it can be mistakenly be understood as a facet;
- There is a proliferation of auxiliary properties and methods in the class that implements the conceptual model. This increases the code and turns it harder to read and to maintain the knowledge base and the application;
- A large set of code conventions must be learnt by the developers. However, the language interpreter is not capable to identify the misuse of conventions. For instance, a code can be syntactically correct, but convention mistakes are not identified;
- The use of code conventions reduces the power of code refactoring. As semantic is defined using conventions, a refactoring can generate bugs in the code. For instance, if a property called *age* is refactored to *personAge*, a facet named *ageMinValue* will not be automatically refactored to *personAgeMinValue*.

To address the above disadvantages, the k-annotation approach is proposed. It is detailed in the next section.

3 K-ANNOTATION APPROACH

The discussed disadvantages can be reduced using the k-annotations approach. Annotations reduce or may eliminate the use of code conventions. This is so because each annotation has a well defined role and the related code is automatically generated by the interpreter. Annotations are checked by the interpreter and code conventions are not checked (Piveta et al., 2007). Avoiding code convention eliminates the problem of ambiguity related to the role of properties and methods. Concepts, attributes, facets, axioms and rules can be clearly identified in the code.

The proliferation of auxiliary properties and methods is reduced using annotations. Annotations can be easily processed by a tool for generating code and documentation.

The developer team does not need to use a large set of code convention which is not verified by the compiler. Annotations are validated by the language interpreter. So, it helps the developer to avoid mistakes.

The problem of code refactoring is reduced. For instance, a facet to restrict the minimum value of a property is defined using *@FacetMinValue*. If the property is renamed, it does not affect the facet.

The K-Annotations process (Figure 1) starts when a conceptual model (CM) specification is received by the developer. It is necessary to implement the specification in the OO using k-annotations (Section 3.1). After implementing the CM, the KA-processor tool verifies if the code contains any mistake. When the KA-processor detects some problem, it cancels the process of annotation interpretation and it sends warnings to the developer. If the code does not contain mistakes, the KA-processor calls the KA-DocGen that generates documentation for the CM model. The KA-DocGen

is provided to allow visual and textual comparisons between the CM specification and the CM implementation. Besides the documentation, the KA-Processor generates the code necessary to implement the semantic associated to an annotation. The generated CM implementation by the KA-Processor is interpreted with all code by the Java or C# language processor and an executable CM implementation is generated.

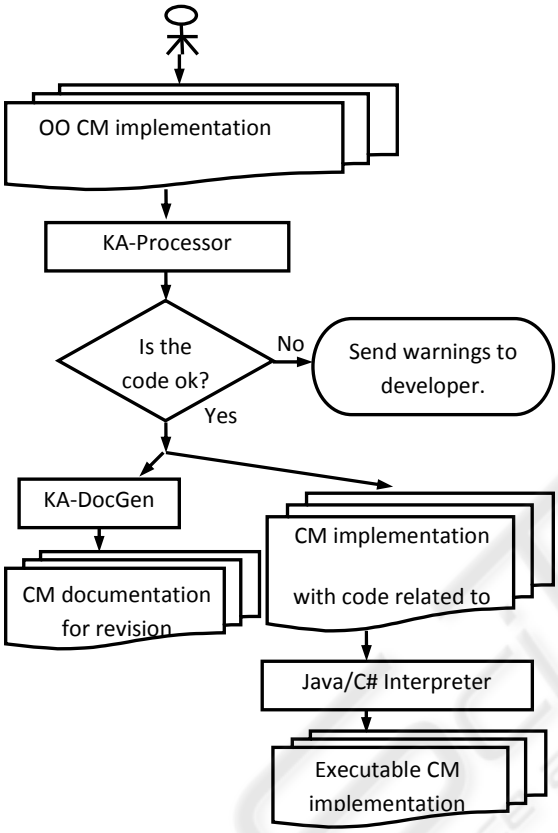

Figure 1: K-Annotation Process.

