Universiteit Antwerpen

Domain-Specific Modeling and Model Transformation to support Collaborative Work on Android

Proefschrift voorgelegd op 15 mei 2012 tot het behalen van de graad van Master in de Wetenschappen, bij de faculteit Wetenschappen, aan de Universiteit Antwerpen.

Promotoren:
Prof. dr. Hans Vangheluwe
Prof. dr. Juan de Lara

Philip De Smedt

Research Group Antwerp
Systems and Software Modelling

Contents

List of Figures

List of Tables

Preface

Preface here...

CHAPTER 1

Introduction

This work has been carried out in cooperation with two academic institutions, University of Antwerp (UA) and Universidad Autónoma de Madrid (UAM).

1.1 Problem Description

The context of this Master's Thesis is Domain Specific Modeling applied to Android development for the generation of collaborative applications. Domain-Specific Modeling (DSM) and Model Transformation (MT) have the potential to support rapid development and synthesis of new applications on Android. This thesis investigates the power and limitations of DSM and MT in the context of Android. The metaDepth meta-modeling framework and tool will be used to create a domain specific language (DSL) that allows the creation of collaborative Android applications.

This work arises from the rising need for mobile applications that solve recurring collaborative problems. The thesis tries to tackle those recurring problems by applying DSM on the Android stack to support collaborative work.

Leek wil ook Android apps maken. Specifiek voor zijn noden. Makkelijker met een DSL. Geen Java/Android SDK kennis vereist.

1.2 Background

In general, traditional software development follows a process that aims at guaranteeing a certain standard. Important stages in this process are *specification*, *implementation* and *verification*. Software testing is an important aspect of this latter

activity. Although the analysis of different software development processes is out of the scope of this thesis, it is an important contribution to the quality of a solution that solves a problem through DSM. In the following, these three stages are described as a natural evolution to DSM. Also *code generation* is discussed as a possible solution to recurring software engineering problems.

1.2.1 Specification

Specification identifies the problems a new software system is suppose to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance [1]. Based on the requirements specification, software engineers can build an architectural design. The final goal of any engineering activity is to create some kind of documentation. If the design truly represents a complete system design, a team of software engineers can proceed to build iterations of the product, dependent on the software lifecycle that is applied. (http://c2.com/cgi/wiki?WhatIsSoftwareDesign)

1.2.2 Implementation

After creating a requirements specification and any accompanying design documents, a software engineering team can proceed to the implementation stage. An implementation usually is a realization of the design documents created in an earlier step. The result of the implementation stage is a software component or system that conforms as close as possible to those design documents.

1.2.3 Verification

Verification is the process of determining that a system, or a module, meets its specification. The verification process always relies on the requirements specification, as this document defines the intended behavior of the system. We could also ask ourselves *"Are we building the system right?"*. Verification is usually performed by (automated) testing of the system. The testing process can involve unit testing (testing a component in isolation), integration testing (components tested as a group), regression testing (uncover new errors in existing functionality after changes have been made to a system) and other ways of verifying that the system behaves as we expect it to.

A complete software development lifecycle is depicted in figure 1.1. One of the goals of this thesis is to minimize or even eliminate the implementation and testing cycles. Using a DSL, an end-user can easily create and customize their own applications by specifying its functionality in a clear and non-verbose way. The requirements and design lifecycle remain the same, as we use these documents to generate code. The activity of generating code is explained in the next subsection.

1.2.4 Code Generation

When developing software, many bugs and errors occur due to coding mistakes in the implementation phase. This can happen due to reusing old software components,

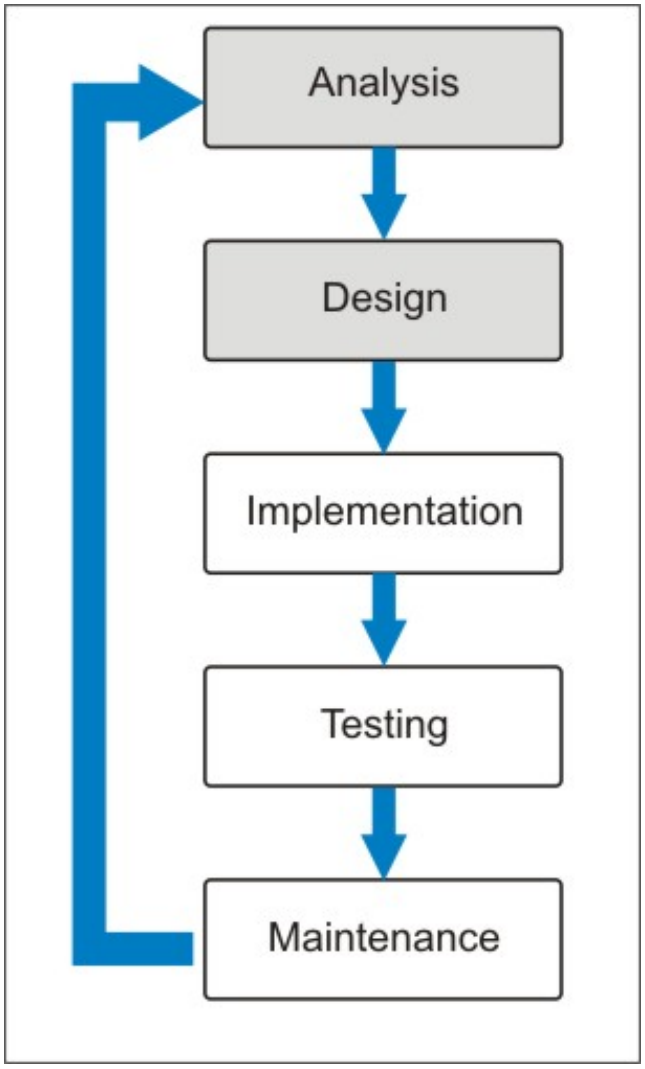

Figure 1.1: Software Development lifecycle

unimplemented methods, uninitialized variables, etc. In order to prevent those mistakes, we have to fall back onto coding standards in the development of software systems.

A possible approach is the use of code generation. Among the problems described earlier, it can also prevent errors that emerge in manually created code. Usually, a model (in e.g. UML) serves as a blueprint for generation the code. Sometimes the code generated from a UML diagram is not complete and merely offers stubs to the users that still have to be implemented manually. The main reason for this is that UML is a generic modeling language. However, if we constrain the required system to a restricted domain, we are able to generate specific code from a DSL.

1.3 Contribution

1.3.1 Aims and Objectives

The goal of this Master's Thesis is to provide a coherent modeling framework for the creation of collaborative applications running on the Android platform.

These are the main objectives of the study:

- Discuss the shortcomings of two level modeling and why we need multi-level modeling
- Analyze the possibilities of model driven engineering in the context of the Android platform
- Analyze and define the different types of collaboration methods
- Create a coherent framework that allows the creation of collaborative Android applications

1.3.2 Research Questions

The research questions are sustained on underlying hypotheses. These are the main hypotheses with accompanying research questions:

- Hypothesis 1. Android is an open source software stack.
  Research Question 1. How can we leverage the Android stack to support the modeling of Android applications?
- Hypothesis 2. There exists a categorization of different types of groupware options.
  Research Question 2. How can we support the different types of groupware options in a modeling framework?
- Hypothesis 3. The validation of the framework can be done based on certain quality attributes (QAs).
  Research Question 3. Which QAs are needed to evaluate the framework properly?

1.3.3 Expected Outcomes

These are the expected outcomes for this thesis:

- A study of the possibilities of modeling Android applications
- A study of the possibilities of multi-level meta-modeling
- An ontology of collaboration patterns

- A standardization of collaborative patterns in a framework (the collaboration stack)
- A complete description and design of a modeling framework for collaborative applications

1.4 Thesis Outline

This document starts with an overview of the fundamentals of domain specific modeling in chapter 2. It also explains why there are shortcomings in the two-level modeling approach. Chapter 3 continues with a solution (multi-level modeling) that resolves problems inherent to two-level modeling. It will also blabla Metadepth... Android dev... Collaborative shizzle... Case Study... Conclusion... Future Work!

CHAPTER 2

Domain Specific Modeling

Domain Specific Modeling (DSM) is a software engineering methodology for designing and developing complex systems. Modeling a system through a domain-specific language (DSL) allows the user to rapidly iterate through different prototypes and represent them at various levels of abstraction. The goal of this chapter is to give the reader an introduction to domain specific modeling and the primary meta-modeling technology, the UML framework.

Modern software design has reached a complexity that requires well- defined engineering methods and model-based approaches to ensure correct- ness. Complex systems are becoming extremely heterogeneous and the many engineering disciplines that are involved in system design all require problem- specific formalisms. As such, a company may resort to modeling techniques to solve complex problems in a specific way.

In the next section, traditional approaches to Meta-Modeling will be discussed. Through Meta-Modeling, a modeling formalism and their respective domain-specific aspects can be described. A meta-model of a modeling language L can furthermore be used to check whether a given model m is in L. They can also constrain the modeler during the incremental model construction process such that only elements of L can be constructed (1).

We continue with the shortcomings and problems of the UML Modeling Framework and current Meta-Modeling techniques in section 3. An example domain model will be presented and used throughout consecutive sections. This example will be based around a computer hardware product hierarchy and can be described in several ways.

