AGH University of Science and Technology
in Kraków, Poland

Faculty of Electrical Engineering, Automatics,
Computer Science and Electronics

Methodology and Tool Supporting Cooperative Composition of Semantic Domain Models for Experts and Developers

Maciej Rząsa

Master of Science Thesis
Institute of Computer Science

Supervisor:
PhD Marian Bubak

Consultancy:
Tomasz Gubała

Kraków, September 2011

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Maciej Rząsa

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki,
Informatyki i Elektroniki

Metodologia i narzedzie umożliwiające współtworzenie semantycznych modeli dziedzinowych przez ekspertów i programistów

Maciej Rząsa

Praca Magisterska
Katedra Informatyki

Promotor:
dr Marian Bubak

Konsultacja:
Tomasz Gubała

Kraków, Wrzesień 2011

Abstract

Development of specialized software requires from computer science specialists deep understanding of the problem domain. To obtain knowledge required for creation proper application model, cooperation with a domain expert is essential. However differences in world description method by experts and developers may lead to inconvenience and failures in the collaboration.

This thesis proposes a method of domain modelling for domain experts and developers. We describe an iterative cooperation methodology where each of participants contribute to created model using methods that he is accustomed with. We also propose methods of model evolution and validation.

As a result of this work, the Domain Model Builder was implemented. It is the tool that supports iterative cooperation oriented to knowledge passing. Using the DMB presented concepts were evaluated during two sessions consisting in model creation according to presented methodology. Experiments results and participants opinions approved that the methodology is suitable to domain modelling in cooperation with an expert in that domain.

The thesis starts with problem definition and its background (Chapter 1). Chapter 1 it describes existing methods of data modelling, especially focusing on a role of a domain expert during the modelling process. It also depicts software that facilitates team collaboration.

Chapter 2 introduces a metamodel used as a framework of the domain modelling process. We describe its elements and transitions. In this chapter the proposed metamodel is compared with similar ideas.

Chapter 3 describes the methodology of domain modelling. It characterises consecutive stages of cooperation and defines roles of participants (an expert and a developer). It also delineates objectives of the cooperation and requirements of the elaborated model. Finally it discusses features that the methodology is distinguished by.

The Domain Model Builder is describes in the Chapter 4. This chapter shows how the methodology is implemented by the tool. It also depicts

design and implementation of the Domain Model Builder.

The methodology and the tool were validated with modelling sessions that involved various experts. Chapter 5 describes conclusions of two of them: modelling flood warning system and a part of the civil engineering domain.

Keywords: domain modelling, domain expert, semantic model, metamodel, software methodology, Ruby on Rails, Redmine

Acknowledgments

First of all I would like to express my gratitude to my supervisor, dr Marian Bubak, for his invaluable suggestions time and help.

I would like to thank Tomasz Gubała for insightful look and inspiring advices that could not be overestimated.

I express my gratitude to Pau Fernandez for his consultancy during my scholarship at the Universitat Politecnica de Catalunya.

I am also grateful to Marek Kasztelnik and Dominik Siwiec for help in evaluation of the concepts proposed in this thesis.

Warm thanks go to my brother, Wojciech Rząsa, because without his inestimable support and counsel it would have been difficult to finalise this work.

Parts of this work was elaborated during Erasmus Programme scholarship at the Universitat Politecnica de Catalunya under supervision of Pau Fernandez.

This thesis is related with the UrbanFlood [1], a project funded under the EU Seventh Framework Programme, Theme ICT-2009.6.4a. ICT for Environmental Services and Climate Change Adaption. Grant agreement no. 248767.

Contents

List of Figures

List of Tables

List of Acronyms

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CASE	Computer-Aided Software Engineering
CMS	Content Management System
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
CWM	Common Warehouse Metamodel
DDD	Domain-Driven Design
DMB	Domain Model Builder
DNA	Deoxyribonucleic acid
ERD	Entity Relationship Diagram
EWS	Early Warning System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MOF	Meta Object Facility
MVC	Model-View Controller
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model

RNA	Ribonucleic acid
SCM	Source Code Management
UML	Unified Modeling Language
VP-UML	Visual Paradigm for UML
XMI	XML Metadata Interchange

Chapter 1

Background

In this chapter we introduce a purpose of this thesis. To obtain this goal we depict a role of a domain expert and domain modelling in a software development process. We also show different technical solutions supporting collaboration and modelling: CASE tools, wiki systems and project management systems. Finally we introduce the objectives of this work: building a methodology and a tool supporting domain modelling for experts and developers.

1.1 Role of domain modelling and domain expert in software engineering methodologies

This section presents the state of the art in the subject of domain modelling in software crafting. Firstly we present Model-Driven Development. Next we focus on one of its implementation that focuses on the understanding of a problem domain: Domain-Driven Design. We also outline agile methodologies with special attention payed to Agile Modelling. The section is summarised with description of the role of a domain expert in the modelling process.

To facilitate understanding of following chapter, we introduce two basic definitions.

Definition 1 (Model) *A model is a simplified version of a certain concept. A model represents several feature of the original, depending on its purpose and may be usable in place of original with respect to this purpose [2][3].*

Definition 2 (Metamodel) *Metamodel is a set of rules and constructs needed for creating models [2][4][3].*

They are a base for further considerations about modelling included in this chapter.

1.1.1 Model-Driven Architecture

Haileper and Tarr in [5] defines Model-Driven Development:

> *Model-driven development* (MDD) is a software-engineering approach consisting of the application of models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle.

To implement MDD, Object Management Group (OMG) elaborated Model-Driven Architecture (MDA). In the MDA the key artifact is the Platform Independent Model (PIM) – a model of a system being designed that describes its functions but not implementation details. PIM is intended to be translated to Platfotm Specific Model (PSM). PIM is expressed in a modelling language whereas PSM is its mapping to programming language and enables deployment the model as an executable application [6].

MDA uses several OMG standards, such as:

⇒ Unified Modeling Language (UML),

⇒ Meta Object Facility (MOF),

⇒ XML Metadata Interchange (XMI),

⇒ Common Warehouse Metamodel (CWM).

The aim of the MDA is to facilitate software development by enabling higher abstraction to its design. Providing models that are platform independent MDA is suppose to increase interoperability introduced by OMG in previous standards (such as CORBA[7]). On the other hand modelling a system from different points of view, MDA introduces redundancy and adding abstraction layers hinders changes [5].

1.1.2 Domain-Driven Design

Domain-Driven Development is an approach to software design that is based on deep understanding of a problem domain [8]. It can be considered as a MDD method, however, DDD has slightly different aims than MDA. Whereas a modelling process in the former method focuses on in-depth domain comprehension, the latter one use models to achieve platform independence. Broadly speaking DDD focuses on transmitting and structuring of knowledge, while MDA solves technological issues [9].

A foundation of a software architecture in DDD is a domain model whose building blocks are outlined below [10][11]:

Entity – represents an object that has identity.

Value Object – represents object without identity, described by its attributes. Value objects are immutable.

Service – a stateless operation that refers to domain element but does not naturally belongs to any of domain elements.

Module – is used to group related entity.

Aggregate – a set of associated objects. Every Aggregate has one root, that is an Entity. A root is the only element accessible from the outside of an aggregate.

Factory – a class responsible for objects (Entities and Aggregates) creation.

Repository – a structure that encapsulates object persistence and enables obtaining a reference to an object.

Domain knowledge is built as a result of cooperation with experts [10]. The communication between a development team and experts consists mainly in conversation concerning the domain. As a result of this collaboration the team builds an *ubiquitous language* – a specialized language that precisely describes the domain and that is intended to use during whole development process.

During its usage a model should be constantly refactored to adapt it to newly gained knowledge and increase its conformity with a domain. DDD practices contains also rules that help to preserve model integration. They focuses on bounding models and providing clear methods of exchanging knowledge between them.

DDD rules supports creation of domain models. From the point of view of this thesis the most important principles are involvement of a domain expert and elaborating ubiquitous language.

1.1.3 Agile methodologies

Agile software development methodologies are a group of lightweight methods of delivering software. They base on competence of individual member of a development team [12]. As they relay on communication amongst a people, they are well suited to small or medium team and projects of such size [13].

The essence of these methods is outlined on *Agile Manifesto* [14]. The Manifesto is elaborated in the *Principles behind the Agile Manifesto* [15]. They emphasize importance of cooperation between customer and developers and personal traits of team members. The aim delineated with these two documents is to provide software that satisfy a customer with possible preservance simplicity, both in the development process and in the software architecture.

The important part of agile concept is to involve a customer in a development process. Although he is not intended to write code, but he may provide feedback basing on earlies prototypes. Thus a customer becomes to some extent a member of a development team.

User stories

Creating user stories is a simple method of requirements modelling that strongly involves a customer. A story is a unit of requirements that describes a user visible functionality *that can be*

developed within one iteration [1] [17]. A story is prepared by a customer, but then development team discusses it and may ask a client for clarifications. Therefore a customer is the one that *owns* the stories [18]. As they are written by him, they are supposed to be understandable for him. User stories are a method of problem decomposition in order to plan development.

Agile modelling

Although agile methodologies focus on providing working code rather than comprehensive documentation, there is an idea agile model creation introduced by Ambler in [19] and outlined by Abrahamsson in [16]. Methods described in those papers consists in applying agile assumptions not only to coding, but also during modelling process. Agile modelling itself is insufficient. To provide working application one need to consider combining it with other agile techniques.

As agile methods focus on the working software not on the documentation, agile models are not required to be comprehensive. Ambler in [19] defines features of model that causes that they are *just barely good enough*[2] Model fulfill this requirement when:

⇒ fulfill their purpose;

⇒ are understandable;

⇒ are sufficiently accurate;

⇒ are sufficiently consistent;

⇒ are sufficiently detailed;

⇒ provide positive value;

⇒ are as simple as possible.

As one can see, these conditions are not strict and may be treated as guidelines.

1.1.4 Unified Meta Language as a data modelling tool

UML is a general purpose modelling language used for designing object oriented software [21]. It is intended to provide possibility of describing different issues of development, beginning at requirements modelling through architecture design until deployment planning.

UML defines two categories of diagrams: behavioral and structural. The former define dynamics of modelled system and its operation, and the latter – its architecture.

The class diagram, one of structural diagrams defined by the UML specification, is a tool that enables data modelling. It depicts a class graph structure and provides a simple method of

[1] In agile methodologies development process is split into parts called iterations that lasts one to six weeks and result running and tested code [16][17].

[2] Noble and Biddle in [20] contrast *good enough software* with previous approach that claims that software should be correct and efficient. They interpret this change as one of symptoms of *the postmodern turn* in the software development.

describing object oriented design. Therefore this diagram is used to depict structure of domain models.

1.1.5 Summary – domain expert role

Domain expert presence is crucial during domain modelling. Without his aid there is always a risk of misunderstanding modelled concept. However his duty differs depending on the used methodology.

In the DDD his role is well defined:

⇒ he explains domain concept to developers;

⇒ he creates and verifies *ubiquitous language*;

⇒ he verifies prepared model.

In case of Agile method it may vary. He may act as a *customer* mentioned in agile methodologies and because of that he should closely cooperate with developers and verify their work.

As can be seen well-known methodologies indicate a need of cooperation with domain experts. Still in most cases they omit to specify such a method of this collaboration that enables seamless knowledge transmitting and verifying.

1.2 Software supporting modelling and collaboration

This section presents application that facilitates collaborative creation of software with special attention to modelling issues. We also focus on the adjustment this software for people that are not computer science professionals (such as domain experts).

1.2.1 Case systems

Computer-Aided Software Engineering tools are applications that facilitate software development process. They aim is to facilitate software development process. CASE systems provides wide functionality.

The simplest focus on creating UML diagrams and exporting them as images. Some of them omit to implement whole UML specification and provide only a few diagrams (e.g. UMLet [22] enable only class diagram creation). More advanced tools support more sophisticated functions. Visual Paradigm for UML (VP-UML)[23] supports modelling using all UML diagrams. Moreover it enables code generation from models and updating models according to the source code (*round-trip engineering*). VP-UML allows also sharing models using XMI. Thus it may be considered as a tool for MDA.

As CASE tools generally provides non-trivial functionality and an advanced user interface, majority of them are desktop application. Popular and widely-used desktop CASE systems are: VisualParadigm [23] and IBM Rational Rose [24]. Comprehensive list of CASE systems compatible with the UML is available on the web: [25].

While it is not very popular solutions, there are several web-based CASE systems. They frequently enables possibility of cooperation that is not available in classical tools. Mackay, Noble and Biddle in [26] describe NutCASE – simple web application that provides functionality of class diagram drawing. There also several tools available on the web:

1. *zOOml* – application that enables UML class diagram sketching and exporting [27].
2. *WebSequenceDiagrams* – application to creating UML sequence diagrams, using simple textual representation [28].
3. *BeoModeler* – CASE tool implemented as a Rich Internet Application (using Flash). Supports diagrams: use case, object, class, package, interaction [29].

Advanced CASE tools supports full development process and may considerably facilitate work of development team. But the rich functionality cause high complexity of these tools. For that reason they may be difficult to learn for people that are not accustomed to programming. The simpler tools, seem to be better suited for such people. Especially useful web CASE applications appear. They provide simple functionality and do not require installation, thus it should be easy to adapt.

1.2.2 Wiki

Wiki is web application oriented on collaborative creation of documents stored as web pages. Cunningham and Leuf in [30] define wiki:

> Wiki is a piece of server software that allows users to freely create and edit Web page content using any Web browser. Wiki supports hyperlinks and has a simple text syntax for creating new pages and crosslinks between internal pages on the fly.

Present wiki systems supports easy markup that allow users to format text stored on pages and embed media files (e.g. images). Many wiki systems provides version history tracking, access controll and other functions facilicating collaboration. For example MediaWiki (the engine of the Wikipedia) enables amongst others [31]:

⇒ editing articles using wiki markup

⇒ discussing about articles

⇒ tracking changes

⇒ spam filtering

⇒ full text search

Wiki systems provides functions enabling collaborative text creating. Designed to be user-friendly, they are supposed to be easy to adapt for non computer science people. Therefore they are a tool that is well suited for collaboration with a domain expert.

1.2.3 Project management tools

During project development, a team demand more specific communication tools than simple wiki documents. Time tracking, task management and work planning are only examples of these needs. To respond these requirements project management software is used. This type of application facilitates cooperation by providing functions mentioned above. Frequently they also enables code repository access, documenting with wiki systems and other specialised activities. Typically they are web applications. Examples of this kind of software are Redmine [32] and JIRA [33].

As project management tools often provides wiki systems and discussion forums, they are well suited to cooperation with domain experts (similarly to standalone wiki systems). Integrating tool used for knowledge passing with project management allow experts to accustom to this type of software and to involve in more specific activities (e.g. checking project progress or task-specific domain problems).

1.2.4 User stories tools

Originally user stories are supposed to be written down on index cards with one story per card. This solution provides simplicity and intuitiveness, but has several disadvantages. Stories on index cards are inconvenient for copying and changing. They lacks auto numbering support and it is difficult to share them (especially in distribute teams) [34].

To respond these needs, various software tools are developed. Rees in [34] discuss adaptation of different project management tools to user stories creation, e.g. issue tracker or wiki system. Although these solutions may be used to manage stories, there are specific user stories tool. Two of them are described in [34] and [17]. The aim of the former is stories management especially easily grouping. The latter focus on attaching a user story to code created on the basis of this story and as a result of that improve software documentation.

There is multitude of other tools that supports user stories. An outline of them is provided in [35]. Besides standalone applications, such as XPlanner [36], there are plugins of development tools. e.g. Redmine Backlogs plugin [37].

1.3 Objectives of this work

Understanding of client requirements is important during every software development process. However in case of specialized software, comprehension of the problem domain is the key issue.

A method for a developer to obtain needed knowledge is cooperation with a domain expert that could be for instance a life scientist, a businessman or an engineer. The problem is that a method of knowledge description used by an expert is utterly different than a software model needed by a developer.

The aim of the cooperation between these two parties is to correctly transform expert knowledge into a domain model. It requires from a developer elementary understanding of the domain

whereas from an expert – adjusting a domain description to modelling requirements.

1.3.1 Motivation

The survey contained in this chapter (section 1.1) describes different methods that support modelling and cooperation. They indicate importance of modelling as design task (MDA, section 1.1.1), provides leads for cooperation with domain expert (DDD, section 1.1.2) and describe modelling guidelines (1.1.3). However none of them provides direct methodology of cooperation between domain experts and developers.

The presented solutions omit to define precisely effective manner of expert involvement during model creation and (especially) validation. They indicate necessity of utilize experts' knowledge, but fail to specify an exact method of transmitting of this knowledge. Moreover defined methods of validation of the cooperation process seems to be insufficient for collaboration with a domain expert, because they consists in usage of working software basing on client requirements (e.g. agile, section), This approach fails to take advantage of expert's knowledge concerning domain structure.

Diverse tools described in the section 1.2 supports different activities of software development. CASE systems (section 1.2.1) are supposed to facilitate a whole development process. Other supports specific tasks: documenting (wiki, section 1.2.2) or project management (e.g. Redmine, section 1.2.3). Software that supports user stories creation are example of tools designed for specific development methodologies.

Amongst variety of applications that facilitate software production it is difficult to find a tool that is adapted for a process of collaborative building of domain model. Therefore there is a lack of a tool that would that would support effective building and validation of a domain model similarly to agile tools that facilitate user stories management.

1.3.2 Aims of the thesis

This thesis responds needs outlined in previous section. Its purpose is to facilitate a process of domain knowledge transmitting between developers and experts. To achieve this goal, we propose a collaboration methodology and a tool that supports it:

1. Requirements for the methodology:

   Knowledge transmitting oriented – the methodology should help developers to comprehend a domain being described. Participants should be able to verify if developer understands the domain knowledge correctly.

   An outcome: semantic domain model – the result of the methodology application should be a set of connected elements that represent and describe a certain discipline.

   Well suited for cooperation with non computer science people – experts involved in cooperation should be able to use a method of description that is convenient for them. They should not be obligated to learn any complex notation or software design principles.

2. Requirements for the tool:

   Implementation of the methodology – the tool should support creation and validation a domain model according to the methodology.

   Possibility of verification of the methodology – the tool should enable evaluation of the methodology by examining if a course of the cooperation is convenient for participants and if an elaborated model conforms with domain knowledge.

   Easy to use for domain experts – experts should be able to easily adapt to work with the tool.

A methodology and a tool that fulfill these requirements should improve cooperation between specialist of various domain and computer science professionals. To prove worth of proposed solutions experimental modelling sessions involving experts of different disciplines should be conducted.

Chapter 2

Metamodel: a Framework of Domain Description

In this chapter we present a metamodel used to build domain model during cooperation between an expert and a developer. It describes main parts of the metamodel (*entity*, *attribute* and *association*) and their transitions (*merge*, *split* and *extract*). We also show relation between formal model and domain description and discuss sources of this solution: DDD and UML class metamodels.

2.1 Elements

The metamodel is a static part of proposed domain modelling methodology. It serves as a framework to develop domain model basing on natural language description. Building blocks are shown in the Fig. 2.1 and described in this section. The most important part of the metamodel is an *entity* (section 2.1.1) which gathers whole knowledge related to specific concept. Simple features of an *entity* are described by a set of *attributes* (section 2.1.2) whereas *entity* interconnections are characterised by *entity associations* (section 2.1.3). *Alternative names* of model elements (section 2.1.4) facilitates resolution of terminology vagueness.

As solutions presented in this thesis are supposed to be rather an evolution than a revolution, the metamodel is based on several concepts, that are described in the Chapter 1. Domain-Driven Design is source of an Entity as central element of the metamodel linking real-word concept with object-oriented structures (see section 1.1.2). As the metamodel bases closely on object-oriented languages structure represented, thus it is similar to the UML class diagram metamodel (see section 1.1.4). Basing on the mentioned solutions the presented metamodel enhanced them adding possibility of semantic modelling.

