Universiteit Antwerpen

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

Proefschrift voorgelegd op 28 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

Almost a year ago I was taking on a new challenge and left Belgium for Spain. A few months before, I was presented this thesis subject. I would have the chance to work out of Madrid to use the Metadepth framework and explore the possibilities of domain-specific modeling and model transformation to support collaborative work on Android.

Now, more than a year later, I can say I'm very satisfied and happy of all the chances I got when taking on this challenge. The regular meetings with my mentors kept me on the right track. Working on this project wasn't always easy, but the thought that I could be doing things I like and achieving real progress kept me going. Throughout the whole year, we conducted research to collaborative patterns and we've been thinking about ways to integrate collaboration. The result is a modeling framework that supports the creation of collaborative Android apps.

First of all, I'd like to thank my promoter Hans Vangheluwe, co-promoter Juan de Lara and mentor Bart Meyers. Without them, I would never have had the chance to pursue a year of studying in Spain. This country changed my life, both on a professional and on a personal level. I always knew I wanted to build things, but here I created my first startup and learned about raising money from actual investors. Not only that, but I saw the importance of a healthy mix of business sense and technical knowledge. Also, without the continuous support of Juan, my work wouldn't be as valuable. It was a pleasure to have someone I could rely on at all times. Thanks to all the friends I made in Spain. I guess most of them didn't really understand what a thesis is about, but they were always listening. And of course I would not have been able to successfully get my computer science degree without the support of my parents.

Let me finish by saying that I hope the information in this work will be as valuable to you as it is to me. Thanks to everyone reading this.

Abstract

Domain-specific modeling (DSM) is a software engineering methodology for designing and developing systems, such as computer software. Among other things, it can be used to rapidly validate ideas through prototypes. Moreover, it also provides a way of doing formal verification. A developer can easily do model checking or formal analysis by transforming a domain-specific language into another formalism. Metadepth is a multi-level meta-modeling framework developed at the Universidad Autónoma de Madrid that supports DSM. The main features of Metadepth are the following:

- Support for an arbitrary number of ontological meta-levels. This feature makes Metadepth especially useful to define multi-level languages.
- Textual modeling. A concrete textual syntax allows us to specify a Metadepth meta-model in detail and uses standardized keywords accompanied by parameters to make computer-interpretable expressions.
- Integration with the Epsilon family of languages. Hosts both Java and Epsilon Object Language (EOL) as action and constraint languages.
- Not based on the Eclipse Modeling Framework (EMF). Metadepth runs as a stand alone application.

At present, there is no all-in-one modeling solution that allows the generation of Android applications. Nevertheless, the potential for Android through (meta-)modeling is huge. In this work, Metadepth is used to create a new modeling framework that allows the creation of collaborative Android applications. This involves the creation of collaborative patterns and the design and implementation of a set of meta-models that model an Android application. These meta-models are transformed to Java code using the Epsilon Generation Language (EGL).

Abstract (Dutch)

Domein-specifiek modelleren (DSM) is een software engineering methodologie voor het ontwerpen en ontwikkelen van systemen, zoals computer software. Het kan onder andere gebruikt worden voor het valideren van ideeën door middel van prototypes. Bovendien biedt het ook een manier om aan formele verificatie te doen. Een ontwikkelaar kan gemakkelijk model checking of een formele analyse uitvoeren door middel van transformatie van een domein specifieke taal (DSL) naar een ander formalisme. Metadepth is een multi-level meta-modeling framework dat DSM ondersteunt, ontwikkeld aan de Universidad Autónoma de Madrid. Metadepth ondersteunt volgende functies:

- Ondersteuning voor een willekeurig aantal ontologische meta-levels. Deze functie maakt Metadepth in het bijzonder nuttig voor het definiëren van multi-level languages.
- Tekstueel modelleren. Een concrete tekstuele syntax stelt ons in staat om een Metadepth meta-model in detail te specifiëren. Het gebruikt gestandaardiseerde trefwoorden vergezeld van parameters om computer-interpreteerbare uitdrukkingen op te stellen.
- Integratie van de Epsilon familie van *languages*. Gebruikt zowel Java als de Epsilon Object Language (EOL) als action en constraint *languages*.
- Niet gebaseerd op het Eclipse Modeling Framework (EMF). Metadepth kan als een stand-alone autonome applicatie uitgevoerd worden.

Op dit moment bestaat er geen all-in-one oplossing die een modeler toelaat om Android applications te genereren. Desondanks is het potentieel voor Android door middel van (meta-)modeling enorm. In dit werk wordt Metadepth gebruikt om een nieuw modeling framework te ontwikkelen. Dit framework laat een modeler toe om snel collaboratieve Android applicaties te ontwikkelen. Het gaat hier om het creëren van collaboratieve patterns en het design en implementeren van een verzameling van meta-models die een Android applicatie kunnen modelleren. Deze

meta-models kunnen vervolgens getransformeerd worden naar Java code met behulp van de Epsilon Generation Language (EGL).

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 (DSM) applied to Android development for the generation of collaborative applications. Domain-Specific Modeling and Model Transformation (MT) have the potential to support rapid development and the 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 higher level framework that allows the creation of collaborative Android applications.

This work arises from the rising need for mobile applications that solve recurring collaborative problems. We try to tackle those recurring problems by applying DSM on the Android stack to support collaborative work. When a non-technical person previously wanted to solve a problem by means of a smartphone application, this person had to learn how to program. In this framework, we tackled this problem by giving the developer a high-level framework that allows creating a sufficiently complex Android application within hours. The types of applications that can be generated fit specific needs, with a focus on collaboration. Moreover, no knowledge of the Java programming language or the Android SDK are required.

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*, which is used extensively in DSM, will be discussed as a possible solution to recurring software engineering problems.

1.2.1 Specification

Specification identifies the problems a new software system is supposed to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance [2]. Based on the requirements specification, software engineers can build an architectural design. This design can not only be used to build the system, but also has as a goal 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 [3].

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 the specified requirements. Therefore, the verification process always relies on the requirements specification, as this document defines the intended behavior of the system. We should 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 [4]), integration testing (components tested as a group [5]), regression testing (uncover new errors in existing functionality after changes have been made to a system [6]) and other ways of verifying that the system behaves as we expect it to. Another process closely related to this is *validation*. Validation is the process of determining that a system meets the *actual* requirements of the user. In this process, we should ask ourselves *"Are we building the right system?"*. Moreover, we should both `verify` and `analyze` a software system.

One of the major benefits of DSM is the ability to use other verification techniques than testing, i.e. formal verification. In the context of this work, formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property, using formal methods, where formal means "mathematically precise" [7]. An example of formal verification is model checking. Domain-specific languages (DSLs) can be verified using model checkers, which are tools that automatically verify that a system cannot exhibit some specified undesirable behavior such as deadlock, livelock, information leakage etc. The model checker explores all logically possible execution traces of the system, making the verification equivalent to exhaustive testing. The model checker returns an execution trace exhibiting the unintended behaviour (e.g., deadlock) if such a trace exists, a useful feature during debugging [8].

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. This makes the design lifecycle is a lot easier, as the modeling of applications is very rapid and processes several cycles on the background. 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, 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 prevent errors that emerge in manually created code. Usually, a model (e.g. in UML) serves as a blueprint for the generation of 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.

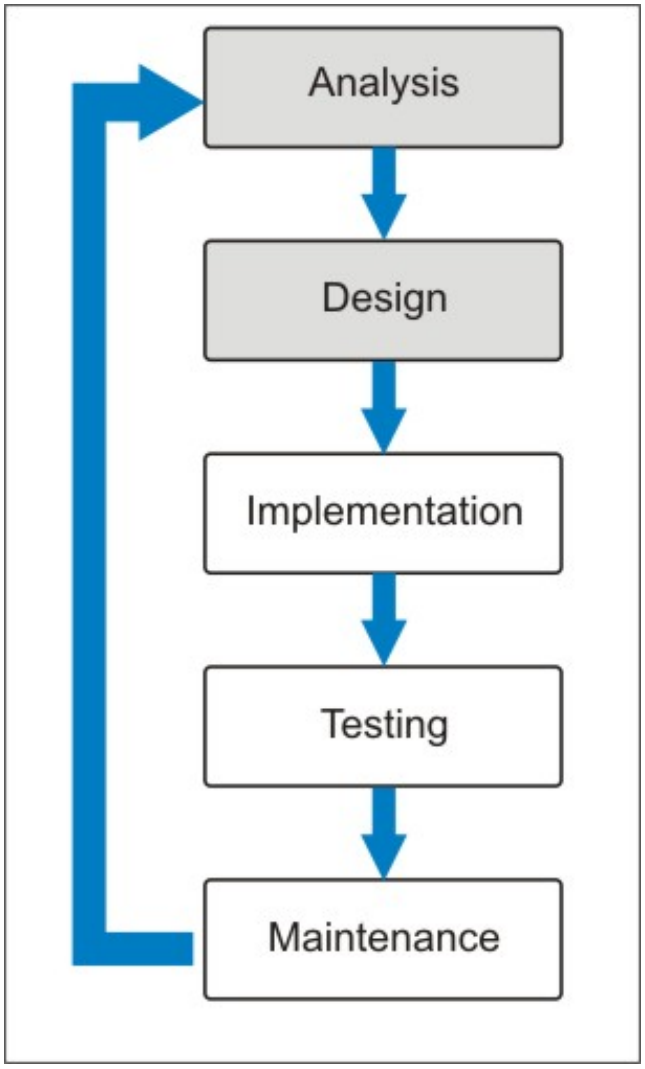

Figure 1.1: Software Development lifecycle

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 domain-specific modeling 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. The following 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.
  Research Question 2. How can we support the different types of groupware in a modeling framework?
- Hypothesis 3. We have a broad offering of collaborative applications and functionalities are offered as separate applications.
  Research Question 3. How can we integrate different functionalities in one collaborative application that is adaptable to a group of (non-technical) people?

1.3.3 Expected Outcomes

These are the expected outcomes for this thesis:

- A study of the possibilities of multi-level meta-modeling
- A study of the possibilities of modeling Android applications
- An ontology of collaboration patterns (the collaboration stack)
- A standardization of collaborative patterns in a framework
- A 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. We illustrate these problems using the Metadepth multi-level modeling framework. In chapter 4, the main components of Android are discussed, such as Activities and the AndroidManifest file that is a blueprint for Android components in an application. Chapter 5 discusses the different collaborative methods such as Groupware, Computer-Supported Cooperative Work (CSCW) and Computer-Supported Collaborative Learning (CSCL). Using the concepts we have learned from chapter 5, we can make a complete framework with the appropriate collaborative patterns included in chapter 6. This chapter will contain all the designs that make the entire collaborative modeling framework. Chapter

7 proposes a case study where we have developed a complete application using the collaborative modeling framework. The application allows a group of users to authenticate (and thus initiate a session) and collaborate with each other on a certain research project. They can communicate asynchronously (by adding list items or comments on research papers) as well as synchronously (chatting). Finally, chapter 8 concludes the thesis and talks about future work.

CHAPTER 2

Domain Specific Modeling

Domain-Specific Modeling 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 DSM and the primary meta-modeling technology, the Unified Modeling Language (UML) framework.

Modern software design has reached a complexity that requires well-defined engineering methods and model-based approaches to ensure correctness. 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.

The most common way for creating domain models is through the modeling language standard, UML. Creating a domain model in 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 chapter 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 class diagrams (type level) and object diagrams (instance 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 [9]. Solutions for this problem will be discussed in chapter 3, Deep Meta-Modeling.

In the next section, traditional approaches to meta-modeling will be discussed. We continue with the shortcomings and problems of the UML Modeling Framework and current meta-modeling techniques. 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.

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 model elements in two 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 dive into the UML modeling framework, 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 formalisms is the most efficient approach. Additionally, meta-modeling exhibits lots of other advantages. First of all, a modeling environment may be generated from a meta-model by a set of language tools. A model of a modeling language can also serve as documentation and as a 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

most likely be less error prone too [10].

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. On their own, UML class diagrams are expressive enough to be expressed by 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.

2.1.3 Linear Modeling Hierarchy

In this section, we briefly discuss the UML Modeling Framework. First, we present the four-layered Meta-Object Facility (MOF) along with a small example.

The overall architecture of the UML modeling framework is most heavily influenced by the CASE Data Interchange Format (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) [11]. The meta-modeling architecture is depicted in figure 2.1. The meta-meta-model (M_3) level is the highest level, from which the UML meta-

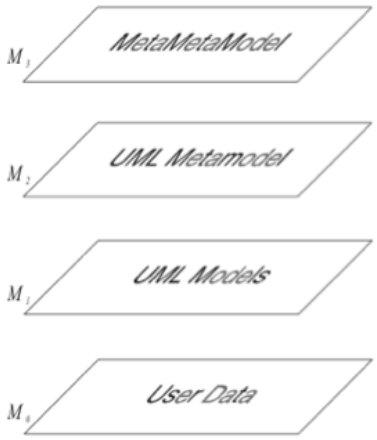

Figure 2.1: The four-level meta-modeling architecture

model is created. In general, it is the core from which the descriptions of specific modeling languages (i.e. specific language meta-models) are 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. For example, an instantiation of the concept

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

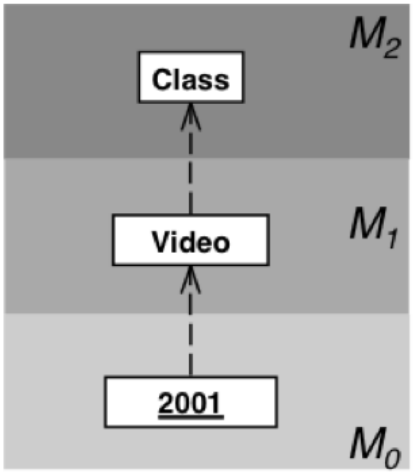

Figure 2.2: Extending the two levels of instantiation

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).

2.1.4 Linguistic Meta-Modeling

In the meta-modeling field, we can distinguish two different approaches of meta-modeling, linguistic and ontological meta-modeling. The linguistic approach relegates ontological instance-of relationships, which relate user concepts to their domain types, to a secondary role [1]. A linguistic instance-of relationship crosses a linguistic metalevel, whereas ontological instance-of relationships do not cross such levels. Ontological instance-of relationships merely relate entities within a given level. Figure 2.3 shows this interpretation of the four layered architecture as embraced by the UML and MOF standards.

2.1.5 Ontological Meta-Modeling

When we want to dynamically extend a modeling language, only linguistic meta-modeling doesn't suffice. Modeling language extensions are facilitated as dynamic user extensions, introducing the need for ontological meta-modeling. This dynamic requirement in turn requires the capability to define domain metatypes (i.e. types of types). We refer to this form of meta-modeling as ontological meta-modeling since it is concerned with describing what concepts exist in a certain domain and what properties they have [1].

When we take a look back at figure 2.3, we already see an ontological instantiation example. The model element `Lassie` is an ontological instance-of relationship of `Collie`, and thus resides at a lower ontological level than `Collie`. This expresses the fact that in the real world, the mental concept `Collie` is the logical type of `Lassie` [1]. Figure 2.3 only contains two ontological levels, both contained within

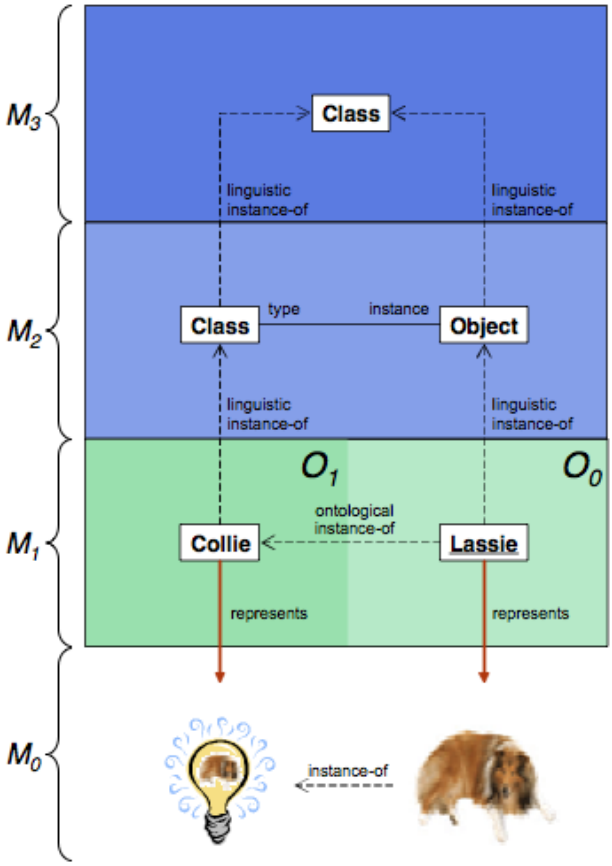

Figure 2.3: Linguistic meta-modeling [1]

the linguistic level M_1. We can extend the ontological approach among multiple levels too. This is featured in figure 2.4, where we introduce three ontological levels. In ontological level O_2, we show that `Collie` can be regarded as an ontological instance of `Breed`. The ontological metatype `Breed` not only distinguishes types like `Collie` and `Poodle`, but also from other types (such as `CD` and `DVD`). Moreover, we can also use this metatype to define breed properties, for example, where a particular breed first originated.

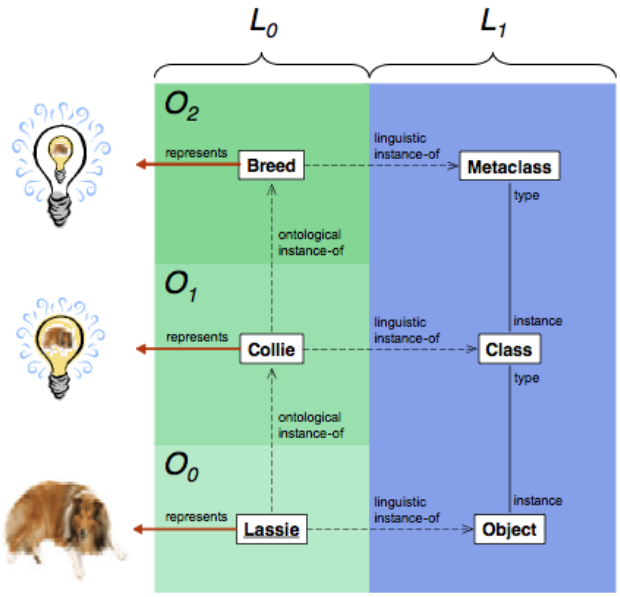

Figure 2.4: Ontological meta-modeling [1]

2.2 Strict Meta-Modeling

In the following section, we will discuss strict meta-modeling and the fundamental problems it exhibits are introduced.