The most common way for creating domain models is through the modeling language standard, UML. Creating a domain model in the UML essentially consists of

composing a model through one or more class diagrams, capturing the important domain concepts and their relationships. The main objective of this report is to address issues with the object-oriented paradigm currently underpinning the UML. UML supports modeling at two levels only, e.g. in the form of object diagrams (instance level) and class diagrams (type level). Thus, it provides only meager support for true multi-level modeling. As a consequence, this two-level modeling approach adds accidental complexity to domain models (2). Solutions for this problem will be discussed in section 5, Deep Meta-Modeling.

Furthermore, the main issues introduced by the *strict* meta-modeling approach, inherent to the UML Framework, will be discussed. These issues arise from the need to capture both *classlike* and *objectlike* features of some model elements in a few modeling levels. As a consequence, the modeler introduces a mismatch between the problem and the technology used to represent this problem. A mismatch that makes models more complex than they need to be, and by which *accidental complexity* is introduced. Solutions for avoiding this accidental complexity will be discussed in depth in chapter 3.

2.1 Traditional Meta-Modeling

This section discusses traditional meta-modeling approaches that are used in domain specific modeling. Before we go into the UML modeling frame- work, we need to define what a formalism is and how meta-modeling is used to describe a formalism.

2.1.1 Formalism

A formalism typically consists of a *syntactic* part and a *semantic* part. The syntax of the formalism deals with form and structure. Additionally, the syntactic part is separated into a *concrete* and *abstract* part. The concrete syntax relates to the actual appearance of the language elements (graphical or textual). The abstract syntax pertains to how the language components may be connected. For example, a Petri Net Transition may only be connected to a Petri Net Place.

The semantic part on its turn relates to the *meaning* of the syntactic constructs. The semantic part is also separated in two parts, *operational* and *denotational*. The operational part explicitly captures how a model can be executed. With a denotational specification, we can provide rules to map a model in a given formalism onto a model in a different formalism for which a semantics is available. We can for example map a statechart to its equivalent petri net. Both semantic approaches ultimately define a transformation.

2.1.2 Meta-Modeling

Meta-Modeling concentrates on the modeling of modeling formalisms. To solve domain specific problems as fast as possible, explicitly modeling the for- malisms is the

most efficient approach. Additionally, meta-modeling exhibits lots of other advantages. First of all, a modeling environment may be gener- ated from a meta-model by a set of language tools. A model of a modeling language can serve as documentation and as specification. Additionally, by modifying a meta-model, new domain-specific languages can be designed in an easy manner. Last but not least, the generation of a modeling tool through a meta-model is possibly orders of magnitude faster than developing such a tool by hand. This generated modeling tool will probably be less error prone too (1).

Meta-models are models on their own, so they must be specified in a modeling language. This modeling language is specified by a meta-meta-model. Most of the time, UML class diagrams are used to express the meta-model. The UML class diagrams are expressive enough to be expressed in themselves, e.g. UML is partly defined in terms of UML. This so-called *meta-circularity* refers to the fact that the meta-model of a language L is a model in language L.

2.2 UML Modeling Framework

In the following section, we will briefly discuss the UML Modeling Framework. First, we present the four-layered Meta-Object Facility along with a small example. Afterwards, strict meta-modeling and the fundamental problems it exhibits are introduced.

2.2.1 Linear Modeling Hierarchy

The overall architecture of the UML modeling framework is most heavily influenced by the CDIF standard developed by a consortium of CASE tool vendors[1]. The CDIF standard describes the different modeling languages used to create specific user models in terms of a single fixed, core model (a meta-meta-model) (3). The meta-modeling architecture is depicted in figure 2.1.

The M_3 level is the highest level, from which the UML meta-model is created. The UML meta-model (M_2) describes the core from which specific language meta-models are created. Specific models are then created as instances of these language meta-models on the M1 level. The User Data depicts instances of such models created in M_1. An example of this approach can be seen in figure 2.2.

In this example, the dashed arrows in the figure describe an "instance of" relationship. Every level in the diagram is an instance of its level above. An instantiation of the concept Video resides at level M_0. Video on its turn is an instantiation of Class and Class is regarded as an instance of the meta-meta-model elements (the M_3 level, not shown in this example).

[1] CASE Tools: Computer-Aided Software Engineering Tools

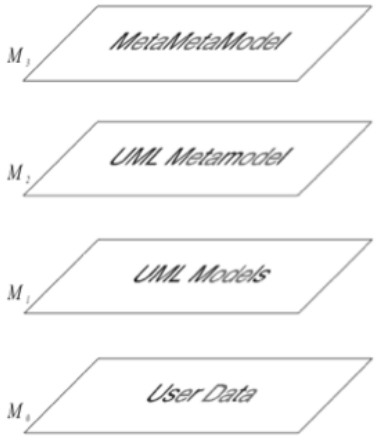

Figure 2.1: The four-level meta-modeling architecture

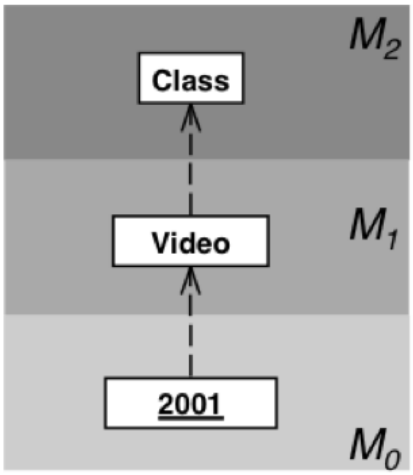

Figure 2.2: Extending the two levels of instantiation

2.2.2 Strict Meta-Modeling

The approach described earlier exhibits a number of fundamental problems. One of them is the fact that the precise meaning of the instance-of relationship is not defined. The UML documentation does not contain a formal definition of the instance-of relationship. It merely states that a modeling level in the hierarchy must be an instance of its level above. Therefore, a formal concept should be defined to encapsulate the exact meaning of the instance-of relationship. This concept is formalized through strict meta- modeling. Strict meta-modeling states that if a model A is an instance-of another model B, then every element of A must be an instance-of some element in B (3). This concept is illustrated in figure 2.3. As a consequence, modeling levels have strict boundaries and may be crossed only by instance-of relationships.

The exact definition is defined as follows:

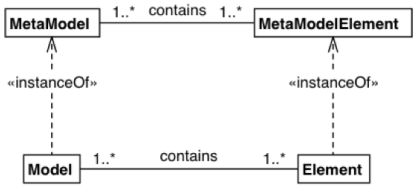

Figure 2.3: Strict meta-modeling.

Definition 1 *In an n-level modeling architecture, M_0, M_1, ..., M_{n-1}, every element of an M_m-level model must be an instance-of exactly one element of an M_{m+1}-level model, for all $m \leq m \leq$ n-1, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X) = level(Y).*

It is clear that the topmost level n is ignored in this definition. Nevertheless, we can model the top level so that its elements are instances of themselves, such that we can terminate the hierarchy of metalevels.

2.2.3 Limitations of Strict Meta-Modeling

There are several issues with the existing UML modeling framework, like *dual classification* and *class/object duality* (3). The class/duality problem arsis from the need to capture both classlike and objectlike facets of some model elements and dual classification is based on the fact that we need to capture both logical as well as physical aspects of model elements. Every solution to these problems ultimately leads to unwanted, accidental complexity. They try to cramp multiple programming levels into a single modeling level, due to the limitations of strict meta-modeling. In what follows, we will focus on both the class/duality problem and the dual classification problem. For example, one could note that in figure 2, the M_2 level aims to address two concerns. This is denoted in figure 2.4. Here, user types at the M_0 level are defined as instances of two concepts. First, the *2002* instance is an instance-of the M_1-level type *Video*, and second of the M_2-level meta-type Object. Although this approach feels natural, it is clearly in violation with our definition of strict meta-modeling:

- Object 2001 has more than one classifier (Video and Object).
- One instance-of relationship crosses more than one meta-level boundary.

It is clear that we have to modify our approach to strict meta-modeling. The following section addresses the issues in which accidental complexity occurs and provides well-defined solutions for them.

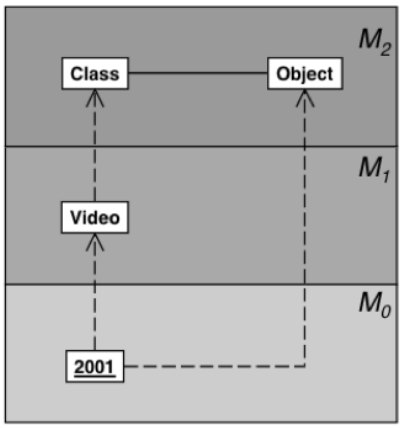

Figure 2.4: Multiple classification in the current UML framework.

2.3 Problems with the two-level UML Modeling approach

In this section, we will introduce an example domain model that is used throughout the remaining sections. We will also demonstrate a few workarounds used to represent multi-level domain models using only two modeling levels.

2.3.1 Example domain model

This sample model adopts a computer hardware product hierarchy using the "Item Description" pattern. This example exhibits a typical technique which is often used to capture a multi-level do- main classification into the two-level UML modeling style. The example class diagram is depicted in figure 2.5.