3.1 K-Annotations

The set of annotations proposed clearly distinguishes the CM elements in the OO implementation, which contains other elements related to other requirements. The k-annotations defined in Table 1 allow defining the following CM elements (examples are presented in section 4):

- @Concept: it is used with a class declaration to define that a class is a concept. The interpretation of this annotations does not directly generate code, but visually distinguishes classes from concepts;
- @Attribute: defines that a property is an attribute of a concept. The interpretation of this annotation does not directly generate code, but visually distinguishes properties from attributes of a concept;
- @HasParts: defines the parts of a concept. Each part is defined in a different class;
- @PartOf: defines that a concept is a part of another concept. It must be used with the class declaration. For instance, when this annotation is declared as @PartOf(value=B.class) in a class named *A*, the KA-Processor verifies if there is a class constructor that contains a parameter of type B. If it does not detect the parameter of the specified type, it cancels the interpretation process and it alerts the developer;
- @FacetMinCardinality/MaxCardinality: these annotations can be used to define the cardinality of collections and thus also relations;
- @Facet<Name>: all annotations whose names start with *Facet* are based on the KBeans facet catalog (Knublauch 2002, p. 109). Facets are used to define restrictions over the valid values of an attribute;
- @OnKnowledgeViolation: defines the class that manages a runtime exception generated by an event that violates a facet of an attribute. For instance, during runtime, an attribute could receive a null value, but a facet denies null value for it. So, an exception must be thrown to alert about the violation and this allows repairing the invalid state. The application should block all operations until the repair of the invalid state;
- @Axiom: it can be used with concepts to define expressions that always must be true. It requires a parameter named *value* that identifies the class that implements an axiom. An axiom must be defined in the object level using OO expressions, for instance, if-then-else. This annotation can be used to create custom facets;
- @Rule: it can be used with concepts to define rules that are related to a concept. It requires a parameter named *value* that identifies the class that implements a rule. When the KA-Processor detects this annotation, it modifies all methods that modify attribute values to call the rule instance after a modification of an attribute value. It is necessary to invoke the rule method after a value modification to validate the state of the instance.

Simple inheritance among concepts is directly expressed using OO. It is not necessary to provide an annotation with this purpose. Multiple inheritance is not supported by OO, neither by this approach. To verify if a concept is a super-type of another concept, Java and C# offer reflection mechanisms that provides this information.

Table 1: The K-Annotations for CM Implementation.

Annotation	Role
@Concept(name = *x*)	Identifies a concept.
@Attribute(name = *y*)	Identifies a concept attribute.
@PartOf (value=Element.class)	Defines that a class is a part of Element.class.
@HasParts(values=' …')	Defines the parts of a concept.
@FacetNotNull	Defines that an attribute value can not be null.
@FacetDefaultValue (value = *x*)	Defines the default value of an attribute as *x*.
@FacetMinInclusive @FacetMaxInclusive (value = *x*)	Defines that an attribute value must be greater/less or equal than *x*.
@FacetMinExclusive @FacetMaxExclusive (value = *x*)	Defines that an attribute value must be greater/less than *x*.
@FacetMinLength @FacetMaxLength (value = x)	Defines the minimal/maximum length of a string.
@FacetFractionDigits (value = *x*)	Defines the maximum length of digits.
@FacetMaxCardinality @FacetMinCardinality (value = *x*)	Defines the maximum/minimum cardinality of a collection.
@FacetValidClasses @FacetInvalidClasses	Defines the classes that a collection (does not) supports. Parameters omitted.
@FacetPattern (pattern = *x*)	Defines the string pattern *x* that an attribute value must respect.
@FacetOrdered	Defines that a collection must be ordered.
@FacetDuplicateFree	Defines that a collection must not contain duplicated values.
@FacetValidValues @FacetInvalidValues	Defines the valid/invalid values of an attribute. Parameters omitted.
@OnKnowledgeViolation(value = Handler.class)	Defines the class that manages the exception generated by an event that violates a facet of an attribute.
@Rule(value = RuleImpl.class)	Defines the class that implements a rule.
@Axiom(value = AxiomImpl.class)	Defines the class that implements an axiom.

3.2 KA-Processor: Code Generation Tool

The KA-Processor tool validates the declaration of k-annotations and it generates any code related to k-annotations. This tool process the CM implementation enriched with annotations. It adds the necessary code to implement the expected behaviour for each annotation. To comprehend this section, it is necessary to bear in mind the concepts *pointcut* and *advice* of Aspect Oriented Paradigm (Kiczales et al., 2001). Pointcut is a point in a software where a transversal functionality must be invoked. For instance, when @FacetMinLength is defined on a concept attribute, at this point, an advice must be invoked. Advice is the additional code necessary to implement an aspect. For instance, the code that validates the length of a string.

The annotations @Concept and @Attribute do not generate additional code. However, they are mandatory to identify elements of the CM models. Also, they are used to extract the model documentation.