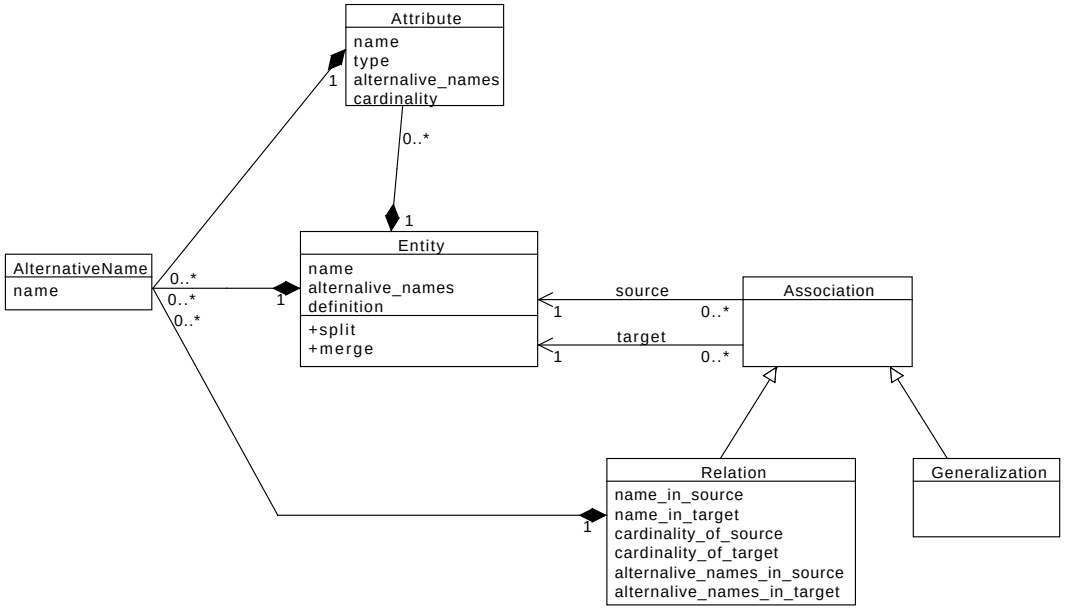


Figure 2.1: Domain builder metamodel.

2.1.1 *Entity*

Main building blocks of any description of the reality are definitions. Each definition consists in two main parts: *definiendum* (a defined word) and *definiens* (a description) [38][39] and fulfill a dual function. Firstly it introduces a proper vocabulary, that apply for the domain. Creation of a set of definitions produces foundations of a specialized language that can be used during discussions. Second function of definitions is depiction parts of the domain. A group of related definitions provides a precise description of a reality part, disambiguates used terms and assigns them an unequivocal meaning.

Definition 3 (Entity) *Entity is an element of the metamodel that describes a single concept of modelled domain. Entity is defined by its name, a set of attributes (Def. 4) and a set of associations (Def. 5). Entity is associated to a concept definition that is written using natural language.*

An *entity* is an equivalent of a definition in the metamodel (see: Fig. 2.3). It is a main building block of the metamodel and is used to depict a single concept of the described domain. An entity is identified by name that corresponds to the definiendum.

A substitute of the definiens is twofold. First part is natural language description enriched with necessary figures, formulas etc. It encapsulate essence of a defined concept and should be rather simple than comprehensive. Therefore it may contain illustrations that depict defined idea, mathematical formulas that provides quantitative information and text fragments that summarize knowledge referring to described concept.

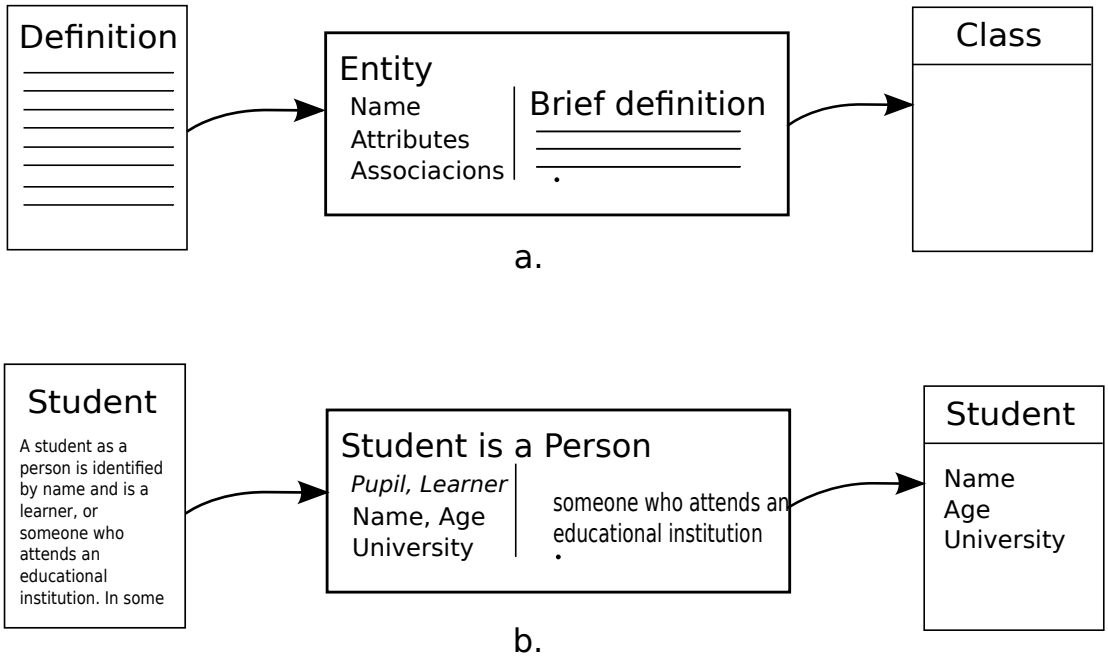


Figure 2.2: An *entity* lays between a real word concept and a programming language class. Intended to be easily understood by an expert and a developer, an *entity* enhances a knowledge exchange process. The entity concept: a. overview; b. example.

Features of the *entity* (*alternative names*, *attributes*, *entity associations*: *generalizations* and *relations*; see: Fig. 2.1) are complementary description of the modelled concept and formalise its definition. Although one entity is identified by one *name*, it can posses several aliases (*alternative* names see section 2.1.4) that occur during the cooperation. Features of the *entity* are described by *attributes* (simple features, section 2.1.2) and *relations* (complex features). A *generalization* servers to encapsulate fact that certain entities have common parts. The *generalization* and the *relation* are types of the *entity association* (section 2.1.3). Associations serves to connect entities to each other. All of elements mentioned in this paragraph are described carefully in successive subsections.

The *entity* is a simplified equivalent of the class in object-oriented modelling, thus it can be directly mapped to a class definition.

2.1.2 *Attribute*

A complex being may be partially described by displaying set of its simple features, those that could be characterised with a short value or set of short values. For a student examples are: name (few words), date of birth (few numbers), or list of used email addresses (a set of addresses). To express this in an *entity*, it has set of *attributes* (Fig. 2.3).

Definition 4 (Attribute) *Attribute is a metamodel element that describes a simple feature of a concept. Attribute is characterised with: name, type, cardinality and a list of sample values.*

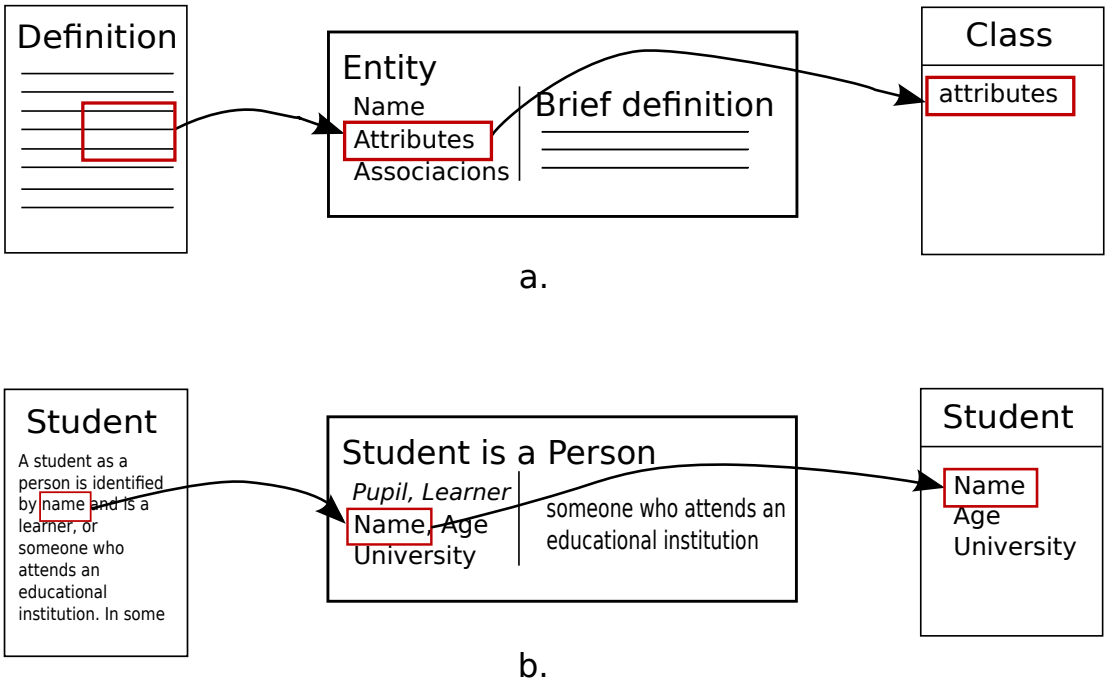


Figure 2.3: *Attribute* serves as a description of simple entity features. Describing simple entity features, *attributes* serve to separate incomplex aspects of a described idea. *Attribute* concept: a. overview; b. example.

The first of them, a name, is the most important one. An *attribute name* should be unique amongst one *entity attributes* set as the name identifies an *attribute*. And as it describes an essence of an *attribute*, a name should be selected with care, taking a domain terminology under consideration. An *attribute* has also an attached list of alternative names. Their role is analogous to another cases and is described in a subsection 2.1.4.

A feature complementary to an *attribute name*, a set of *sample values* characterizes exemplary data that could be described by the attribute. *Sample values* help to understand a purpose of an *attribute* and enable possibility to infer further features of it (*type* and *cardinality*). Although they do not posses strict equivalent in object oriented modeling, they can be useful during preparing test cases during development phase.

Type is one of those *attribute* features that are more important from the programming point of view than from the semantic one. It enables to determine data type of a class field during development phase. Classifying an *attribute* structure, a type definition helps to understand what kind of data is represented by it. Basic *attribute* types are: whole numbers (`integers`), real numbers (`floats`) and texts (`strings`). This list can be extended in specific cases.

Similarly to type, *cardinality* is especially useful during development, but what is more it also represents significant part of domain knowledge. By defining cardinality, one can distinguish between features that can be singular (*one*) or plural (*many*). Singular ones are name or age, and plural – list of favourite quotations or used emails.

The *attribute* is an equivalent of the class field in object-oriented programming. The *name* is mapped to a field name, the *type* to a field type. *Cardinality* decides if a field is a simple variable or an array.

2.1.3 *Association*

Describing only simple features of the *entity*, *attributes* fail to depict concepts interrelationships and complex features introduced by them (cf. Def. 4). These connections are essential to obtain model integrity and relate all dependencies inside the described domain. A representation of these junctions in the metamodel are *associations*.

Definition 5 (Association) *Association is an ordered link between two entities (called association ends): one of them is a source of the association and another one is its target.*

Associations has two subgroups: *generalizations* an *relations*.

Generalization

The main reason of using *generalization* is categorisation of *entities*. This type of *entity* interconnection can be described as *is a kind of* interrelation: *Student* is a kind of *a person*. It enables clustering *entities* by its superentity.

Definition 6 (Generalization) *Generalization is type of association (Def. 5) whose source and target represent similar ideas, but the idea described by the source is more specific than the one described by the target.*

The secondary purpose of introduction of *generalization* is reusability of *entity* elements. When several *entities* posses similar features it is highly possible that the intersection of the features could be extracted as *generalizing* entity. This technique facilitates modelling by reducing model complexity.

Generalization is mapped onto class inheritance. With regard to model simplicity one entity can posses only one superentity. This limitation is also important to obtain seamless code generation because many programming languages (e.g. Java, Ruby) do not support multiple inheritance.

Relation

Definition 7 (Relation) *Relation is type of association (Def. 5) that represents complex features of modelled concepts or their strong connection. Each end of relation is characterised with a name and cardinality.*

Containment and strong connection are two types of inter-concept link represented by the *relation*. In first case (containment) *relation* describes complex features of an *entity*, that cannot be depicted by *attributes*. Semantics of this *association* is: *source entity consists of target one*

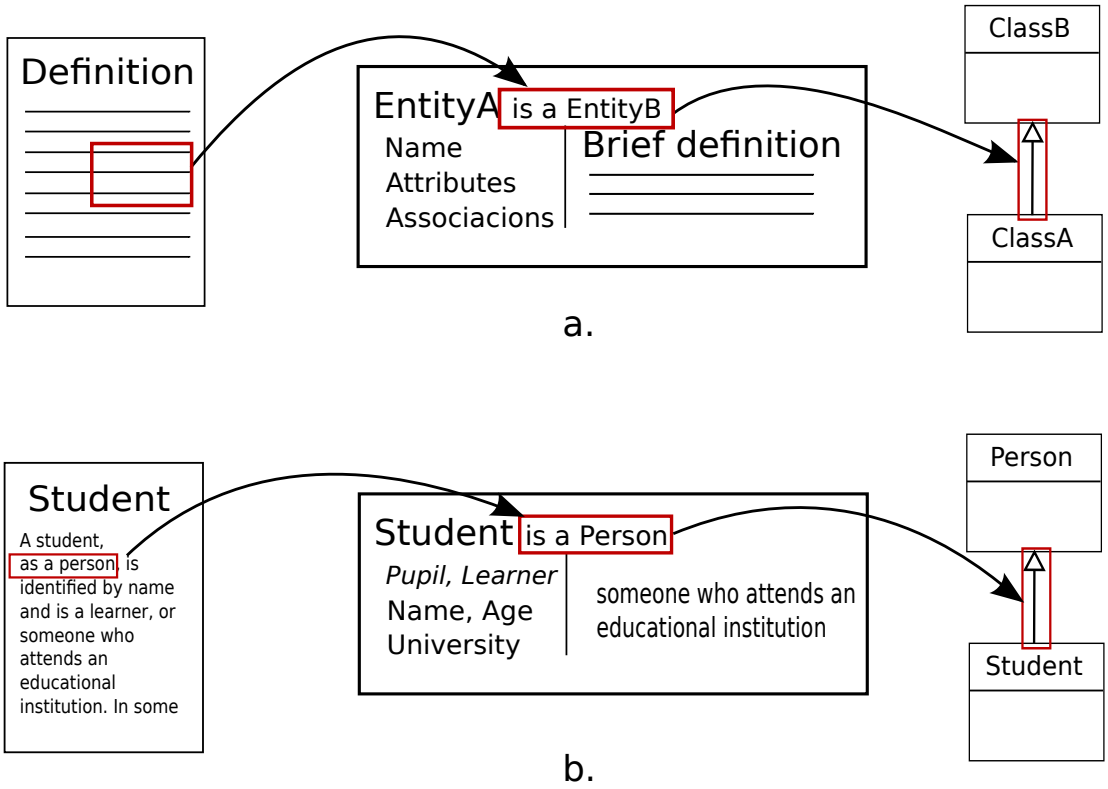


Figure 2.4: A *generalization* is an *entity association* that serves to represent fact when one concept is kind of another (e.g. a student *is a kind of* a person). It is also directly mapped to class inheritance in object oriented programming. The *generalization* concept: a. overview; b. example.

and *target is part of* a *source* (Fig. 2.5). The second case (strong connection) corresponds with two *entities* that are not containment relation in real word, but they require one another to be understood. Semantics of this meaning is: *source has a target* and a *target belongs to* a *source*.

The relation extends the *association* definition by adding several fields that describe both ends. *Name in source* is a *relation* field identifier in *source entity*. *Alternative names* in source enables resolve naming problems. *Cardinality* (*one* or *many*) of a *source entity* determine multiplicity of source entity in this relation (one or many). A *target* end posses equivalent features that acts likewise.

The *relation* in both cases is mapped onto a class aggregation. Names in a *source* and in a *target* becomes fields names that relates to respective *aggregations*. *Cardinalities* are base to decide if *relation* should be represented as single reference or as a array of references.

2.1.4 *Alternative names*

During creating a description of a domain naming doubts can arise. Furthermore one element may be described by many different names. Methodology participants have to decide, which

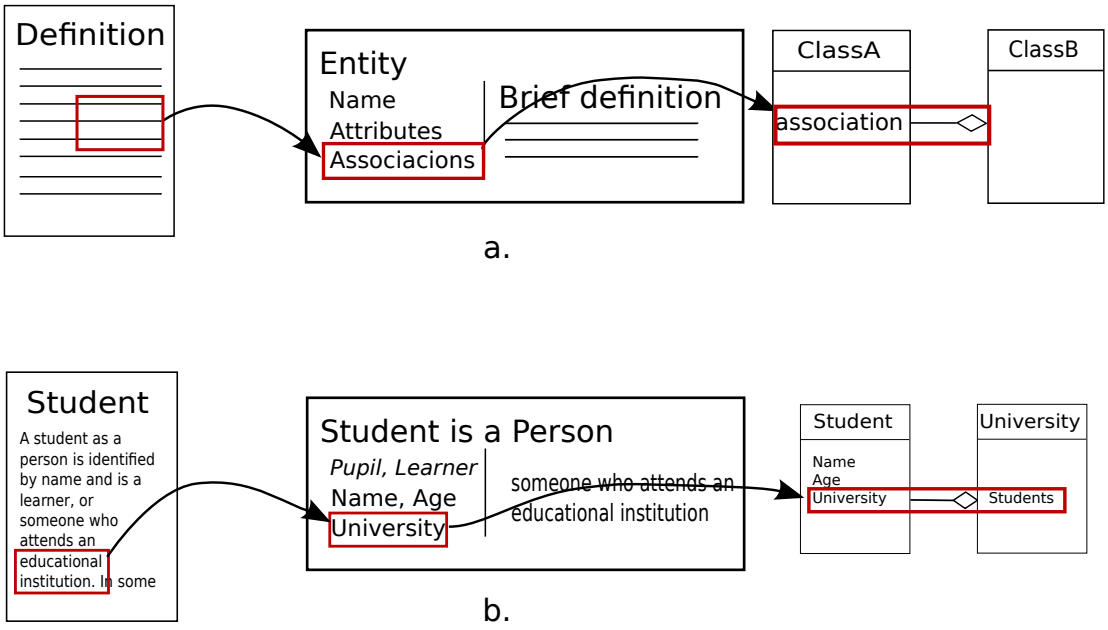


Figure 2.5: A *relation* has two-fold semantics. Basically it expresses that a being *consists of* other ones (e.g student group *consists of* students). Second meaning of the *relation* is strong connection between two beings, represented by a statement *has a*, e.g. a student *has a* supervising professor. Both meaning of association are mapped to a class aggregation. The relation concept: a. overview; b. example (second case).

name is the most adequate one and set is as main name of element. Remaining ones could be saved as *alternative names*. Moreover one of them could be set as code name - name used to identify element is generated code.

Definition 8 (Alternative names) *Altenative names are a list of names that equivalently define a metamodel element. A list of alternative names is attached to every metamodel element that is characterised with a name, i.e. entity (Def. 3), attribute (Def. 4), and relation (Def. 7).*

This feature is especially important if code and model languages are different (e.g. model description is made in Polish and code identifiers are in English). Elements that can have *alternative names* are: *entity*, *attribute*, and *relation* (*alternative names* in *source* and in *target*).

2.2 Transitions

Because of iterative nature proposed methodology, participants should be able to evolve a model in an easy way. A final model should represent distinct concepts in distinct entities and furthermore one concept should be mapped to exactly one entity.

Definition 9 (Model transition) *Model transition is operation conducted at the same time on several model elements that changes their state.*

To facilitate a process of obtaining such consistency we define three operation on the model (called transitions): *split* (section 2.2.1), *merge* (section 2.2.2) and *extract* (section 2.2.3).

2.2.1 *Split*

As we propose to develop a model in top-down approach, an initial coarse classification of domain into entities must be elaborated and specified in further cooperation. With growing amount of details in its definition and features, an entity may become describe too wide part of a domain. It is also possible that a feature that initially seemed to be simple enough to be represented as an attribute, needs more exhaustive description. In these and similar cases *split* is the operation that facilitates model evolution (Fig. 2.6).

Definition 10 (Split) *Split is a model transition that creates a new entity using elements and definition of an existing one.*

During *split* any subset of entity elements can be chosen as a base to create new one. A part of the features transferred to a new *entity* may remain attached to original *entity*. In this case they are copied instead of being moved and become an intersection of these entities.

Split operation is intended to achieve state when one entity describe only one real-word concept. It is useful for step-wise process, when the modelled domain is described as a couple of entities (in particular as one entity) and then iteratively split into smaller ones.

2.2.2 *Merge*

During development of a model it may happen that information related to one concept is spread over several entities. Lowering the cohesion, described situation is cause reduction of the model clarity. There are two cases when described problem emerges: recurrent entities and overlapping ones.