2.2.1 Definition

The linear modeling approach described in the previous section 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 [11]. This concept is illustrated in figure 2.5. Here we see that strict meta-modeling interprets the instance-of relationship at the level of individual model elements. As a consequence, modeling levels have strict boundaries and may be crossed only by instance-of relationships. The exact definition of strict meta-modeling is defined as follows:

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 $0 \leq m \leq n\text{-}1$, and any relationship other than the instance-of relationship between two elements X and Y implies that level(X) = level(Y).*

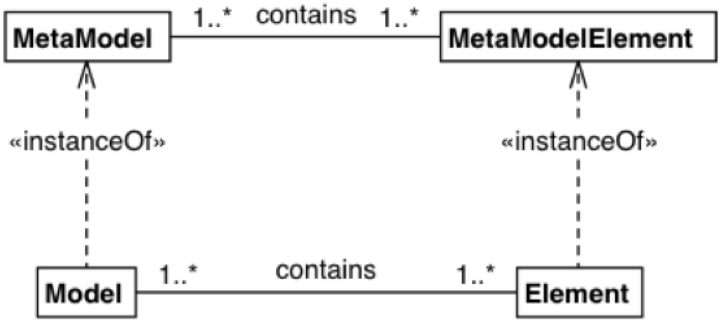

Figure 2.5: Strict meta-modeling.

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 meta-levels.

2.2.2 Limitations of Strict Meta-Modeling

There are several issues with the existing UML modeling framework, such as *dual classification* and the *class/object duality* [11]. The class/duality problem arises from the need to capture both classlike and objectlike facets of some model elements in one level and dual classification is based on the fact that we need to capture both logical (e.g. `Video`) as well as physical (e.g. `Object`) aspects of model elements. Every solution to these problems ultimately leads to unwanted, accidental complexity. Those solutions try to include 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 2.2, the M_2 level aims to address two concerns. This problem is denoted in figure 2.6. Here, user types at the M_0 level are defined as instances of two concepts. First, the *2001* 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).
- An 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.

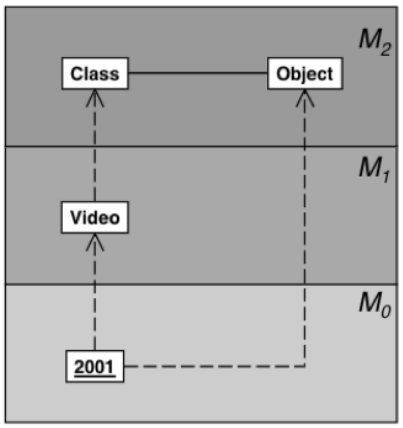

Figure 2.6: Multiple classification in the current UML framework.

2.3 Problems with the two-level UML Modeling approach

In this section, we will introduce an example domain model. 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 domain classification into the two-level UML modeling style. The example class diagram is depicted in figure 2.7.

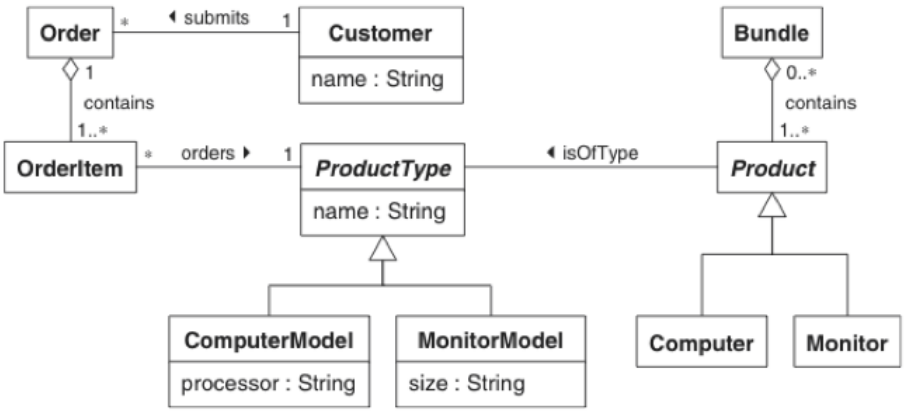

Figure 2.7: Example class diagram of the Item Description pattern.

The above class diagram is separated into two parts. One part models products sold by an enterprise and the other part models 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 the role of classes in order to explicitly represent class-level information [9]. In this way, we can dynamically change information about a product and keep this information even when no representation of this product type exists. An example of an instance of the above class diagram is depicted in figure 2.8.

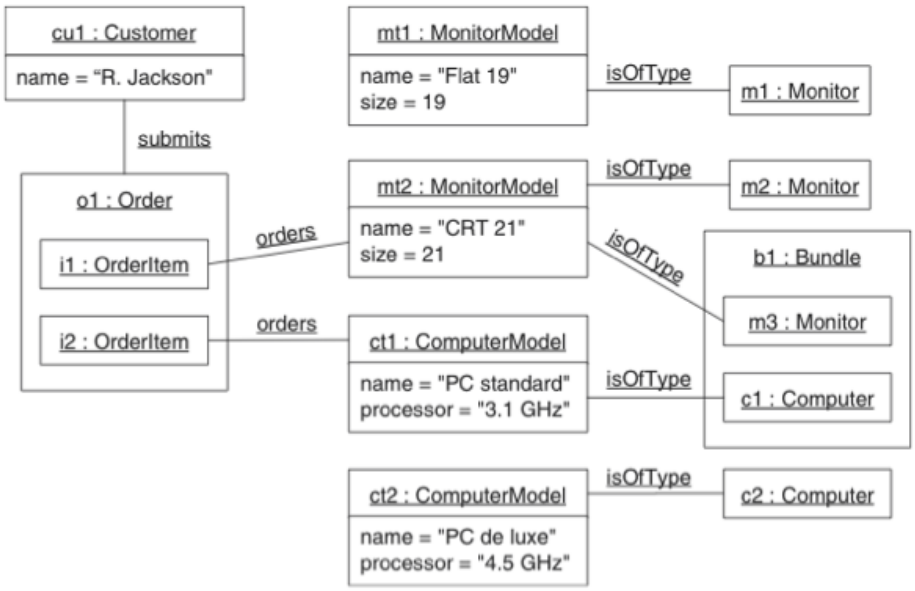

Figure 2.8: Domain instance (object diagram).

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. This verbosity does not violate strict meta-modeling, but introduces a workaround at the object level in order to create new objects at runtime.

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 distinguish 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 2.9. 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, we have 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.

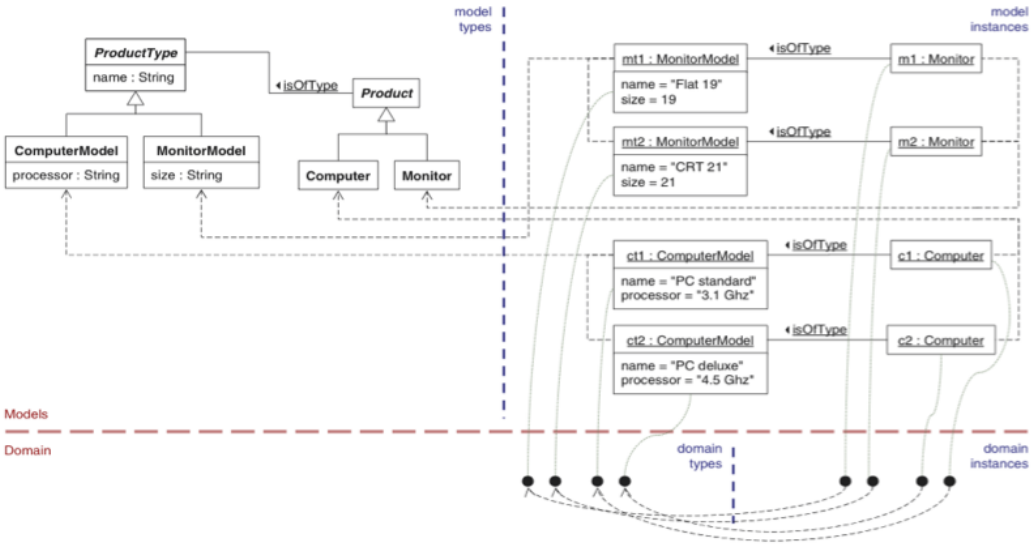

Figure 2.9: Combination of the class and object diagram.

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 2.10. 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.

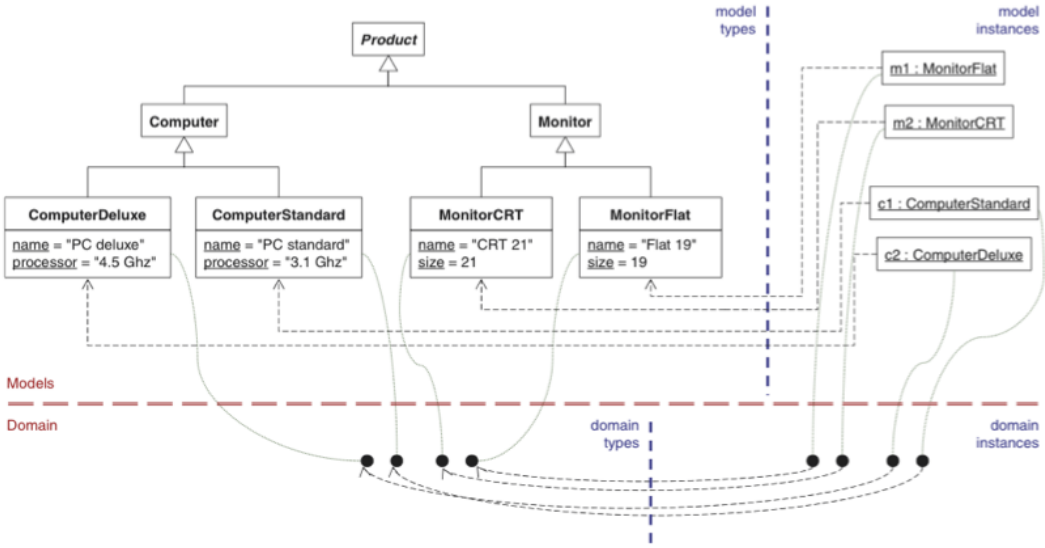

Figure 2.10: Types as classes.

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 chapter we will see whether it is possible to keep the simplicity of figure 2.10 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 (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. A possible solution involves the use of *powertypes*. Another solution introduces an object that can capture both *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 solves 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 the following: "Is it possible to keep the simplicity of figure 2.10 while still supporting 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 can be created and what features they possess. Therefore, we need (domain-)meta-types controlling the features of (domain-) types [9]. 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.

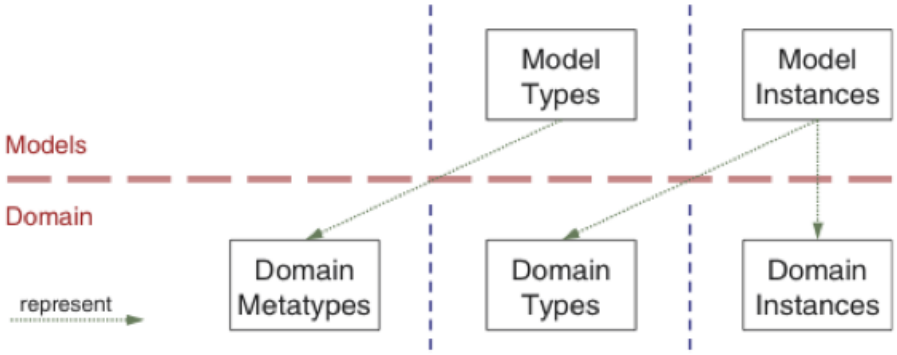

Figure 3.1: Mapping domain levels to modeling levels.

In the following section, 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, we need modeling concepts that can be applied in a uniform way across all levels in a multilevel hierarchy [9]. 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 the 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, which are instantiations of attributes found in a *higher* modeling level. In figure 3.2, the model instances do not contain any slots, because all attributes are already instantiated as slots at the model types level. 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 3.3. This direct mapping requires that the solution structure reflects the problem structure as closely 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 neighboring 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

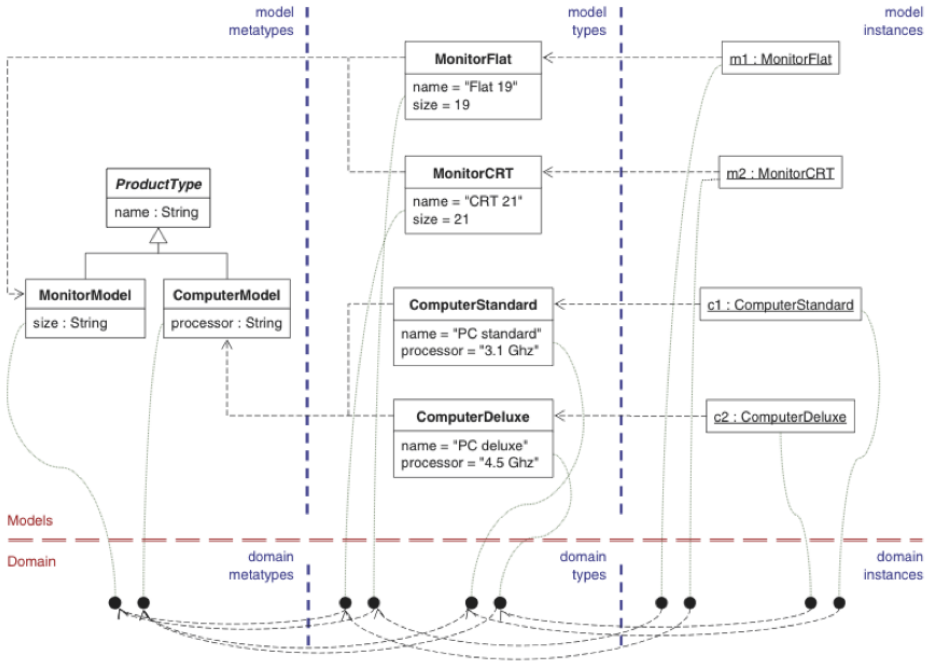

Figure 3.2: Example modeled using clabjects.

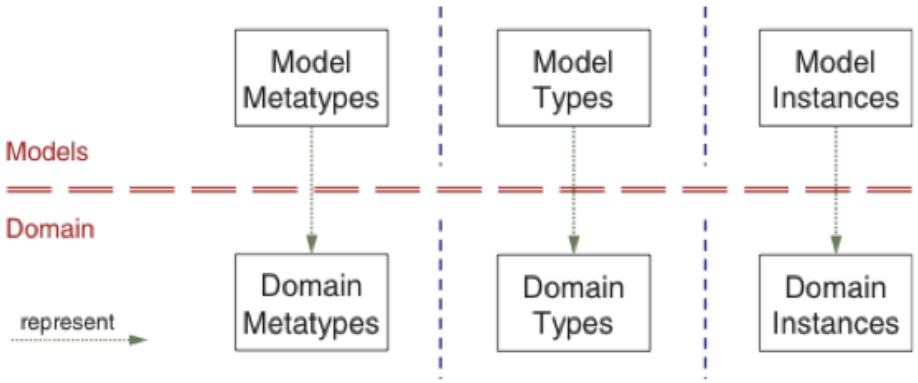

Figure 3.3: Direct mapping between ontological domain levels and modeling levels.

multiple classification levels to be modeled naturally. This is the issue of *deep characterization*. Deep characterization means that a type can influence the attributes of the types of its instances, as well as their object facets. Figure 3.4 shows an example of how Powertypes can be used to represent deep characterization. In this figure, we see two Powertypes, `ComputerModel` and `MonitorModel`. They respectively have slots `processor` and `size`.

Let's have a look at the class `Computer`. This class has a *power type* relationship

with its Powertype `ComputerModel`. In general, 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`). Because the Powertypes contain slots (`processor` or `size`), we should instantiate those slots in a subclass (e.g. `ComputerStandard`). The slot `price` is only instantiated at the model instances level.

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*.

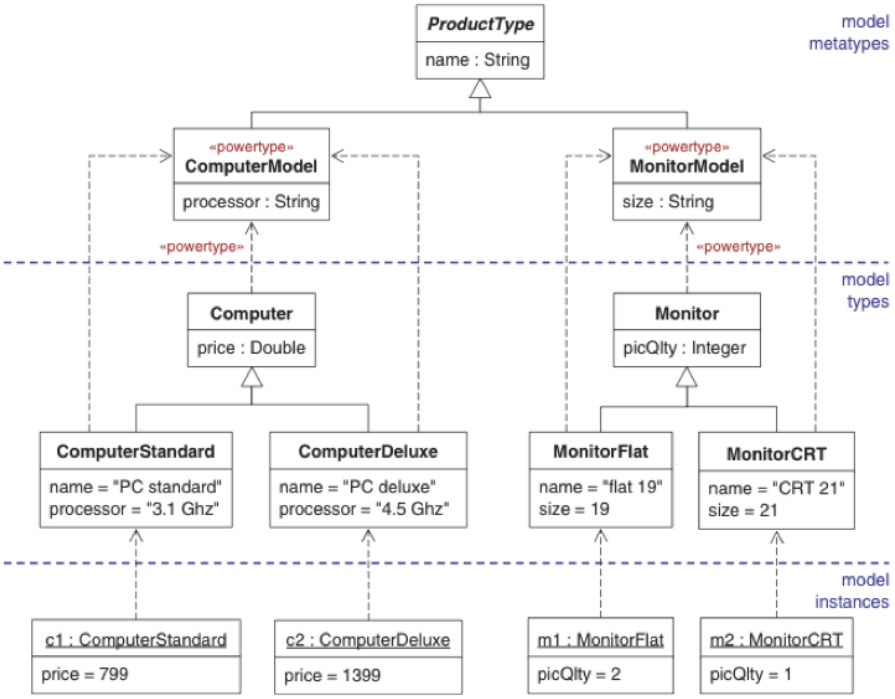

Figure 3.4: Deep characterization with powertypes.

3.1.3 Deep Instantiation

Deep characterization is needed 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.