For annotations of facets, the KA-Processor generates a pointcut that activates the advice related to the defined facet. For instance, if the facet @FacetPattern is used, before invoking the method that defines the value of a string variable, the pointcut is called to validate the pattern of the value. This group of annotations is responsible for constraint validations.

The annotation @HasParts is used by the tool to generate validation code related to the method with the signature *addPart(Object part)*. This method allows adding parts to its owner. So, part instances can be directly accessed by the owner. It must validate the class type of the parameter *part*. If the class type is not declared in @PartOf, a knowledge violation must be thrown by the method. Section 4 shows a detailed example.

The annotation @PartOf is used by the tool to verify if the annotated class contains a constructor with a single parameter. The class type of this parameter must be the same type defined in the annotation. If the constructor is not detected, the tool alerts the developer. See section 4 for a detailed example.

By using axioms, it is possible to define custom facets. An axiom is declared with a class and the processor tool generates a pointcut in each method that modifies an attribute value. So, when a value changes, all axioms are validated by invoking their advices.

Rules can be linked to concepts using the annotation @Rule. The KA-Processor tool generates a pointcut in each method that modifies attributes values. The advice is invoked after the modification of the attribute value. For instance, if an attribute has a value dependent of another attribute, when the value of the later attribute changes, the advice must be invoked.

The next section presents a case study using k-annotations and the KA-Processor tool.

4 CASE STUDY

The following case study is initially presented on a semi-formal specification based on CML (Schreiber et al., 2000) of a conceptual model (Table 2). CML is the language used by CommonKADS to build knowledge models.

Table 2: Model of the concepts Sample and Identification.

Concept Sample	
Is-a	Object
Name	string(40)
Concept Identification	
Is-a	Object
Part-of	Concept Sample
Depth	real, range [0.0 - 9999.99]
Use	string, list-of [Depositional, Diagenetic, Ecologic, Provenance], MAX (3 occurrences)
Date	date, (DD/MM/YYYY)

Table 3: Java implementation of the concept Sample.

Concept Sample

```
@Concept
@HasParts(values=Identification.class)
public interface Sample{
  @Attribute(name = 'Name')
  @FacetMinLength(value = 1)
  @FacetMaxLength(value = 40)
  @OnKnowledgeViolation(value =
    SampleExceptionHandler.class)
  String name;

  addPart(Object part);
}
```

In this case study, only a small fraction of the conceptual model is presented, the complete model can be found in (Abel, 2001). This model is part of a complex knowledge system for underground oil reserves evaluation. After the presentation of the model fragment (Table 2), an implementation in Java using k-annotations is presented (Tables 3 and 4) and the results are evaluated.

The concept named *Sample* (Table 2) results in the code defined in Table 3. It is possible to notice that k-annotations can be declared in both Java/C# interfaces and classes. The support for interfaces is offered to increase the re-use of the model implementation. The annotation @Concept is mandatory to define that KA-Processor must process any k-annotation declared in the class or interface.

Table 4: Java implementation of the concept Identification.

Concept Identification

```
@Concept(name='Identification')
@PartOf(value=Sample.class)
@OnKnowledgeViolation(value =
    MacroExceptionHandler.class)
public class Identification{
  @Attribute(name='Depth')
  @FacetMinInclusive(value = 0)
  @FacetMaxInclusive(value = 9999.99)
  @FacetNotNull
  Float Depth;

  @Attribute(name='Use')
  @MaxCardinality(value = 3)
  @FacetValidValues(values =
    'Depositional, Diagenetic,
     Ecologic, Provenance')
  List<String> Use;

  @Attribute(name='Date')
  @FacetPattern(pattern =
'DD/MM/YYYY')
  String date;

  //Constructor
  Identification(Sample owner);
  Sample getOwner(){…}; //Parent
}
```

For identifying the parts of this concept, @HasParts is used with the list of parts. This annotation also requires the definition of a method with the signature *addPart(Object part)*. This method is mandatory to allow the storage of parts of *Sample*. When the mandatory method is not identified, the KA-Processor alerts the developer. The annotations @Attribute and @Facet<*Function*> is used with properties (e.g.: *name*) and it identifies attributes of a concept and its facets, respectively.