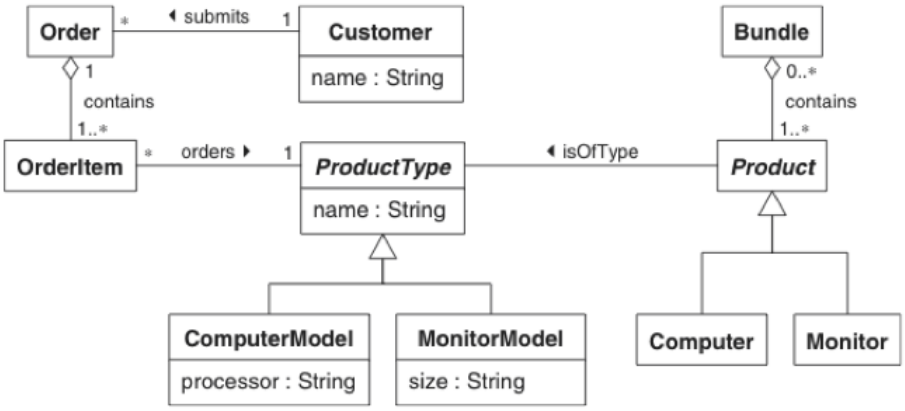

Figure 2.5: Example class diagram of the Item Description pattern.

The above class diagram is separated into two parts, one modeling products sold by an enterprise and the other part modeling the descriptions of these products.

Here, instances of `ProductType` (`ComputerModel` and `MonitorModel`) are descriptions of the `Product` types (Computer and Monitor). The idea is to let objects play to role of classes in order to explicitly represent class-level information (2). In this way, we can dynamically change information about a product and keep this information even when no representation of this product type exists. When we create an instance of the above class diagram, we get something as depicted in figure 2.6.

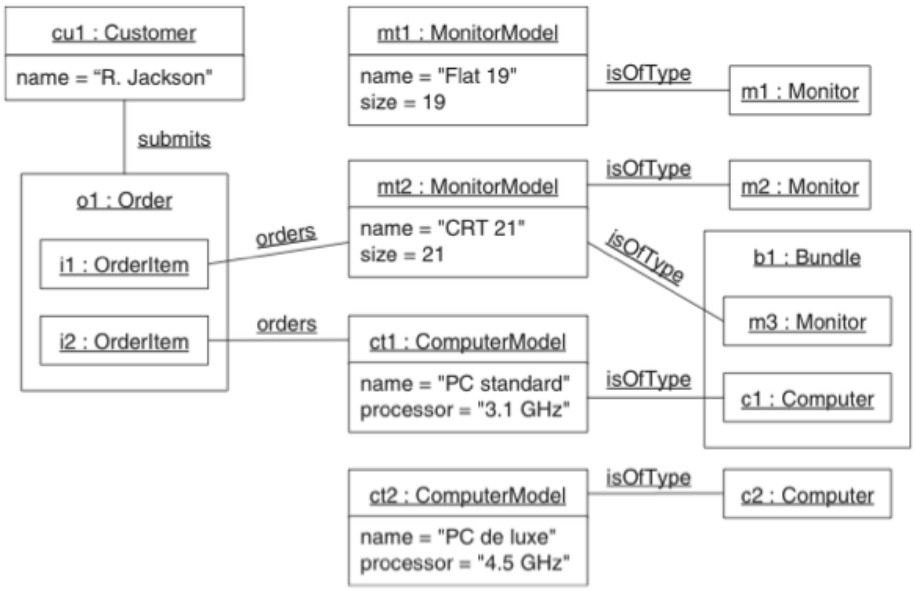

Figure 2.6: Domain instance (object diagram).

The "isOfType" links between "Monitor"/"Computer" instances and "MonitorModel"/"ComputerModel" instances conceptually represent a form of instance-of relationship (as explained in previous section), although all objects involved in the example are at the instance level in an object diagram.

2.3.2 Domain types as objects

We have used the graphical "isOfType" links to model the instance/type relationships at the object level. However, we can distinct up to three different kinds of instance-of relationships:

- the UML built-in "instance-of" relationship between the elements in the object diagram and the elements in the class diagram (e.g. `m1` *instance-of* `Monitor`)
- the "isOfType" associations in the class diagram between `Product` classes and `ProductType` classes (e.g. `Computer` *isOfType* `ComputerModel`)
- the "isOfType" links between instances of `Product` and instances of `ProductType` (e.g. `c2` *isOfType* `ComputerModel`)

Undoubtedly, the combination of these relationships introduce some kind of accidental complexity. To see the consequences caused by this approach, we have combined the object and class diagram in one figure that explicitly represents all the relationships that conceptually exist. This diagram can be found in figure 2.7. This figure actually highlights two aspects of the two-level modeling paradigm that was not explicitly clear from the separate diagrams:

- It shows all UML "instance-of" relationships explicitly in addition to the "isOfType" relationships.
- It highlights the fact that the object diagram contains model information that represents domain instances and types of the domain conceptualization.

The black dots in the figure below the dashed lines represent the domain entities. On the one hand, we have got domain types and on the other hand, weve got domain instances. The relationships between the domain entities use the same "instance-of" relationships as between the model types and model instances. They represent *ontological* "instance-of" relationships, because they show which domain entities are considered to be types of other domain entities.

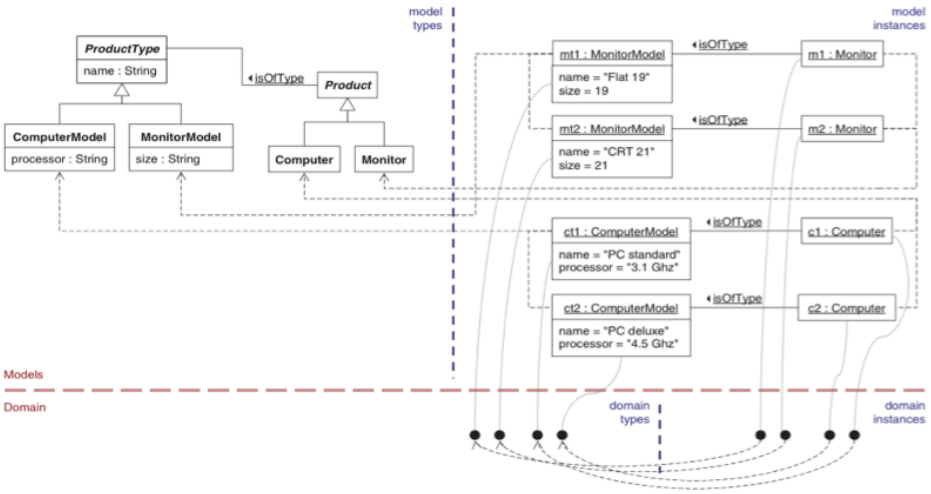

Figure 2.7: Combination of the class and object diagram.

After investigation of the figure above, we see that it contains some redundant information. There are some model elements that do not represent any domain elements. Type `Product` and its subclasses can be considered as abundant model attributes if the model is interpreted as a domain model, because they have two classification relationships instead of just one. The subclasses `Computer` and `Monitor` merely exist to set up the *Item Description* pattern. For instance, `m1` has type `Monitor` plus a modeled type `mt1`. The redundancy introduced here is an example of accidental complexity, because some information only exists to realize some workaround technique.

2.3.3 Domain types as classes

The model presented in the previous subsection is often used when we need to introduce new types of products dynamically. When we do not need to create objects dynamically at runtime, we can simplify the model significantly. This approach is shown in figure 2.8. It shifts some modeling elements that represent domain types (i.e. `mt1`) to the type level (into the class diagram). According to the UML class/object modeling conventions, this is a more natural place for these elements to be.

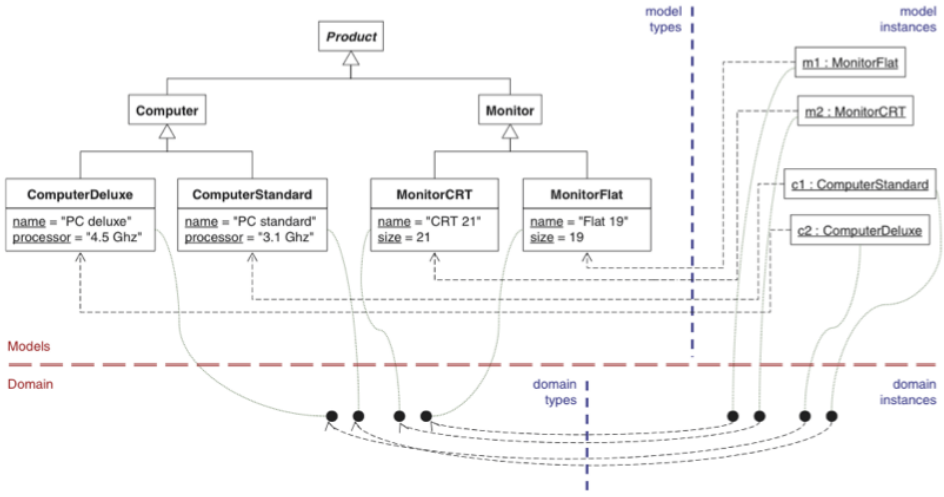

Figure 2.8: Types as classes.

Although the figure above removes some accidental complexity from the previous example, we also lost some features. It is no longer possible to instantiate new types at runtime, because product types are now represented by classes. Without additional information, we cannot conclude whether the workaround methods of previous section were necessary or not. Therefore, it is hard to compare the models and evaluate them on the level of accidental complexity. In the next section we will see whether it is possible to keep the simplicity of figure 2.8 and still support the ability to create new product types at runtime.