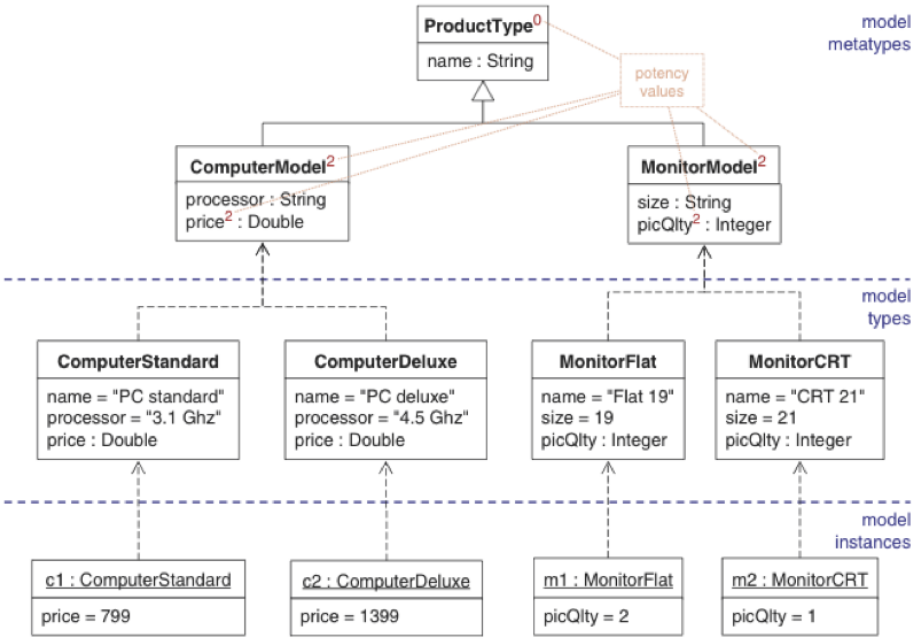

Figure 3.5: Deep characterization with potency.

Every clabject has a maximum instantiation depth associated with it. This instantiation depth is indicated by the superscript next to the name of the clabject. For example, the clabject `ComputerModel` has an instantiation depth of two, but also has a field `price` of potency two at the model metatypes level. This is indicated by the superscript *2* at the name of the field. After instantiating the clabject once, the potency of field `price` is reduced by one. Note that price acts like an attribute on this level (not visually indicated, but fields or clabjects without an explicit potency have a default value 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 introduces 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 doesn't 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 introduces the concept of potency in meta-models and allows a developer to work in two ontological (domain) instantiation modes: *strict* and *extensible*, as shown in figure 3.6.

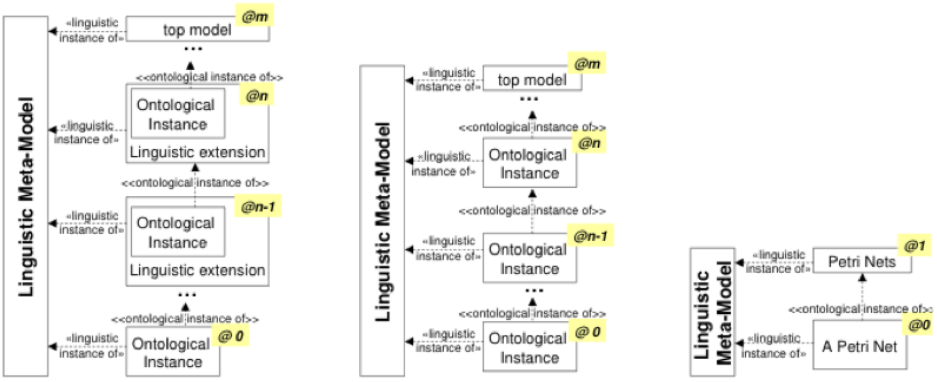

Figure 3.6: MetaDepth instantiation schemes: extensible ontological instantiation (left) and strict ontological instantiation (center). Example of strict instantiation (right).

In the extensible case, it is possible to linguistically extend each ontological instance model (as depicted in the figure, on the left side). This means that instances of elements marked as `ext` can be extended with new linguistic 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 ontologically. Note that such instances cannot be extended linguistically. This situation is depicted in the center of figure 3.6. For example, we could use this mode to describe the MOF meta-model at the highest level (and give it a potency of 2). This model could then be instantiated ontologically to describe meta-models for languages at potency 1. In turn, we can create instantiations (models) at potency level 0. Thus, the strict mode is similar to most meta-modeling environments, without restrictions on the number of meta-levels (i.e. deep meta-modeling). We will be using this *strict* case in the collaborative modeling framework.

On the right of figure 3.6, we see an example of the strict instantiation. Here we use the linguistic meta-model (see next section) to create a Petri Nets meta-model. The user then creates an ontological instance (i.e. a model) of the Petri Nets meta-model on the same linguistic meta-level.

3.2.1 The Linguistic Meta-Model

MetaDepth uses its own linguistic meta-model, inspired by MOF [12]. The framework is modified to accommodate an arbitrary number of meta-levels, using the concepts of deep instantiation and potency. Part of the linguistic meta-model is shown in figure 3.7.

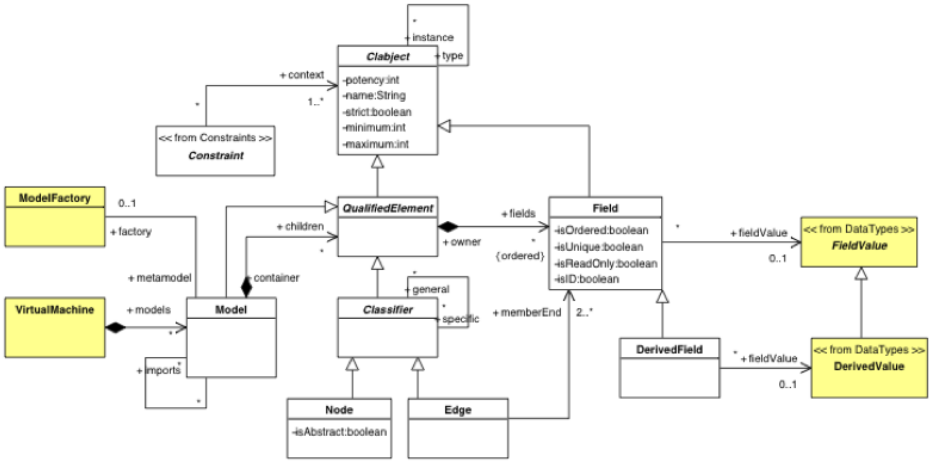

Figure 3.7: MetaDepths linguistic meta-model.

The root class `Clabject` takes responsibility of handling the dual type/object facet of elements [12]. 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. Notice that both fields `VAT` and `price` respectively have a potency of one and two in the `ProductType` meta-model. When instantiated one level lower, we fill in the `VAT` field, because we lowered its potency by one, which made it zero. The `price` field will subsequently be instantiated at the instance level.

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 [13]. The storage of models is also done in this format. As an example, figure 3.9 shows the three model dual classification

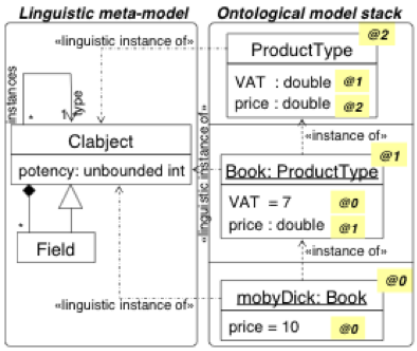

Figure 3.8: Dual classification example.