Definition 11 (Merge) *Merge is a model transition that transfer elements or parts of definitions between two entities.*

In first case two entities depicts the same concept, but each consists different elements (Fig. 2.7.a). Given that merge joins two *entities* and create one that consists of a sum of recurrent *entities* elements. Second situation happens when one *entity* contains elements that refer to another already created one (Fig. 2.7.b). In that case *merge* omit to delete an entity, but only move or copy several elements between *entities* (information exchange).

2.2.3 *Extract*

The aim of an *extract* operation is to facilitate knowledge-to-model transition. For an expert the most natural way of creating domain description is writing it as a single document. Especially during initial stage of the cooperation, it can be problematic for an expert to divide the description into separate concepts. Therefore preferable way of starting the cooperation is writing one

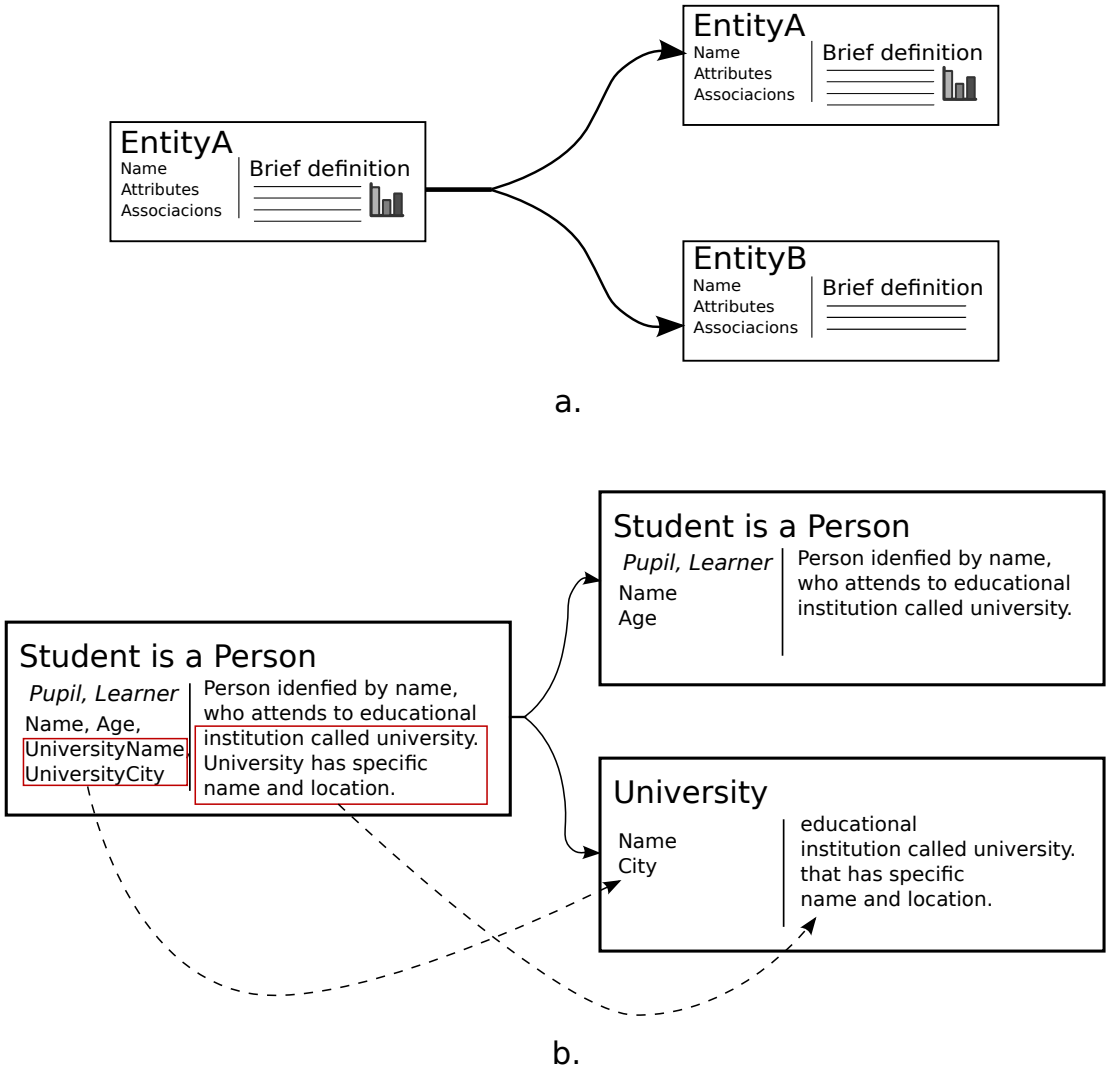


Figure 2.6: During top-down model development, a split operation is an elementary transformation. When one entity describes several fundamental domain concepts, collaborators separate new entities by consecutive splits. As a result cohesion of the model grows. The *split* operation: a. overview; b. example.

document that depicts whole domain. During next steps the document is transformed into a set of *entities*. This operation is intended to help participants to perform this process.

Definition 12 (Initial definition) *Initial definition is a document that coarsely describes modelled domain and is prepared as a first document during the cooperation process.*

Definition 13 (Extract) *Extract is a model transition that creates a new entity using parts of an initial domain definition.*

Extract creates new entity basing on fragments of *initial domain* definition. Selected definition parts becomes a definition of a newly created *entity* (Fig. 2.8).

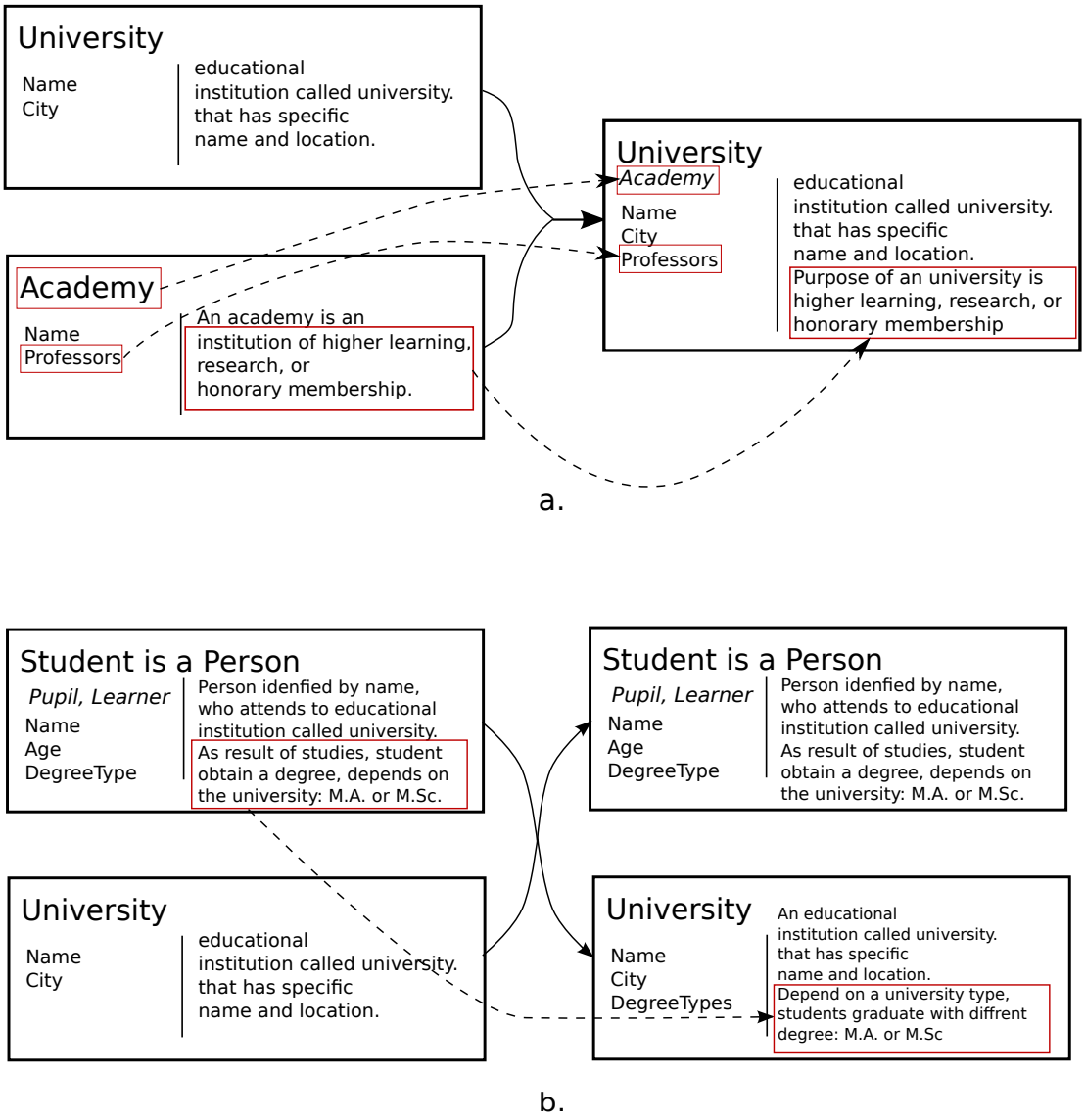
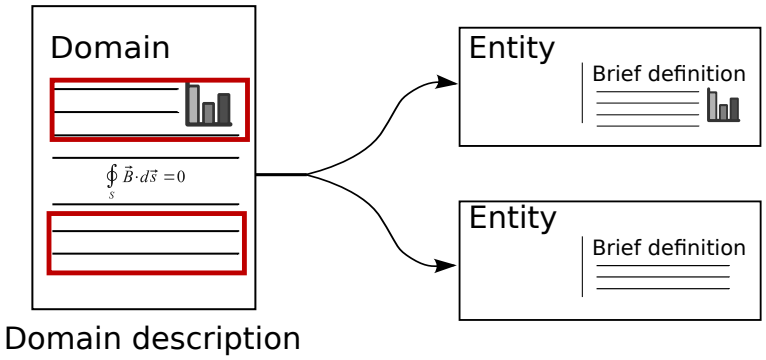


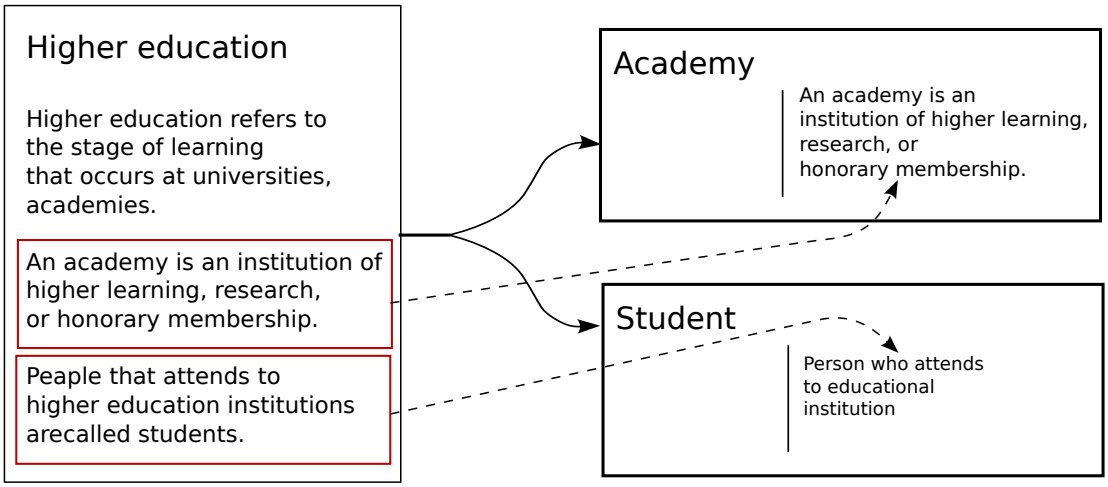
Figure 2.7: A *merge* operation has dual purpose. Firstly it serves to eliminate recurrent or similar *entities* by merging them into one. Secondly it may be used to exchange data between two overlapping *entities*. Examples of two types of *merge*: a. simple merge; b. information exchange.

2.3 Conclusions

The presented metamodel is a bridge between domain description used by experts and class structure that is a language used by developers. The metamodel enables relating this two methods of world description by creation a semantic domain model. Elements that are parts of the metamodel provides a language to formalise definitions provided by an expert and structurise it into a form that allows code generation of domain classes.



a.



b.

Figure 2.8: *Extract* is creation of new entities basing on a domain description document that is an outcome of initial steps of the collaboration. It is similar to *split* because it divides an extensive text into smaller definitions encapsulated in *entities*. The *extract* operation idea (a) and example (b).

Chapter 3

Methodology of Domain Model Composition

In this chapter we describe a methodology of the cooperation that aim at creation of a semantic domain model. Firstly we characterise participants of the collaboration: a domain expert and a software developer. Then work steps: initialization, iterations and stop conditions are introduced. Next we depict tasks assigned to each participant of model creation and we define objectives and a stop condition of the collaboration. Finally we characterise methodology features that distinguish this solution from similar ones: semantic integration, early development possibility and knowledge flow tracking.

3.1 Participants of the cooperation

The methodology that is presented in this document is intended to facilitate cooperation between two collaborators: a domain expert and a developer. Before we define their roles in the modelling process, we will describe them.

3.1.1 Domain expert

A domain expert is a person that posses extensive knowledge concerning discipline that is intended to be modelled. He could be a scientist, for instance a biologist that needs a DNA simulation, a businessman e.g. an owner of small factory that want to computerise the production or a qualified clerk that assists development of new software that supports communication with citizens.

In some cases he could be identified with a client in development process. However intimate knowledge concerning the modelled domain is a necessary condition. A shop owner that needs

a simple website to advertise his products is evident example of client that is not a domain expert at the same time, as his broad understanding of sales is not useful during development of a CMS (Content Management System) application.

3.1.2 Software developer

A developer is computer science professional that specialize in software development. He is not required to be experienced in modelled domain, but he needs knowledge about the domain to create proper software.

3.2 Overview

The aim of this section is to depict stages of the modelling process To achieve this goal, we describe consecutive phases of the collaboration: initialization (section 4.3.4) and iteration (section 3.2.2). We also define validation of the model and a stop condition that finalize the process (section 3.2.3).

3.2.1 Initialization: Defining and extracting

At the beginning of the cooperation participants should collect as much domain knowledge as possible. They are not required to care about organizing collected data. It is not essential to eliminate duplication and minor disambiguation. The point of this step is to outline main points of modelled domain and describe it in holistic way. This phase is shown in the Fig. 3.1.

An expert task during the initialization is to define main concepts of the domain. To make it as effortless as possible, he should use a language that is specific to his discipline. Besides natural language descriptions, an expert can use figures, formulas and graphs. Whole description may be put in one document (an *initial definition*).

Developer should split document made by an expert into parts that describe distinct concepts. Data (text, figures, formulas etc.) from each part is foundation to build *entity*. Developer should name each *entity*, propose its *attributes* and connects created *entities* with *associations*.

After this initial step basic domain model is attained. It consists of several *entities* describing main concepts of modelled domain. Each *entity* should have initial definition *name* and *definition*. It can have *attributes*, but it is not essential to define them precisely (by putting their *cardinality* and *type*). More important is to define a *name* and *sample values* of each of them. The goal of adding *entities associations* is similar: they should be identified rather than precisely defined.

The best circumstance to conduct this step is face-to-face meeting of participants. Personal contact of would be for expert a good opportunity to explain to developer building blocks of a domain. After such session conclusions could be recorded as a document or even as bunch of *entities*. However, if organising a meeting is not possible, cooperation could be conduced via the Internet. In this case participants should start with creating coarse domain description and then divide it into separate *entities*.

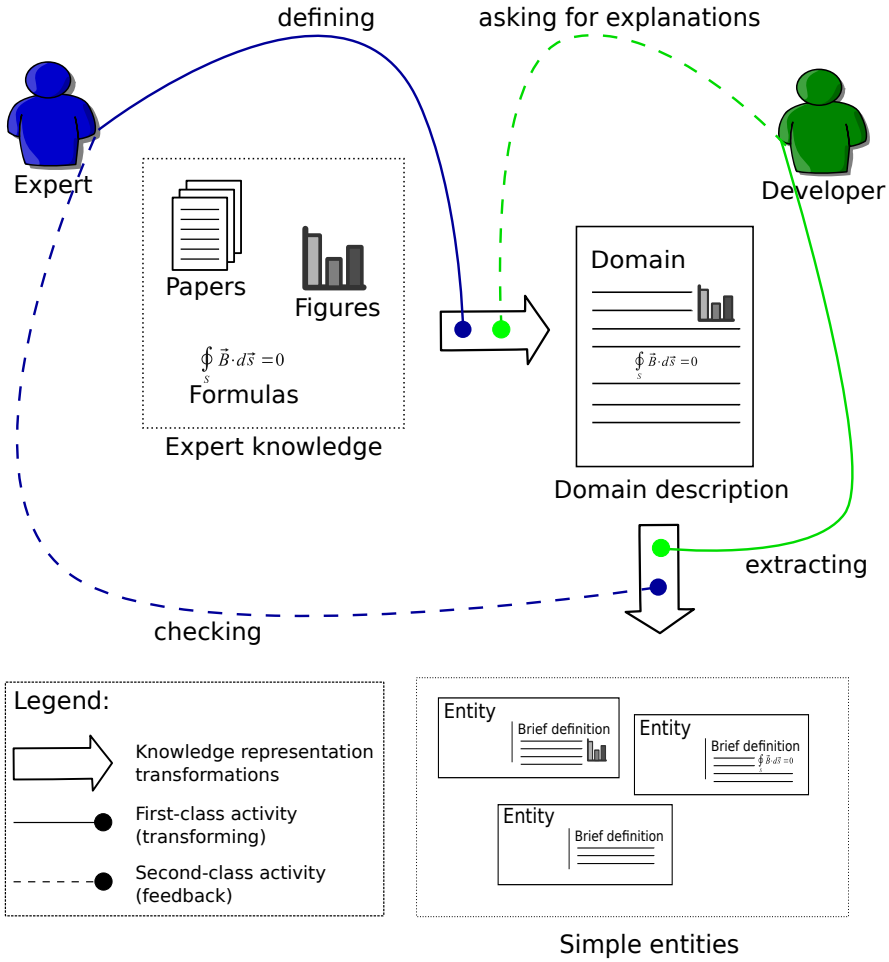


Figure 3.1: Initialization of cooperation. The goal is to create coarse domain description and extract several basic entities. Collaborators should focus on knowledge gathering (first step) and basic entities extracting. Information redundancy is acceptable at this point. In the event of effective communication between parties, creation of domain description document might be omitted. In this case the knowledge transmitted by an expert could be directly transformed into simple entities set.

3.2.2 Iteration: Correcting and adding details

After establishing foundation of model during initialization phase, participants should specify its details. The goal of this phase is to obtain model state when one *entity* describes precisely one concept and information related to specific domain part is encapsulated by exactly one *entity*. Furthermore *entities* should be connected by proper *associations* and defined by their *attributes*. Outline of the iteration phase is shown on the Fig. 3.2.

While during the first phase of the communication participants concerns a general vision of the domain, throughout iterations they should take care about details. Correct names of elements should be specified. *Alternative names* should be recorded and one of them should be

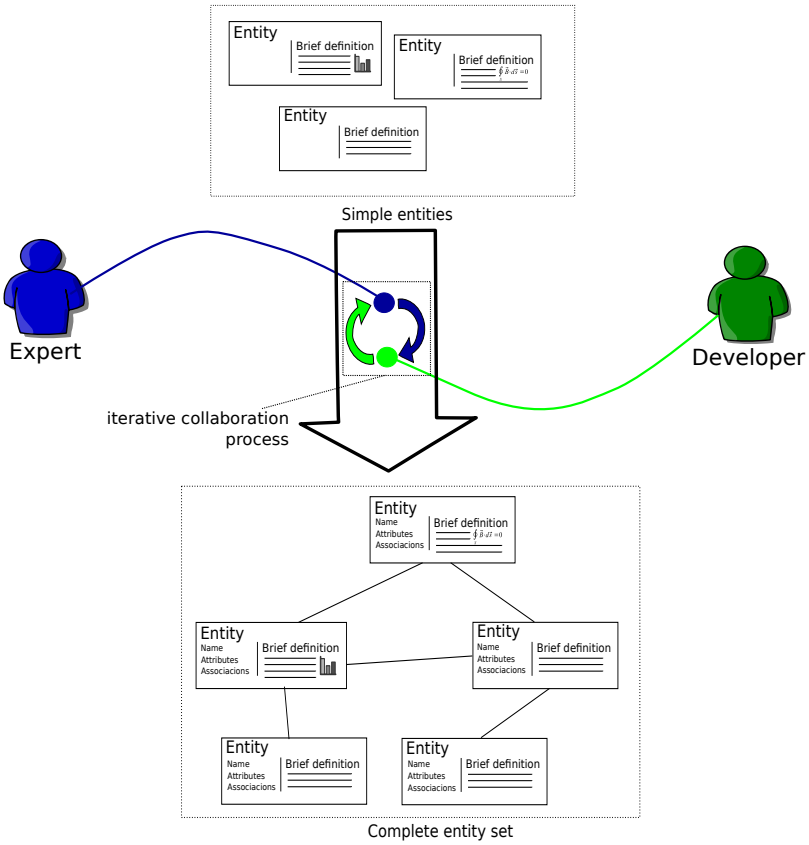


Figure 3.2: Iterative model elaboration. The goal is to create a cohesive and consistent set of *entities* that precisely describes the modelled domain. Parties should focus on extracting *entities* features and *association*. Collaborators ought to concentrate on redundancy removal and specifying model details.

chosen as a *code name*. *Cardinalities* and *types* should be defined.