The annotation @OnKnowledgeViolation identifies a class that manages the violation of one or more facets. A class that manages knowledge violations must contain a method with the signature *manageException (Map context)*. Variable *context* is

a map that contains the instance of the class that generated the exception and the value that violates the facet. The map is offered to allow the developer to manager or alert the user about the exception. Standard method signatures are used instead of pre-defined classes to avoid blocking the use of inheritance by classes of the conceptual model. A pre-defined abstract class would consume the single inheritance offered by OO languages.

The concept named *Identification* (Table 2) results in the code defined in Table 4. In this case, k-annotations were used directly in a class declaration. As *Identification* is a part of *Sample*, it used @PartOf. When this annotation is used, it is mandatory to define a constructor with a single parameter. This class type of the parameter must be the same as the part owner (e.g.: *Sample*). Also, the method *getOwner* is implemented for obtaining the instance of the owner from its parts.

It is possible to notice that @OnKnowledgeViolation is defined with the class declaration instead of with a property. When this annotation is declared with a class, all knowledge violations are managed by it. This functionality avoids defining the same annotation multiple times in the same class. Also, the facet @FacetPattern is used with an attribute that is a string. This facet is very useful to avoid errors related to string patterns, for instance, the pattern of dates.

5 CONCLUSIONS

This paper presents an approach based on metadata annotations for implementing the conceptual model of a KS. The proposed approach defines a common vocabulary and tools that can be shared among multiples projects. Also, it is compatible with multiple knowledge modelling languages.

The approach defines a standard path for the task of implementing the conceptual model, so the problems related to the use of *ad-hoc* solutions can be reduced and even eliminated. This may help to reduce the implementation time and improve the reusability of conceptual knowledge models.

As future work, the CM model presented in section 4 will be completely implemented using k-annotations to identify important improvements to this approach. In addition, the next step of this research is to define connections between the CM implementation based on k-annotations and the implementation of the task model. A link between both implementations can provide ways to develop inferences machines based on the use of aspects. Using aspects, redundant code for each inference machine implementation could be reduced or eliminated. The future investigation will focus on k-aspects, an approach to build reusable inference machines using the aspect oriented paradigm.

ACKNOWLEDGEMENTS

This work is supported by Finep and ENDEEPER Co. that retails PETROLEDGE, the described KS.

REFERENCES

Abel, M 2001, 'Estudo da perícia em petrografia sedimentar e sua importância para a engenharia de conhecimento', PhD thesis, Universidade Federal do Rio Grande do Sul.

Angele, J, Fensel, D, Landes, D & Studer, R 1998, 'Developing Knowledge-Based Systems with MIKE', *Automated Software Engg*, vol. 5, no. 2, pp. 389-418.

Auer, S 2006, 'Towards Agile Knowledge Engineering: Methodology, Concepts and Applications', PhD thesis, University of Leipzig.

Castro, ESE, Victoreti, FI, Fiorini, SF, Abel, M & Price, RT 2008, 'Um Caso de Integração de Gerenciamento Ágil de Projetos à Metodologia CommonKADS', In *Proceedings of the 1st Workshop of Software Project Management,* Brazil, pp. 12-21.

Chalmeta, R & Grangel, R 2008, 'Methodology for the implementation of knowledge management systems', *Journal of American Society for Information Science and Technology*, vol. 59, no. 5, pp. 742-755.

Fowler, M 1997, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, California.

Kiczales, G, Hilsdale, E, Hugunin, J, Kersten, M, Palm, J & Griswold, W 2001, 'An overview of AspectJ', In *Proceedings of the 15th European Conference on Object-Oriented Programming*, UK, pp. 327-353.

Knublauch, H 2002, 'An Agile Development Methodology for Knowledge-Based Systems', PhD thesis, Universidade of Ulm.

Meseguer, P & Preece, A 1995, 'Verification and Validation of Knowledge-Based Systems with Formal Specifications', *The Knowledge Engineering Review*, vol. 10, no. 4, pp. 331-343.

Piveta, E, Moreira, A, Pimenta, M, Araújo, J, Guerreiro, P & Price, T 2007, 'Avoiding Bad Smells in Aspect-Oriented Software', In *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering*, USA, pp. 81-84.

Schreiber, G, Akkermans, H, Anjewierden, A, Hoog, RD, Shadbolt, N, Velde, D & Wielinga, B 2000, *Knowledge Engineering and Management: The CommonKADS Methodology*, MIT Press, Cambridge.

Stephens, RT 2004, 'Utilizing Metadata as a Knowledge Communication Tool', *in Proceedings of IEEE IPCC2004*, UK, pp. 55-60.