example of figure 3.8 in the textual notation. In this example, the model Store (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. The `price` attribute has a potency of two. This is not visually indicated, but attributes are automatically assigned the parent (that of Model `Store`) potency if no explicit value is given. The majority of the collaborative modeling framework will be written in this textual syntax.

```
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) [14], by communicating with the models through a connectivity layer. EOL can work with EMF models [15], 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) [16]).

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 3.11 extends our Store example.

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

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

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

A linguistic extension is interesting to permit unforeseen extensions to Domain Specific Languages spawning more than one level [17]. 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.

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

Figure 3.12: Linguistic extensions and associations in Metadepth.

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 new Node `Author`, which is an 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.

In figure 3.12, we also notice an association `writer`, that associates an author with a book. In Metadepth, associations can be annotated with fields by explicitly defining an `Edge` between their association ends. So in our example above, the association `writer` is 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.

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 integrated solutions. For example, DeepJava [18] 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 [19]. 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 [19] and [12].

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 the reuse and replacement of components. Components such as Activities, Tasks and Processes form the basics of an Android applications. An Activity typically represents one screen in an Android application, it is a "molecule". A task is a collection of Activities and a process represents a standard Linux process.
- **Dalvik virtual machine** optimized for mobile devices. It is the software that runs the apps on Android devices. Programs are commonly written in Java and compiled to bytecode. They are then converted from Java Virtual Machine-compatible .class files to Dalvik-compatible `.dex` (Dalvik Executable) files before installation on a device.
- **SQLite** for structured data storage. It is a powerful and lightweight relational database engine available to all applications.
- **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.

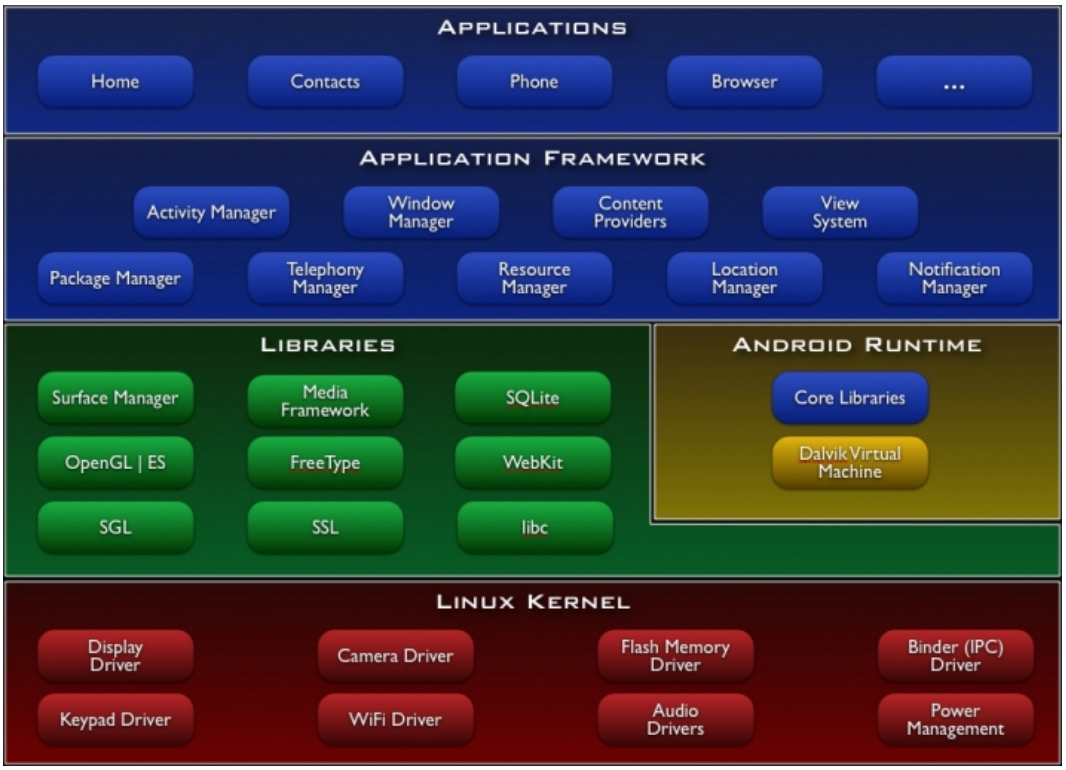

Figure 4.1: Android Architecture.

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 [20].

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 [21]. 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. For example, we could store a list of books in a `ContentProvider` if we need that list in multiple applications and need to access this resource in a RESTful way.
- `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 [22].
- `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 [23]. 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 of them can be found in the next section.
- A `Service` class supports long-running background tasks. They may be active, but not visible on the screen. For example, 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.
- 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.

4.1.3 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 [20]:

- **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 programming

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

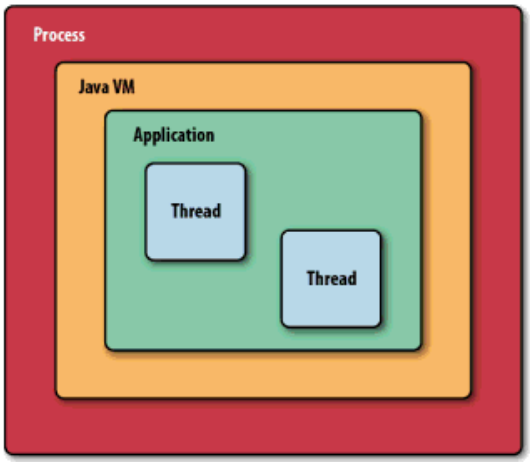

Figure 4.2: A Java application running in a JVM.

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 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.

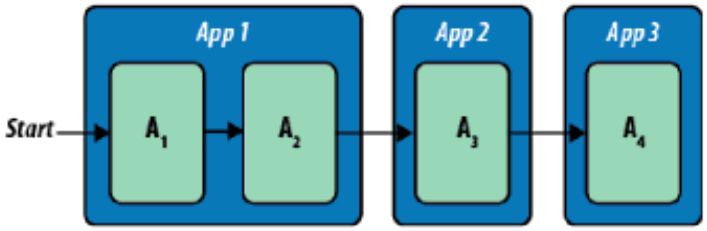

Figure 4.3: Activities in a single task spanning multiple applications.

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 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.

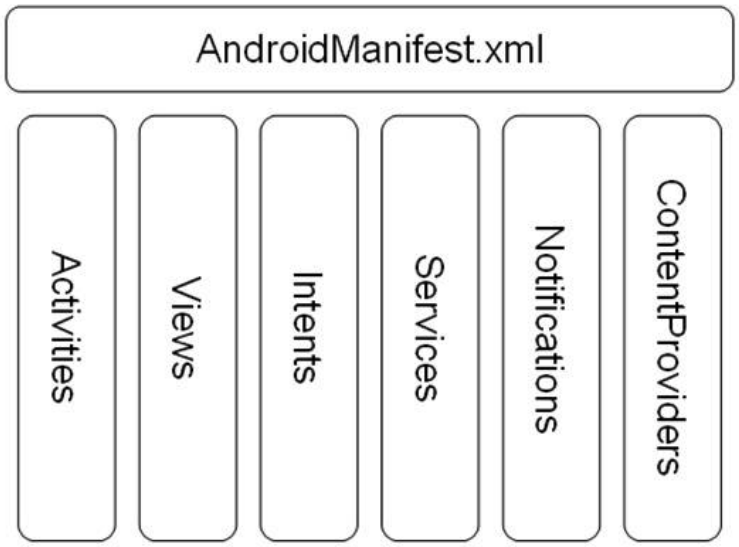

Figure 4.4: Structure of an AndroidManifest.xml.

4.2.5 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. This difference between the two states is that an Activity can still be running, but it is not active, whenever it is still visible on the screen, but partially obscured by another Activity [24].
- *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 *major* states an Activity can be in are the colored ovals:

- **Activity launched** corresponds to the `Active` or `Running` state.
- **Activity running** corresponds to the `Running` state.
- **Activity shut down** corresponds to the `Stopped` state.
- **App process killed** corresponds to the `Stopped` state.

The rectangles introduce the callbacks a developer can implement to perform operations when moving between states.

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.
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.

4.4 Android compilation

In this section we will take a look at the Android compilation process. In our collaborative modeling framework, it was not feasible to use the Android Development Tools Plugin for Eclipse, so I had to look for an alternative solution. Both the problem and the solution are discussed next.

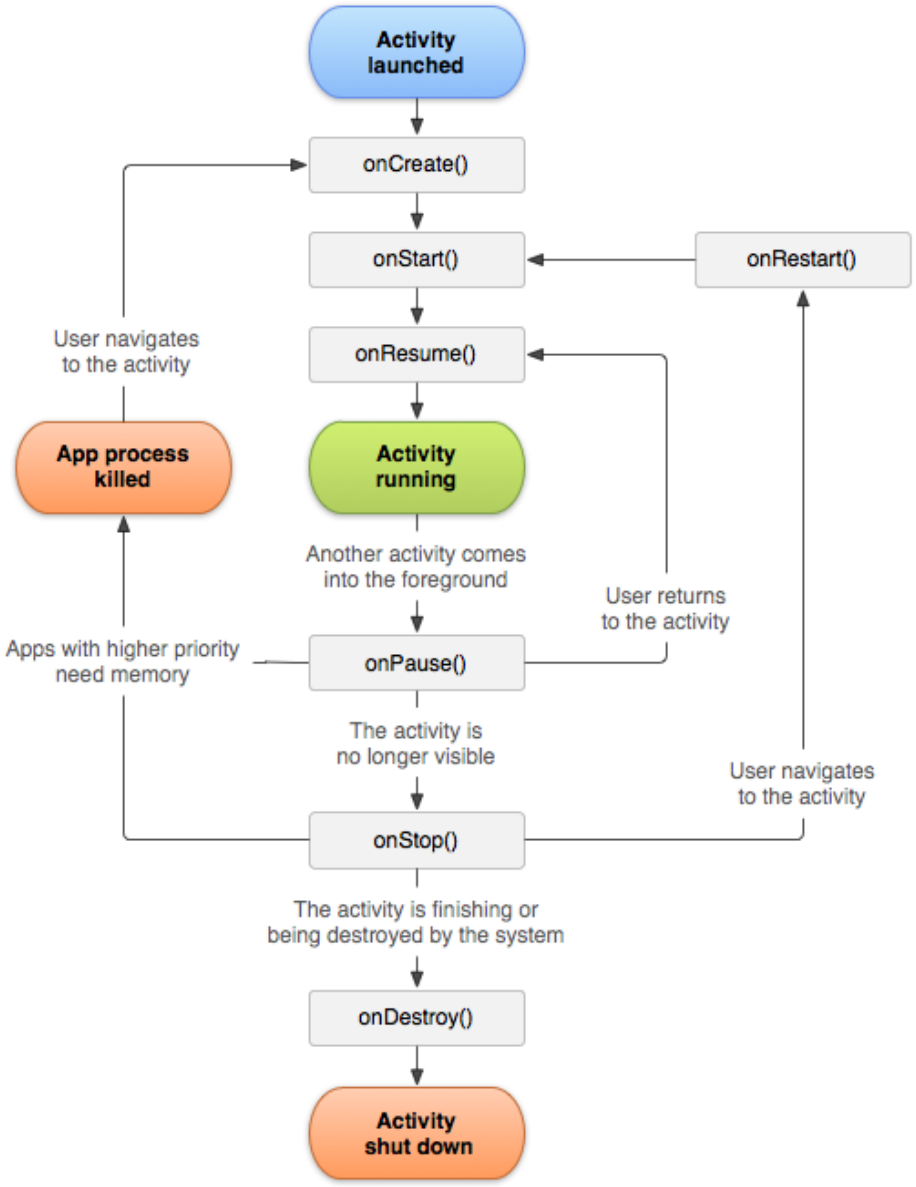

Figure 4.5: The lifecycle of an Android Activity.

4.4.1 Problem

Usually when a developer creates an Android application, the Android Development Tools (ADT) Plugin for Eclipse is used. It gives the developer a powerful and integrated environment in which to build Android applications. These tools extend the capabilities of Eclipse to set up new Android projects, rapidly create user interfaces, debug applications using the Android SDK tools and it also lets one create a binary `.apk` file of their Android project. After investigation of this solution, it

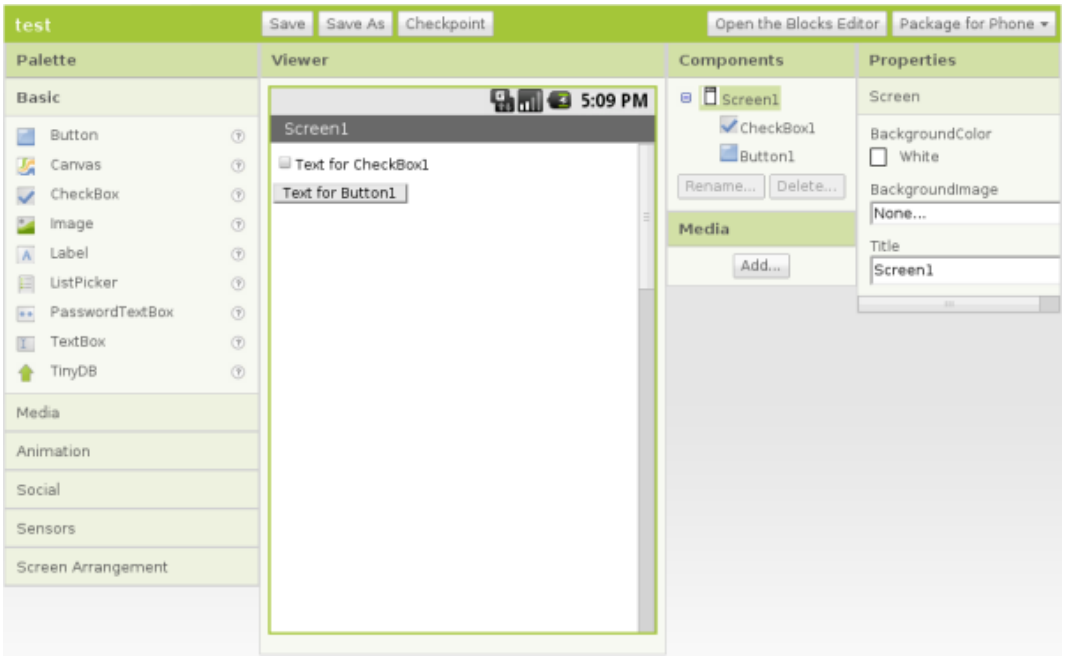

Figure 4.6: Google App Inventor.

proved unfeasible to apply this to our modeling framework. A modeler would have to generate the Java code, import it in Eclipse and ultimately generate the binary `.apk` file. This is a suboptimal solution.

4.4.2 Solution

The best solution is to create a custom build file that generates the code together with the resources and compiles it into a `.apk` file afterwards. This process is defined as follows:

1. Generate Android sources using the Metadepth model
2. Generate resource files and packaged Resources using the Android Asset Packaging Tool (aapt). This utility creates files such as `AndroidManifest.xml` and also produces `R.java`, which contains references to resources in the Java code (resources such as layout elements).
3. Compile the generated java source code + R.java
4. Convert compiled classes to Dalvik byte code. Any 3rd party libraries are also converted into `.dex` files so they can be packaged into the final `.apk` file.
5. Create an unsigned `.apk`. All non-compiled resources (such as images), compiled resources and the `.dex` files are sent to the apkbuilder tool to be packaged into a `.apk` file.
6. Sign the `.apk` with either a debug or release key

This process is also depicted in figure 4.7.

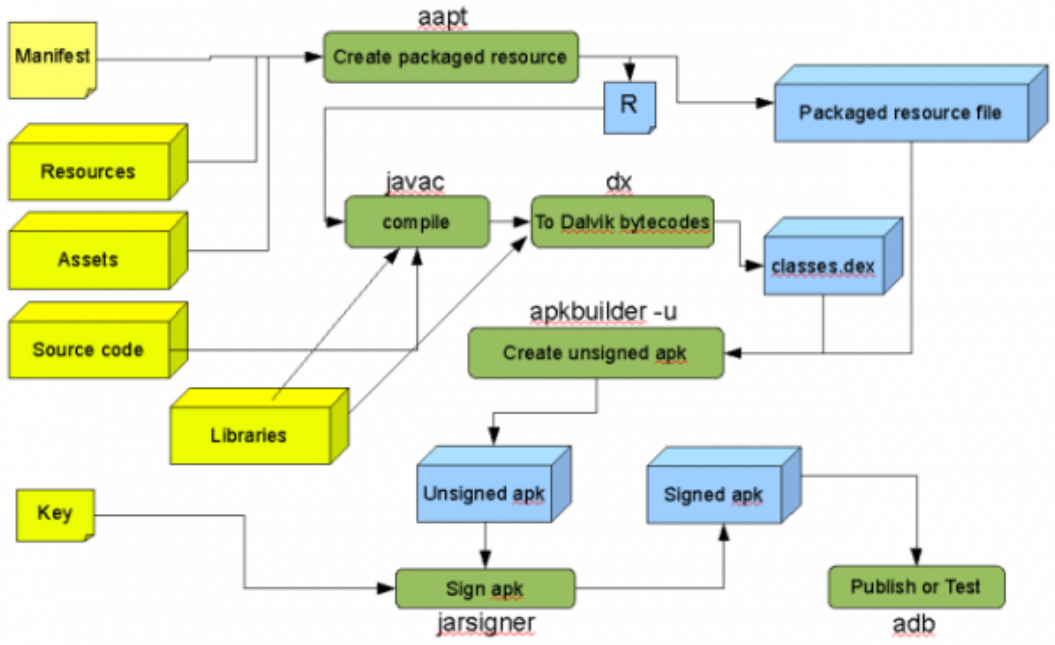

Figure 4.7: Android build process

CHAPTER 5

Collaboration

In this chapter, we will introduce different types of collaborative methods. People will continue to interact in new and different ways. One outcome of the marriage between technology and communication is a technical workplace. The study of such workplaces gives rise to a new field: *Computer-Supported Cooperative Work* (CSCW). CSCW analyzes the way groups work and seeks to discover how technology can help them work. Sometimes we refer to CSCW as *groupware*. Often this term is used synonymously with CSCW technology. However, this is not completely true. In the following sections, we will define the concept of groupware and CSCW applications. We will also create a taxonomy of groupware and compare this taxonomy to CSCW applications. There exists another collaborative method, especially targeted at collaborative learning, *Computer-Supported Collaborative Learning* (CSCL). This collaborative method is a refinement of CSCW. When studying these methods, we should make a clear distinction between cooperative and collaborative work. Cooperative work is accomplished by the division of labour among participants. It is an activity where each person is responsible for a portion of the problem that's being solved. Collaboration involves the mutual engagement of participants in a coordinated effort to solve the problem together [25].

The second part of this chapter defines the collaborative patterns typically used in a collaborative application (whether it is defined as groupware, CSCW or CSCL). We finish this chapter with a collaboration stack that defines the different levels in collaborative applications.

5.1 Groupware

In the following section, we will first give a definition for Groupware. Next, we will discuss different taxonomies of Groupware and concepts of real-time (distributed) Groupware, which is the emphasis of the collaborative modeling framework. Subsequently we will discuss the benefits and drawbacks of distributed interaction, which is a central concept in distributed Groupware. Finally, we will give a classification of Groupware which states the different concepts a Groupware application should support.

5.1.1 Definition

Several definitions for groupware exist. Some define groupware as collaborative software for small focused groups, that do not give organization-wide support. Another definitions says that groupware can be viewed as the class of applications arising from the merging of computers and large information bases and communications technology. These applications may or may not specifically support cooperation. Ellis et al define groupware as follows:

Definition 2 ***Groupware****. computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment. [26]*

In this definition, the writer uses the notions of *common task* and *shared environment*. This definition excludes multi-user environments where people do not share a common task. Moreover, the definition does not require users to be active at the same time. We can make a distinction between *real-time groupware* and *non-real-time groupware*. Our modeling framework tries to support both real-time as well as non-real-time groupware.

Furthermore, we can argue what distinguishes groupware from non-groupware. According to Koch, '*the core characteristic of groupware is the non-separation or non-isolation of users from each other. Groupware explicitly provides awareness of the co-workers and their activities and does not separate the users from each other as it is common in distributed systems in general.*' [27].

5.1.2 Communication, Collaboration and Coordination

A groupware application does not only support interactions between a user and the system. Its main strength is the interaction among users, moreover group interactions. As we focus on how to support this group interaction, we must address issues in three areas: communication, collaboration and coordination.

The main problem we currently stumble upon in collaborative applications is the separation of synchronous and asynchronous communication. For instance, asynchronous communication such as electronic mail and bulletin boards still exists separately from synchronous communication such as telephone or face-to-face conversations.

Similar to communication, a collaborative application should support collaboration. Collaboration demands that people share information. The last years, we saw a lot of improvement on this part because of the rise of social networks such as Facebook [28] and purely collaborative applications such as Google Docs [29] or Skype [30]. However, current information systems (such as a database system) seldom allow users to modify different parts of an object simultaneously. Usually, a user must check out this object after which it is locked for other users. Only then a user can manipulate this object. Afterwards, this user has to commit its changes again to release the lock on the object. Ideally, we need a shared environment where users are notified and updated of other's actions.

Finally, if we can effectively coordinate all the actions each user performs, the effectiveness of communication and collaboration can be enhanced significantly. Coordination is a requirement in a groupware application if we want to avoid conflicting or repetitive actions between users.

5.1.3 Taxonomies

If we look at groupware, we can define a time-space separation. In terms of space, groupware can be helpful to a face-to-face group or a group that is distributed over many locations. Furthermore, the communication and collaboration can be enhanced synchronously (in real-time) or asynchronously (in non-real-time). This separation suggests four categories of groupware, as depicted by the 2x2 matrix in table 5.1.

	Same Time	Different Times
Same Place	face-to-face interaction	asynchronous interaction
Different Places	synchronous distributed interaction	asynchronous distributed interaction

Table 5.1: Groupware 2x2 Time Space Matrix

For instance, meeting room technology would belong to the upper left cell, a real-time document editor within the lower left cell, a physical bulletin board within the upper right cell and an electronic mail system within the lower right cell. Our modeling framework allows the generation of applications that support both synchronous and asynchronous distributed interaction.

This taxonomy can be extended by the fact that our activities can be predictable or unpredictable [31]. The extended taxonomy is shown in figure 5.1. Activity can be carried out in a single place (top row), in several places known to the participants (middle row) or in numerous places, not all of which are known to the participants (bottom row). Each of these activities can then be carried out in real time (left column), at different times that are highly predictable (like sending a mail to a colleague and expecting it to be read within a day) or at different times that are

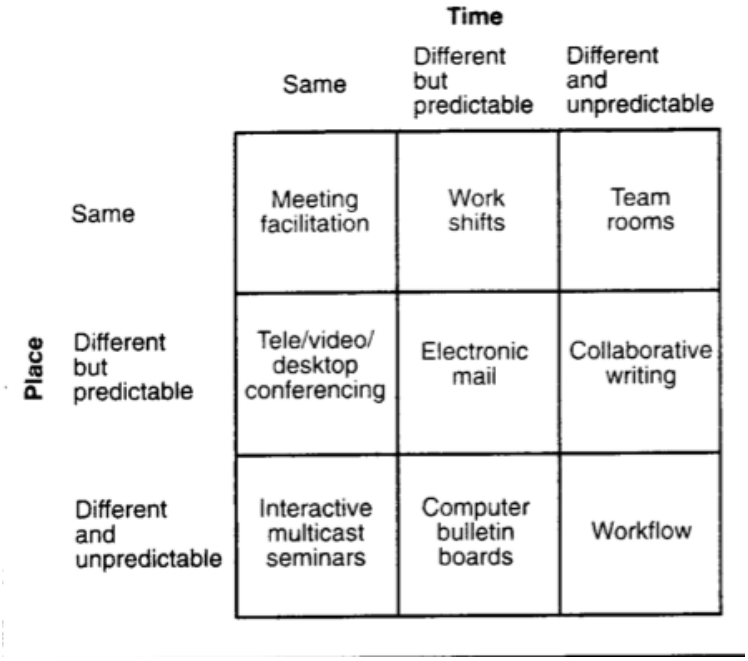

Figure 5.1: Groupware 3x3 Time Space Matrix

unpredictable (like collaborative writing).

5.1.4 Real-time Groupware Concepts

The concept of groupware is not new. It has been around since the early nineties, but it is still constantly evolving. In this section, we define some important terms for comparing groupware systems. These concepts will mainly be applicable on real-time groupware. In the remainder of this chapter we will also mainly focus on real-time applications, because they are mainly used for easy and effective communication and collaboration.

- *Shared context.* A shared context is a set of objects where the objects and the actions performed on the objects are visible to a set of users. For example, a group of users can make notes on files shared in a Dropbox shared folder or class notes within electronic classrooms.

- *View.* A view is a representation of some portion of a shared context. Different views might display the same information in different ways or they can use the same presentation but refer to different parts of the shared context. For example, we can show a dropbox folder as a list of filenames or as a group of images showing an in-file view.

- *Synchronous and asynchronous interaction.* Synchronous interaction happens when people interact in real-time and asynchronous action happens when people interact over an extended period of time. Usually, a groupware application

only supports one of the two interaction types. Our modeling framework allows the creation of both interaction types at the same time.

- *Session.* A session usually represents a period of synchronous interaction supported by a groupware system. When a user logs in into a groupware system, he starts his session and can start interacting with other users that currently own a session.

- *Role.* A role is a set of privileges or responsibilities related to a user. For instance, a user can be assigned the role of admin to control a groupware application.

5.1.5 Benefits and drawbacks of distributed interaction

If we look at the taxonomy of groupware systems and only consider systems at different places, we get a distributed interaction model. From a user's perspective, these distributed interaction sessions are completely different experiences from face-to-face (i.e. same place) sessions. Because our modeling framework primarily targets interaction at different places, we list the benefits and drawbacks of these distributed interaction types here:

- *Encourages parallel work within the group.* Usually, people divide into subgroups to work on different parts of an assignment. It is easy for distributed members to drop out for a while, do something else and then return. In most face-to-face situations, this is not socially acceptable.

- *Increases information access.* Apart from sources on the internet, participants in distributed sessions have access to local books and files.

- *Makes discussion more difficult.* In comparison to face-to-face sessions, it is a lot harder to have a discussion or host a panel over a distributed session. Usually, people tend to take turns and are unusually polite.

- *Cuts down on social interaction.* This item has both a positive and a negative connotation. Distributed sessions are usually more serious. As a consequence, there is less interchange about nontask-related topics and people tend to focus on the task immediately. The effect is a possible efficiency gain from time saved and a possible loss from social needs.

- *Can be efficient.* Distributed sessions allow for parallel work. In contrast to face-to-face sessions, people in a distributed session are usually more concentrated on the to be accomplished task itself.

5.1.6 A groupware classification

Groupware is about crewing computer-based systems that support groups of people involved in a common task. This is a very open definition and in practice it is often hard to decide what kind of functionality a particular group needs. In general, we can distinguish five application classes:

- **Awareness support** - This class is one of the core concepts in groupware. Groupware tries to connect people and allows to coordinate each other. Almost every groupware application has some sort of Awareness support.
- **Communication support** - In order for people to be connecting with each other, we need some sort of communication in the groupware system. This sort of communication can be of a synchronous or asynchronous nature. Asynchronous communication examples are email or forums. Synchronous communication examples are chat or video.
- **Coordination support** - There is already some contribution to coordination through awareness, but in a very general way. We might need explicit coordination support in the form of workflow solutions.
- **Team support** - Not every person in a group of people will have the same needs and abilities within a groupware application. There is a need for special group types and their special needs. We might need to define roles that allow people to do different things within the application.
- **Community support** - Teams and communities are a different thing. To begin with, they have a different structure and probably completely different needs. A team is a group of people usually working on a bigger cause or goal. A community often does not consist of equals working together on the same things. There are many different functions in a community that make the groupware application complex.

When we are faced with a particular situation, it is quite straightforward to identify one or two classes that cover the requirements in this situation. From there on, we can select the appropriate tools to start building the groupware application.

5.2 Computer-Supported Cooperative Work

Like groupware, CSCW is concerned with understanding social interaction and the design, development and evaluation of technical systems supporting social interaction in teams and communities. It researches the use of computer-based technology for supporting collaboration. In this section, we will define CSCW exactly, look at a few design challenges and study the technology behind CSCW.

5.2.1 Definition

Many researchers in CSCW have their own definition of what CSCW exactly is. Bowers and Benford have the most general sight: '*In its most general form, CSCW examines the possibilities and effects of technological support for humans involved in collaborative group communication and work processes*' [32]. Greif defines CSCW as '*computer-assisted coordinated activity such as communication and problem solving*

carried out by a group of collaborating individuals.' [33]. Wilson on his turn defines CSCW as '*a generic term which combines the understanding of the way people work in groups with the enabling technologies of computer networking, and associated hardware, software, services and techniques*' [34]. In general, we can make a distinction between the social and technological definition of CSCW. The following CSCW definitions are the most appropriate definitions for what we have in mind.

Definition 3 ***CSCW: Social Definition***. *CSCW should be conceived as an endeavor to understand the nature and characteristics of cooperative work with the objective of designing adequate computer-based technologies. [35]*

Definition 4 ***CSCW: Technological Defintion***. *computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment. [26]*

Note that the technological definition of CSCW is the same as that one of Groupware. The exact difference between CSCW and Groupware will be explained in the next subsection.

5.2.2 CSCW is Groupware

CSCW is an interdisciplinary field where researchers from various fields contribute with '*different perspectives and methodologies for acquiring knowledge of group work and for suggesting how the group's work could be supported*' [26]. For instance, '*computer scientists might bring in their technical knowledge and social scientists contribute their sociological and anthropologistic knowledge for questions of design*' [36].

How does all of this relate to Groupware? In general, groupware is the term coined for the technical system resulting from CSCW research and development. Groupware is the technical part of the CSCW system. Moreover, many software systems that have collaborative functionality will be groupware to some extent. However, CSCW as a research field will persist, because it addresses larger questions about the design and refinement of groupware [33].

5.2.3 Challenges in CSCW design

CSCW is not a typical software design activity. Several authors in the field have been analyzing CSCW projects and have been identifying core challenges of collaborative system design compared to software design in general [27].

- It is hard to capture the requirements for a collaborative system, because the requirements are not clearly known to all participants or they change over time.
- All of the potential users have to use the CSCW system actively in order for the system to be a success.

In the next subsection, we will address the requirements engineering of groupware in more detail. The other challenge, the adoption of a collaborative system, is largely due to a network effect and cannot be easily manipulated.

5.2.4 Requirements engineering in CSCW

When a CSCW project was developed in the future, more often than not it was not able to satisfy its intended goals. The main reason for this problem is that most of these systems have been regarded as a technical system only. However, as we have seen there are two aspects of CSCW, a social and technological aspect. It turned out that very often the social aspect was not satisfied in the CSCW system and it was not seen as a socio-technical system. The fact that the success of such a system depends on the social group that is using it is overlooked. This fact results in several problems:

- It is hard to get the requirements from users. The social group using the end-product might change and thereby changing the requirements.
- When designing a CSCW system, there should be both a technical and a social design.
- It might be hard to get the potential users to accept the resulting CSCW system.

The first problem of capturing the requirements from users in an initial development phase is a known problem in software engineering. To address this issue, iterative development has been introduced. If we build a prototype that engages users and produces some feedback, we might be able to create a much better product even from one iteration more.

If we look at typical collaborative applications today, we notice the trend that functionalities are offered as separate applications. In order to make collaboration a true success in an enterprise, there is a need to integrate the different functionalities and to adapt applications to the need of the individual and the group. There is a need for an all-in-one collaborative applications. This is one of the issues our modeling framework tries to tackle.

5.3 Computer-Supported Collaborative Learning

Computer-supported collaborative learning (CSCL) has some interfaces with CSCW and Groupware, but is fundamentally different in a few aspects. The focus in CSCL is on learning, as the name suggests. It is concerned with studying how people can learn together with the help of computers [37]. In this chapter, we will see several definitions for collaboration in the context of learning. It will become clear that the interplay of learning with technology turns out to be quite intricate.

5.3.1 Education and CSCL

Computers have become an important artifact in formal education (e.g. high school or college) as well as informal education (e.g. museums). In the broader learning sciences, there also exists a trend in learning together in small groups. However, the ability to enhance learning through technology and education remains a challenge that CSCL is designed to address.

CSCL proposes the development of new software that bring learners together and that can offer creative activities of intellectual exploration and social interaction. It is often conflated with e-learning, the organization of instruction across computer networks. E-learning is often associated with the creation and delivery of digital content such as slides, text or videos. There are a few problems with this view:

- Digital content such as slides, text or videos does not necessarily make for compelling instruction. It may provide important resources for students, but there must exist a larger interactive context.
- Online teaching requires as much effort by human teachers as classroom teaching.
- CSCL stresses collaboration among the students, so that they are not simply reacting in isolation to posted materials.
- CSCL is concerned with Face-to-Face collaboration, either synchronously or asynchronously.

5.3.2 Definition

Even more than in groupware, a distinction between cooperation and collaboration is conceptually central in this review of CSCL. Cooperative work is accomplished by the division of labour among participants. It is an activity where each person is responsible for a portion of the problem solving. Collaboration involves the mutual engagement of participants in a coordinated effort to solve the problem together.

Before CSCL became a research topic, there had been a lot of research conducted on group learning. To distinguish CSCL from group learning, we can draw a distinction between *cooperative* and *collaborative* learning:

Definition 5 *In cooperation, partners split the work, solve sub-tasks individually and then assemble the partial results into the final output. In collaboration, partners do the work 'together' [38].*

In cooperation, the learning is done by individuals who contribute their results and present these as their group product. Cooperative learning in groups is an activity that takes place individually. In contrast, collaborative learning occurs socially as the collaborative construction of knowledge. The activities that individuals engage in are not individual-learning activities, but group interactions like negotiation

and sharing.

Finally, a formal, programmatic definition of CSCL was presented by Koschmann [39]:

Definition 6 *CSCL is a field of study centrally concerned with meaning and the practices of meaning-making the context of joint activity, and the ways in which these practices are mediated through designed artifacts.*

"Practices of meaning-making in the context of joint activity" might be hard to understand. It means that learning is not merely accomplished through interaction, but it is *constituted* of interactions between participants.

5.4 Collaborative Patterns

In the previous sections we have discussed the different kinds of collaborative applications. In general, we may conclude that a collaborative application provides a group of users with the facility to communicate and share data in a coordinate way. In this section, we will propose several design patterns to help in the design and development of collaborative applications. These patterns cover the basic aspects such as data sharing and communication.

Every pattern proposed will describe a *context* within which we can use this pattern, the *problem* the pattern solves and the *solution*. We will describe the *forces*, *implementation*, *examples* and *related patterns* as well. All these individual patterns can be interwoven into a *patterns system* that describes how these patterns are connected, how they complement one another and how software development with patterns is supported [40]. If we combine the following patterns, we arrive at a patterns system. I implemented the following patterns that form the patterns system as closely as possible in the modeling framework.

5.4.1 Session

In all collaborative applications, we have users working together in a session. A session can be both synchronous (at the same time) or asynchronous (at different times). Therefore, a session is usually tracked and we can identify who is currently connected in this session.

Problem
How do we manage and coordinate a session in a collaborative application?

Requirements

- Users work in groups
- We need to know which users are connected and working synchronously

- Data is protected from users outside of the session
- It is possible to register new users
- It is possible to unregister new users
- We need to check a user's credentials when creating a session

Solution
Register a list of all users belonging to the session and show a list of users currently working and connected to the session.

Examples
The collaborative modeling framework allows a developer to model a session that persists users and keeps track of currently connected users.

5.4.2 Repository

In all collaborative applications we need to store and share data.

Problem
How can we provide tools to users currently connected in the session that allow them to get recourses that have been created? How can users share their own resources?

Requirements

- Ability to store data
- Possibility to share data
- Edit or remove data that belongs to the repository

Solution
Provide a repository that can be managed and controls access to users.

Examples
In the collaborative modeling framework, Dropbox is used as a repository. Users may upload new resources to Dropbox and a developer may model a view that shows the Dropbox repository.

5.4.3 Object

Every object (e.g. chat message, dropbox resource, generic list) that is created should be saved in order to provide data awareness. We typically save the meta-data related to the object, such as the user, the object type and the time.

Problem
Data is generated by the members of a workgroup. How can we manage this data?

Requirements

- Users can create new objects/resources

Solution
Provide a meta-data information object for each data object. We typically store an object's creation data and time, the user and the object type.

Examples
In the collaborative modeling framework, a developer can model and include a MongoDB [41] database to persist all in-app objects and their meta-data.

5.4.4 View

In a collaborative application, it is necessary to have a graphical view of the repository's data. Various groups of users may have various views.

Problem
How can we provide a view of the data in the repository?

Requirements

- All users should have access to the data they have created
- Some data needs to be hidden from certain users

Solution
Users are able to view the data in the repository, optionally in different views.

Examples
In the collaborative modeling framework, it is currently only possible to view the repository's resources in a list view. In the future, multiple views might be supported. The repository might show both the resource itself as well as its meta-data.

5.4.5 Broadcast

A user typically wants to send or share objects with other users in a session. Depending on the object type, a user may be able to send the object to all connected users, to all users or to a group of users.

Problem
How can we send objects to other users?

Requirements

- Users can send an object to a single connected user
- Users can send an object to all connected users
- Users can send an object to all users (connected or disconnected)

Solution
We need a broadcast technique to send objects to other users. This is a technical detail discussed in the next chapter.

Examples
In the collaborative modeling framework, it is possible to model a chatroom and allow users to send messages to each other. Messages are persisted within MongoDB, so other users don't have to be connected at the time of sending the message. Currently it is not possible to send arbitrary binary data, but only text.

5.4.6 User

A user is an object that identifies a person in a group. Certain personal information about each user has to be saved.

Problem
How can we manage information about users?

Requirements

- User awareness is needed in applications, so we need to keep track of certain information about each user

Solution
A user object contains all necessary personal information for a person to initiate a session and participate in the application.

Examples
In the collaborative modeling framework, in order to participate in a session, a developer can require every user to log in with a username and a password. The developer can model these accounts and restrict public access. In order to provide user awareness, every application also display all users currently connected.

5.4.7 Role

In a collaborative application, we may have different access rights for groups of users.

Problem
How can we manage the access rights of users? Not all users will have the same permissions within a collaborative application.

Requirements

- Collaborative applications have different types of users
- Roles are used to restrict data access
- Some views can be restricted by roles

Solution
We assign roles to each user in order to define their rights and permissions.

Examples
In the collaborative modeling framework, we can restrict access to certain objects with different user roles. For instance, we can model a list that asks questions to a user, but only a user with the admin role can see the answers.

5.4.8 Environment

A collaborative application may contain multiple sessions, with users working on different repositories. This means that some groups might not interfere with others, in which way we can re-use an application.

Problem
How can we provide access to collaborative applications with users identified among corresponding sessions? How can we control the 'environment'?

Requirements

- A session member's work should not interfere with another session member
- All users among the sessions should have a uniform way to enter a collaborative application

Solution
The *Environment* object has information on all sessions of users connected to the collaborative application.

Examples
In the collaborative modeling framework, a developer may model a server that maintains an environment. This *server/environment* object pertains all sessions and handles calls between client and server.

5.4.9 Pattern System for Collaborative Applications Design

All these previous patterns together define a pattern system for the design of collaborative applications. The overall system design through these patterns is depicted in figure 5.2.

The environment object keeps track of all sessions in a collaborative application. A session can broadcast objects and is persisted through a user with a specific role. Furthermore, we have a Repository that is accessed through the session object and shown through a View.

One more thing, the patterns covered in the pattern system are so-called low-level patterns. It is possible to combine those low-level patterns to form higher-level patterns. For example, if we combine the Role, User and Session pattern, we can come

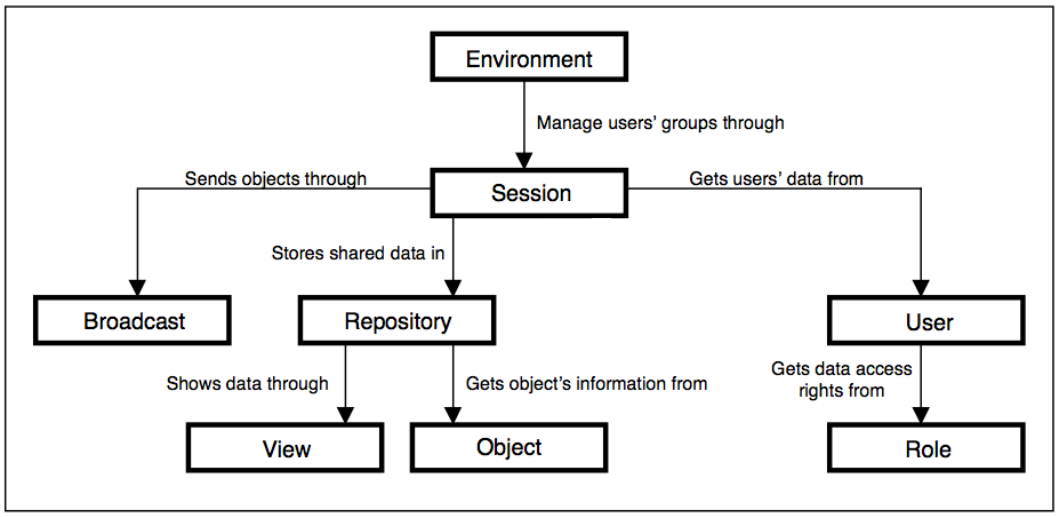

Figure 5.2: Pattern system of a collaborative system

up with a full-fledged authentication/authorization pattern. Some of these patterns also feature cross-cutting concerns [42]. Cross-cutting concerns are parts of a program which rely on or must affect many other parts of the system. For instance, the Role pattern depends on the View pattern, because a user's role affects what can be seen in the view.

5.5 Conclusion

In this chapter, we have described the concepts of Groupware, CSCL and CSCW. Afterwards, we discussed several collaborative patterns that help us in the design and development of collaborative applications. These patterns are important in providing collaboration support in the modeling framework created in this work. To conclude, our framework tackles the following problems using the proposed pattern system:

- Without a pattern system there is no defined structure for solving problems. The use of a pattern system allows for a structured solution to solve recurring collaborative problems, such as session creation, object persistence or broadcasting of data.
- Having a collaborative pattern system, it makes it more obvious how to implement the meta-models and code that support collaboration.

Using this pattern system in the modeling framework, we allow the rapid validation of an idea or prototype that involves collaboration. Moreover, the pattern system incorporated into the framework gives us the ability to create an all-in-one integrated collaborative solution. An example of such an application will be shown in chapter 7.

Now that we have a clear idea of our pattern system, we can describe the design of our collaborative framework. Important components and their design will be described in the next chapter.

CHAPTER 6

Designing a Collaborative Modeling Framework

In this chapter, the collaborative modeling framework will be explained in depth. The target platform for applications modeled through the framework is Android. Once a modeler has a basic understanding of how Android development works, the modeler can use this framework to rapidly develop Android collaborative prototype applications. In the first section, we will briefly state the requirements for the collaborative modeling framework. Next, several meta-models that make up an Android application are discussed. In the third section we describe the server component, written in Node.js [43] and Javascript. The server component is used to persist data created by a user and to distributed this data to other users currently involved in a session. Finally, we end the chapter with a section on code generation and EGL [44].

6.1 Framework requirements

todo

6.2 Main component Meta-models

In this section, we will discuss the main component meta-models featured in the collaborative modeling framework. First, a high-level overview in the form of a megamodel will be given. Next, individual meta-models will be explained in more detail.

6.2.1 High-level megamodel

In figure 6.1, we see a high-level overview megamodel. This megamodel contains all the important meta-models that make up an application model in the framework. There is one important piece missing in the megamodel, which is the `Component` hierarchy. This hierarchy will be reviewed in section 6.3. The meta-models `Application`, `Manifest`, `Activity`, `AndroidAction`, `UIAction` and `Server` will be explained in detail in this chapter.

6.2.2 Application

The top-level meta-model that represents an Android application is the `Application` meta-model. `Application` has a `name` field and contains an `AndroidManifest` and a set of `Activities`. If we want to communicate with a server, we also need to specify an instance of the `Server` meta-model. The complete `Application` meta-model is described as follows:

Listing 6.1: Application meta-model