The expert role (Fig. 3.3) is to review elements created by developer and to correct mistakes. He should check if extracted *entities* correspond with domain concepts. His task is also to specify definition details, to explain ambiguities and to answer questions submitted by a developer. He should also propound developer corrections of model elements.

A developer should take into consideration observations of an expert. His task is also to transform model in order to obtain consistency. He can obtain it applying transformations to

already created *entities*.

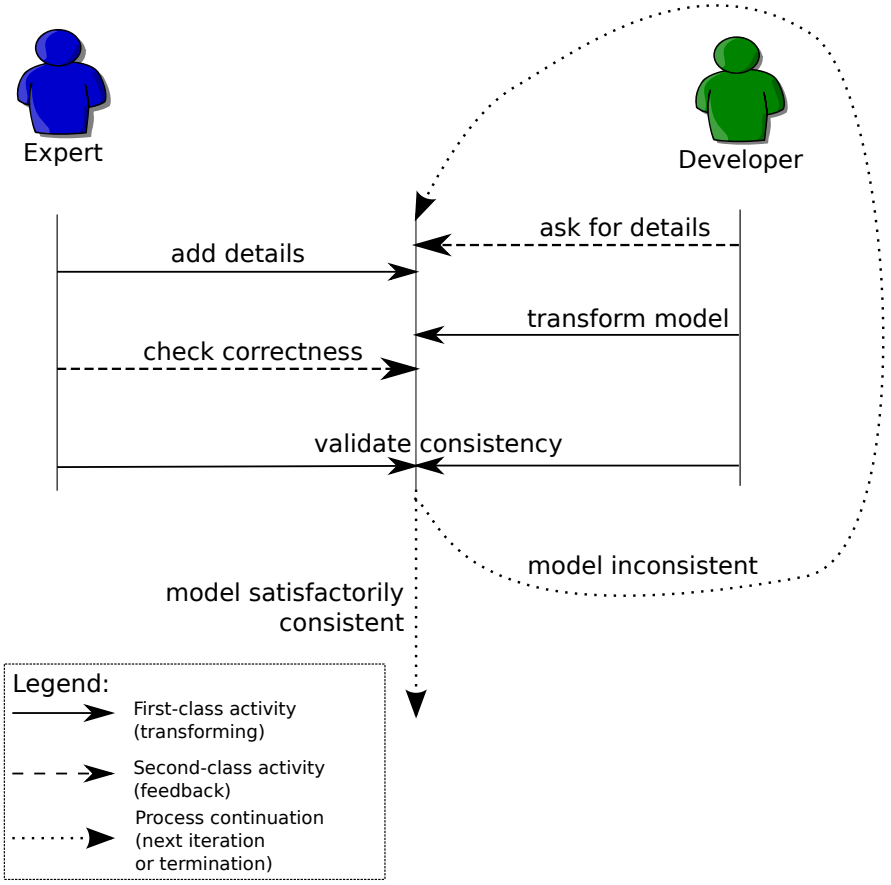


Figure 3.3: Iteration step. Subsequent tasks done by the parties aim at creating a consistent set of *entities*. The role of an expert is to provide required details of domain description and check developer work. The task of a developer is to transform model by specifying entity elements or transforming them. After several steps of model elaboration, collaborators should validate model consistency and decide on process continuation or termination.

3.2.3 Stop condition: Consistent model

After several iteration steps when participants are concerned about detailing domain description and organizing, they should validate the model. It is done by examining various perspectives: entities, generated code or even working application (Fig. 3.4).

First phase of consistency validation is *entities* review. Expert's role is to check if all entities elements are related to the domain. He should also assure that none of important concepts is omitted. Developer's task is to examine if the model is consistent. He ought to inspect correctness of *entities associations* and elements, especially focusing on *type* and *cardinality* accuracy.

If state of the *entity* set is satisfactory, further methods of validations may be applied. De-

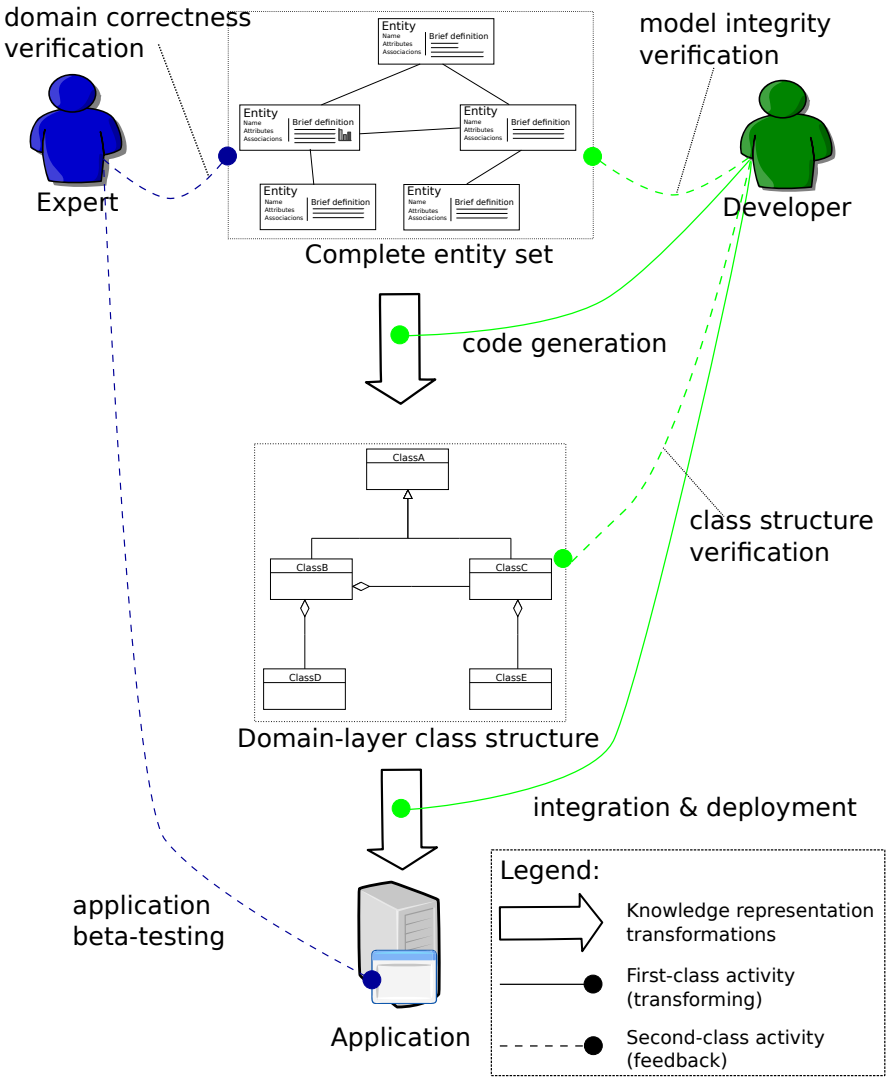


Figure 3.4: Model validation could be done using different perspectives. Firstly both parties should examine an entities set. If it is satisfactory, a developer generates code and check if it is convenient for development. Finally the model could be used in working application to facilitate providing an expert feedback. Code generation and possible deployment concentrate on a possibility of such model review that is convenient for both parties.

veloper generates domain model code. Then he checks if it is conforming with information aggregated in the entity set. If this test passes, code is ready to use in an application. A programmer can utilize it in the application development. Then working prototype may be shown to an expert. This is last stage of model validation as during beta tests conducted by an expert several model lacks might be revealed. If validation fails in any stage, collaborators should return to iterative process of model development (section 3.2.2).

3.3 Participants' tasks

The presented methodology is distinguished by precisely defined roles of both types of participants: experts and developers (for description of collaborators see section 3.1). Tasks performed by these two parties of modelling process are twofold: model development and feedback. Model development tasks are also known as first-class activities because they are a direct contribution to the model. Feedback (second-class activities) is a model review that enables opportunity to correct created model or ask for clarification. Interleaving and complementing one another, these two types of action constitutes process of iterative domain model composition.

3.3.1 Expert tasks

Defining

The most important task of an expert is knowledge transmitting by defining of domain concepts. The method of information passing should be convenient for him so as he can describe without unnecessary obstacles. An expert may use any knowledge representation that helps to understand domain depiction. Therefore it is desirable to use mathematical formulas, diagrams, figures or even movies to illustrate definition text.

Second aspect of this task is responding to developer's questions and clarifying ambiguities. By this activity an expert helps a programmer to understand the domain properly.

Correcting

The second-class activity of an expert is checking model elements prepared by a developer. This task is important because when a developer transforms information provided by an expert, he could misunderstand some part of it. As a result of his mistake, prepared model will not completely correspond with described domain. To avoid this situation, an expert should supervise model information extracting process and correct any error that occurred.

3.3.2 Developer tasks

Extracting and transforming

The main task of developer consists in transforming knowledge from raw form provided by an expert to consistent structure that enables code generation. This first class activity of the programmer is two-folded: he extracts entities from textual domain definition (*initial definition*) and transforms them and in this way he improve model state.

Extraction introduces structure to plain and linear domain description. Not only extraction from initial text this task encloses, but also inferring attributes and association from entity definition. This activity consists also in specifying entity elements details (e.g. cardinality or type). This actions transform expert knowledge into a domain model.

Transitions applied to *entities* increase consistency of the model and organizes it. They enclose *entity splits* and *merges* that are precisely described in section 2.2. This activity causes

that the model is easier to understand. It also enables possibility of generating correct code.

Consistency checking

The second-class developer activity is model validation of consistency. Whereas an expert task is checking if model conforms to a domain knowledge, a duty of a developer is reviewing model formal features.

3.4 Cooperation objectives

The model in its final state should be characterized by several features that decide about its quality. They are more guidelines than strict requirements but familiarity with them would facilitate model validation. The set of these rules consists in three elements: cohesion, completeness and consistency.

Cohesion refers mainly to a single *entity* state. A coherent entity relates to one and only one concept. Its definition and elements describes only one modelled being. At the same time entity that describes a concept is the only place in the model that contains this concept's definition. In other words cohesion can be expressed as: *one entity per a concept and one concept per an entity*.

Completeness is a feature of a whole model that describes a well defined and strictly limited part of a domain. All concepts referenced in definitions are represented as entities. Model completeness requires coherent entities that are connected with precisely described associations.

Consistency of model mean that parts of it neither contradict against each other nor are mutually exclusive. Although inconsistency may occur inside a scope of a single *entity*, it is more probable amongst several definitions.

Whereas cohesion and completeness are desirable, consistency is necessary condition of model correctness. Therefore participants concentrate on obtaining this feature even at the expense of remaining ones.

3.5 Features

This section briefly recapitulate these features of methodology that distinguish it among other approaches. Aspects described below are mentioned in various parts of this chapter, but the aim of this section is to emphasise their significance.

3.5.1 Semantics

The metamodel proposed in this document constitutes a rendezvous-point of domain knowledge and programming techniques. Definitions written in natural language specific for a domain are transformed into formal model consisting in entities and then into source code.

As a result of this process the code is enriched with domain information what causes that programming structures posses semantics that refers to real-word concepts. It facilitates later understanding of code because domain definitions provides documentation of code of a model. Thus a development team, that use such enriched code, is able to adopt ubiquitous language required by DDD process [10].

Furthermore structuring domain knowledge with formal defined entities makes this knowledge available for further automated processing. First example of such automation is code generation mentioned above. Another possibilities are for instance model evolution tracking, visualization or advanced search.

3.5.2 Iterative cooperation

Iterative characteristic of modelling process enables possibility of repeated attempts of domain comprehension. Therefore aspects omitted or misunderstood during initial steps might be clarified during later ones. This approach is especially convenient when participants are not able to work full-time on model development because of other activities.

3.5.3 Early development

Code generation is a part of model validation, thus it creates an opportunity to start using the model early during development. Through this feature, programmer is able to utilize expert's knowledge in initial application prototypes. As a result of that collaborators can precise requirements early basing on beta tests.

3.5.4 Is it agile?

The idea of proposed process has its origin in agile methodologies. User stories were an inspiration of entities method of creating their definitions. Although stories describes usage scenarios whereas entities defines part of a domain, the common feature is partiality and conciseness of both descriptions. Similarly to the stories entities' definitions are created using a language understandable by a customer (expert) and he *owns* them (cf. section 1.1.3). A customer (an expert) is actually part of development team. This fact improves requirements collecting and enables utilization of his domain knowledge.

Why agile approach adapts to this case? We describe cooperation that characterised by special features.

1. It involves small group of people (even as few as two of them: a developer and an expert).
2. The cooperation must be very close and efficient, because of non-trivial character of knowledge being transmitted.
3. As the modelled domain are often related to research projects, requirements may change during developments process.

Problems characterised by such features suits well to agile methodologies.

On the other hand, the one of Agile Principles [15] is:

> Working software is the primary measure of progress.

In this case working software is not an absolute aim of the cooperation. More important is to pass knowledge between participants that may lead to create software. In most cases working application is one of artifacts of the modelling process, but a model itself is a satisfactory result of the cooperation. Thus this solution is similar to Agile Modelling mentioned in the section 1.3.1.

To conclude, one would seem that proposed methodology could be classified as Agile. However it is essential to remember, that the final outcome of typical agile method is working software whereas the methodology described in this thesis concerns rather organising knowledge by creating semantic models. Thus it is better adapted to scientific applications.

3.6 Conclusions

The methodology presented in this paper is derived from existing methods of modelling. It uses the metamodel that bases on the UML and DDD metamodels. We also adapted DDD expert interaction and join it with agile practice of customer being a part of a team. However it introduces several innovations and improvements:

⇒ well defined developer and expert tasks,

⇒ well defined model transitions,

⇒ joining textual domain definition with formal one,

⇒ adaptation to small teams (especially one to one cooperation).

For these reasons the methodology can be easily adapted by a small team that is preparing to elaborate domain model.

Iterative conduct of the cooperation facilitate creation of cohesive, complete and consistent model. As a result of the collaboration, expert knowledge is structured and stored as a semantic model.

Chapter 4

Domain Model Builder: a Tool for Cooperative Domain Modelling

The Domain Model Builder – the tool, that implements the methodology proposed in the thesis is described in this chapter. It shows prototype application than enables model composition and presents programming methods that provide possibility of effective implementation.

Firstly we present requirements for a tool that supports presented methodology. Then a general architecture of the DMB is outlined, with special attention to Redmine with its plugin mechanism. Next we depict tool functionality proving, that it fulfill requirements of the methodology. Finally DMB implementation is described focusing on used algorithms and special mechanisms provided by the Ruby programming language.

4.1 Requirements

The goal of the Domain Model Builder is to support an iterative cooperation between an expert and a developer. Therefore it is intended to enable model building and facilitate participants' communication.

4.1.1 Functional

Metamodel implementation The most important function of the tool is possibility of model creation. Therefore it should implement the metamodel structure presented in this thesis (Chap-

ter 2) and provide functions to manipulate instances of its elements (CRUD[1] functions). The tool is also required to implement model transitions: split, merge and extract (see: section 2.2).

Methodology support The tool should provide functions to creation of model in a manner described in the methodology. Thus the tool should enable iterative model creation and validation. The tool also needs to enable text editing and media embedding.

Model visualisation A created model should be visualised in a diagram. The diagram should contain all important information contained in the model. It also need to be a gateway to all parts of the model: user should be able to access information concerning any part of model using its diagram.

Model evolution tracking The tool should not only support the methodology, but also enable its validation. The tool should record a history of the cooperation and present it to users. It needs to track changes and their authors. In consequence usefulness and convenience of the methodology could be validated.

4.1.2 Nonfunctional

Collaboration The tool should provide convenient method of collaboration. As it will be used by distributed teams, it should be available via the Internet and should be usable by many people at the same time.

Intuitiveness As domain expert often omit to be computer science professionals, the tool should be easy to use for such people. Although a user interface needs not to be self explanatory but still messages presented to its user should not contain technical terminology.

4.2 Architecture

Web application is a type of distributed application. Main part of computation is done in a server side. A thin client, that is a web browser mainly interprets data sent by a server. A synchronous communication of these kind of applications is conducted using the HTTP protocol (or, in secure version, HTTPS). A server response for a client request with a HTML page. This is a solution that is widely used in modern applications, especially these that requires cooperation between significant number of users.

The Domain Model Builder is a web application implemented as a Redmine plugin. Redmine is project management tool written in Ruby on Rails [40] – a framework based on the Model-View-Controller design pattern Fig 4.1. This section describes basis of Rails and Redmine with special attention to a method of integration the DMB with Redmine.

[1] Create, Read, Update, Delete

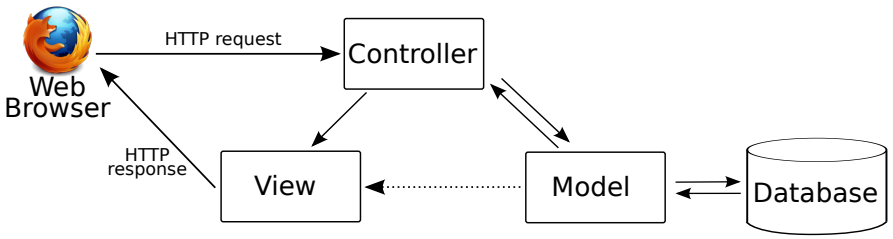


Figure 4.1: Ruby on Rails implements the Model-View-Controller design pattern. A web browser sends HTTP request that is processed by one of controllers. Models encapsulates data and communicates with database. Basing on models data and website template stored in a view, the controller generates a HTTP response that is sent back to the browser [41]

.

Redmine provides various functions facilitating cooperation. Issue tracker, Gantt chart and calendar facilitate task planning. Wiki system, news and forums enable cooperative project documenting. Integration with SCM repositories facilitates code management. Changes contributed by users are stored and presented in the activity log allowing to review cooperation history. Access control is based on user roles [32].

Redmine functionality can be extended or changed with plugins. According to the Redmine documentation [42][43] it may be done in several ways:

1. Overriding views – defining a view with the same name as original one overrides the latter.

2. Adding methods to existing classes – Ruby provides mechanism of adding methods to existing classes by redefining them or including modules to them (mixins).

3. Wrapping existing methods – methods in Rails application can be aliased and then wrapped by another method. That mechanism facilitated by a method `alias_method_chain` [44] enables extending existing Redmine methods (e.g. in a model or in a controller).

4. Using Rails callbacks – `ActiveRecord`, a standard ORM in Rails 2 defines several callbacks that can be registered and triggered during changes in model state [45]. Callbacks enable injecting methods into Redmine classes life cycle.

5. Registering Redmine hooks – Redmine also provides callback API called hooks. They enable Redmine extending in a elegant way, without changing its code [43]. List of hooks is available in the Redmine documentation [46].

6. Adding own classes or modules – the most straightforward way of extending Redmine. Created plugin can act similarly to regular Rails application, but use Redmine classes. A plugin integrates with it during plugin initialization using configuration options (as described in the Redmine plugin tutorial [47])

The Domain Model Builder may be described as a separate application embedded in Redmine and integrated with it. Thus the DMB is intended rather to use Redmine functions than to change them. Therefore it uses mechanisms described in points 2 and 6. It uses Redmine user management, wiki system and activity logger and it is presented using Redmine layout. However it is not typical Redmine plugin that just changes its functionality.

4.3 Functionality

The DMB is an implementation of concepts characterised in the preceding chapters. It enables creating domain model using the metamodel described in the Chapter 2 according to the methodology proposed it the Chapter 3. This section presents functions of the tool that supports these ideas.

4.3.1 Metamodel implementation

As an *entity* in the metamodel is a knowledge aggregation, an *entity* page (Fig. 4.2) contains references to all significant information related to presented entity. The page provide access to model elements defining functions. It also contains links to model transition actions.