CHAPTER 3

Multi-Level Modeling and Metadepth

This chapter continues where chapter 2 left off. Now that we know we can capture the important domain concepts and their relationships through the modeling language standard UML, we can address issues with the object-oriented paradigm currently underpinning the UML. UML supports modeling at two levels only, e.g. in the form of object diagrams (in- stance level) and class diagrams (type level). Thus, it provides only meager support for true multi-level modeling. As a consequence, this two-level modeling approach adds accidental complexity to domain models. A possible solution involves the use of powertypes. Another solution is to introduce an object that can both capture *classlike* and *objectlike* features, *clabjects*, together with the concept of potency to support an arbitrary number of modeling levels. Both concepts are explained in depth. Finally, Metadepth, a framework for multi-level meta-modeling is presented. This approach is trying to solve the mismatch between the two level modeling paradigm and the information it models.

3.1 Deep Meta-Modeling

A question that undoubtedly comes to ones mind is it is possible to keep the simplicity of figure 2.8 and still support the ability to create new product types at runtime. In order to make this possible, the type level should be as dynamic as the object level. We should of course have some control over what types are created and what features they possess. Therefore, we need (domain-) meta-types controlling the features of (domain-) types (2). However, the problem with the "Item Description" pattern is that it uses three classification levels, where only two level (class/object) mechanisms are supported. This introduces a level mismatch problem, depicted in figure 3.1. It shows that our example actually features *three* ontological levels, while the corresponding models only contain *two* modeling levels. The level of the

model instance is used to represent two ontological levels, domain types and domain instances.

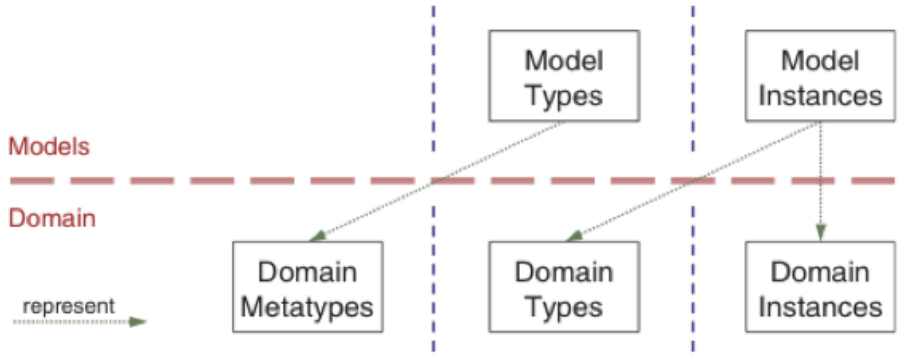

Figure 3.1: Mapping domain levels to modeling levels.

In the following, we will discuss how it is possible to avoid squeezing multiple domain levels in only two modeling levels.

3.1.1 Clabject

In order to truly support multiple levels, modeling concepts that can be applied in a uniform way across all levels in a multilevel hierarchy should exist (2). A modeling construct that supports the representation of the dual type and object-property of some domain concept is necessary. Therefore, the concept of *clabject* was introduced. Clabjects are a combination of *class* and *object*. They have a name and a set of attributes, like classes, and a set of slots, like objects. The example of previous section, modeled using clabjects, is depicted in figure 3.2. This solution features less accidental complexity than solutions of previous sections, because it maximizes the mapping between the problem structure and the solution structure. Note that the model types do not contain attributes, they only fill in slots, and subsequently instances do not contain any slots. Furthermore, we have used only a single notion of the "instance-of" relationship.

Clabjects support any number of model classification levels, which creates a *direct mapping* between ontological domain levels and modeling levels. This concept is visualized in figure 11. This direct mapping requires that the solution structure reflects the problem structure as close as possible. Now that we have introduced clabjects, a uniform way to support multiple domain levels, we can address the issue when elements in one level need to influence elements beyond its neighbouring levels.

3.1.2 Powertypes

Using clabjects, the accidental complexity in the domain model was reduced significantly. However, there is one additional issue that needs to be addressed to allow multiple classification levels to be modeled naturally. This is the issue of *deep characterization*. Deep characterization means that a type can influence the attributes

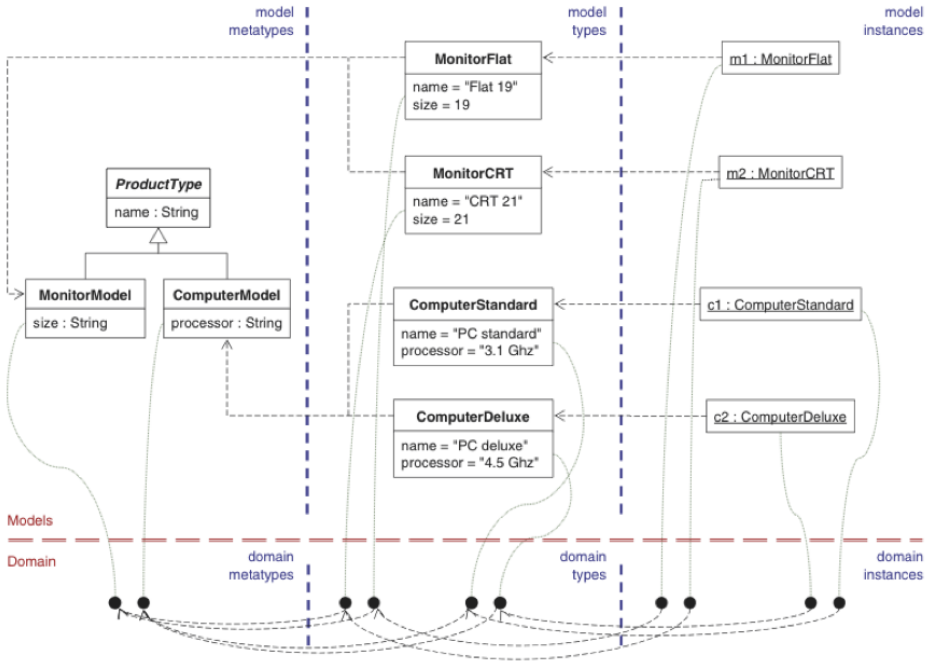

Figure 3.2: Example modeled using clabjects.

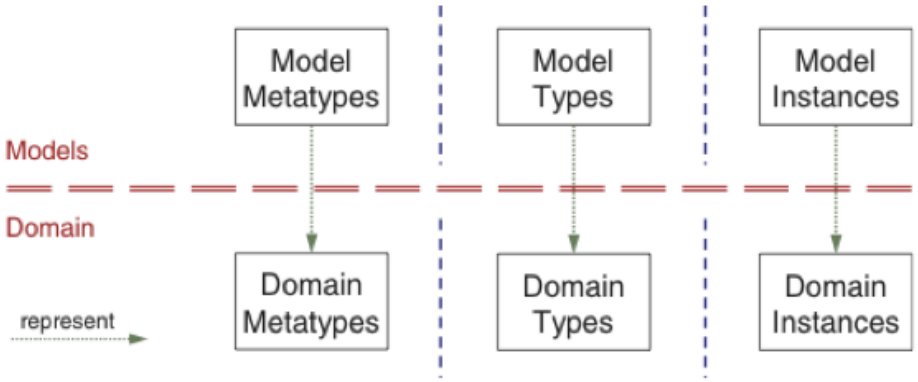

Figure 3.3: Direct mapping between ontological domain levels and modeling levels.

of its instances types as well as their object facets. Figure 3.4 shows an example of how powertypes can be used to represent deep characterization. If a superclass (e.g. `Computer`) is said to have a powertype (`ComputerModel`), then an instance of the powertype (`ComputerStandard`) is only well-formed when it inherits from the superclass (`Computer`). Every derived class of a class inheriting from a powertype should also be an instance of that powertype (e.g. `ComputerDeluxe` inherits from `Computer` and is also an instance of the powertype `ComputerModel`). Hence, all subclasses in our example will have a `processor` or `size` *slot* and a `price` or `picQlty` *attribute*.

For example, the `size` attribute of `MonitorModel` is a slot in `MonitorFlat`, because it is an instance of `MonitorModel`. Powertypes therefore control the type facet of powertype instances by means of inheritance. The powertype mechanism supports deep characterization, but as can be seen in the example only at the cost of introducing supertypes whose only purpose might be to provide a type facet for their subclasses. In the next section, we will solve this accidental complexity by introducing *deep instantiation*.