```
1  load "Manifest"
2  load "Activity"
3  load "Server"
4
5  Model Application imports Activity, Manifest, Server {
6      name             : String;
7      manifest         : ManifestDescription[1];
8      activities       : Activity[*];
9      server           : Server[0..1];
10 }
```

6.2.3 Manifest

One field in the `Application` meta-model is the `AndroidManifest`. The *AndroidManifest.xml*, as described in chapter 4, contains all activities and other Android components. The equivalent meta-model is declared as follows:

Listing 6.2: Manifest meta-model

```
1  Model Manifest {
2
3      Node ManifestDescription {
4          namespace        : String="http://schemas.android.com/
               apk/res/android";
5          package          : String;
6          versionCode      : String="1";
7          versionName      : String="1.0";
8          sdk              : String;
9          app_info         : AppInfo[1];
10         permissions      : Permission[*];
11     }
12
```

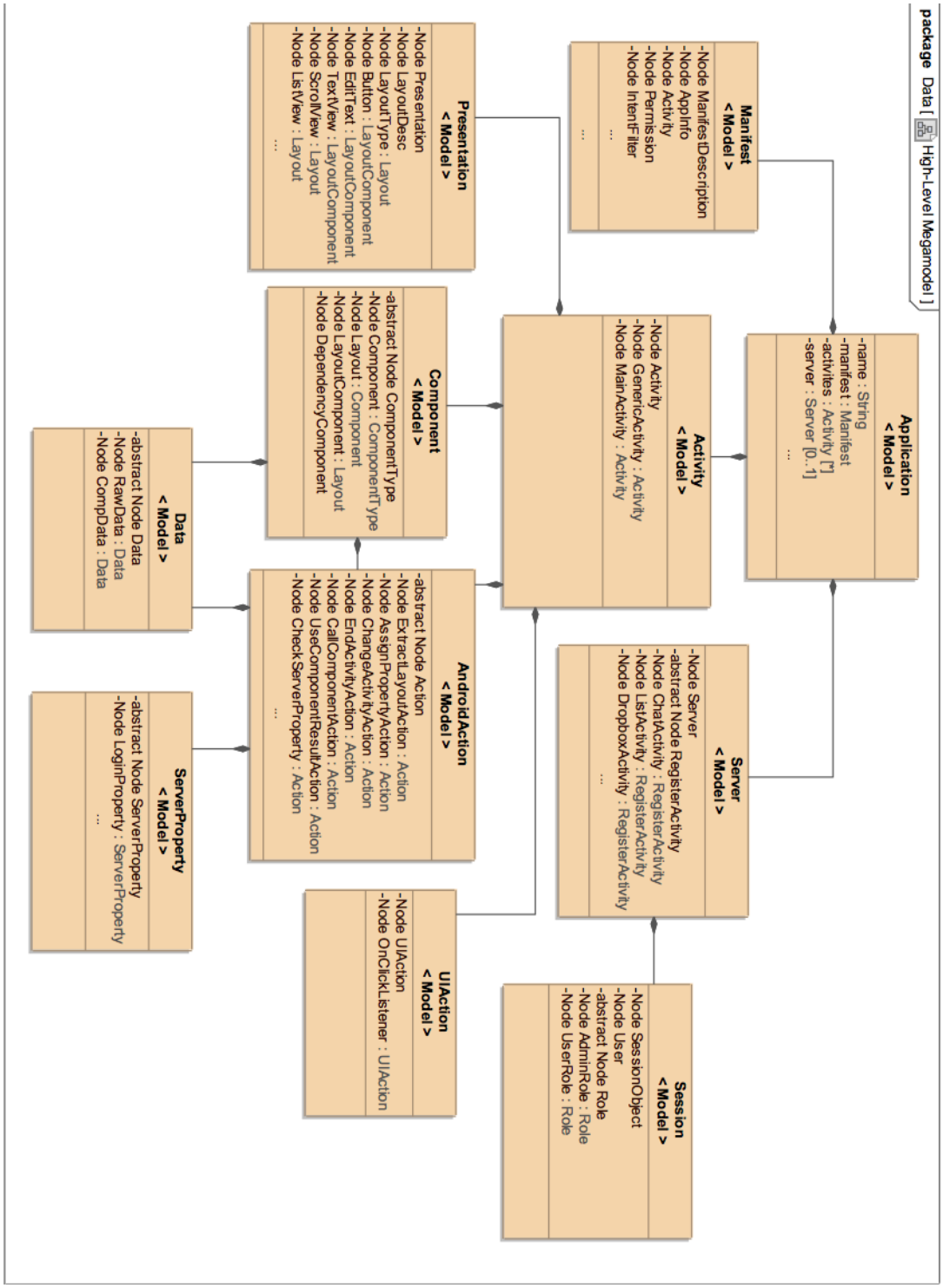

Figure 6.1: High-level Megamodel