The *entity* page is divided into two main parts. The right one is natural language description of modelled concept. The left one presents formal entity definition: its attributes and associations.

A concept encapsulated by an *entity* is defined using a wiki page. Intended to be used for collaborative document creating, wiki seems to be the best tool for definition creating. Easy media files (images, data files) embedding and edition history tracking are two most important amongst wiki features.

The Domain Builder directly implements concepts of *association* and *attribute* defined in the metamodel description. *Attribute* and *association* summary are shown in the *entity* page. Showing inherited elements and helper texts near *relations* facilitates understanding model semantics (Fig. 4.3).

The initial page is a welcome page of each project for the Domain Model Builder and starting point for the cooperation. Using this page an expert is supposed to prepare a general domain description (*initial definition*) that is afterwards used as a basis for *entities* creation. Thus the only two operation that can be performed on the initial page is a definition edit and *extract* (a simplified *split*).

The Domain Builder implements all main model transformations: *split*, *merge* and *extract*. The transition page (similar for all transformations) shown on the Fig. 4.4 is divided horizontally into two main parts. The first one contains *entities names* and definition and another one refers to *entity* elements: *associations* and *attributes*. The main difference between *split* and *merge* (Fig. 4.5) forms lays in the kind of presented data. Whereas the *merge* page shows elements of both merged *entities*, the *split* form has data of an *entity* being divided and a new one. The *extract* operation is a simplified *split*, but it contains only definition texts.

Conlcuding, the DMB implements CRUD operations one the model and model tranitions.

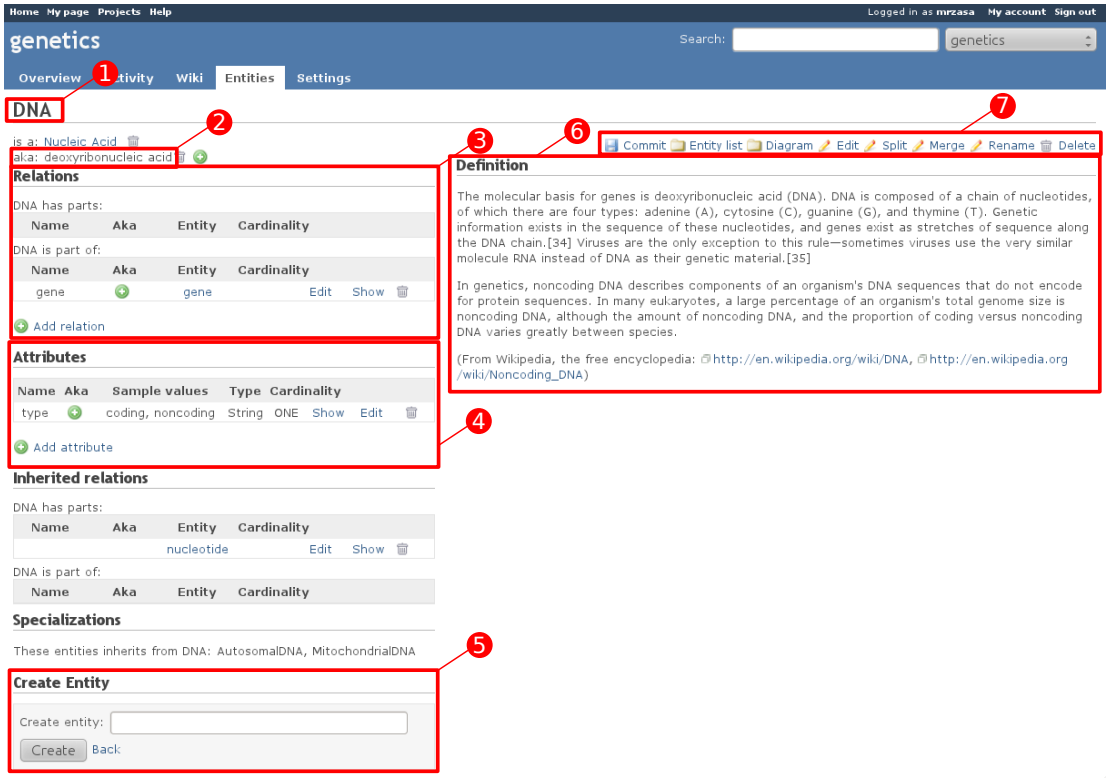


Figure 4.2: Sample *entity* page. 1 – *entity name*; 2 – *alternative names*; 3 – *relation* with its most important features; 4 – *attributes* (one's own and inherited); 5 – new *entity* form; 6 – definition; 7 – menu with main functions

Therefore it building semantic models in accordance to the idea presented in the Chapter 2. Then the DMB fulfil the first of functional requirements outlined in the section 4.1.1

4.3.2 Model visualization

During development of a model, its complexity grows. What is more during the cooperation some information might be duplicated and that reduces clarity of the model. A standard point of view at the model: a single *entity* with its dependencies seems to be insufficient for model validation. Therefore a model in its advanced stage can be difficult to understand without a holistic view.

A feature that responds to this need is a model diagram representation (Fig. 4.6). The diagram is based on UML class diagram and shows *entities*, its attributes and associations. The diagram is also a place that facilitates navigation, because each of a diagram element is enriched by a link to its definition. This feature corresponds to third of the funcional requirements (cf. section 4.1.1)

Figure 4.3: *Association* features. 1 – *generalization*; 2 – explaining text for a *relation*; 3 – inherited *relations*; 4 – specializations list.

4.3.3 Cooperation process logging

Although the purpose of the tool is creation of a domain model, the process of cooperation is also worth of exploration. It is not only for the reason that the knowledge transmitting process itself is absorbing; examination of it also enables a possibility of methodology validation. Recorded communication between participants and a model evolution may show if the procedure of cooperation is well suited and efficient.

To obtain this aim successive edition of model elements are archived and grouped in changesets. Every model change is stored with information about its author and date. A changeset groups consecutive changes and adds to them a comment and a user that created the changeset.

After metamodel transitions, changeset is created automatically. User is also able to create

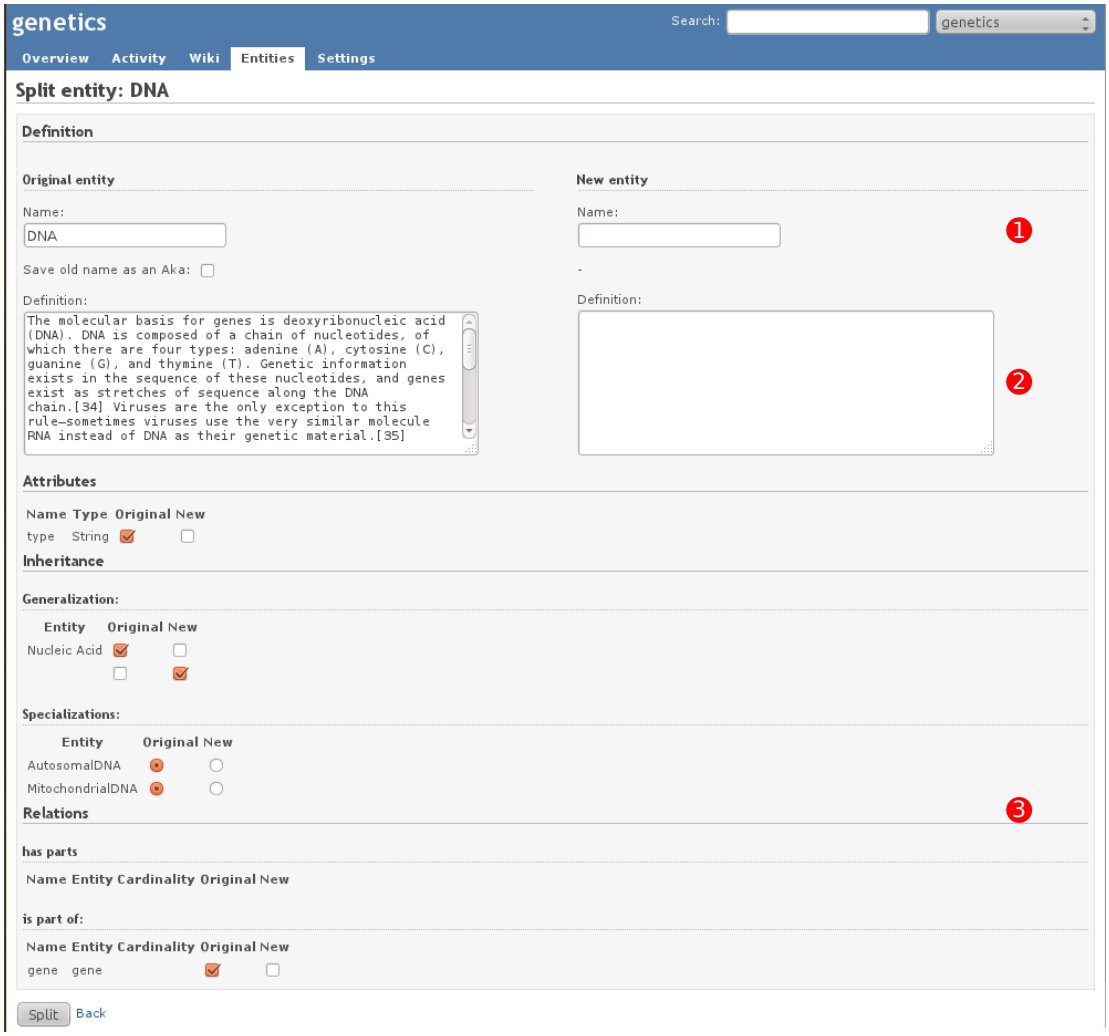


Figure 4.4: *Split* page. 1 – names; 2 – definitions; 3 – *associations* (*generalizations* and *relations*). The tool enables using all of elements of original *entity* during creation of new one.

a changeset manually to store and comment other changes in the model (e.g. definition edits or new element creation). Changeset are presented in the Redmine activity log (Fig. 4.7). User is also able to examine changeset details (Fig. 4.8). One may examine process of cooperation and commitment of every participant. It may also support error tracking process as users are able to check simple history of editions.

To sum up, the DMB tracks evolution of a model and because of that feature provides possibility of methodology validation. Therefore the tool fulfils the last functional requirement (cf. section 4.1.1).

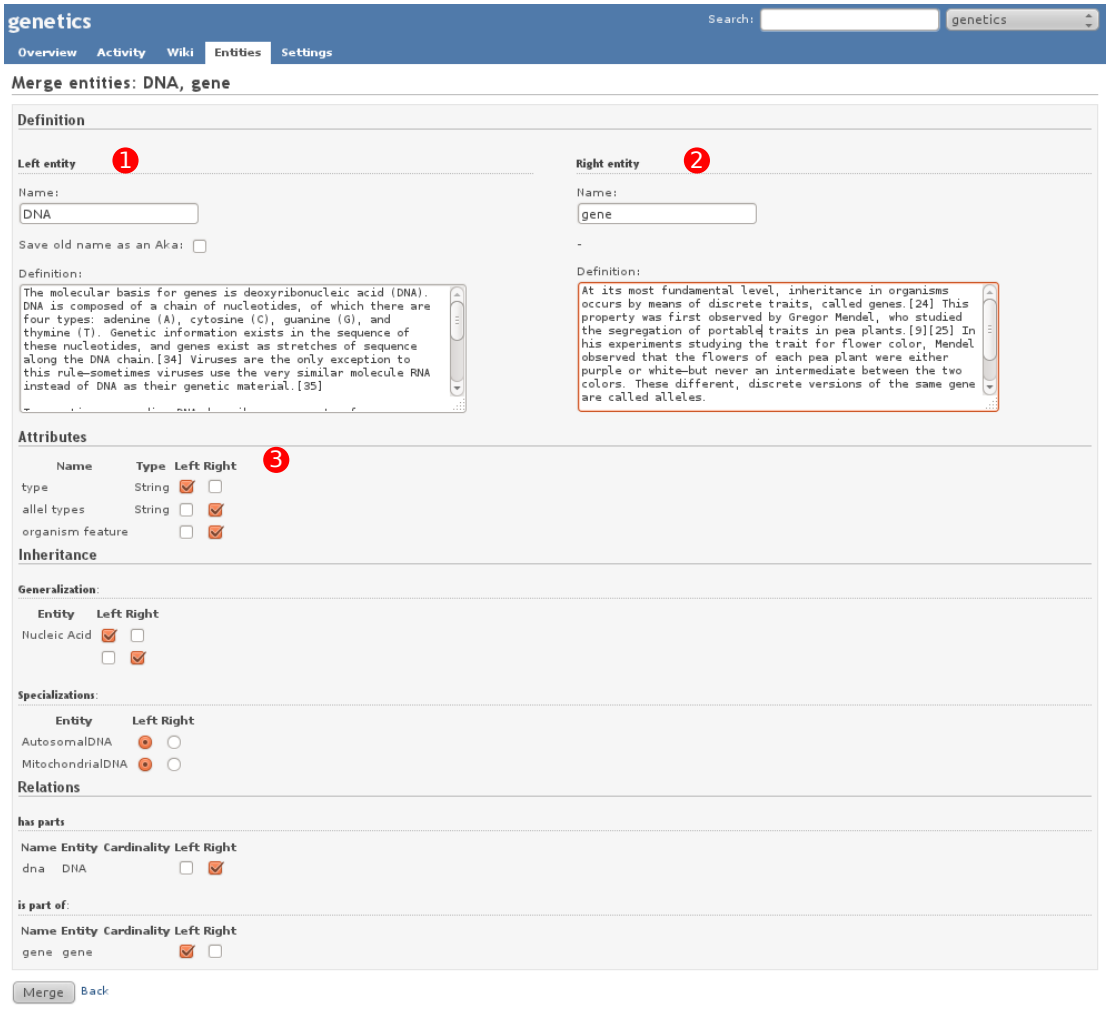


Figure 4.5: A *merge* page is very similar to a *split* form. The difference is that instead of original and new *entity*, there are two *entities* (1,2) to exchange data.

4.3.4 Methodology support

To prove that the tool enables metamodel building according to the methodology we discuss phases of the cooperation and the most important features of this concept, characterised in the sections 3.2 and 3.5.

Defining

The first and the most important feature of the DMB is domain concepts defining. Firstly participants are able to create general definition of the described discipline using an initial page. That is especially useful during first stage of the cooperation (initialization phase, see section). Text that is attached to each entity and rendered with it enables defining particular parts of the domain. As definitions (general ones in the initial page and partial in *entity* pages)

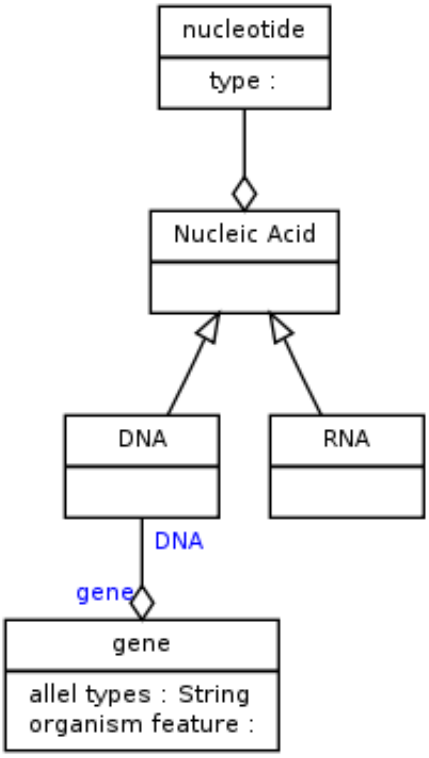


Figure 4.6: Model is visualised on a diagram that is based on UML class diagram. *Entities* are depicted as classes; *attributes* are listed inside an *entity*; *association* are described by *names* at both ends (blue text).

are implemented as wiki page, they enable enriching description with embedding images (e.g. figures, photos).

Textual definitions are not the only way of concepts defining. The DMB supports creating formal definitions using metamodel elements. The definitions written in the language natural for the expert may be easily transformed into formal ones structured with the metamodel. Thus the tool enables creating of the domain model using approach presented in the methodology.

Correcting and evolving

The DMB provides functions to model evolving and correcting. Firstly it implements model transitions (see section 2.2) that are crucial for convenient model development. Moreover as all of model elements are editable, the tool enables evolving and correcting each element after its creation.

Very important feature is wiki page attached to each entity (what was already mentioned). As wiki enables convenient way of text editing and stores history of its changes, it can be used not only for simple definition editing, but for discussing about it as well.

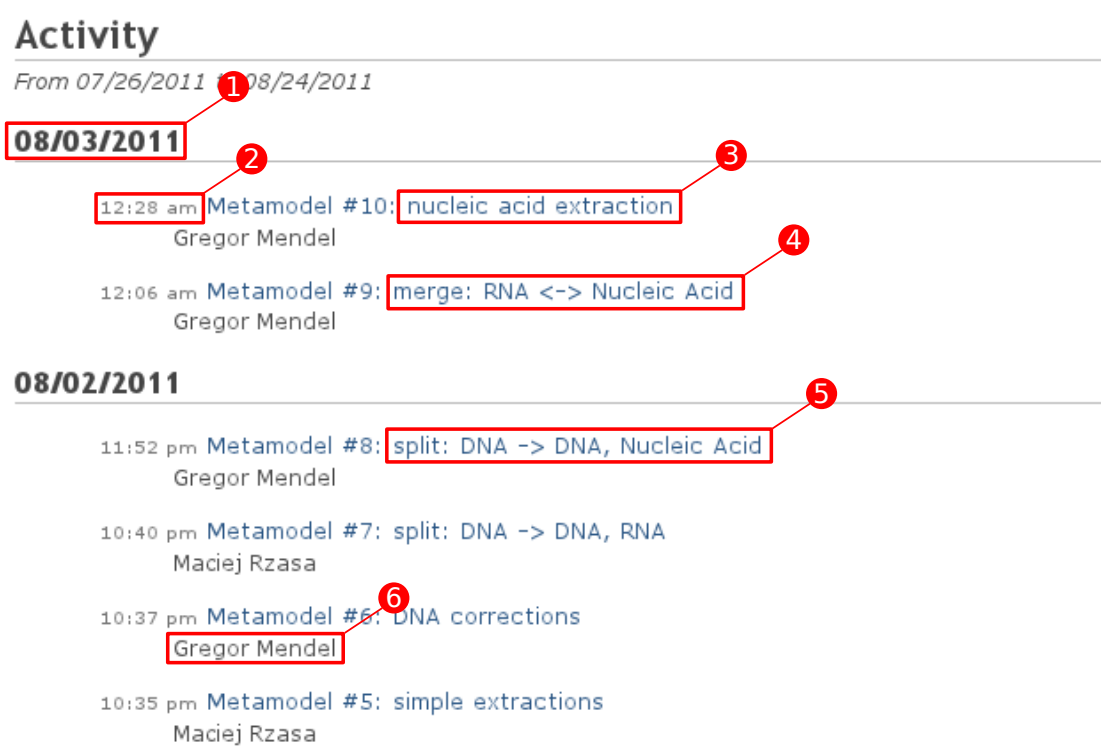


Figure 4.7: Activity log page. 1 – activity date; 2 – activity time; 3 – comment: normal changes; 4 – comment: merge of `RNA` and `Nucleic Acid` *entities*; 5 – comment: *split* `DNA` into `DNA` and `Nucleic Acid`; 6 – author.

Validation

As a text definition of an entity is displayed next to the formal one, participants are able to check correctness of the latter one in a convenient way. The diagram provides cooperation result overview and enables completeness checking of the model.

The metamodel implementation in the DMB is designed to preserve model consistency. To obtain this aim several validation and constraints were developed. *Name* of an *entity* must be unique in the *project* and a *name* of an *attribute* – in scope of one *entity*. Additionally an *entity* cannot be its own superentity and cannot be merged with itself. These solutions facilitate maintenance of the model and protect model validity.

Iterative cooperation

Participants are able to edit model elements in any stage of the cooperation. They can exchange knowledge using wiki pages and create new entities using model transitions. Model elements are easy to edit and correct thus modifications may be made any time.

Changesets groups adjacent editions of a model and definitions. Owing to this mechanism consecutive iteration of a cooperation process may be examine.