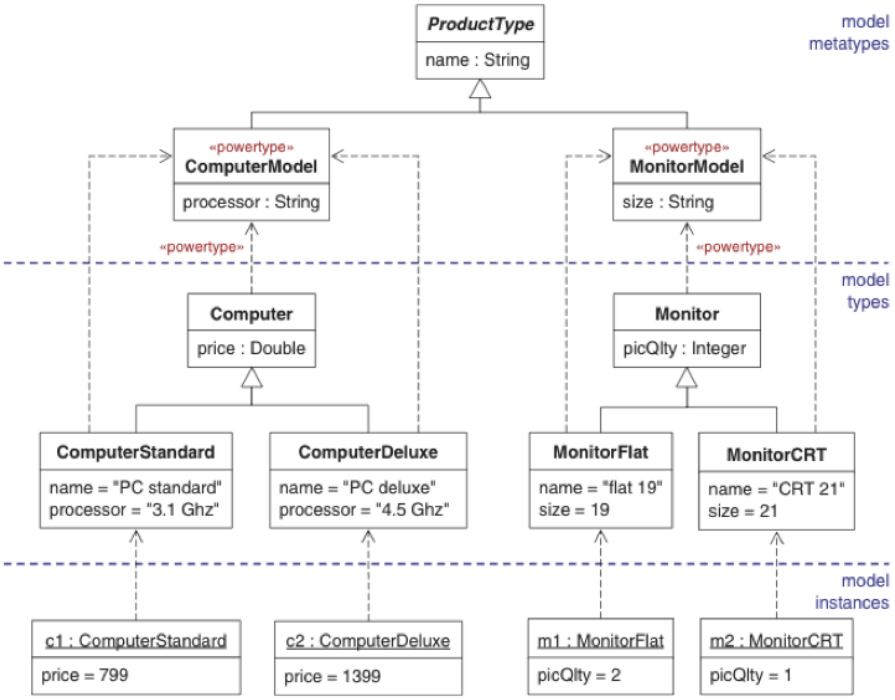

Figure 3.4: Deep characterization with powertypes.

3.1.3 Deep Instantiation

Deep characterization exists when a type should influence an entity beyond its immediate instances. In order to support deep characterization, we need a mechanism that backs it up in a concise way and minimizes accidental complexity introduced by deep characterization with powertypes. This mechanism is referred to as *deep instantiation*.

Potency

Deep instantiation covers two concepts. One is the unification of attributes and slots into a single concept, which we refer to as `field`. The other concept extends clabjects and fields with an additional property known as *potency*. Potency defines how deep an instantiation chain produced by a clabject or field may become. For example, when we create a field of potency two, it can produce an instantiation depth of two. Each instantiation of the field lowers the potency value by one, until the

potency value equals zero. Entities with potency zero cannot be further instantiated and act like regular objects or slots. Figure 3.5 shows our domain model represented by deep characterization.

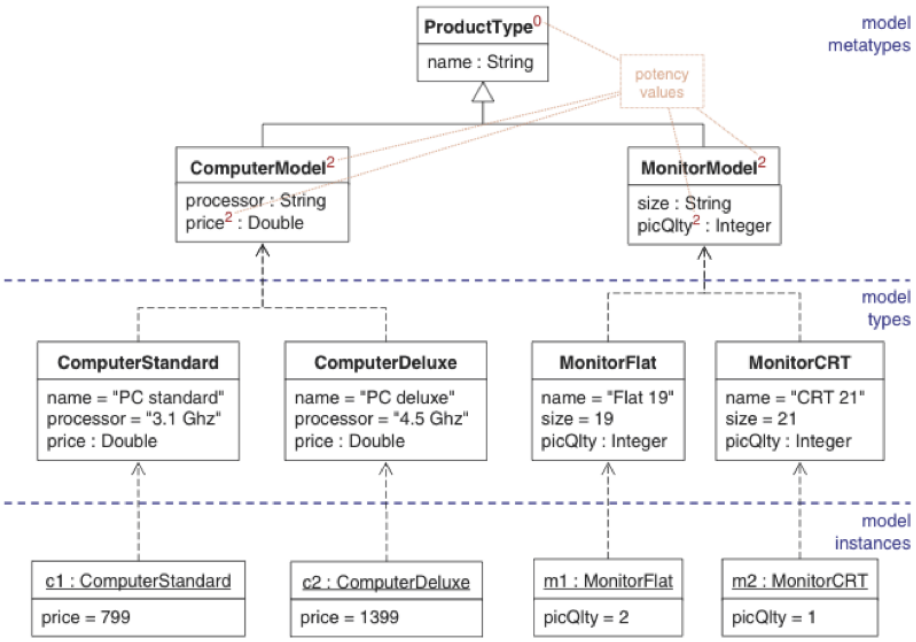

Figure 3.5: Deep characterization with potency.

The clabject `ComputerModel` has a field `price` of potency two at the model metatypes level. This is indicated by the superscript *2* at the field `name`. After instantiating the clabject once, the potency of field `price` is reduced by one. Note that price acts like an attribute on this level (indicated by a potency of 1). When we instantiate this clabject to an object `c1`, field `price` has turned into a slot that has a real value. Note that a potency value of zero for clabjects with non-zero potency fields allow us to create an *abstract* (meta-)class.

The example using potency features the least accidental complexity yet. It only allows computer and monitor types for computer and monitor instances respectively, without enhancing it with extra constraints. Furthermore, it doesnt require an extra superclass for merely providing a type facet like with powertypes.

3.2 Metadepth

MetaDepth is a meta-modeling framework written in Java that uses a deep meta-modeling approach to support multiple programming levels. It allows a developer to work in two ontological (domain) instantiation modes: *strict* and *extensible*, as

shown in figure 3.7.

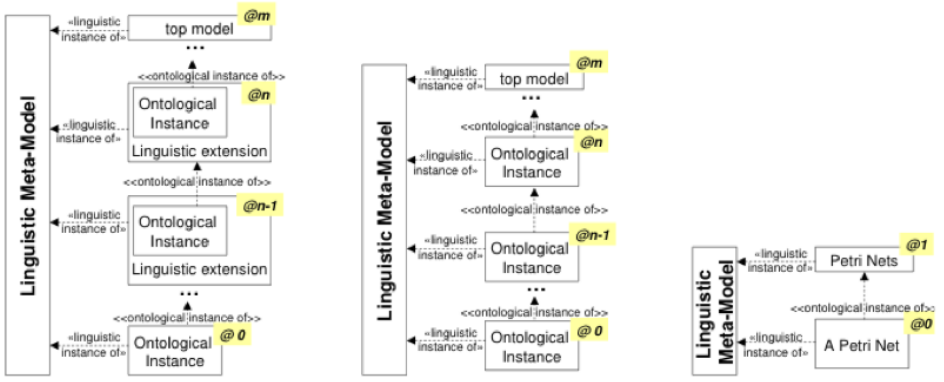

Figure 3.6: MetaDepth instantiation schemes: extensible ontological instantiation (left) and strict ontological instantiation (center). Example of strict instantiation (right).

In the extensible case, it is possible to linguistically extend each ontological instance model (as depicted in the figure). This means that instances of elements marked as `ext` can be extended with new attributes. We can also mark a complete model as extensible, allowing us to add new types and extend its elements. The alternative *strict* case is similar to most meta-modeling environments in the sense that the top-level meta-model hardcodes all language concepts and can be subsequently instantiated. This is depicted in the center of figure 3.7.

3.2.1 The Linguistic Meta-Model

MetaDepth uses its own linguistic meta-model, inspired by MOF (5). Its modified to accommodate an arbitrary number of meta-levels, deep instantiation and potency. Part of the linguistic meta-model is shown in figure 15. The uncolored classes are those the designer usually instantiates when building his own model.

The root class `Clabject` takes responsibility of handling the dual type/object facet of elements (5). Therefore, it contains a potency attribute and links to its instances and types. `Constraints` can be attached to clabjects, as shown in the class diagram. All working models are managed by a `VirtualMachine` container, which is a *singleton* object. An example of the dual classification using a clabject and the two instance-of relationships is shown in figure 3.8. The ontological model stack depicts the user-defined (meta-)models, instantiated through the linguistic meta-model shown above.

3.2.2 Tool Support

MetaDepth models can be built through a Java API or through a `CommandShell` and a textual syntax, built with ANTLR (7). The storage of models is also done in this format. As an example, figure 3.9 shows the three model dual classification example of figure 3.8 in the textual notation. In this example, the model Store

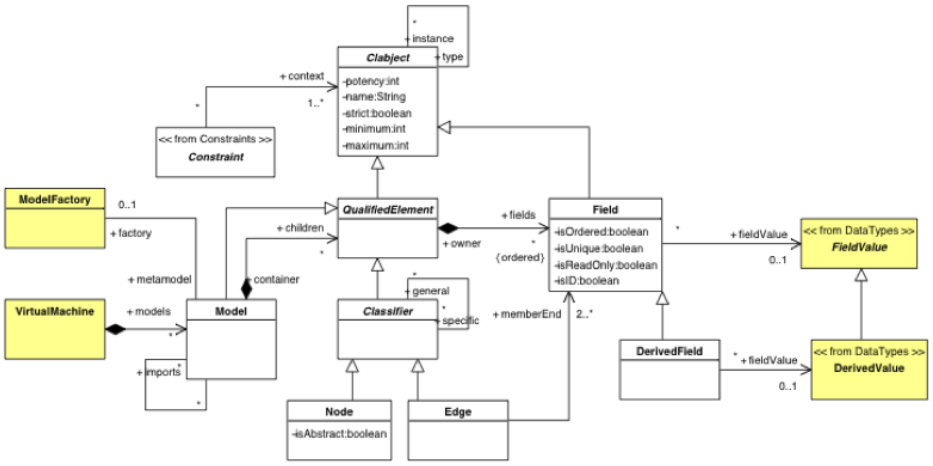

Figure 3.7: MetaDepths linguistic meta-model.

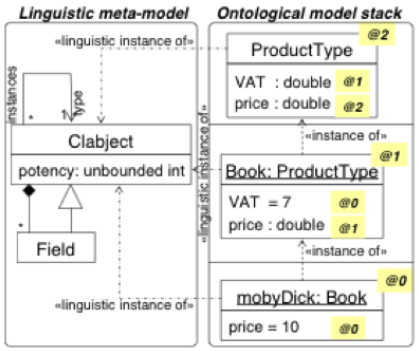

Figure 3.8: Dual classification example.