```
13      Node AppInfo {
14          icon         : String="@drawable/icon";
15          label        : String="@string/app_name";
16
17          activities   : Activity[*];
18      }
19
20      // e.g. <uses-permission android:name="android.permission
            .SEND_SMS"/>
21      Node Permission {
22          name     : String;
23      }
24
25      Node Activity {
26          name     : String;
27          label    : String="@string/app_name";
28          intent   : IntentFilter;
29      }
30
31      // e.g. http://developer.android.com/guide/topics/intents
            /intents-filters.html
32      Node IntentFilter {
33          action       : String="android.intent.action.MAIN";
34          category     : String="android.intent.category.
                LAUNCHER";
35      }
36  }
```

Some of the fields have a pre-defined value, such as `namespace`, `icon` or `label`. Other fields need to be set explicitly by the modeler. The `package` field describes the Java package used to logically organize the source code. Other fields describe the current SDK version of Android and the Activities that correspond with the ones that are modeled in the `Activity` meta-model. The `Permission` Node models the permissions that the modeled application needs. For instance, if the application needs to access the internet, we need to add an INTERNET permission. Every Android application can have a variety of permissions set to use the functionality provided by the SDK [45]. An `IntentFilter` is associated to an Activity, we indicate that this Activity is the entry point of the application. In Android, other options can be specified through intent filters [46], but these are not supported in the collaborative modeling framework.

6.2.4 Activity

The most important part of the collaborative framework is the `Activity` meta-model. It encapsulates all Android components, the layout and all actions on (layout) components. The `Activity` meta-model is designed as follows:

Listing 6.3: Activity meta-model

```
1  load "Presentation"
```

```
2  load "Component"
3  load "AndroidAction"
4  load "UIAction"
5
6  Model Activity imports Presentation, Component, AndroidAction
       , UIAction {
7
8      Node Activity {
9          name                : String;
10         // Is this the main activity that is launched when
               launching the application?
11         main                : boolean;
12
13         content             : Component@0[*];
14         presentation        : Presentation[1];
15         onClickListeners    : UIAction[*];
16     }
17
18 }
```

As we can see, an Activity contains an arbitrary number of components (explained in next section), a presentation model and a set of UI actions. Both the `Presentation` and `Action`/`UIAction` meta-models will be explained in next subsections.

6.2.5 Presentation

In this section, we will review the `Presentation` meta-model. This meta-model represents a mapping from the Metadepth syntax to the XML representation in Android. The `Presentation` meta-model is described in Listing 6.4

Listing 6.4: Presentation meta-model

```
1  load "Component"
2
3  Model Presentation imports Component {
4
5      Node Presentation {
6          activityname    : String;
7          layout          : LayoutDesc[1];
8      }
9
10     Node LayoutDesc {
11         name            : String;
12         layoutType      : LayoutType[1];
13     }
14
15     // e.g. LinearLayout
16     Node LayoutType : Layout {
17         name        : String;
18         namespace   : String="http://schemas.android.com/apk/
               res/android";
19         orientation : String;
```

```
20          width       : String="fill_parent";
21          height      : String="fill_parent";
22
23          children    : Layout@0[*];
24      }
25
26      Node Button : LayoutComponent { }
27
28      Node EditText : LayoutComponent {
29          password        : String;
30          requestFocus    : boolean;
31      }
32
33      Node TextView : LayoutComponent { }
34
35      Node ScrollView : Layout {
36          width       : String;
37          height      : String;
38          weight      : String;
39          components  : Layout@0[*];
40      }
41
42      Node ListView : Layout {
43          width       : String;
44          height      : String;
45          weight      : String;
46      }
47  }
```

The relevant part of the meta-model is defined through the `LayoutType` node. This node describes the type of layout (e.g. LinearLayout, a layout that arranges its children in a single row or column [47]) to use in the `Activity`. Apart from the type, it also contains a description of the elements contained in the layout itself. We usually want to create a `LinearLayout` type, but other types are possible too [48]. Examples of layout elements are `Button` or a child view that has its own layout elements, for example a `ScrollView`.

6.2.6 Action

The `Action` meta-model is an important part in the functionality of the collaborative modeling framework. It provides a skeleton for executing actions on Android components or elements in a layout. Another example of an action would be the exchange of data between activities. A part of the `Action` meta-model is displayed in listing 6.5.

Listing 6.5: AndroidAction meta-model

```
1  load "Component"
2  load "ServerProperty"
3  load "Data"
4
```

```
5  Model AndroidAction imports Component, ServerProperty, Data {
6
7      abstract Node Action {
8          ctype           : String{id};
9          condition       : Action;
10     }
11
12     // e.g. use the value of a text field to call the action
           method of a component
13     Node ExtractLayoutAction : Action {
14         source          : Component@0;
15         name            : String;
16     }
17
18     // Specify the target activity
19     Node ChangeActivityAction : Action {
20         oldActivity     : String;
21         newActivity     : String;
22         data            : Data[*];
23     }
24
25     // Call the action method of a component.
26     // Might save a value if it's a sensor (i.e. geo)
27     // Or execute a real action if it's an actuator (i.e.
           send an SMS)
28     Node CallComponentAction : Action {
29         // properties needed to call the action method of the
               component
30         properties      : Data[*];
31     }
32 }
```

The actions listed above are the three most used and important actions. The `ExtractLayoutAction` takes as arguments a source layout component and a name for the data that has to be extracted. This data will be extracted from the source component that is specified. Usually this is an Android `TextView` or `EditText` component. The `ChangeActivityAction` will take two activities and an arbitrary data structure as input. The first Activity specified should be the current Activity and the other Activity should be the Activity that is to be launched. When executing this action, the application will change to the destination activity and the data structure will be available in the new activity. Finally, the `CallComponentAction` calls the *action* method that has to be implemented in every `AndroidComponent`. The *action* method will be explained in the next section, Component Meta-Model.

Now, in order to execute those actions, we usually want to associate them with a `UIAction` model within an `AndroidComponent` or an `Activity`. The `UIAction` meta-model is listed in Listing 6.6.

Listing 6.6: UIAction meta-model

```
load "Presentation"
load "AndroidAction"

Model UIAction imports Presentation, AndroidAction {

    Node UIAction {
        target      : LayoutComponent;
        actions     : Action[*];
    }

    Node OnClickListener : UIAction { }

}
```

In `UIAction`, we specify a target to execute a list of actions on. This target should be a `LayoutComponent` (hence the name `UIAction`). For instance, if we want to change activities when clicking a button (i.e. in a menu), a `ChangeActivityAction` should be created that is added to the list of actions of the `OnClickListener` instance.

6.3 Component Meta-Model

Another important part of the framework is the `Component` meta-model. Every implemented Android component should be inherited from this meta-model. Examples of such components are `SMSComponent`, `TwitterComponent` or `ChatComponent`. In the following sections we will describe the abstract `Component` meta-model, the derived `AndroidComponent` and an instantiation of an `AndroidComponent` as an example.

6.3.1 Component megamodel

The component megamodel describes the whole `Component` meta-model hierarchy. As seen in figure 6.2, we see that the `Component` meta-model is used to model both layout components as well as Android framework components. The general Android framework components are instantiated through the Node `AComp`. Examples of these components are `Dropbox` or `Twitter`. When we need a component that communicates with our server (e.g. a `List` component), we need to instantiate the `SComp` component. The layout components `Button`, `EditText` and `TextView` are layout elements visible in an Android application. `ListView` and `ScrollView` are used to divide the layout into parts, respectively when we want to show a list of items or have a scrolling view in the UI. The structure of a layout is described through the `LayoutType` Node. An example of such a structure is a `LinearLayout`, that arranges its children (layout elements such as `Button`) in a single column or a single row.

6.3.2 Component

The `Component` meta-model is listed in Listing 6.7.

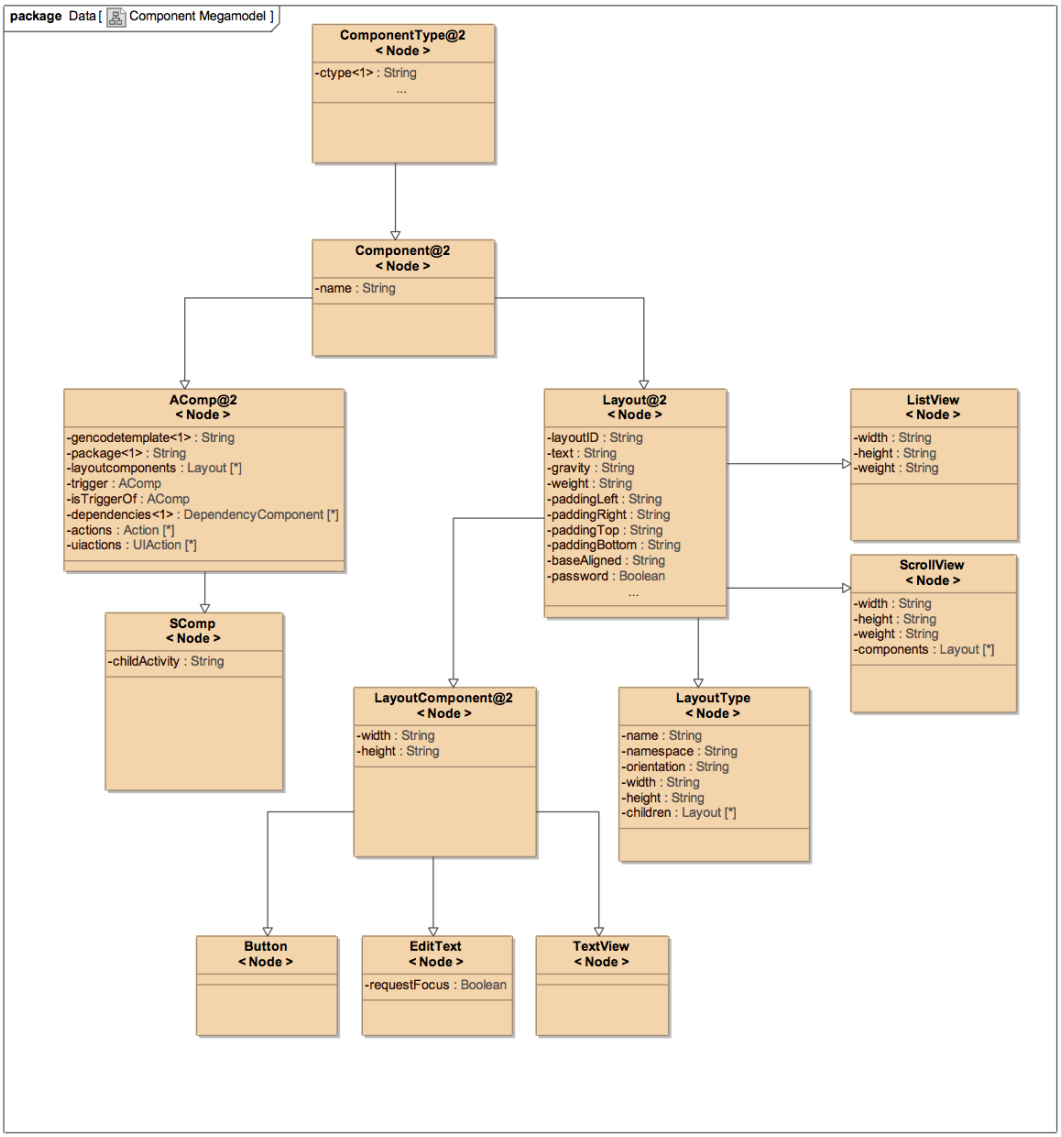

Figure 6.2: Component Megamodel

Listing 6.7: Component meta-model

```
1  Model Component@2 {
2
3      abstract Node ComponentType {
4          ctype@1      : String{id};
5      }
6
7      Node Component : ComponentType {
8          name             : String;
9      }
10
```

```
11      Node ComponentData@1 {
12          name         : String;
13          value        : String;
14      }
15
16      Node Layout : Component {
17          layoutID         : String;
18          text             : String;
19          gravity          : String;
20
21          paddingLeft      : String;
22          paddingRight     : String;
23          paddingTop       : String;
24          paddingBottom    : String;
25          baselineAligned  : String;
26      }
27
28      // e.g. Button, EditText, TextView, ...
29      Node LayoutComponent@2 : Layout {
30          width            : String;
31          height           : String;
32          weight           : String;
33      }
34
35  }
```

As we can see, this meta-model has a potency level of 2. This means that we have three levels in the meta-model. At the highest level, we define an abstract `Component` meta-model. Going down one level, we have defined the `AndroidComponent` itself (e.g. *SMS*) and at level 0, we instantiate the component in our model. All components are of type `AndroidComponent` which is a subtype of `Component`. This component will be discussed in the next section. `LayoutComponent` is another component that inherits from our `Component` meta-model. It is used to describe the layout elements of an `Activity` and has a potency of two. It follows the same instantiation scheme as an `AndroidComponent` does.

6.3.3 AndroidComponent

Listing 6.8: AndroidComponent meta-model

```
1  load "Component"
2  load "AndroidAction"
3  load "UIAction"
4
5  Model AndroidComponent@2 imports Component, AndroidAction,
       UIAction {
6
7      // Android component for semantics and extensibility
8      Node AComp : Component {
9          gencodetemplate@1        : String;
```

```
10
11          package@1                   : String;
12          layoutcomponents@2          : LayoutComponent@0[*];
13
14          // Trigger can be another component (e.g.
                GeoComponent)
15          trigger                     : AComp@0;
16
17          // isTriggerOf defines the bi-directional connection
                with another component
18          // a component could be the trigger of another
                component or a layoutcomponent
19          isTriggerOf                 : AComp@0;
20          dependencies@1              : DependencyComponent[*];
21
22          properties@1                : ComponentData[*];
23
24          // actions extend the action() method, so users can
                extend a component's functionality
25          actions                     : Action[*];
26          // uiactions are actions on a layoutcomponent
27          uiactions                   : UIAction[*];
28      }
29
30      Edge Trigger@(2)(AComp.trigger, AComp.isTriggerOf) {
31          actions                     : Action[*];
32      }
33
34      Node SComp : AComp {
35          childActivity    : String;
36      }
37  }
```

The `AndroidComponent` meta-model contains a `Node` that inherits from `Component`. `AComp` contains all necessary slots to instantiate and generate an Android component. At potency level one, we instantiate the EGL template to be used for generating the component code (in Java), the Java package that should be used, optional dependencies (regular Java classes) or properties to be associated with the component. An example of a possible property is a telephone number for sending an SMS. At the instantiation level, we can instantiate the remaining fields of `AComp`, such as *layoutcomponents* or *actions*. A `trigger` or `isTriggerOf` field can be specified, which enables local communication between two components. For example, when the trigger of a `GeoComponent` is fired, it passes its location to another component (e.g. `SMSComponent`), after which this component can use the location data to perform an action (e.g. send an SMS). This trigger can additionally be extended by using the `Trigger` edge. Through the `Trigger` edge, we can specify a list of actions to be executed when a trigger is fired. The trigger behavior is implicit to each component and cannot be changed. For instance, a `GeoComponent` will only trigger when a new location is found.

Apart from a normal `AComp` node, a modeler can also instantiate components of type `SComp`. These components are used to represent client/server components. Typically, an instantiation of `SComp` communicates with a server. An example of this is the `DropboxComponent`. These type of components are usually embedded in an activity and could optionally have a child activity. For instance in a `DropboxComponent`, we usually want to show additional information on documents found in a Dropbox repository. This additional information will be shown through a child activity that is specified in a Dropbox instantiation of type `SComp`.

6.3.4 Example Component

As an example, we will show the Dropbox component, both on potency level one as well as on the instantiation level. The model for potency level one is listed in Listing 6.9.

Listing 6.9: Dropbox model

```
1  load "Component"
2
3  AndroidComponent dropboxComponent {
4
5      SComp Dropbox {
6          ctype = "dropbox";
7          gencodetemplate = "dropbox.egl";
8
9          folder      : String;
10         key         : String;
11         secret      : String;
12     }
13
14 }
```

At this level, we specify the template to generate the Java Dropbox component. We also introduce new slots that represent the name of the folder that should be used as a repository, together with the token key and token secret to authenticate with OAuth [49]. This level should and will never be visible to a developer. A modeler is only concerned about the instantiation level. The model instantiation is listed in Listing 6.10.

Listing 6.10: Dropbox instantiation

```
1  Dropbox dropbox {
2      layoutcomponents = [dropboxButton, dirContentButton,
           contentText, dirContent];
3      actions = [];
4      uiactions = [triggerDropbox];
5
6      childActivity = "DropboxItemActivity";
7      folder = "/Thesis/";
```

```
8       key = "KEY";
9       secret = "SECRET";
10  }
```

We have defined a few layout components that make up the layout of the Dropbox activity. For instance, the `dirContentButton` value is used to grab the content of the Dropbox folder and the `dirContent` value represents the list that contains all filenames after the content is fetched from the Dropbox folder, that is specified by the folder, key and secret fields. When clicking on an item in a list, the activity that is specified in the `childActivity` field is initiated. This activity allows a user to make comments on a file (e.g. a research paper) in the specified Dropbox folder.

6.4 Server

In this section, we will review the `Server` meta-model and very briefly discuss the implementation details. The server component was written in Javascript and uses WebSockets for bi-directional communication, i.e. communication from client to server and vice versa.

6.4.1 Meta-model

The `Server` meta-model is quite straight-forward and the more complex parts are implemented in Node.js. `Server` is embedded into the `Application` meta-model. The code is listed in Listing 6.11.

Listing 6.11: Server meta-model