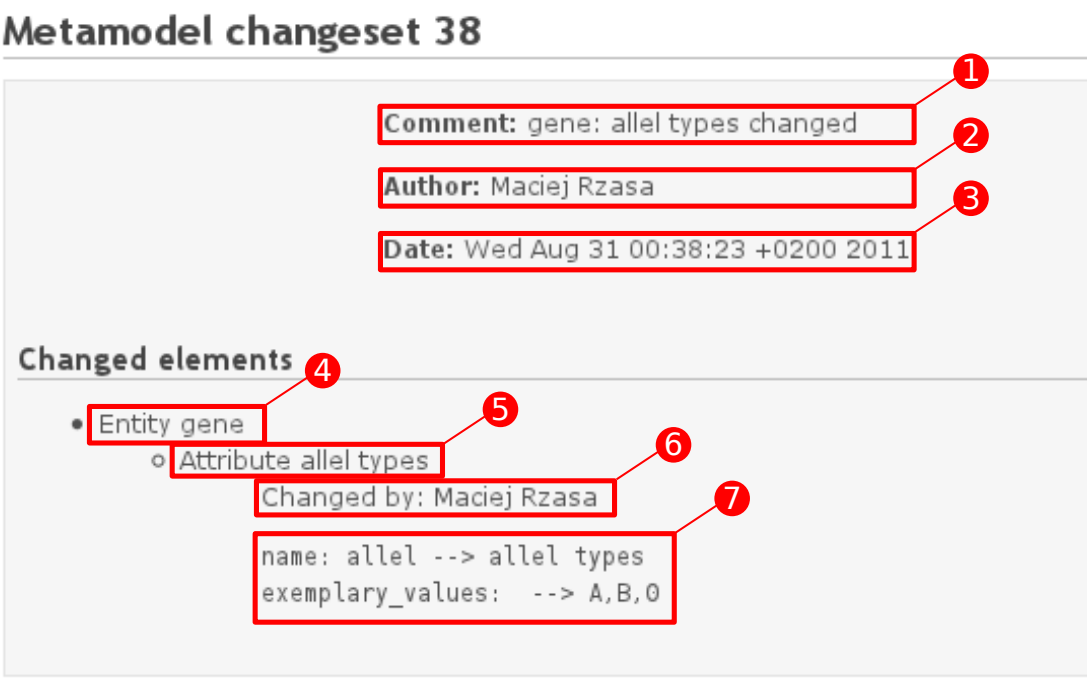


Figure 4.8: Changeset page. 1 – comment; 2 – changeset author; 3 – changeset creation date; 4 – root of changes tree: entity; 5 – changed element: attribute; 6 – date of attribute change; 7 – changes.

Semantics

Creating a set of connected definitions formalised with attributes and associations, participants build semantic domain model. The result of the cooperation is a formal model that is understandable for a machine. The model is ready for code generation. On the other hand formal model is enriched by textual definitions that document it. Therefore the model created using the DMB is a bridge between domain knowledge and formal software models and is an exact implementation of the semantics concept explained it the methodology description (section 3.5.1).

Conclusions

The tool is characterised with the same features that the methodology (cf. section 3.5). Therefore it supports creating a domain model according to this methodology. We could conclude that DMB responds to the second functional requirement (cf. section **??**).

Nonfunctional requirements

The tool responds also to the nonfuctional requirements described in the section 4.1.2. As it is a web application, it enables cooperation of multiple users that works simultaneously and remotely. Clear messages and tooltips enhances intuitiveness and facilitates using of the tool.

4.3.5 Summary

The DMB responds to all of requirements outlined at the beginning of this chapter (section **??**). It provides implements the metamodel and provides functions to create and evolve it conforming to the methodology. Moreover the created model is visualised using the diagram. The process of cooperation is tracked and thus the tool enables validation of usefulness of the methodology.

4.4 Implementation

4.4.1 Metamodel elements implementation details

Entity

An *entity* is the central point of the presented metamodel. Therefore the `Entity` class has associations with all important classes of the metamodel. Associations in the Rails models are always bidirectional. As classes `Attribute`, `Association` and `AlternativeName` are part of the DMB, creating references is easy. To obtain references from the `WikiPage` the original class must have been patched. A required method was added to class by encapsulating a function in the module that is included to the `WikiPage` class during the DMB initialization.

Attribute

An `Attribute` class is direct implementation of the Attribute concept presented in the Chapter 2.1.2. It enables defining name, alternative names, type and sample values of the attribute.

Association

Similarly to the presented metamodel `Association` has two subclasses: `Generalization` and `Aggregation` (the latter one is code name for the relation).

To store this generalization structure, Single Table Inheritance is used. In this object-relational mapping, described by Fowler in [48], all classes of the inheritance hierarchy are stored in the one table (in described case called `associations`) That table has columns corresponding to fields of all subclasses and a super class. To determine which class is stored in a particular row, additional column (`type`) with class name is recorded.

In case of the association hierarchy, only `Aggregation` class adds new attributes to the table (Fig. 4.10). To avoid repetition of code, data related to a relation endpoint is stored in the class `AggregationEndpoint` (Fig. 4.9).

Alternative names

Alternative name are implemented by a class `Aka` (*Also known as*), that is associated to each metamodel class. To represent the situation, when one association links different classes, Rails provides *polymorphic associations*. This mechanism is extension of normal associations: besides the id of the associated object, its type is stored. During fetching an associated object,

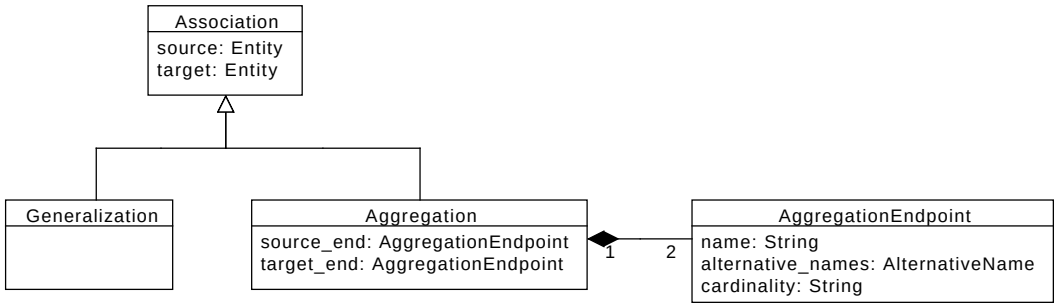


Figure 4.9: *Association* implementation: class diagram

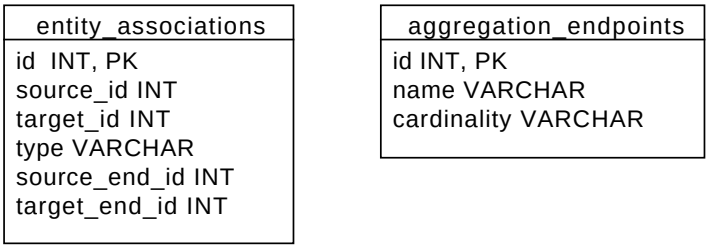


Figure 4.10: *Association* implementation: an entity relationship diagram of Single Table Inheritance mapping of the `Association` hierarchy. Because of Ruby on Rails persistence strategy, tables are not connected by foreign keys. Inter-table integration rules are maintained by the Rails framework, not by the database engine.

not only is its id used to specify the query but also its type. In the DMB the table `akas`, that stores instances of `Aka` class, has column `aliasable_id` and `aliasable_type`. The type column is used to determine a class (and a table) that is a target of the association. The id points to a particular instance of this class (and a row in its table) [49][41][50].

Model elements that can posses alternative names provides additional methods. To improve implementation and follow the DRY[2] principle, the declaration of association with `Aka` class and all related methods are encapsulated in `Aliasable` module.

Modules in Ruby serve two purposes: they are namespaces for functions classe and constants and they can encapsulate methods to use them in mixin mechanism. Module can be mixed into a class using `include` statement and as a result the class has access to the functions defines in the module as if they are defined in this class [51][52].

[2] *Don't repeat yourself*

Initial page

An initial page is implemented as a simplified `entity`. It has only a natural-language definition and no formal elements. The definition can be edited by each participant. The only transition that can be conducted is simplified `split` (concerning only the textual definition) that implements extract operation.

4.4.2 Model transitions

All operations are conducted using the same web form template and the same backend function. User interface is adapted depending on the transition type and data sent by a user are transformed to fit the interface of the generic transition method.

During `split` and `merge` operation several elements (`attributes` and `relations`) may be assigned to both `entities`. In this case an element is deeply cloned (with association). The original is saved with its `entity` and the copy is saved with an another `entity`.

4.4.3 Activity log

Redmine provides an activity log that presents changes made by users in the data stored in the system. To use the log in class provided in a plugin, one must fulfill three conditions:

⇒ use two plugins provided with Redmine inside the class (`acts_as_event`, `acts_as_activity_prov`

⇒ register event type in the plugin `init.rb` file;

⇒ add permission type corresponding to the event type: `:view_EVENT_TYPE`.

To enable model changes tracking, plugins `acts_as_audited` and `acts_as_paranoid` are used. The first one serves to record changes in model elements and their authors (provenance). Each change is stored with differences between old and new version, date of the change and contributing user. The second plugin has auxiliary role: it prevents model elements from being completely deleted, because in this case they are untraceable. This plugin causes that objects instead of being removed are marked as deleted but they are still stored in the database.

A class `MetamodelChangeset` is used to group stored changes and enable commenting them. Audits are connected to a particular changeset basing on the creation date: audit is associated to a first changeset that was created after it.

Changeset page shows changes in entities. To properly assign their elements, `entity` searching process depicted in Fig. 4.11 is conducted. Starting from an `audit`, an *entity* is being searched by subsequent *jumps* creating branches:

⇒ `Audit` to `Auditable` (it may be any model element),

⇒ `Aka` to `Aliasable` (any model element with alternative names),

⇒ `AggregtionEndpoint` to `Aggregation`,

⇒ `Association` to `Entity` (two branches to source one and to target one),

⇒ `Attribute` to `Entity`.

Afterwards branches are merged to create trees with an entity in the root, other model elements in branches and audits as leafs. This is achieved in a top-down approach. The first branch is an initial forest (with only one tree). When a next branch is being added to a forest, a tree with the same entity in a root is searched. If found, duplicating elements in the branch are removed and the rest of the branch is attached in a place of first difference.

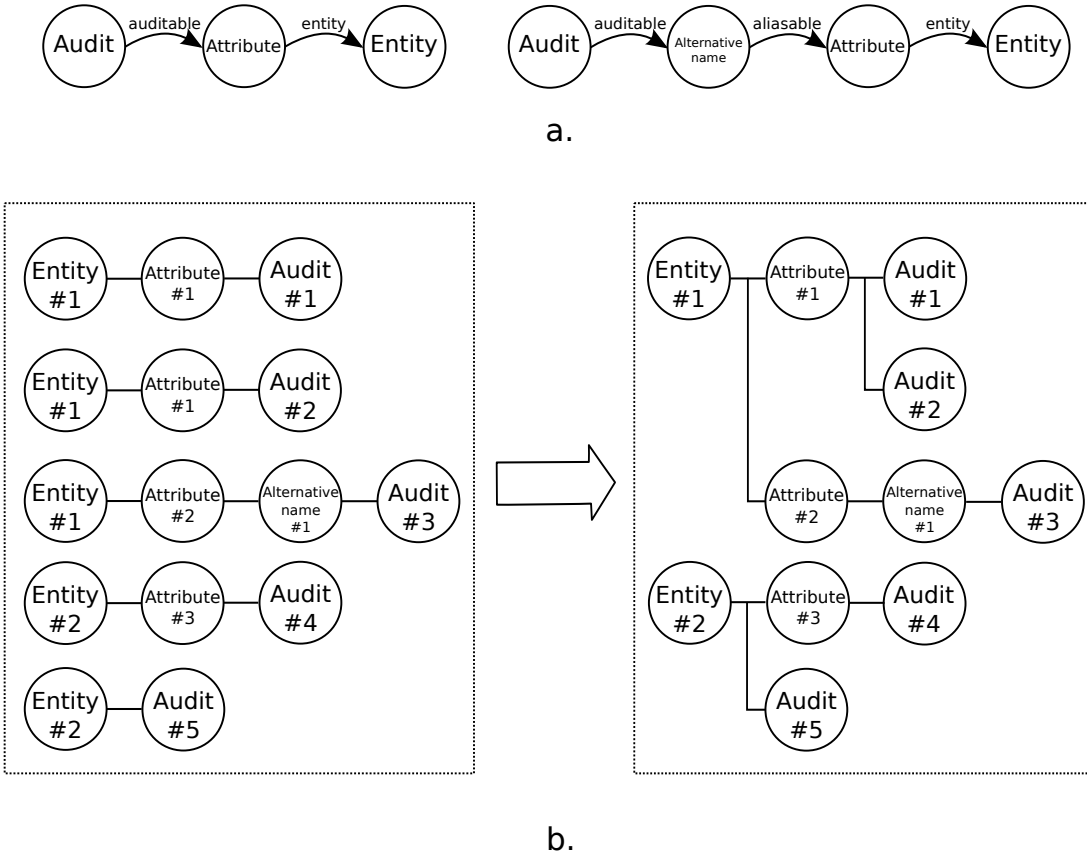


Figure 4.11: `Changeset` forest is created in two-step process basing on changes stored in `audit` objects. During first phase (a) every link to an `entity` is created for every `audit` in the `changeset`. An `attribute` is connected straight to its `entity`; `alternative name` is linked to its `aliasable` (`attribute`) and then to an `entity`. The aim of the second phase (b) is to merge redundant parts of branches and produce changeset forest.

4.4.4 Diagram generation

The domain diagram is generated using Graphviz. Graphviz is software for manipulating and viewing graphs. It enable creation a graph description using textual representation (e.g. *dot* language). Graphviz create diagrams with one of the choosen layout. It can produce various types of output, including image maps what is especially useful during presenting a graph on a website [53].

Dot output is generated basing on the model class structure. It is stored in a wiki page and processed by redmine plugin – `wiki_graphviz_plugin`. The diagram is generated each time user access a diagram page that is non optimal, but sufficient for prototyping purposes.

4.4.5 Used tools and mechanisms – summary

The DMB is implemented using Ruby (version 1.8.7) – an interpreted, dynamic programming language. This project take advantage of several features of this language.

The first one is mixin mechanism: adding methods to a class by including a module. This feature enables reusability of methods without changing inheritance hierarchy. The second one is changing existing classes without modifying their original source code. It is one of the methods of Redmine extending. This practice is informally called *monkey patching* or *duck punching* [3] [54][55].

The most important mechanism provided with Rails and used in the implementation are polymorphic associations, that link `Aka` and `Audit` with model elements. The feature that simplified development process was also database independence. Database engine was changed several times without additional effort.

The DMB uses several Rails or Redmine plugins. They are summarised in the Fig. 4.12.

4.5 Technological dependence

The methodology described in this thesis omit to rely upon any particular technology. However the second part of this work, the Domain Model Builder, is strongly dependent on the tools that are used during the implementation.

4.5.1 Ruby and Ruby on Rails

As the tool is implemented using Ruby on Rails, it is strongly dependent on this framework. Abandonment of using Ruby or Rails would cause necessity or redevelopment of the DMB. However, as several frameworks similar to the RoR exist (e.g. Django [56]), another implementation would be far simpler than the first one.

Both the language and the framework are mature, open source projects with vital communities. Thus it seems not very possible that they suddenly disappear. Still it may well be that the DMB will become difficult to deploy because of version the language or the framework. Nowadays there exists two maintained branches of Ruby: 1.8.x and 1.9.x, and the latter is not backward compatible. The latest stable Rails version is 3.1[4] but there are still applications using

[3] Happe in [54] cites a Patrick Ewing quote from RailsConf 2007:

> Well, I was just totally sold by Adam, the idea being that if it walks like a duck and talks like a duck, it's a duck, right? So if this duck is not giving you the noise that you want, you've got to just punch that duck until it returns what you expect.

[4] Rails 3.1 were released 30th Aug 2011 [40]

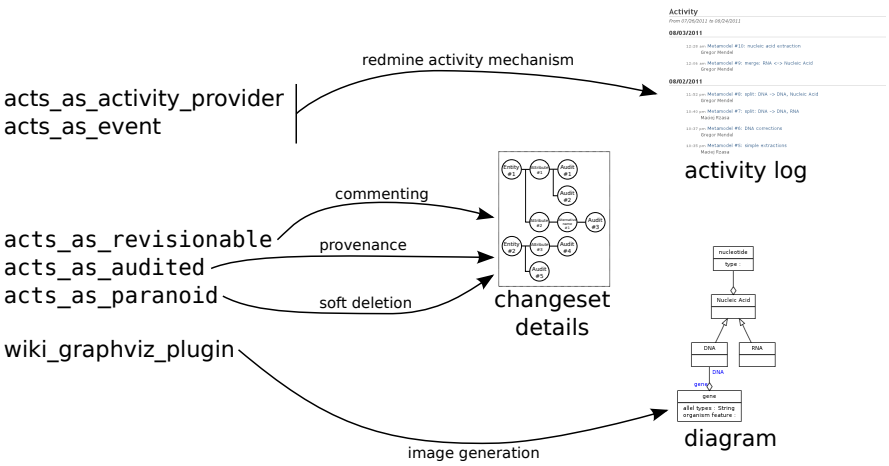


Figure 4.12: Redmine core plugins (`acts_as_activity_provider` and `acts_as_event`) are used to implement activity log. Details of changesets are provided using three plugins. `acts_as_revisionable` enables adding comments to stored changes. `acts_as_audited` stores information of what and by whom changes are made. `acts_as_paranoid` prevent permanent removal of model elements and thus enables tracking changes in deleted ones. `wiki_graphviz_plugin` interprets dot language stored at Redmine wiki pages and processes it to a model diagram using graphviz.

Rails 2.3. Used Redmine version is developed in Rails 2.3.5 and Ruby 1.8.6 and so does the DMB. With outdating these pieces of software, the tool will have to be adapted to new versions.

4.5.2 Redmine

The Redmine project is still evolving. The DMB evaluation deployment uses version 1.1.1 and the current stable is 1.2.1. During Redmine evolution, the DMB may become incompatible. It is especially required in case of significant changes in the activity log or wiki system.

In case of abandoning Redmine usage in the project, one must consider finding other vendor of several functions (or he may implement them on his own):

⇒ user management and access control,

⇒ multiple project management,

⇒ wiki system,

⇒ activity log,

⇒ diagram drawing (currently the diagram is drawn with a Redmine wiki plugin),

⇒ user interface: CSS layout.

From above list it can be seen, that the core of the DMB, the metamodel implementation omit to depend on functions provided by Redmine. In case of implementing the tool without Redmine,

the metamodel implementation is the only part that remains unchanged. The most complex Redmine element used by the DMB is wiki system. To replace it one may consider usage of other wiki implemented in Ruby on Rails, e.g. instiki [57] or olelo (gitwiki)[58].

4.5.3 Database

Ruby on Rails provides for a developer convenient database interface and for that reason Rails applications are slightly dependent on the database. Thus a database engine may be easily changed during development. Several nearly semless migrations from PostgreSQL to MySQL and back, that took place in the middle of application crafting, proved this claim. Therefore an application is not bound to any particular relational database management system.

4.5.4 Graphviz

Graphviz is used to draw a diagram. It is a stable open source tool and it can be assumed that it will be maintained. To remove Graphviz without changing the functionality of the DMB, one must implement diagram drawing, for instance using one of JavaScript graph libraries. The disadvantage of such solution is necessity of implementing diagram layout (that is implemented in Graphviz and might not be in JavaScript libraries).

4.5.5 Plugins

The last dependency of the DMB are used plugins. They are not always as stable as Redmine or Ruby on Rails whereas their role in the tool design is significant. Therefore abandoning any of them may require serious changes in the the tool implementation. However, as they provide well-defined functionality, they can be replaced by other plugins that have similar purpose. The replacement may require several changes in the remaining parts of the project, but is feasible.

`acts_as_activity_provider` and `acts_as_event` These plugins are provided with Redmine thus they are supposed to be as stable as Redmine. To replace them, one must implement module responsible for recording events.

`acts_as_audited` This plugin records changes in the model. To replace it one should use plugins that provides two kind of action: versioning (e.g. `acts_as_versioned`) and provenance (e.g. `paper_trail`).

`acts_as_paranoid` The only purpose of this plugin is soft-delete mechanism (records in a database are marked as deleted instead of being deleted). Thus to replace it one should find plugin with similar functionality (e.g. `acts_as_trashable`).

`wiki_graphviz_plugin` This plugin interprets a wiki page text that is written in dot language and execute Graphviz commands that produce diagram image displayed on the final

Table 4.1: Technology replacement possibilities

Technology	**Replacement probability**	**Replacement cost**	**Possible technologies**	**Required changes**
Ruby, Ruby on Rails	very low	very high	Django, Sinatra	redevelopment of the tool
Redmine	low	high	instiki, olelo	wiki system, diagram drawing, activity log, CSS layout
Database	medium	very low	PostgreSQL, MySQL, Oracle	deployment configuration
Graphviz	low	medium	JS libraries (e.g. JointJS [59])	diagram generation, diagram layout
Plugins	medium	medium	plugins with similar functionality	adaptations to another plugin

page. Replacement of this plugin would require preparing a function that calls Graphviz and produces proper image. Therefore abandoning usage of this plugin seems to be easy.