(with potency two) is extended linguistically through the Node `ProductType`. The VAT attribute in node `ProductType` has a potency of one (and a default value of 7.5), which means that it will be instantiated in its lower neighboring level. We could also declare *anonymous* clabjects, like `Book{price=10;}`.

```
1 Model Store@2 {
2   Node ProductType {
3     VAT@1 : double = 7.5;
4     price : double = 10;
5   }
6 }
```

```
7 Store Library {
8   ProductType Book { VAT = 7; }
9 }
10 Library MyLibrary {
11   Book mobyDick { price=10; }
12 }
```

Figure 3.9: Three model example.

The Metadepth framework is integrated with Epsilon, a family of languages built on top of the Epsilon Object Language (EOL) (8), by communicating with the

models through a connectivity layer. EOL can work with EMF models (9), but also with any other model technology that implements the interface of this connectivity layer. Metadepth works through this kind of interface and adds support to make EOL aware of multiple ontological levels. Using this approach, we can use EOL programs to create models, as shown in figure 3.10. We could also use EOL to define the semantics of a model (e.g. write a Petri Net simulator based on a model using the Epsilon Transformation Language (ETL) (10)).

```
1 context MyLibrary
  ::entering context MyLibrary
2 # EOL
  :: entering eol execution mode
3 for (i in Sequence{1..1000}) {
4   var b: new Book;
5   b.price:=10+i/500;
6 }
```

Figure 3.10: An EOL program that populates a MetaDepth model.

3.2.3 Constraints and Derived Attributes

Constraints and actions in Metadepth are usually defined using Java or EOL. Just like clabjects and model fields, they have an assigned potency that indicates at which meta-level they have to be evaluated. Figure 19 extends our Store example.

```
1 Model Store@2 {
2  Node ProductType@2 {
3   VAT@1      : double = 7.5;
4   price@2    : double = 10;
5   discount@2: double = 0;
6   minVat@1   : $self.VAT>0$
7   minPrice@2: $self.price>0$
8   maxDisc@2 : $self.VAT*self.price
8    *0.01+self.price<self.discount$
9   /finalPrice@2: double =
9    $self.VAT*self.price/100
9    +self.price-self.discount$;
10  }
11 }
```

Figure 3.11: Constraints and derived fields in Metadepth.

We have declared three constraints, on line 6, 7 and 8. An example of a derived field is given on line 9. The constraints are specified between two "$" symbols and can be defined at the level of a clabject (as done in our example) or outside of it. The `minVat` constraint was given a potency of one, which means that it will be evaluated on the meta-level directly below. This constraint cannot access the value of fields with bigger potency, like `price`, as they may not have a value yet. The declaration of the derived field `finalPrice` is similar to a normal field, except that it is preceded by a backslash and may include fields with a lower potency.

3.2.4 Controlling linguistic extensions

Linguistic extension is interesting to permit unforeseen extensions to Domain Specific Languages spawning more than one level (6). In these languages, the top-most meta-model is usually highly generic, and linguistic extensions in lower levels could therefore be useful. Figure 3.12 shows an extension of our running example.

In the scenario above, we are interested in associating an author with `ProductType` instances. To achieve this, we linguistically extend the `Store` model by adding a

```
1 Store Library {
2  ProductType Book {
3   VAT    = 7;
4   title  : String;
5   author : Author;
6  }
7  Node Author {
8   name    :String;
9   nonRep@1:$Author.allInstances().
9     forAll(x|x<>self implies
9            x.name<>self.name)$
10  books : Book[1..*]{unique};
11 }
12 Edge writer (Book.author,
12                Author.books) {
13  year : int;
14 }
15 }
```

Figure 3.12: Linguistic extensions and associations in Metadepth.

new node `Author`, instance of `Node` in the linguistic meta-model (briefly discussed earlier). Authors are related to one or more books, modelled through the field books. The {`unique`} modifier ensures that a given `Author` is not related to the same `Book` twice.

Associations can be annotated with fields by explicitly defining an `Edge` between their association ends. In the example above, we have an association `writer` instantiated as an `Edge` that relates books and authors. This `Edge` has got an extra field `year` that includes the year in which the book was written by that author. We could also define a more generic type of association. We could refer to linguistic types, like `Node`, when defining association ends. This makes sense if we want to specify that a certain association end is to be taken by any (linguistic) instance of `Node`.

3.3 Other deep meta-modeling solutions

Metadepth offers one solution to the problem of deep meta-modeling. However, other solutions exist, but not all are offered as complete frameworks. For example, DeepJava (4) is an extension of Java with the concept of potency, thus it cannot be considered as a framework. It does provide methods with potency, but needs special keywords to navigate up the type hierarchy in order to find attribute values. The constraints and computations for derived attributes in Metadepth can access type fields in a uniform way. This approach is preferred because it has the advantage that the number of meta-levels do not matter for a given field.

Next to DeepJava, another proposal for deep meta-modeling has been developed (11). This tool is largely based on *Ecore*. It considers multi-level constraints and proposes extending OCL to cope with multiple ontological meta-levels. This approach is similar to MetaDepth, but MetaDepth has got the ability of assigning *potency* to constraints, which makes them easier to define on multiple meta-levels. More information about these concepts can be found in (11) and (5).

CHAPTER 4

Android Development

Android offers a software stack for mobile devices. Apart from what people refer to as the Operation System, it also contains middleware and key applications. The Android SDK offers the necessary tools and APIs to start developing Android applications. These applications are developed using the Java programming language. The following are the most important features of the Android stack:

- **Application framework** enabling reuse and replacement of components
- **Dalvik virtual machine** optimized for mobile devices
- **SQLite** for structured data storage
- **Media support** for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- **GSM Telephony** (hardware dependent)
- **Bluetooth, EDGE, 3G, and WiFi** (hardware dependent)
- **Camera, GPS, compass, and accelerometer** (hardware dependent)
- **Rich development environment** including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Eclipse IDE

4.1 Android Architecture

This section will explain the major components of the Android architecture. The complete architecture is depicted in figure 4.1. We will follow a top-down approach, starting with the lowest level of abstraction (Applications) first all the way down to

the Linux Kernel, that provides core system services such as security and memory management.

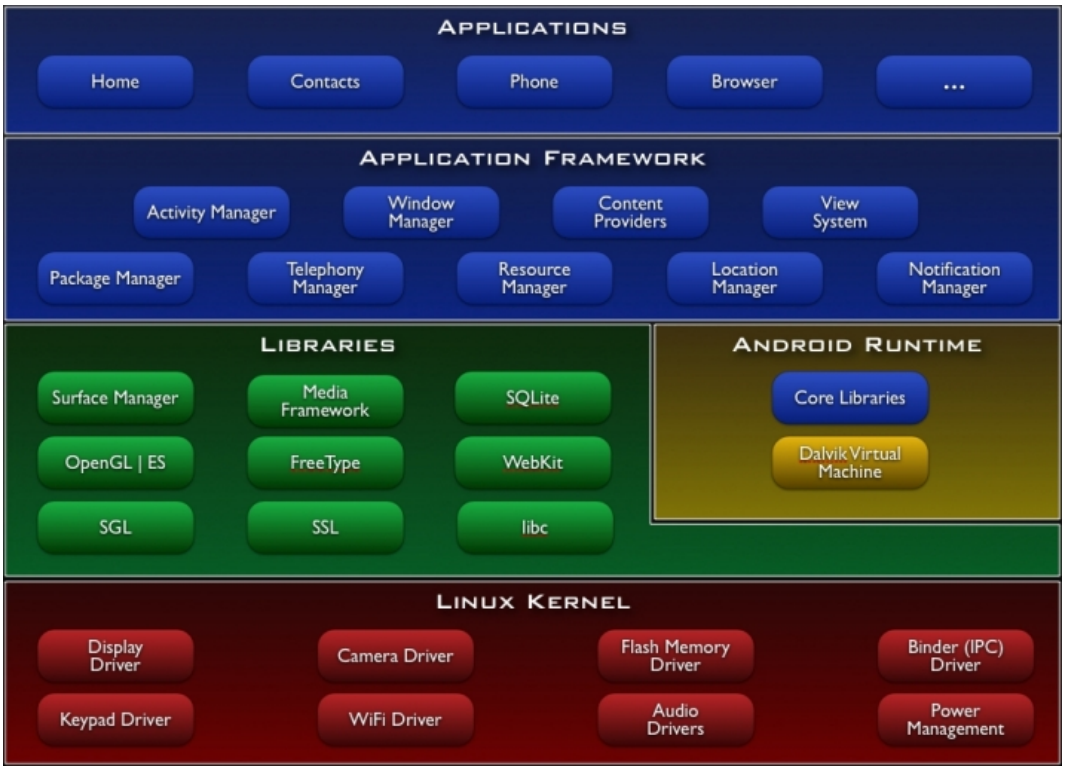

Figure 4.1: Android Architecture.

4.1.1 Applications

Android applications are developed using the Java programming language. By default, Android ships with a number of default applications written in Java. These include an email client, SMS program, calendar, maps, browser, contacts and others. In the next section, we will see the major components of a typical Android application. In other words, the building blocks of a minimal Android application.