```
1  load "Session"
2  load "AndroidComponent"
3
4  Model Server imports Session, AndroidComponent {
5
6      Node Server {
7          name            : String;
8          session         : SessionObject;
9          host            : String;
10         port            : String;
11         components      : Component@0[*];
12     }
13 }
```

The host and port for the server can be set, together with the components that should be included in the server. Apart from that, a modeler should also set a `SessionObject` node. This node simply contains a list of users that are able to authenticate to the server, together with their role.

6.4.2 Javascript and WebSockets

The most relevant part of the `Server` meta-model lies in the implementation in Javascript. In order to realize both asynchronous as well as synchronous communication, Node.js [43] was used together with WebSockets:

> The WebSocket specification - developed as part of the HTML5 initiative - introduced the WebSocket JavaScript interface, which defines a full-duplex single socket connection over which messages can be sent between client and server. The WebSocket standard simplifies much of the complexity around bi-directional web communication and connection management [50].

If we send all our data packets through a `WebSocket`, we can communicate with our server in *near real-time*, because there is bi-directional communication between the client and server. Combining this with a server implementation in Node.js, we leverage Javascript to implement a data-intensive real-time application suited for multiple devices:

> Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Since all communication works through WebSockets, the server implementation is decoupled from any Android application a developer may model. If at some point the modeling framework has to be adapted to generate iOS applications, nothing has to be changed to the Javascript server implementation. The only requirement is a `WebSocket` library on the client side (iOS/Android/web).

6.5 Code Generation

In this section, we will explain the code generation details of Metadepth and the collaborative modeling framework. First, we will briefly explain the Epsilon Generation Language (EGL) syntax. Afterwards, we will show an example of the code generation of the activities in an Android application.

6.5.1 Epsilon Generation Language

EGL is a template-based model-to-text language for generating code, documentation and other textual artefacts from models. EGL supports content-destination decoupling, protected regions for mixing generated with hand-written code, and template coordination [51].

The concrete syntax of EGL closely resembles the style of other template-based

code generation languages, such as PHP [52]. Any text enclosed in a tag pair *[% %]* is used to delimit a dynamic code section. Text not inclosed in a tag pair is considered static code. Listing 6.12 illustrates the use of dynamic and static code sections to from a basic EGL template.

Listing 6.12: EGL template

```
1 ?[% for (i in Sequence{1..5}) { %]
2     i is [%= i %]
3 [% } %]
```

The structure of this basic template is used for the generation of Java code in the collaborative modeling framework. The next subsection shows an example of a real EGL template used in the collaborative modeling framework.

6.5.2 Example

Finally, after a model has been created, a developer can initiate the code generation phase. This phase loads a set of EGL files and traverses each model that has been created. A typical code generation process is implemented as follows:

Listing 6.13: Android code-generation

```
1  // generate main Android app
2  var t : Template := TemplateFactory.load(basePath+'
       genActivity.egl');
3  if (application.activities.isDefined() and application.
       activities.size() > 0) {
4      for (activity in application.activities) {
5          if (activity.main == true) {
6              "populating server var".println();
7              if (application.server.isDefined()) {
8                  t.populate('server', application.server);
9              }
10         }
11         t.populate('activity', activity);
12         t.populate('application', application);
13         t.populate('path', path);
14         t.populate('basePath',    basePath);
15         t.populate('compPath',    compPath);
16         t.populate('codePath',    codePath);
17         t.process();
18         t.generate(codePath + 'src/' + path + activity.name +
               '.java');
19
20         var l : Template := TemplateFactory.load(basePath+'
               genDependency.egl');
21         if (activity.content.isDefined() and activity.content
               .size() > 0) {
22             for (c in activity.content) {
23                 if (c.dependencies.isDefined() and c.
                       dependencies.size() > 0) {
```

```
24                          for (d in c.dependencies) {
25                              l.populate('dependency', d);
26                              l.populate('basePath',     basePath);
27                              l.populate('compPath',     compPath);
28                              l.process();
29                              l.generate(codePath + 'src/' + path +
                                    d.name + '.java');
30                          }
31                      }
32                  }
33              }
34          }
35      }
```

This listing iterates all defined activities and generates the Java `Activity` files. Lines 11 through 16 populate the template with variables we need to generate the target (Java) code. The `populate` method passes variables in the source formalism and map these onto a name. For example, in line 11, we assigned the activity variable to 'activity' in the EGL template that will be called. Line 17 processes all this information and line 18 finally calls the template we initiated in line 2. The template is initiated by calling the `generate` method. This method expects one parameter, the location on the file system of the file that is to be generated.

CHAPTER 7

Case Study: Collaborative Knowledge-Management Platform

In this chapter, we will discuss an example application of the collaborative modeling framework in detail. Using the framework, I modeled a collaborative knowledge-management platform. This application allows a group of people to create sessions and collaborate on a research project. Most of the components and collaborative patterns incorporated into the framework have been used in order to model this case study application.

7.1 Collaborative Knowledge-Management Platform

The goal of this case study is to show the capabilities of the collaborative modeling framework. I modeled an application, using the framework, that proposes a prototype that should make it easier for researchers to collaborate on research projects.

7.1.1 Problem

Currently, there is no collaborative Android application that allows people to work on a research project. This prototype proposal should help researchers on communicating and collaborating through Android phones. For instance, when reading papers, a researcher typically wants to make notes on them. It should be possible to create and share notes easily with a research group. Currently, people make notes in the margin of a paper and pass this paper on to other people. This method does not encourage collaboration and it's hard to collaborate in real-time when both people want to read the same paper. Moreover, this classical approach is not scalable to a group of researchers.

A second problem goes back to collaboration in real-time. When two researchers are not in the same place (and/or time), it is hard to collaborate in real-time on a

research project. How do people working on a research project keep in sync at all the times? We currently have access to the technology, we only need to build the right tools with this technology. If we use technology as an enabler to work more efficiently (i.e. create the tools), how will this make us more efficient on a research project?

7.1.2 Solution

It looks like the solution to the problem that I analyzed is twofold:

- In order to solve the first problem, we can introduce an annotation layer on top of a repository. In the collaborative modeling framework, a repository is modeled using a Dropbox folder. The metadata of every item in this folder can be fetched and annotated with a note by one of the registered researchers of the application. This allows people to simultaneously share their insights on new papers.
- When we want to allow a group of researchers to truly collaborate in real-time when they are not in the same place, we have to introduce a way to make it possible to easily communicate with current technology constraints. This problem is solved by integrating a chat box that allows a team to have a discussion in real-time, whether they are in the same place at the same time or not.

In addition to solutions to those problems, the case study also includes other mechanisms that create awareness between researchers and helps them identify their research goals. In the following section, implementation details of these needs and features will be discussed.

7.2 Platform implementation

The main entities of the model for the Collaborative Knowledge-Management Platform are the Android activities. These activities represent the main features found in the application. Each of those are embedded in a main `Application` object. All source code can be found on Github[1] [53]. We will discuss the most important activities in depth in this section[2]:

- LoginActivity
- MainActivity
- ListActivity

[1] https://github.com/philipdesmedt/Thesis-Android

[2] Note that all references not explained in the meta-models, are simply references to other Metadepth meta-models in the collaborative modeling framework

7.2.1 LoginActivity

The model for `LoginActivity` is listed as follows:

Listing 7.1: LoginActivity model

```
Activity act1 {
    name = "LoginActivity";
    main = true;

    content = [login];
    presentation = pres1;
}
```

Following the `Activity` meta-model, we have set the `LoginActivity` model to be the main `Activity` (which means it will be the first one to be started by the Android OS). As a component, we include a login component and set the layout (which contains two textfields to authenticate, one for the username and for the password, and a button to initiate the login action). The login component is defined as follows:

Listing 7.2: Login component model

```
Login login {
    host = "10.0.2.2";
    port = "3000";

    layoutcomponents = [usernameView, usernameText,
       passwordView, passwordText, loginButton];
    actions = [ela1, ela2];
    uiactions = [triggerLogin];
}
```

Both the `host` and `port` fields are settings that should be set to connect to the server users should authenticate with. The layout components are, as mentioned earlier, the inputs for authentication with the server. The `actions` list extends the functionality of the `action` method of a component and the `uiactions` list contains actions that will be executed on UI elements in the application. In the login component, `ela1` and `ela2` are both `ExtractLayoutAction` elements that extract both the username and the password from UI elements and use these in the `action` method to pass credentials to the server. The UI action `triggerLogin` specifies what has to be done when the login button is triggered. This specific action effectively calls the `action` method of our login component and changes the Activity to `MainActivity` when the given credentials are verified. If the credentials are incorrect, nothing happens at all.

7.2.2 MainActivity

The `MainActivity` class is the entry point for our application once the user is logged in successfully. It contains shortcuts to all functionality within the app. This activity is modeled as follows:

Listing 7.3: MainActivity model

```
1 MainActivity act2 {
2     name = "MainActivity";
3     main = false;
4
5     content = [twitter];
6     presentation = pres2;
7     onClickListeners = [triggerChat, triggerDropbox,
          triggerGoals, triggerLogout];
8     userView = connectedUsers;
9 }
```

We only modeled one component in `MainActivity`, a Twitter component. We don't want a separate Activity for a Twitter component, so that's why we just embedded it into the current Activity and used a `Button` as a trigger for it. Because this component does not need a lot of customizations (apart from the text to tweet), it is offered as an integrated component solution in the framework. The `onClickListeners` specify all shortcuts to the other Activities in the current Activity. Finally, we also specified a `userView` field that, if set, shows all currently connected users in the application.

Note that we don't just use the `Activity` meta-model, but the `MainActivity` meta-model. This meta-model has no difference with the `Activity` meta-model on a modeling level, but it has some differences on the implementation level. The `MainActivity.java` implementation inherits from the `Activity` implementation included in the Android SDK. It is an `abstract class` that requires the implementation of the `setConnectedUsers` method. This method takes a `String` of users as input and allows a modeler to show the currently connected users. That way, our collaborative framework can easily support the awareness of users in a collaborative application.

7.2.3 ListActivity

Another important Activity is the model of the `ListActivity`. We use this generic Activity to allow the implementation of lists of various forms in our application. This way we could model the annotation of Dropbox notes, a list of goals or a questionnaire that can have an arbitrary number of answers. The model for this generic activity is listed in listing 7.4.

Listing 7.4: ListActivity model

```
1 GenericActivity act5 {
2     name = "ListActivity";
3     main = false;
4
5     content = [list];
6     presentation = pres5;
7 }
```

The most important aspect of this model is the use of the `GenericActivity` meta-model type. Just like the `MainActivity` meta-model, this meta-model has no difference with the `Activity` meta-model, but the implementation again is slightly different. `GenericActivity` is an abstract class that contains two abstract methods, `AndroidComponent getComponent()` and `void onMessage(String user, String message)`. The `getComponent()` method is used when we need to notify the embedded component within this Activity and the `onMessage()` method is a callback function that is called when we receive a message from the server. Usually, those `GenericActivity` classes also have a child Activity associated with them, in particular when we implement a list component. This child activity is used to display the comments or notes on a particular list item. For instance, if we model a list of goals, we can comment on these goals. These comments are wrapped into the child activity.

7.3 Demonstration

In this section, we will have a look at the running modeled application. In the previous section, the several activities and their models were discussed and now we can see them in action. In figure 7.1, we see the main activity after a user successfully logged in. In this activity, a user can navigate to all other activities in the collaborative application.

When we start the chat activity, a simple screen with the latest chat messages comes up. A user can then send a message to all other users. This message will be persisted in a MongoDB database and subsequently be sent to all users. A view of the chat box is depicted in figure 7.2.

Finally, the functionality of the Dropbox repository is depicted in figures 7.3 and 7.4. The first figure shows the content of the folder we passed when modeling the Dropbox repository. This directory contains all research papers used to write this thesis. The other figure shows notes on a particular research paper in this directory. Users are able to comment on the paper and make additional notes to e.g. remember the summary of a paper. Users with the `admin` permission can view all notes on a file. Other users with non-admin permission can only see their own comments.

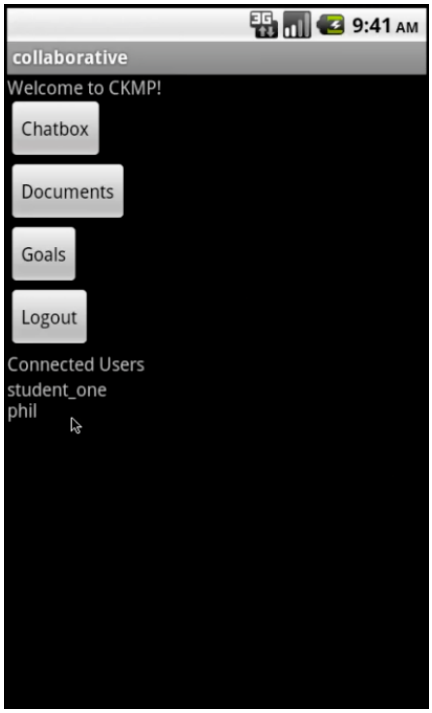

Figure 7.1: In the main menu of the collaborative application with two connected users

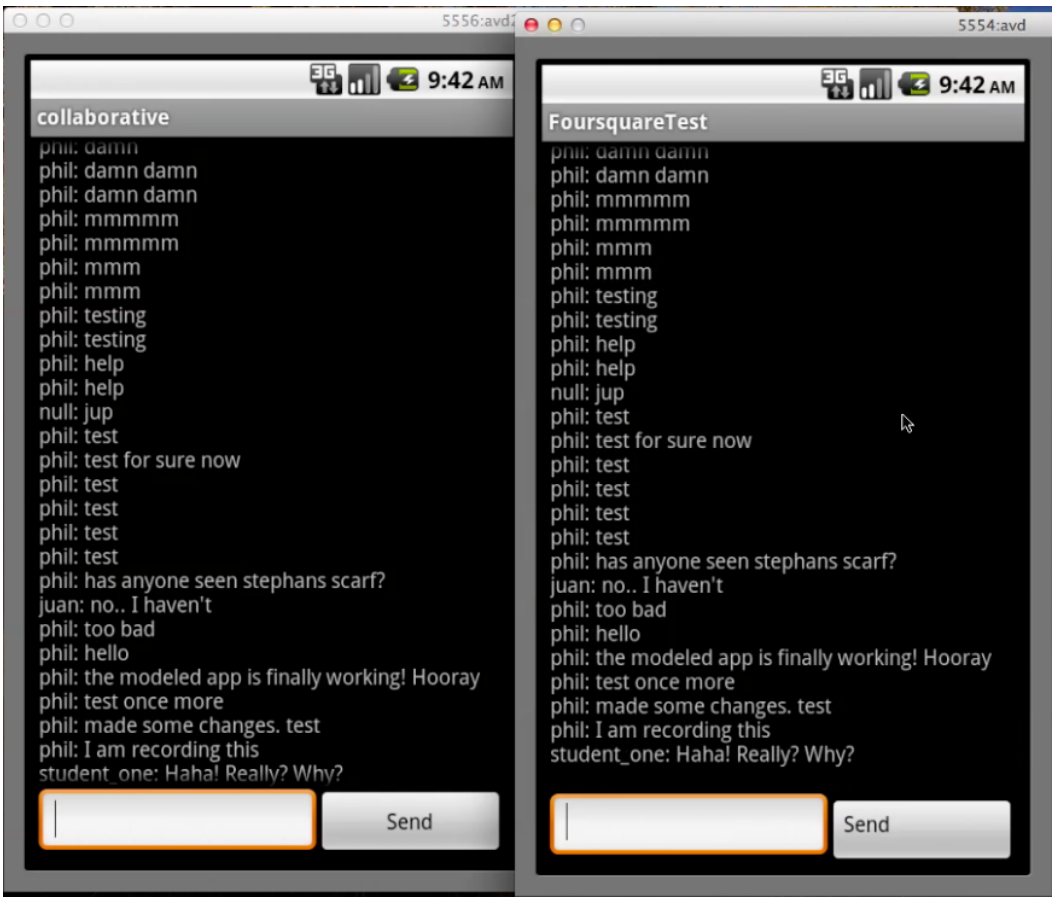

Figure 7.2: In the chat box of the collaborative application

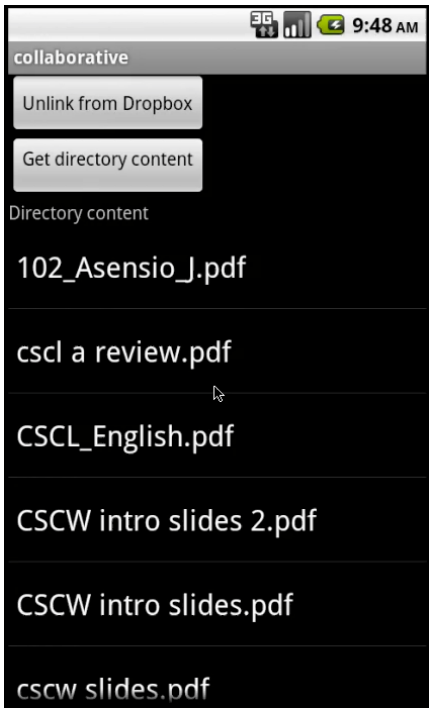

Figure 7.3: Fetched all research papers from the Dropbox repository

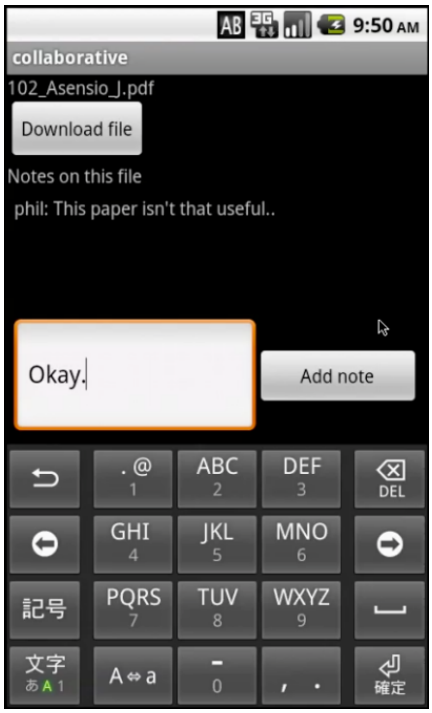

Figure 7.4: Notes and comments on a research paper in the Dropbox repository

CHAPTER 8

Conclusion

This chapter concludes the thesis 'Domain-Specific Modeling and Model Transformation to support Collaborative Work on Android' by giving an overview of the chapters contained in this work. In the second and last section, a number of possible extensions to the collaborative modeling framework are proposed.

8.1 Summary

The document started out with the problem description and an introduction to a traditional software development cycle in chapter 1. The traditional software development cycle established a workflow of analysis, design, implementation, testing and maintenance. With domain-specific modeling, we can test assumptions we make very early in the development process. Moreover, another benefit of DSM is the ability to use other verification techniques apart from testing, i.e. formal verification. Formal verification techniques include formal analysis and model checking.

Next, an overview of domain-specific modeling was given in chapter 2. Here we discussed the traditional modeling approach using UML and its shortcomings. In this chapter, a definition for strict meta-modeling was given. Afterwards, the limitations of strict meta-modeling and the two-level UML modeling approach were discussed. In strict meta-modeling, 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 $0 \leq m \leq n\text{-}1$. Any relationship other than an instance-of relationship between two elements X and Y implies that both elements reside on the same modeling level. This approach introduces some problems that can be solved through multi-level modeling.

In chapter 3, we proposed a multi-level modeling solution (applied to Metadepth) to

solve the problems discussed in chapter 2. The chapter started off introducing new modeling elements (e.g. clabjects, powertypes) and properties (e.g. potency) used in a multi-level modeling approach. Afterwards, the specifics of Metadepth, such as tool support, constraints and linguistic extensions were covered.

Chapter 4 gives an introduction to the Android SDK and Android development in general. Moreover, the layered Android architecture, together with the most important components modeled in the collaborative modeling framework have been covered. The chapter ends with an overview of the Android Activity lifecycle, the most prominent Android SDK component, and Android App Inventor, a What You See Is What You Get (WYSYWYG) editor for building Android application.

The theory and thought process behind the collaborative components included in the collaborative modeling framework were discussed in chapter 5. First the chapter covers two collaborative fields, Computer-Supported Cooperative Work (CSCW) and Computer-Supported Collaborative Learning (CSCL), together with Groupware, a generic term for collaborative work. The second part of the chapter defines the collaborative patterns typically used in a collaborative application and a collaboration stack that defines the different levels in a collaborative application.

Chapter 6 covers the design of the collaborative framework. First, the most prominent meta-models such as `Application`, `Activity` and `Manifest` are explained. Next, we describe the `Component` meta-model hierarchy that has two implementing meta-models `LayoutComponent` and `AndroidComponent`. The other sections explain the `Server` meta-model and how instantiations of these meta-models are used to generate Java code with EGL and the Android SDK.

Finally, chapter 7 describes a case study application that shows off the capabilities of the modeling framework described in chapter 6. In this case study, I modeled a collaborative Android application that allows a group of people to create sessions and work together on a research project. This application uses the majority of the capabilities of the modeling framework.

8.2 Future Work

Based on the previous section, the main conclusion of this work is that it *is* possible to build a modeling framework for Android. However, the Android SDK is *very* extensive and therefore requires an enormous amount of work to map all the SDK components to meta-models. Therefore, the collaborative modeling framework can still be extended with different functionalities and components. Unfortunately, the time frame in which this research was conducted was too small to elaborate on certain aspects. Here are a number of possible extensions to the collaborative modeling framework:

- Support for custom resources (images, graphics, icons, backgrounds,...). Right now the framework does not support custom UIs or visualizations. The consequence is that all applications will look the same, which does not give much flexibility to the framework.
- Improved constraints checking. The collaborative modeling framework consists of minimal constraint checking, but when those constraints are violated, no errors are thrown. Ideally, we would need a building process that throws an error once we violate a high-priority constraint. Unfortunately, it is not easy to give priority to constraints in Metadepth, but it should be possible to catch all errors thrown by violating constraints.
- Additional components to extend functionality. The current functionality in the framework allows a modeler to create a fine range of collaborative Android applications, but there are missing pieces. Right now, there is not a lot of social media support (i.e. in the form of a Facebook or Foursquare component). Adding new components to the framework would give it more commercial value.
- Web service to model and generate .apk Android apps. Right now, we have a minimal visual environment to model Android applications (see appendix A). It would be beneficial to create a web service that allows one to model applications and downloading binaries that can be deployed to Android phones immediately.

Appendices

APPENDIX A

Visual Editor

The collaborative modeling framework allows a developer to model a working Android application that has significant complexity, with a minimal knowledge of the Android SDK. However, the specification of the framework and the meta-models itself are quite verbose. In this appendix, an alternative approach to modeling an Android application is shown, by using a visual editor. This editor restricts the modeler to a certain domain, but allows him/her to easily model an application within a few minutes. In the next sections, we will show how the editor was created and how we use EGL and ETL to create the right target models that conform to the meta-models of the original collaborative modeling framework.

A.1 Domain model

For the creation of the visual environment, we need a domain model that represents the basic elements a modeler wants to create in an Android application. With the creation of this environment, we narrowed down the scope of our applications to a very specific domain. The goal of this domain model is to give the modeler the ability to create Android applications that contain lists of text items. These lists can contain a set of questions the user of the application should answer, a to-do list that can be commented on or a even a city trip info guide. The newly created domain meta-model should provide a mapping onto a more sophisticated target meta-model, one that represents the collaborative modeling framework. Support for the following domain models is included:

- List
- Item

- Server
- User

The complete domain meta-model is listed in listing A.1.

Listing A.1: Domain meta-model