4.6 Conclusions

4.6.1 Requirements fulfillment

The metamodel presented in this thesis is fully implemented in the Domain Model Builder. The tool enables creation and transition instances of metamodel elements. It also supports the methodology that is described in this work. Participants are able to iteratively develop a model gradually transforming domain knowledge into a formal model. The result of the cooperation is visualised on a diagram and collaboration is recorded using activity log. As it is a web application, the tool supports collaboration in a distributed team.

It can be seen that the DMB implements all requirements that are stated in the beginning of this chapter. Part of the nonfunctional requirements concerning usability and intuitiveness is additionally evaluated and summarised in the next chapters, because it requires reference to users' opinions.

4.6.2 Summary

The Domain Model Builder is a prototype implementation of the methodology presented in this thesis (Chapter 3). The tool fully supports metamodel creation and transforming in an iterative way. Used technologies meet development requirements and omit to cause a risk of vendor lock-in. As the subject considered in the thesis is extensive, there are still several features that might improve and extend the tool.

Chapter 5

Validation of Metamodel and Methodology

This chapter presents conclusions based on applying the methodology to real-world problems. Firstly it introduces a usage scenario, next describes a course of modelling and its result and finally contains opinions of users. The aim of this chapter is to verify ideas described in previous chapters and possibly suggests changes in the methodology and the tool.

5.1 Experiment description

To evaluate the methodology and verify the tool two experimental modelling session took place. The first domain was flood forecasting, and the second one – road designing. In both of them domain experts were involved, whereas the author of this thesis acted as a developer. This section outlines the domains, characterises the experts and describes the circumstances of experiments. The results of the cooperations are described in the next section. The purpose of this one is to summarise the cooperation course.

5.1.1 Controlled experiment: flood forecasting

The first experiment was related to the UrbanFlood [1]: a forecasting system aiming to respond natural threats (in this case floods). In fact, this experiment was a modelling of an existing system, so the final model could be compared with a UrbanFlood software design. Thus it was an opportunity to validate the methodology not only by checking the conduct of the modelling, but also its outcome.

The expert involved in the experiment was one of developers working on the UrbanFlood. Although he was not a flood expert, he possessed detailed domain knowledge that he had ac-

quired during project elaboration. His computer science skills were substantial facilitation of modelling process.

The cooperation process started with a meeting that consisted in quick tutorial of the DMB (presented to the expert by the developer) and description of the domain. The meeting was summarised in an `initial definition` written in the DMB by a developer. That was an initialization phase of the cooperation.

Further iterative collaboration was conducted remotely. Participants was communicating using the DMB and email. The definitions stored in the tool was edited by the developer whereas the expert was checking results and sending his comments using email. After approximately one week the model was found complete and the cooperation was finished.

This experiment proved importance of personal contact between participants. After the initial meeting the outline of the domain was established and further cooperation was mainly transforming this description into a formal model. This experiment indicated a need of a tool for discussion about a model, since in this case participants use email for this purpose.

5.1.2 Full experiment: road design

The second experiment was related to a domain model of road design application. The application was supposed to aid the design of road surface layers with special attention payed to selection of building materials.

The expert was a civil engineer specialised in roads and motorways. Although he had basic computer science knowledge, it was insufficient to model the domain on his own.

The cooperation was conducted without personal contact between participants. They used the DMB, email and phone to communicate. The general description of a domain was prepared by an expert and send as via email and then stored in the DMB as the initial definition. Relying on this document, first entities were created by a developer.

The iterative improvement of the model was conducted using initial definition. The wiki page that contained that definition was used similarly to a forum: the developer was asking there for clarifications and an expert was answering using the same page. Iterations lasted three weeks, but there were significant intervals between them.

This experiment showed that the cooperation without a personal contact, though diffucult, is possible. It also indicated a need of communication tool embedded in the DMB, as wiki pages are intended to store definitions not discussions.

5.2 Results of modelling

This section describes outcome of the modelling session: a domain model and a cooperation course diagram. Firstly we explain meaning of these diagram, then, using diagrams, we present results of modelling. The experiments are summarised with experts' opinions about the methodology and the tool.

5.2.1 Result types

Domain model The model is represented with a diagram shown in the tool description (Fig. 4.6). As an simplified result of the cooperation process it enables final model verification. Thus may be used to evaluation of the methodology.

Modelling course The another type of modelling outcome is cooperation process record. Basing on the information from the DMB activity log (see: 4.3.3), the modelling course diagram is prepared. This infographics, that bases on the subway maps, shows the process of model evolution by presenting changes in entities (Fig. 5.1). Each metro line represent an entity. Stations represent model changes: ordinary ones – adding or editing elements to entities; junctions – model transitions.

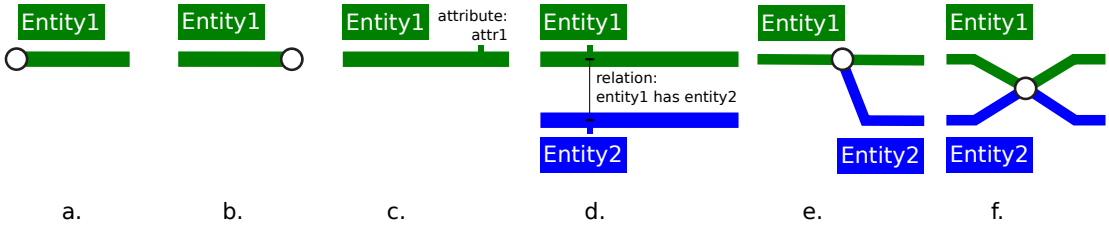


Figure 5.1: The modelling course diagram is based on subway maps. Metro lines represent entities and stations – model changes events.
a. entity `Entity1` created; b. entity `Entity1` deleted; c. attribute `attr1` added to `Entity1`; d. relation between `Entity1` and `Entity2` added; e. `Entity1` split operation creating `Entity2`; f. merge operation of `Entity1` and `Entity2`.

5.2.2 Flood forecasting

The UrbanFlood project relates to the problem of effective filtering and processing environmental data collected with sensors placed on dikes. The aim of the project is to forecast flood threat and minimise risk related to them. The modelled domain was briefly summarised by an initial definition elaborated during the experiment:

> Data from sensors are collected by Sensor Cabinet (tools allowing to connect to the sensor using e.g. GPRS) and sent to computer system (AnySense - sensor storage and JMS for further processing). Next sensor data are sent into filters which estimate a level of danger. Estimation is done by various simulations (e.g. Artificial Intelligence Anomaly detection, Reliable – Risk Calculation tool, Hydrograph or Flooding Simulator) . A simulation can be started automatically or manually by a user. Simulations have different importance level and depending on it have more or less resources (e.g. critical simulations during real danger should be calculates as fast as possible). The result of the simulations (and the state of the dam) are presented to the use on multi touch table.

Set of simulations and monitoring tools working together compose Early Warning System, which allows to monitor the environment threats. Basing on the EWS results crisis management centres takes decision while crisis occurs.

Domain model

The result of the cooperation process is a domain model visualised on the diagram (Fig. 5.2). To represent input data of simulations, the model contain entities `Dike` and `Sensor`. `Simulation` can posses different `importance levels` and it affect what `Resources` it could obtain.

The model contains also two helper entities: `ResourceAmount` and `SensorState`. They connect two entities with a relation of *many to many* semantics and they add data (a value) to this relation. They are equivalent of linking table in the relational model.

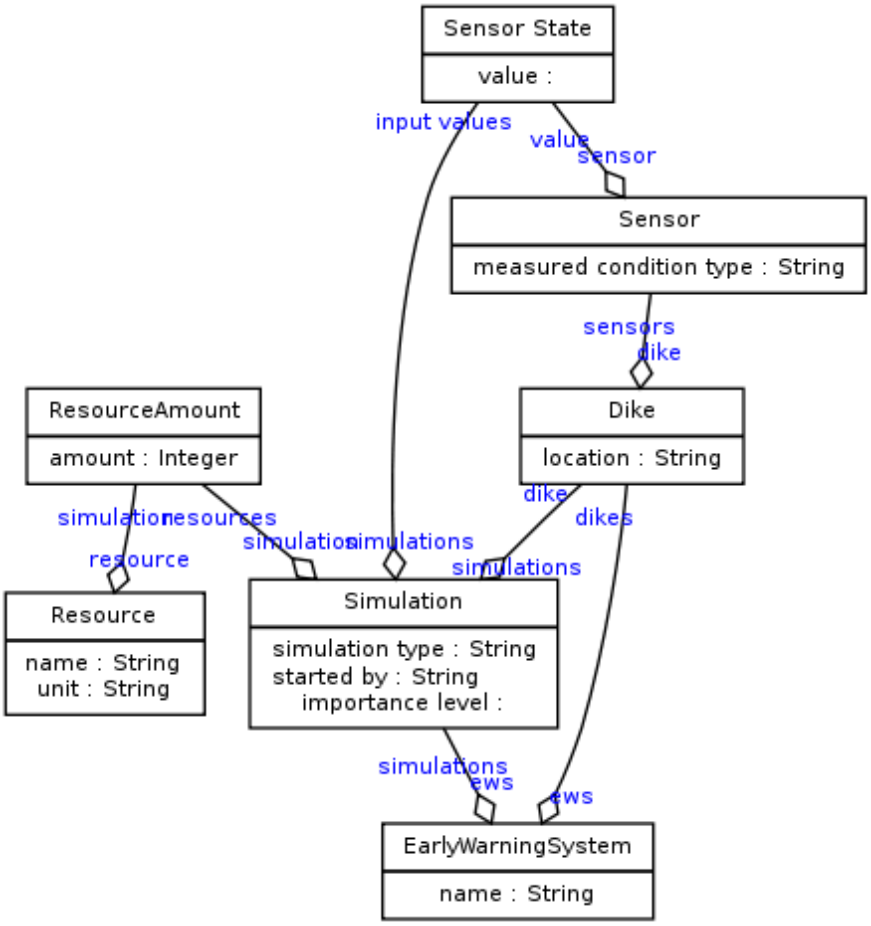


Figure 5.2: The model represents a system (Early Warning System) used to collect and process data concerning state of dikes. The data is an input to simulations of dikes behaviour and possible floods. Resources amount given to the simulation depends on its importance level.

Modelling course

The cooperation process is summarised in the Fig. 5.3. As described in the beginning of the chapter (section 5.2), the figure presents model evolution.

Initialization In the beginning of the cooperation a meeting of participants took place. The main part of it focuses on presenting the domain for developer. Opportunity of asking for clarification enabled for a developer well understanding of the domain. An outcome of the meeting was an initial definition of the domain saved in the DMB. Important role in the creation of this definitions played deliverables of the UrbanFlood project that deeply describe the domain. The initialization phase finished with validation of the initial definition by the expert.

Iterations Further cooperation consists in extracting and precising entities. The process might be divided into two branches developed in parallel: `Dike-Sensor` branch (domain entities) and `Simulation-Resource` branch (computational entities). Firstly three main entities was created (`Dike`, `Sensor`, `Simulation`) and then detail were being added to the model.

Validation The expert validated a model inspecting the diagram and initial definition checking their compliance to the domain. The developer verified model consistence. Although it failed to posses complete information about each element, it enabled understanding of a domain and would facilitate possible implementation.

5.2.3 Road design

The second experiment consisted in elaboration of a domain model for an application supporting road design with special care to planning material usage and optimising building cost.

From technical point of view the main part of road is a surface. In this case *surface* means a whole structure that is above the ground. The surface consists on several layers that are strictly composed. Each layer is built with a specific materials that have to fulfill requirements depending on road type. Properties of a road depends on traffic characteristic.

Domain model

The model diagram (Fig. 5.4) presents outcome of the cooperation process. As the language of the collaboration was Polish, all names on the diagram are writtnen in this language. Next paragraph provides translation.

1. `Droga` (`Road`) describes road characteristic, thus it has following attributes: `lenght` (`dlugosc`), `traffic type`(`kategoria ruchu`), `width of roadsides`(`szerokosc poboczy`), `width of traffic lanes`(`szerokosc pasa ruchu`), `width of pavements`(`szerokosc chodnikow`).
2. `Road` is connected with `Surface` (`Nawierzchnia`) that consists of several

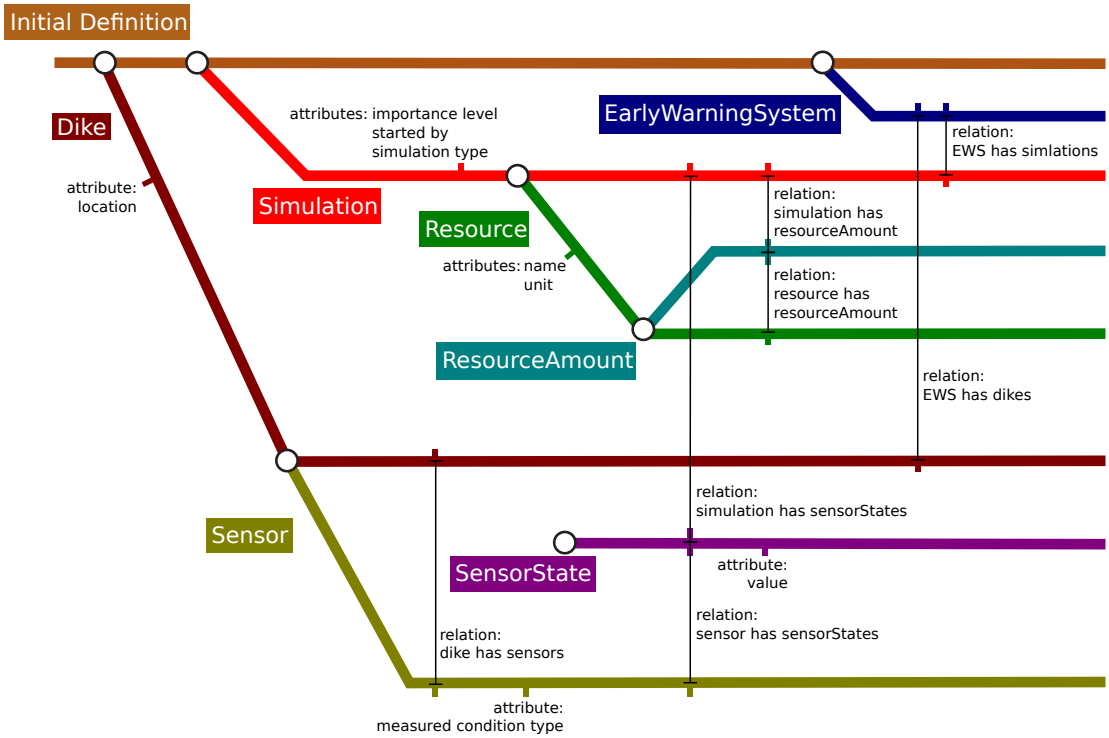


Figure 5.3: `Dike` as a was the first extracted entity. With `Sensor` it create a *domain branch* in the modelling process. The second branch (*computational*) consists in `Simulation` and `Resource`. `Early Warning System` was added in the end to encapsulate and link main elements. `SensorState` and `ResourceAmount` are helper entities that are attached to relations between other model elements.

3. `Layers` (`Warstwa`). Each layer has an `upper layer` (`warstwa wyzsza`) and `lower layer` (`warstwa nizsza`). A `Layer` is characterised with two attributes: `name` (`nazwa`) `thikness` (`grubosc`).

4. `Layers` are built with `Materials` (`Material`) characterised with attributes: `unit price` (`cena jednostkowa`), `transportation cost` (`koszt transportu`), `avalaible amount` (`dostepna ilosc`), `using cost` (`koszt wbudowania`), `type` (`typ`). As can be seen most of `Material` attributes describe the cost of its usage. Only the last one (`type`) describes its constructional characteristic.

5. Each material has a set of

   `Parameters` (`Parametr`) and linked with them
   `ParameterValues` (`WartoscParametru`) – a helper entity. A `Parameter` is characterised with two attributes: `name` (`nazwa`) and `unit` (`jednostka`).

6. The last group of entities is `Requirements` (`Wymagania`) hierarchy. There are two types of `Requirements`.

⇒ The first is `RangeRequirements` (`WymaganiaPrzedzialowe`), characterised with `minial value` (`min`) and `maximal value` (`max`) – it models situation, when a parameter value have to lay inside the range created with these attributes.

⇒ The second is `EnumerationRequirements` (`WymaganiaWyliczeniowe`) and it has one attribute: `permissible values` (`dopuszczalne wartosci`) – it corresponds to the situation, when a discrete parameter may have several acceptable values.

The created model corresponds with the domain and enalbes its comprehension. Therefore it is a good starting point for developing an CAD application supporting road design.

Modelling course

The cooperation process is shown in the Fig. 5.5. This section evaluates conformity of the cooperation with the methodology presented in this thesis.

Initialization The cooperation started with a document prepared by the expert. The document described basics of the domain and suggested application of the model being prepared (CAD tool). The text of the document were stored in the DMB and first entities with their attributes were extracted by the developer (`Material`, `Droga`, `Nawierzchnia`).

Iterations Iterations can be divided into three phases. The first was specifying entities extracted during initialisation, related to the road construction. The new one (`Warstwa`) was extraxted and merged with other ones. In the next phase a developer made an attempt to represent material types as `Material` subentities. This solution appeared to be wrong. Finally this problem was solved by adding attribute `typ` to the entity `Material`. The last part of cooperation was representing parameters and modelling requirements. As a result a flexible structure to requirement modelling was prepared.

Validation The model integrity was validated by the developer. The cooperation process was conducted mainly using the initial page. Thus the expert omitted to check particular entities.

5.3 Experts' opinions

After the experiments the experts were asked to give on opinion on the methodology and on the tool. The most important question they were inquired was if the methodology was a convenient way of knowledge transition and verification. They were also asked about readability of the user interface of the DMB. Finally they were supposed to suggest functions that might be added to the tool.

This section firstly refers experts' opinions and then summarises and comments them.

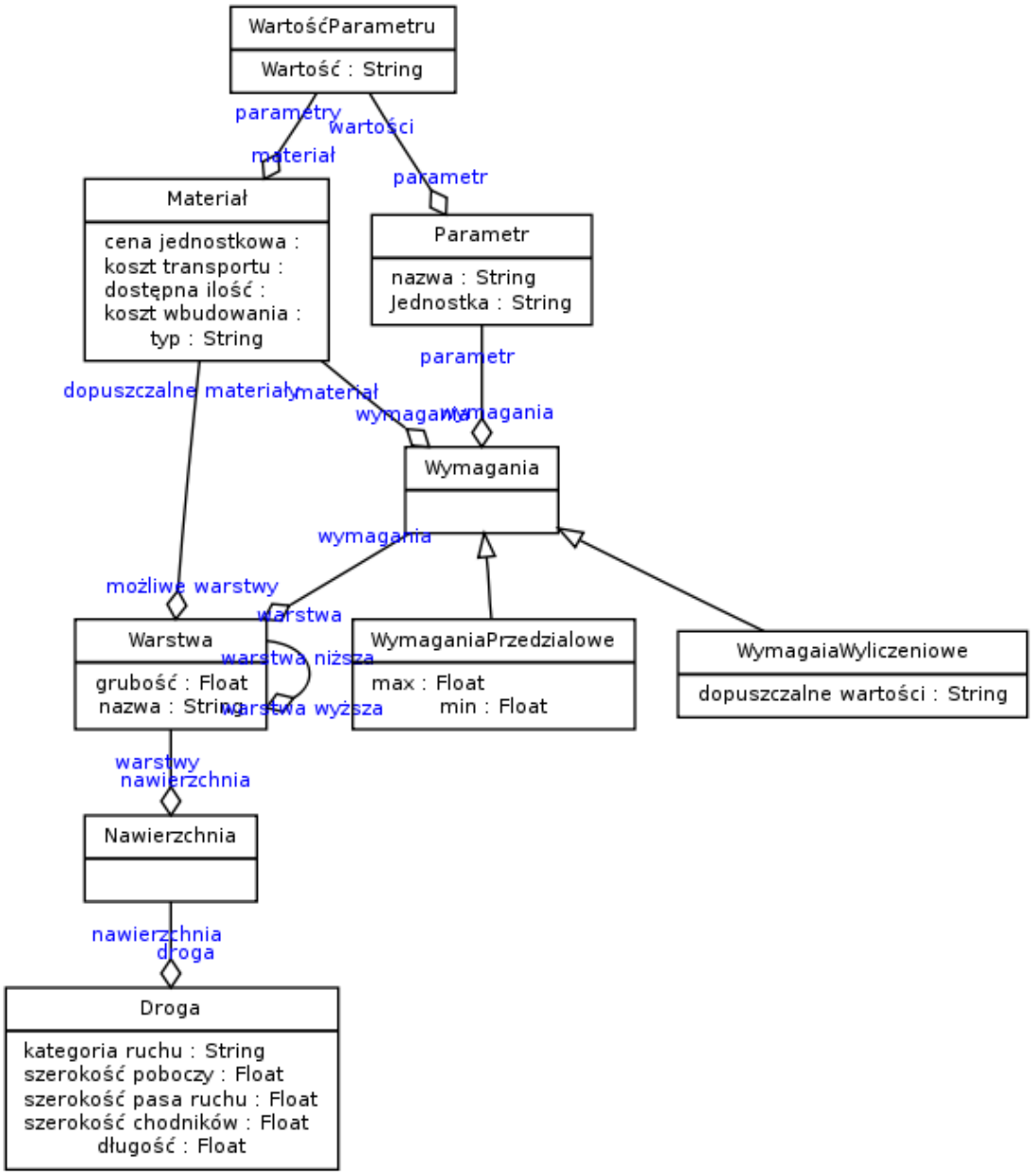


Figure 5.4: The model describes road design problem. A `Road` (`Droga`) consists in `Surface` (`Nawierzchnia`) that has `Layers` (`Warstwa`) with specific order (thus every layer has relation to its neighbours). Layers are built with `Materials` (`Material`) that has `Parameters` (`Parametr`) and have to fulfill requirements (`Wymagania`). `Requirements` depend on the `Layer` and have two types: `RangeRequirements` (`WymaganiaPrzedzialowe`) and `EnumerationRequirements` (`WymaganiaWyliczeniowe`).