4.1.2 Application Framework

Android offers the ability to build extremely rich and innovative applications. Developers are granted the same access to the framework APIs used to build the core applications on the top level. The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities [**?**].

Underlying all applications is a set of systems and services:

- A set of `Views` to build the front-end of the application, including lists, grids, text boxes and buttons. A subset of these Views will be supported on the

modeling framework.

- `Content providers` that manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process. [**?**]
- `Resource Manager` that provides access to non-code resources such as images and strings, so they can be maintained independently. Externalizing your resources also allows you to provide alternative resources that support specific device configurations such as different languages or screen sizes, which becomes increasingly important as more Android-powered devices become available with different configurations. [**?**]
- `Notification Manager` that notifies the user of events that happen. It allows a developer to tell the user that something has happened in the background [**?**]. For instance, the receival of an SMS can trigger a notification.
- `Activity Manager` manages the lifecycle of one Activity. More details on Activities and the management can be found in the next section.

4.1.3 Libraries

Although the Android SDK is offered in the Java programming language, the lower level libraries are a set of C/C++ libraries used by various components in the Android system. Like in a typical stacked architecture, a lower level layer is exposed through a higher level layer. Therefore, these libraries are accessible through the Android application framework. Some of the core libraries [**?**]:

- **System C library** - a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices
- **LibWebCore** - a modern web browser engine which powers both the Android browser and an embeddable web view
- **SGL** - the underlying 2D graphics engine
- **SQLite** - a powerful and lightweight relational database engine available to all applications

4.1.4 Android Runtime

As seen in the above layers, Android has quite a unique application component architecture. In order to make the Android environment suitable for multiple applications, Android executes multiple instances of its own customized Virtual Machine, Dalvik. Basically, each Android application runs in its own process, with its own instance of the Dalvik Virtual Machine.

As a result of this approach to multiprocessing, Android must efficiently divide

memory into multiple heaps, where each heap is as small as possible, so that many applications can fit in memory at the same time. In order to be space-efficient, Android uses a special component lifecycle, which enables objects to be garbage-collected and recreated. Next to this, Dalvik is able to run a bytecode system specifically developed for Android, called dex. Dex bytecodes are approximately twice as space-efficient as Java bytecodes, halving the memory overhead of Java classes for each process.

4.1.5 Linux Kernel

Finally, Android relies on a linux kernel for core system services such as security, memory management, process management, network stack, etc. This kernel also acts as an abstraction layer between the hardware and the rest of the software stack. It controls the hardware elements throughout the software stack and serves as a gateway.

4.2 Android Components

This section will describe the different Android components and their respective managers in depth. We will start with a comparison of traditional versus Android programming. While every Android application is developed in the Java programming language, the approach of writing such an application is different from writing traditional (desktop) applications.

4.2.1 Traditional versus Android programming

When starting applications in a traditional Operating System, there usually exists a single point of entry called `main`. The OS will load the program code into a process and starts executing it. Additionally, if we look at programs written in Java, it gets a little more complex. A Java virtual machine (JVM) that resides within a process loads bytecode to instantiate Java classes as the program uses them. This process is depicted in figure 4.2.

The Android system works a little different and supports multiple application entry points. Instead of a sequential code hierarchy, an Android program is a cooperating group of components that may be started from outside the normal flow of the application. For example, if your application starts the activity in the camera application that captures a photo, that activity runs in the process that belongs to the camera application, not in your application's process. The photo capture function can be integrated by many applications in their UI flow.

4.2.2 Activities and Intents

Typically, an Android activity is a unit of interaction. It usually fills the whole screen of an Android mobile device. It also functions as a unit of execution. Note that an Android application does not have a single point of entry; an application

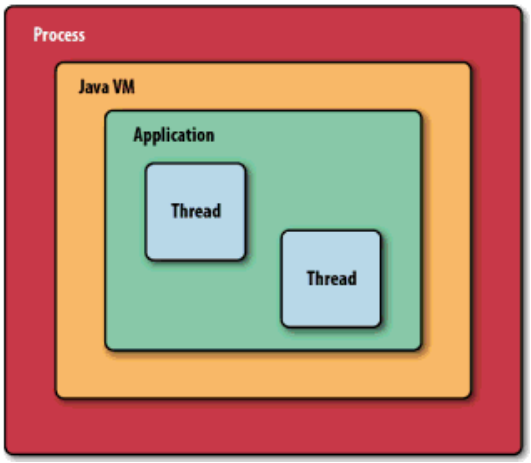

Figure 4.2: A Java application running in a JVM.

can have entry points to multiple activities. They are the reusable, interchangeable parts of the flow of UI components across Android applications. An activity interacts with the Android runtime to implement key aspects of the application life cycle.

Now if one activity invokes another, we usually want to pass some information to it. The unit of communication is called Intent. They are the basis of a system of loose coupling that allows activities to launch one another. It is usually a bad idea to keep references to activities in memory, because of the way Android does garbage collection and the restrictions of memory on a mobile device. In general, an activity is an isolated and independent object that only communicates with other activities through intents.

4.2.3 Tasks

So communication in an Android application is defined by means of an intent. Unlike in traditional desktop applications, the UI flow in Android applications is also described through intents. Using these intents, an Android developer can create a chain of activities that spans more than one application. This is referred to as a `task`. An example of a task spanning multiple activities is depicted in figure 4.3.

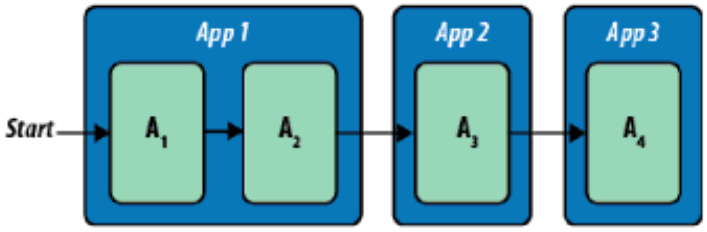

Figure 4.3: Activities in a single task spanning multiple applications.

An example of this task can be found in table 4.1. In this example, app 1 represents the Android Messaging app. First, a user views all messages and after that decides to read a specific message. These two actions map onto two activities. Afterwards, he decides to view the contact, and the user is sent to another application (and another Activity). When the user decides to call this contact, we need another application again. This one task thus involved three applications and four activities. The UI flow in this task is completely defined by means of intents.

Table 4.1: A single task across multiple applications and spanning multiple activities

App	Activity	User's next action
Messaging	View list of messages	User taps on a message in the list
Messaging	View a specific message	User taps Menu - View Contact
Contacts	View a contact	User taps Call Mobile
Phone	Call the contact's mobile number	nothing

4.2.4 Other components

Apart from the `Activity` class, there are three other important components that contribute to Android applications: services, content providers and broadcast receivers.

A `Service` class supports long-running background tasks. They may be active, but not visible on the screen. A typical application that may contain a `Service` class is a music player. We usually want to continue listening to music while doing some other task.

Next, a `ContentProvider` class provides access to a data store for multiple applications. An Android developer specifies a special URI starting with `content://` that gives access to the data store. A `ContentProvider` class works analogous to a RESTful web service. They have a specific URI and associated operations such as putting and getting data.

Finally, the `BroadcastReceiver` class allows multiple objects to listen for intents broadcast by applications. `BroadcastReceiver` is similar to `Activity`, but does not have its own user interface.

While these components can possibly be important in Android applications, currently only the `Activity` class will be supported in our modeling framework. The other components can additionally be implemented as future work.

4.2.5 AndroidManifest

In order for an Android application to know what its contents are, we need to explicitly describe them in an XML file called *AndroidManifest.xml*. In this file, we declare all our activities, services, content providers and broadcast receivers along

with their intents. Since an Android application does not have a single point of entry, we also need to specify which component is the main component (also done through another `Intent`). A visualization of the structure of the AndroidManifest can be found in figure 4.4.

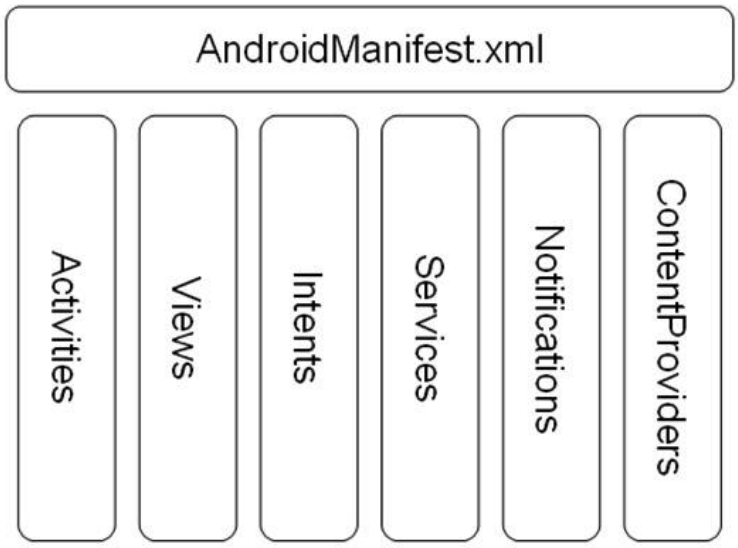

Figure 4.4: Structure of an AndroidManifest.xml.