```
strict Model Android@1 {
    abstract Node NamedElement {
        name: String{id};
    }

    Node App : NamedElement {
        appName: String;
        content: List[*];
        server: Server[1];
    }

    Node List : NamedElement {
        items: Item[*];
    }

    Node Item : NamedElement {
        text: String;
    }

    Node Server : NamedElement {
        host: String;
        port: int;
        users: User[*];
    }

    Node User : NamedElement {
        username: String;
        password: String;
        isAdmin: boolean;
    }
}
```

The `List` and `Item` Node are used to create a generic list of items specified by the modeler of the application. For example, if we model a list of questions, each item will contain a question in the `text` field. These items are then referred to in the `List` Node. The `List` on its own is contained in the `App` Node. The `List` Node in the domain model will be transformed into a `List` Node in the target model (the collaborative modeling framework).

The `Server` Node on its turn contains the fields required to a Node.js server instance. The user should specify the host and port, together with the users that should be able to authenticate to the running server.

Finally, the `User` Node represents a simplified model of the `User` model found in the collaborative modeling framework. For this model, we only require a username and password, together with a boolean that specifies whether this user is an admin user or not (i.e. has special privileges).

A.2 Creating the visual environment

For the creation of the visual environment, I extended the work of Leonardo Diez Dolinski [1]. The environment exists in the form of a web application, that allows people to create their own visual models, backed by a Metadepth meta-model. The core HTML and Javascript was written already when I started extending this project. In order to create a visual environment that allows the creation of models in our newly created domain model, we need to create an instance of the `VModel` meta-model included in the existing visual environment. All names prepended with the character `V` model a visual node in the environment.

Listing A.2: Domain meta-model

```
1  VModel VAndroid imports Android, VWidgets {
2      VNode VApp : Draggable, Connectable {
3          refApp: App{ref};
4          templateHtml = "<span>${children.refApp.id}</span>";
5      }
6
7      VNode VList : Draggable, Connectable {
8          refList: List{ref};
9          templateHtml = "<span>${children.refList.id}</span>";
10     }
11
12     VNode VItem : Draggable, Connectable {
13         refItem: Item{ref};
14         templateHtml = "<span>${children.refItem.id}</span>";
15     }
16
17     VNode VServer : Draggable, Connectable {
18         refServer: Server{ref};
19         templateHtml = "<span>${children.refServer.id}</span>
               ";
20     }
21
22     VNode VUser : Draggable, Connectable {
23         refUser: User{ref};
24         templateHtml = "<span>${children.refUser.id}</span>";
25     }
26 }
27
```

[1] http://www.linkedin.com/in/leio10

```
28 VAndroid ide {
29     VNewItem newApp{text="New app";ref="VApp";}
30     VNewItem newList{text="New list";ref="VList";}
31     VNewItem newItem{text="New item";ref="VItem";}
32     VNewItem newServer{text="New server";ref="VServer";}
33     VNewItem newUser{text="New user";ref="VUser";}
34     VDelete delete{text="Delete";}
35     VDownload download{text="Download";}
36     VProperties objinfo{}
37 }
```

The imports `Android` and `VWidgets` respectively include our domain model and pre-defined layout functionality in the visual editor, such as `VDelete` for a delete button or `VDownload` for a download button. The elements in the `VAndroid` meta-model represent the items we have defined in the `Android` domain model. As an example, `VApp` is a visual node that is draggable and connectable across the visual environment. It contains a reference to the `App` Node in the `Android` domain model and has an HTML template specified with it. The other elements in the `VAndroid` meta-model are similar in functionality.

The model of the `VAndroid` meta-model is created in the same file. The model, `IDE`, creates all the items we want to be able to instantiate in the visual environment. The resulting visual environment is visualized in figure A.1.

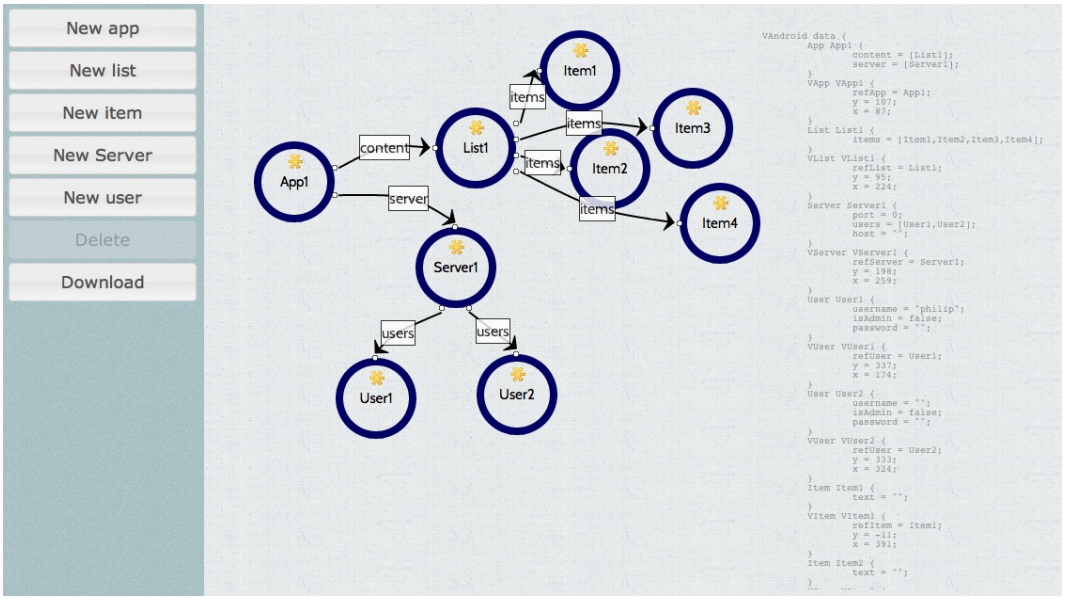

Figure A.1: Visual environment.

A.3 Creating the target model using EGL

Now that we have created the visual editor, we can construct simple models in the domain model. After creating a model, a user can download this model, which produces a file containing Metadepth code. This Metadepth code follows the textual syntax of the domain model we created and is an instance of the `Android` model created before. The source model that contains a list component with items and a server component associated with a list of users now has to be transformed into a target model.

One approach to transforming the source model into the target model is to use an EGL template that interweaves static code sections with dynamic code sections. The static code sections contain the `Presentation`, `Activity` and `Actions` models. The dynamic code sections are filled in with the parts found in the generated source model. For example, we use the following EGL template to construct a list component in the target model:

Listing A.3: List component in the target model

```
1  [% for (li in app.content) { %]
2      List [%= li %] {
3          childActivity = "MainChildActivity";
4          items = [[% for (it in li.items) { %][%= it %], [% }
               %]];
5      }
6      [% for (it in li.items) { %]
7          ListItem [%= it %] {
8              text = "[%= it.text %]";
9              type = "MainActivity";
10         }
11     [% } %]
12 [% } %]
```

In this EGL template we iterate the `app.content` field, which is a collection of items (in the source model). These items are created on their own in a `ListItem` node, which re-uses the `text` field from the `Item` source model. Executing the complete EGL template results in a target model that is conform to the collaborative modeling framework. Using this target model, we can generate an Android binary that runs on all Android phones that have a version of 2.2 or higher.

A.4 Conclusion

Now that we have created a visual editor for the collaborative modeling framework, the barriers of entry to modeling an Android application have been lowered significantly. Although the applications that can be created using the visual environment

are limited to a narrow problem domain, they allow a modeler to easily create domain specific Android applications, eliminating the need of technical knowledge.

Bibliography

[1] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation." http://homepages.mcs.vuw.ac.nz/~tk/publications/papers/mda-foundation.pdf.

[2] W. Scacchi, "Process Models in Software Engineering," *Encyclopedia of Software Engineering*, 2001.

[3] J. Reeves, "What is Software Design?." http://c2.com/cgi/wiki?WhatIsSoftwareDesign.

[4] M. Kolling, "Unit Testing in BlueJ." http://www.bluej.org/tutorial/testing-tutorial.pdf.

[5] Microsoft Developer Network, "Integration testing." http://msdn.microsoft.com/en-us/library/aa292128(v=vs.71).aspx.

[6] Microsoft Developer Network, "Regression testing." http://msdn.microsoft.com/en-us/library/aa292167(v=vs.71).aspx.

[7] T. Szmethy, "Domain-specific models, model analysis, model transformation." http://etd.library.vanderbilt.edu/available/etd-06052006-110407/unrestricted/Dissertation_12pt.pdf.

[8] M. Cohen, "Domain-specific model checking: exhaustive testing in model-based development (position paper)." http://www.foi.se/upload/projects/fusion/TAMSEC_2011_p62.pdf.

[9] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," 2007.

[10] P. Mosterman and H. Vangheluwe, "Computer Automated Multi-Paradigm Modeling: An Introduction," 2004.

[11] C. Atkinson and T. Kühne, "Rearchitecting the UML infrastructure," 2002.

[12] J. de Lara and E. Guerra, "Deep Meta-Modeling with MetaDepth," 2010.

[13] T. Parr, "ANTLR." http://www.antlr.org, 2010.

[14] Eclipse Development Team, "Eclipse EOL." http://www.eclipse.org/gmt/epsilon/doc/eol/, 2012.

[15] Eclipse Development Team, "Eclipse Modeling Framework." http://www.eclipse.org/modeling/emf/, 2012.

[16] Eclipse Development Team, "Eclipse Transformation Language." http://www.eclipse.org/gmt/epsilon/doc/etl/, 2012.

[17] J. de Lara and E. Guerra, "Generic Meta-modeling with Concepts, Templates and Mixin Layers," 2010.

[18] T. Kühne and D. Schreiber, "Can Programming be Liberated from the Two-Level style? Multi-Level Programming with DeepJava.," 2007.

[19] M. G. C. Atkinson and B. Kennel, "A flexible infrastructure for multilevel language engineering," *IEEE Trans. Soft. Eng.*, vol. 35 (6), pp. 742–755, 2009.

[20] Android Development Team, "What is Android?." http://developer.android.com/guide/basics/what-is-android.html.

[21] Android Development Team, "Content Providers." http://developer.android.com/guide/topics/providers/content-providers.html.

[22] Android Development Team, "Application Resources." http://developer.android.com/guide/topics/resources/index.html.

[23] Android Development Team, "Notification Manager." http://developer.android.com/reference/android/app/NotificationManager.html.

[24] StackOverflow, "Difference between the Active and Running state in an Android activity." http://stackoverflow.com/questions/10685973/android-activity-difference-between-active-and-running.

[25] J. Roschelle and S. Teasley, "The construction of shared knowledge in collaborative problem solving," *Computer Supported Collaborative Learning*, pp. 69–97, 1995.

[26] S. Greenberg, "Computer-Supported cooperative work and Groupware," *Academic Press, London*, 1991.

[27] M. Koch and T. Gross, "Computer-Supported Cooperative Work - Concepts and Trends," *Proceedings of the 11th Conference of the Association Information and Management*, pp. 165–172, 2006.

[28] Facebook, "Facebook." http://www.facebook.com.

[29] Google, "Google Docs." http://docs.google.com.

[30] Skype, "Skype." http://www.skype.com.

[31] J. Grudin, "Computer-supported Cooperative Work: History and Focus," *University of California, Irvine*, 1994.

[32] J. Bowers and S. Benford, "Studies in Computer-Supported Cooperative Work: Theory, Practice and Design," *Elsevier Science; Amsterdam*, 1991.

[33] I. Greif, "Computer-supported cooperative work: A book of readings," *Morgan Kaufman, Los Altos*, 1988.

[34] P. Wilson, "Computer-supported cooperative work: An Introduction," *Intellect Books, Oxford, UK*, 1991.

[35] J. Bannon and K. Schmidt, "CSCW: Four Characters in Search of a Context," 1989.

[36] T. Gross and R. Traunmüller, "Methodological Considerations on the Design of Computer-Supported cooperative work," *Cybernetics and Systems: An International Journal*, vol. 27(3), pp. 279–303, 1996.

[37] T. K. S. Gerry and D. Suthers, "Computer-supported collaborative learning: an historical perspective," *Cambridge handbook of the learning sciences*, pp. 409–426, 2006.

[38] P. Dillenbourg, "What do you mean by "collaborative learning"?," *Collaborative learning: cognitive and computational approaches*, pp. 1–16, 1999a.

[39] T. Koschmann and T. L. Ernbaum, "CSCL: Theory and Practice of an emerging paradigm," *Paradigm shift and instructional technology*, pp. 1–23, 1996.

[40] L. Guerrero and D. Fuller, "Jigsaw method in the context of cscl," *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2002*, pp. 789 – 794, 2002.

[41] 10gen, "Mongodb." http://www.mongodb.org/.

[42] Microsoft Developer Network, "Crosscutting concerns." http://msdn.microsoft.com/en-us/library/ee658105.aspx.

[43] Joyent, "Node.js." http://nodejs.org/, 2012.

[44] Eclipse Development Team, "Epsilon Generation Language." http://www.eclipse.org/epsilon/doc/egl/, 2012.

[45] Android Development Team, "Manifest.permission." http://developer.android.com/reference/android/Manifest.permission.html.

[46] Android Development Team, "Intents and Intent Filters." http://developer.android.com/guide/topics/intents/intents-filters.html.

[47] Android Development Team, "LinearLayout." http://developer.android.com/reference/android/widget/LinearLayout.html.

[48] Android, "Common Android Layout Objects." http://developer.android.com/guide/topics/ui/layout-objects.html, 2012.

[49] OAuth, "OAuth: An open protocol to allow secure API authorization in a simple and standard method from desktop and web applications.." http://oauth.net/, 2012.

[50] WebSocket, "What is WebSocket?." http://websocket.org/, 2012.

[51] Eclipse Development Team, "Epsilon Generation Language." http://www.eclipse.org/epsilon/doc/egl/.

[52] Eclipse Development Team, "The Epsilon Book." http://www.eclipse.org/epsilon/doc/book/.

[53] Github, "We make it easier to collaborate with others and share your projects with the universe." http://www.github.com/, 2012.