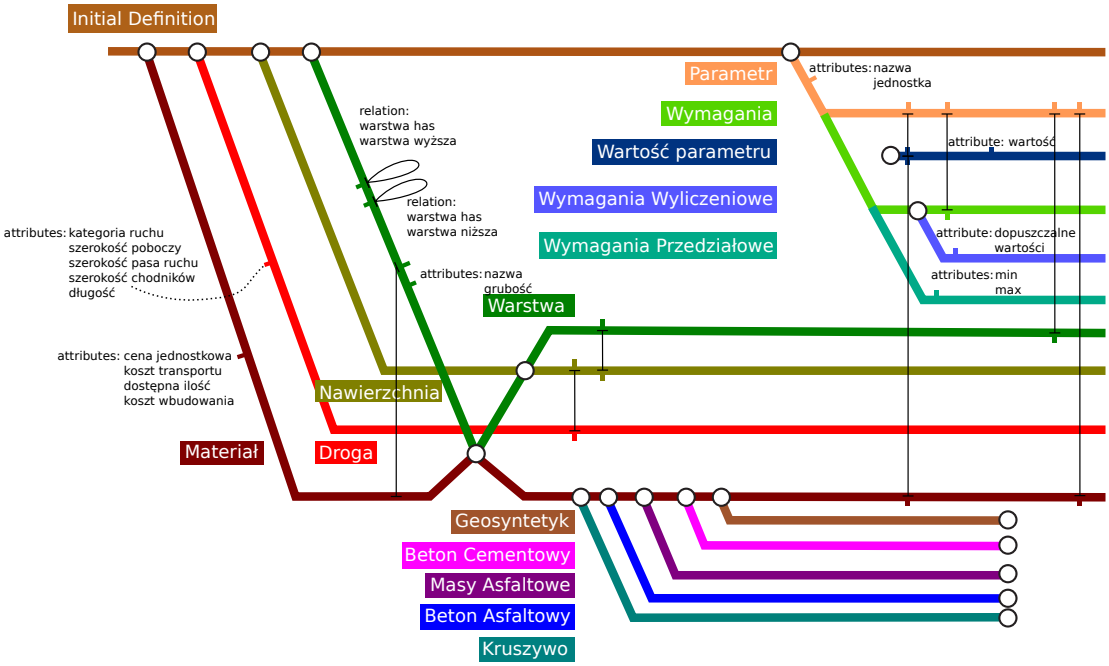


Figure 5.5: The modelling course may be divided into three parts. The first phase was extracting of main domain building blocks (`Material`,`Road`, `Surface` and `Layer`) and building their formal definitions. Second part was an unsuccessful attempt of representing different material types as subentities of a `Material` entity (`Geosyntetyk` – `Geosynthetic`; `Beton Cementowy` – `cement concrete`; `Masy asfaltowe` – `asphalt mastic`; `Beton Asfaltowy` – `asphalt concrete`; `Kruszywo` – `aggregate`). This solutions was finally abandoned and replaced with attribute `typ` (`type`) in the `material` entity. The last part of the modelling related to material parameters and their requirements.

5.3.1 Flood forecasting – software engineer

The first expert affirmed that the method provides an easy way of knowledge extraction. He also pointed several possible improvements of the tool. He observed that overlapping elements of the diagram decrease clarity of the model visualisation. He also suggested that the diagram might be difficult to understand for people without strong computer skills thus it would be reasonable to enhance the diagram with a legend. Readability of entities was assessed to be decent yet their presentation require improvements.

The suggestions of new functions concerned mainly usability of the tool. The expert claimed that navigation of the tool was troublesome. He also suggested that model elements might be linked to parts of definitions that relates to them. Another improvement could be editing elements of a model from the diagram.

5.3.2 Road designing – civil engineer

The second expert approved the method as convenient and well suited for cooperation involving engineers. He affirmed that the method enabled precise transmission and verification of knowledge. However, the expert pointed that it lacked feedback questions: in his opinion a expert should be able to ask if a developer understands a domain concepts correctly.

The expert stated that the diagram and entities interface is clear and understandable. He also pointed that at the end of the cooperation the initial definition had become chaotic and hard to understand because in this case it contained domain description, expert suggestions and a list of questions and answers. Therefore he proposed to split these types of information and display them in different pages.

5.3.3 Summary

The experts acknowledged the methodology as a well suited to the knowledge transition. The only improvement was concerning the cooperation method was verifying, if a developer understands a domain correctly. This suggestion may result from usability lacks mentioned by experts. They might have cause that the civil engineer has difficulties with model verification. Apart from the usability issues the experts indicted two important tool improvements: linking definition parts with model elements and separation definition from discussion.

5.4 Lessons learned

The experiments confirmed worth of the presented solution. The iterative methodology is well suited to the knowledge transmitting. The metamodel enables convenient and precise formalise a domain. Especially useful are the transitions. Facilitating model evolution they supports iterative aspect of the methodology.

The DMB proved its usefulness for domain modelling in accordance with the methodology. Its main functions: defining, model evolution and visualisation enables knowledge transmitting. Recording cooperation history made possible validation of the methodology. The experiments indicated several possible improvements in both (theoretical and practical) parts of this work.

The most important function missing in the metamodel is possibility of attaching entities to relations. It would facilitate creating *many to many* relations and improve model clarity.

The extract operation was intended to act similarly to split: to create new entity basing on the initial definitions. The experiments showed a need of second type of extract that is similar to merge and enable transfer parts of an initial definition to already created entities.

The phases of methodology failed to be strictly distinguished. After establishing initial definition of the domain iterative correcting was starting. Therefore extracting part of the initialization phase (section 4.3.4) may be considered as a first step of the iteration phase.

The use of the tool showed several possible enhancements. Minor ones were introduced in the initial phase of experiments. Other ones require considerable amount of work and are only outlined below. Firstly the usability of the tool should be improved. Navigation could more

smooth and presentation more intuitive for non computer science users. Thus pages should contain more explanation text, especially important is a legend for the diagram. Secondly the diagram itself requires elaboration, because (as can be seen on the Fig. 5.2 and Fig. 5.4) sometimes elements overlap what reduces legibility of the diagram. Moreover the tool misses a discussion function that would separate model description and comments. Finally possibility of linking definition parts to model elements is also worth considering.

The suggestions gathered in this section approve that experiments helped evaluation of the concepts presented in this thesis. These observations are important advice for adaptation of the methodology and further development of the tool.

Chapter 6

Conclusions

This chapters summarize domain modelling solutions proposed in this paper.
It contains conclusions and suggestions related to future work in this field.

6.1 Summary

The objectives introduced in the section 1.3 were developed in subsequent chapters of this thesis. In this section we present how this work relates to the objectives and fulfills requirements.

6.1.1 Methodology

Metamodel

The Chapter 2 presents the metamodel – the framework for building and evolving semantic domain models.

The elements of the metamodel 2.1 are suited to real-word concept modelling as well as to generating domain layer code. Therefore the metamodel is a bridge between a discipline of knowledge and programming techniques. As a result of that each both participants (expert as well as developer) are able to interpret a model using a method that is natural for each of them.

The metamodel transitions: *split*, *merge* and *extract* are a formal method of evolving model. This innovative approach enables transforming model to obtain its consistency.

Cooperation method

The Chapter 3 introduces a methodology itself: a method of collaboration that enables creation of semantic domain models.

The cooperation starts with defining a general discipline description (an initial definition) and extracting simple entities. Iterative model transitions and editions leads to transmitting expert knowledge to a developer and verifying his understanding of a domain. Finally the

model characterised with three features: consistency, cohesion and completeness (see section 3.4) is ready to use in a working application.

The main advantage of the methodology are precisely defined tasks of each participants. An expert defines concepts and checks correctness of model from a domain point of view whereas a developer extracts model elements and is concerned about its consistency (formal features).

Summary

To summarise this section we would say that the requirements concerning the methodology are sufficiently satisfied:

1. The cooperation enables transmitting a domain knowledge from an expert to a developer and verifying its comprehension.

2. A result of the cooperation is semantic domain model. It posses semantics from the human point of view (the formal model is enriched with definitions) as well as from the machine point of view (domain description is formalised and understandable for a machine).

3. As an expert's main task is writing textual definition that he is accustomed to use for description of the discipline, the method should be usable for him.

6.1.2 Tool

The Domain Model Builder, the tool that implements the methodology is described in the Chapter 4.

The DMB enables creating and evolving a model according to rules introduced in the Chapter 2. It provides model transitions that facilitates its management. To achieve easy validation, formal definition of an *entity* is displayed together with its textual description. Furthermore, to facilitate verification of its whole structure, a model is outlined using a diagram basing on the UML class diagram.

Not only is the Domain Model Builder an implementation of the methodology, but also provides possibility of its evaluation. The cooperation between participants is recorded thus one may verify difficulties that were encountered during the modelling process. It enables adapting the method basing on conducted modelling sessions.

Concluding: the tool fulfills requirements defined in the beginning of this thesis:

1. It is implementation of the methodology: it directly realises assumptions presented in the metamodel description (both elements and transitions) and it provides functions to build the model in iteratively and verify it.

2. Recording the history of the cooperation enable review of conduct of the cooperation and thus – verification of the methodology.

3. Experiments proved that the tool is usable for domain experts.

6.1.3 Evaluation

To verify reasonableness of presented solution two experimental modelling sessions were conducted. Cooperation with a software developer and a civil engineer proved usefulness of the methodology.

6.2 Future work

This section summarises all task that were not realised and may be executed in the future. They concerns both the methodology and the tool however the methodology appeared to be better elaborated, thus more improvements is proposed for the DMB.

6.2.1 Methodology

Representation of the entities attached to a relation This type of entities would facilitate modelling of complex *many to many* relations. An example of such a relation occured during road design modelling (see section 5.2.3): `WartoscParametru` (`ParametrValue`) was a helper entity attached to relation `Parametr – Material`.

Merge with initial definition As during modelling process information about created entity may be added to an initial definition, merge operation should be available to apply to an initial definition.

Experiments Two modelling sessions that were conducted proved usability of the method. Yet to develop it, more experiments with various type of experts is required to check if establish approach enables knowledge verification in various conditions.

6.2.2 Tool

Although the DMB implements the presented methodology and enables building domain models according to its assumptions, there are several functions by whom the tool may extended.

Code generation

To obtain seamless transition from modelling to implementation, automated code generating is required. Developer should be able to transform domain model into code stubs in an implementation technology (e.g. Java classes, ActiveRecord migrations and classes).

The prototype version of the DMB omit to provide such functionality, because the subject of the code generation is well known and implemented by numerous tools.

Model versioning

Present DMB implementation provides simple history of model evolution and presents differences between subsequent version. To facilitate cooperation it is reasonable to consider adding

versioning similar to one provided by source code management systems. Model would be reverted to a particular version, and owing to that participants could consider more possible solutions.

Implementation of this functionality may be an extension of current version of the DMB with a database as a model persistence backend. However there is another possible solution: model stores in a DSL and a version control system to store the changes. In this implementation finding elements might be difficult, but versioning is natural and relays on VCS functionality.

Knowledge tracking

Currently in the DMB two definitions of an entity: textual and formal, although displayed together, are hardly linked. Participants should be able to connects parts of a textual definition with model elements. Then a process of transforming domain knowledge into a formal model could be tracked and monitored. Moreover, linking parts of text with an entity elements would improve understanding of a formal model.

Model-focused discussion

The tool should enable a discussion about the model. It should be possible attaching discussion threads. This function would allow to consult specific problems concerning a model. It would also improve communication between participants.

Usability

As the DMB is a prototype, it requires several improvements in GUI layer and application navigation. The interface should contain more explanations. A diagram requires a legend to be understandable and pages related to model elements should clarify details of performed operations. Navigation through the DMB may be inconvenient: passing from one element to another ought to be smoother. To improve user experience and speedup the application introducing of AJAX (asynchronous communication) may be considered.

Bibliography

[1] UrbanFlood project homepage. http://urbanflood.eu. Accessed September 10, 2011.

[2] Jean Bézivin. On the Unification Power of Models. In *Software and System Modeling*.

[3] Thomas Kühne. What is a Model? In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[4] What is metamodeling, and what is it good for? http://infogrid.org/wiki/Reference/WhatIsMetaModeling. Accessed September 5, 2011.

[5] B. Hailpern and P. Tarr. Model-Driven Development: the good, the bad, and the ugly. *IBM Syst. J.*, 45:451–461, July 2006.

[6] John D. Poole. Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[7] CORBA home page. http://www.corba.org/. Accessed September 12, 2011.

[8] What is Domain-Driven Design? http://domaindrivendesign.org/resources/what_is_ddd. Accessed June 7, 2011.

[9] Karsten Klein. Domain Driven Design and Model Driven Software Development, 2007.

[10] Floyd Marinescu and Abel Avram. *Domain-Driven Design Quickly*. Lulu.com, 2007.

[11] Glossary of Domain-Driven Design Terms. http://domaindrivendesign.org/resources/ddd_terms. Accessed June 7, 2011.

[12] Alistair Cockburn and Jim Highsmith. Agile Software Development: The People Factor. *IEEE Computer*, 34(11):131–133, 2001.

[13] Dan Turk, Robert France, and Bernhard Rumpe. Limitations of agile software processes. In *In Proceedings of the Third International Conference on Extreme Programming and*

Flexible Processes in Software Engineering (XP2002), pages 43–46. Springer-Verlag, 2002.

[14] Kent Beck et al. Agile manifesto. `http://agilemanifesto.org/`. Accessed June 7, 2011.

[15] Kent Beck et al. Principles behind the Agile Manifesto. `http://agilemanifesto.org/principles.html`. Accessed June 7, 2011.

[16] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods. Technical report, VTT Publications, 2002.

[17] Karin K. Breitman and Julio Cesar Sampaio do Prado Leite. Managing User Stories. In Armin Eberlein and Julio Cesar Sampaio do Prado Leite, editors, *Proceedings of the International Workshop on Time Constrained Requirements Engineering*, Essen, Germany, September 2002.

[18] Rachel Davies. The Power of Stories, 2001.

[19] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[20] James Noble and Robert Biddle. Postmodern Prospects for Conceptual Modelling. In Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki, editors, *Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006)*, volume 53 of *CRPIT*, pages 11–20, Hobart, Australia, 2006. ACS.

[21] OMG. Unified Modeling Language, Infrastructure and Superstructure (Version 2.2, OMG Final Adopted Specification), 2009.

[22] UMLet home page. `http://www.umlet.com/`. Accessed September 5, 2011.

[23] Visual Paradigm home page. `http://www.visual-paradigm.com/`. Accessed June 7, 2011.

[24] IBM Rational Rose home page. `http://www.ibm.com/software/awdtools/developer/rose/`. Accessed June 7, 2011.

[25] OMG UML Vendor Directory. `http://uml-directory.omg.org/`. Accessed June 7, 2011.

[26] Daniel Mackay, James Noble, and Robert Biddle. A Lightweight Web-Based Case Tool for UML Class Diagrams. In Robert Biddle and Bruce Thomas, editors, *Fourth Australasian User Interface Conference (AUIC2003)*, volume 18 of *CRPIT*, pages 95–98, Adelaide, Australia, 2003. ACS.

[27] zOOml home page. `http://www.zooml.com/`. Accessed September 5, 2011.

[28] WebSequenceDiagrams home page. `http://www.websequencediagrams.com/`. Accessed September 5, 2011.

[29] BeoModeler home page. `http://www.beotic.org/us/projects/beomodeler/index.php`. Accessed September 5, 2011.

[30] Bo Arne Leuf and Ward Cunningham. What is wiki? `http://www.wiki.org/wiki.cgi?WhatIsWiki`. Accessed September 5, 2011.

[31] Manual:MediaWiki feature list. `http://www.mediawiki.org/wiki/Manual:MediaWiki_feature_list`. Accessed September 5, 2011.

[32] Redmine web page. `http://www.redmine.org/`. Accessed June 7, 2011.

[33] JIRA home page. `http://www.atlassian.com/software/jira/`. Accessed June 7, 2011.

[34] Michael J Rees. A Feasible User Story Tool for Agile Software Development? In *Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, APSEC '02, pages 22–, Washington, DC, USA, 2002. IEEE Computer Society.

[35] User stories: All products. `http://www.userstories.com/products`. Accessed September 9, 2011.

[36] XPlanner home page. `http://www.xplanner.org/`. Accessed September 9, 2011.

[37] Redmine Backlogs home page. `http://www.redminebacklogs.net/`. Accessed September 9, 2011.

[38] James W. Moore. The Logic of Definition, 2009.

[39] Patryk Burek. Adoption of the Classical Theory of Definition to Ontology Modeling. In Christoph Bussler and Dieter Fensel, editors, *AIMSA*, volume 3192 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2004.

[40] Ruby on Rails homepage. `http://rubyonrails.org/`. Accessed September 9, 2011.

[41] Sam Ruby, Dave Thomas, and David Hansson. *Agile Web Development with Rails, Third Edition*. Pragmatic Bookshelf, 3rd edition, 2009.

[42] Redmine: Plugin internals. `http://www.redmine.org/projects/redmine/wiki/Plugin_Internals`. Accessed August 24, 2011.

[43] Redmine plugin hooks. `http://www.redmine.org/projects/redmine/wiki/Hooks`. Accessed August 24, 2011.

[44] Ruby on Rails API: alias_method_chain. `http://apidock.com/rails/ActiveSupport/CoreExtensions/Module/alias_method_chain`. Accessed August 24, 2011.

[45] Ruby on Rails API: Callbacks. `http://apidock.com/rails/ActiveRecord/Callbacks`. Accessed August 24, 2011.

[46] Redmine plugin hooks list. `http://www.redmine.org/projects/redmine/wiki/Hook_List`. Accessed August 24, 2011.

[47] Redmine: Plugin tutorial. `http://www.redmine.org/projects/redmine/wiki/Plugin_Tutorial`. Accessed August 24, 2011.

[48] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[49] Chad Fowler. *Rails Recipes*. Pragmatic Bookshelf, 2006.

[50] Ruby on Rails API: Associations. `http://apidock.com/rails/ActiveRecord/Associations/ClassMethods`. Accessed August 27, 2011.

[51] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, 2nd edition, October 2004.

[52] Alexandre Bergel and Stéphane Ducasse. Analyzing module diversity. *Journal of Universal Computer Science*, 11:2005, 2005.

[53] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[54] Andreas Happe. Agile Provenance. Master's thesis, Technische Universität Wien, A-1040 Wien Karlsplatz 13, 2010.

[55] Plone Glossary: Monkey path. `http://plone.org/documentation/glossary/monkeypatch`. Accessed August 30, 2011.

[56] Django home page. `https://www.djangoproject.com/`. Accessed September 12, 2011.

[57] Instiki homepage. `http://www.instiki.org/show/HomePage`. Accessed September 4, 2011.

[58] Olelo homepage. `http://www.gitwiki.org/`. Accessed September 4, 2011.

[59] CORBA home page. `http://www.jointjs.com/`. Accessed September 12, 2011.