4.2.6 Activity Lifecycle

As explained earlier, the Android system can enable objects to be garbage collected and recreated. Due to this mechanism activities have a special customized life cycle, because they can easily be garbage collected when inactive. When a user wants to activate the activity again, the Android system will need to be able to recreate the `Activity` object. Activities are saved on an *activity stack*. The `Activity` on top of the stack is always the running activity. Activities on the stack below the current one are always invisible and inactive. An activity can be in one of four states:

- *Active* or *Running* if the activity is in the foreground of the screen
- *Paused* if the activity has lost focus but is still visible. This activity is still alive but can be killed by the Android system in case of low memory.
- *Stopped* if the activity is obscured by another activity. All state information is still in memory, but the activity is not visible and will most likely be killed when memory is needed elsewhere.
- When the activity is *paused* or *stopped*, the system can ask to finish it or simply kill its process. When a user requests access to the activity again, it should be restored to its previous state.

In figure 4.5, the important states of an Activity lifecycle are visualized. The states an Activity can be in are the colored ovals. The rectangles introduce the callbacks a developer can implement to perform operations when moving between states.

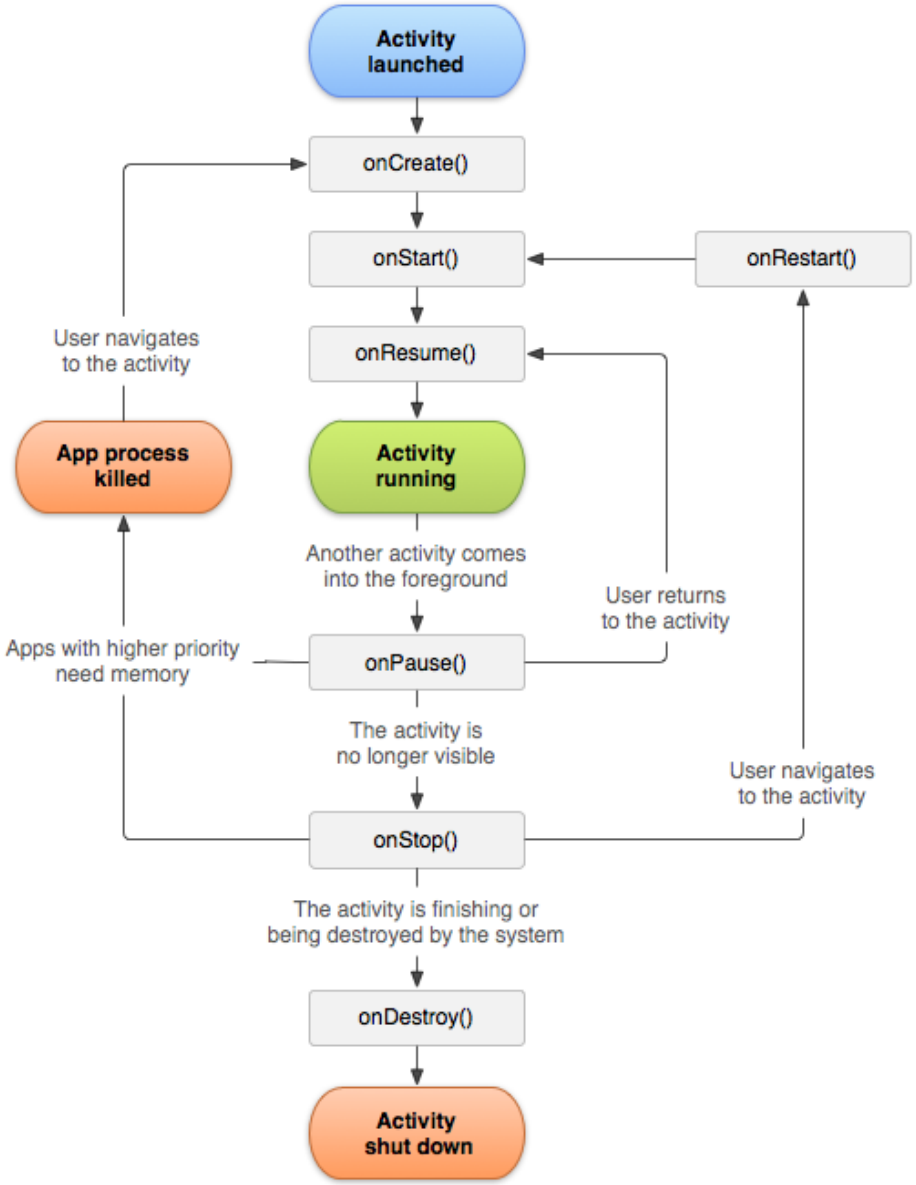

Figure 4.5: The lifecycle of an Android Activity.

4.3 Android App Inventor

Android App Inventor is an application provided by Google that allows anyone to create applications for the Android system. Google wanted to encourage people to create software for the Android platform, even people unfamiliar with computer programming. App Inventor provides a graphical interface that has drag-and-drop functionality, so that users can easily create new applications or prototypes. An example of the application is depicted in figure 4.6.

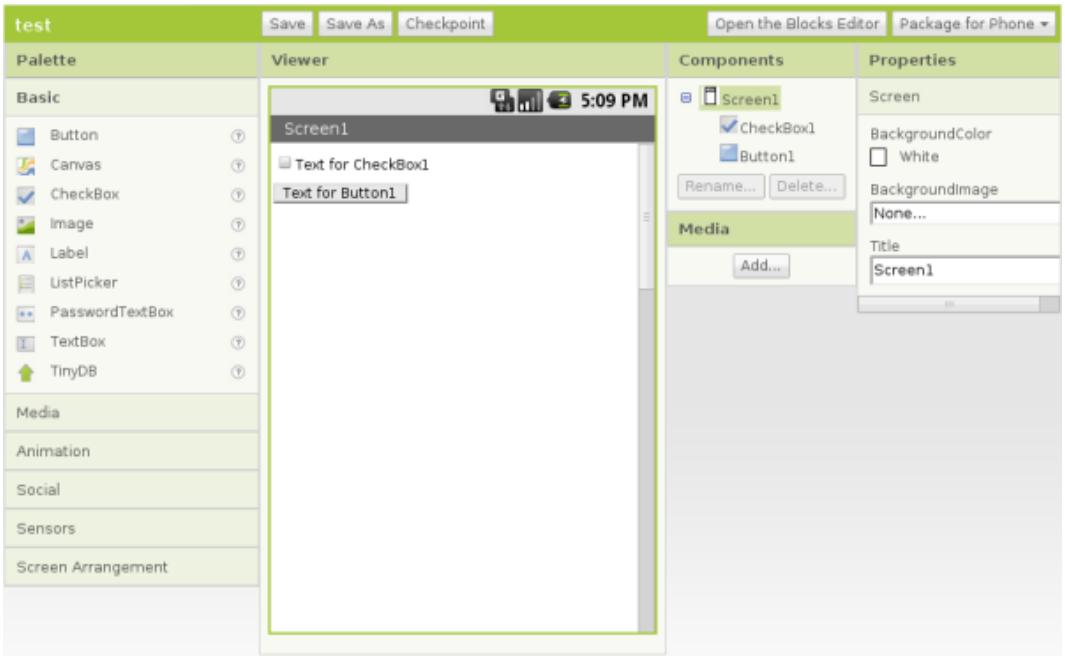

Figure 4.6: Google App Inventor.

However, Google terminated the App Inventor on December 31, 2011. MIT Center for Mobile Learning is now supporting it under the name "App Inventor Edu". The Android App Inventor has been a source of inspiration in the development of my own modeling framework.

CHAPTER 5

Collaboration

5.1 Groupware

5.1.1 Definition

5.1.2 Communication, Collaboration and Coordination

5.1.3 Taxonomies

5.1.4 Real-time Groupware Concepts

5.1.5 Groupware Sessions

5.1.6 Groupware Research and Problems

5.2 Computer Supported Coopartive Work

5.2.1 Definition

5.2.2 CSCW versus Groupware

5.2.3 Challenges in CSCW design

5.2.4 Requirements engineering in CSCW

5.2.5 Technology and CSCW application integration

5.3 Computer Supported Collaborative Learning

5.3.1 Definition

5.3.2 CSCW versus CSCL

5.3.3 Jigsaw method in CSCL

5.4 Collaborative Patterns

5.4.1 Session

5.4.2 Repository

5.4.3 Object

5.4.4 View

5.4.5 Broadcast

5.4.6 User

5.4.7 Role

5.4.8 Environment

5.5 Collaboration Stack

5.5.1 Ontology of collaboration patterns

CHAPTER 6

Designing a Collaborative Modeling Framework

6.1 Components

CHAPTER 7

Case Study: Collaborative Knowledge-Management Platform

7.1 Environmental Analysis

7.2 Collaborative Knowledge-Management Platform

CHAPTER 8

Conclusion

8.1 Summary

8.2 Future Work

Appendices

APPENDIX A

Meta-modeling and Petri Nets

This chapter is an overview of the project done in the Petri Nets course.

Bron	Resource
Bronbeheer-systemen	Resource Management Systems
...	

Table A.1: Meta-modeling and Petri Nets

APPENDIX B

Model to model transformation [Research Internship 2]

This chapter is an overview of the project done in the Petri Nets course.

Bron	Resource
Bronbeheer-systemen	Resource Management Systems
...	

Table B.1: Meta-modeling and Petri Nets

Bibliography

[1] W. Scacchi, "Process models in software engineering," *Encyclopedia of Software Engineering*, 2001.