UNIVERSITY OF OSLO
Department of Informatics

Domain Specific Languages versus Frameworks

Master Thesis
60 credits

Martin Fagereng Johansen

May 4, 2009

Preface

This thesis was written at the Department of Informatics at the University of Oslo. My supervisor has been Birger Møller-Pedersen. I really want to thank him for valuable guidance in writing this thesis and for being able to do this particular thesis, which is well suited for my long term goals. I also want to thank Hilde Galleberg Johnsen for giving feedback on the thesis and Dag Langmyhr for answering questions on LaTeX.

Oslo, Norway. May 4, 2009

Martin Fagereng Johansen

Contents

List of Figures

Chapter 1

Introduction

Solutions to similar problems are often similar. For the software solutions that are similar, the similarities can be extracted into a separate application. This application can then be used as a basis for each specific solution. This separation, into one general and several specific parts, ensures a division of labor. Software developers who are also domain experts can specialize and focus on working on the general part, in which they solve the general problems within a domain. Other developers can then use the general part as a basis for their software solution, and focus on solving their own specific problems instead.

The general solutions should place as few restrictions on its users as possible, while at the same time making sure the users do not do things they should not do. A too restrictive solution unnecessarily burdens the users while a too permissive solution allow users to create problems for themselves. Domain experts know how to avoid domain specific problems. But, they must be able to implement these restrictions in an application and at the same time retain the flexibility of using it.

1.1 The goal of this thesis

It is interesting to look at different techniques for constructing general solutions. It gives insight into which techniques allow the users to have a high degree of flexibility while at the same time being able to ensure appropriate restrictions. Domain specific languages (DSLs) and frameworks are two construction techniques for general software solutions. These two techniques and their application to the construction of general software solutions is the problem area of this thesis.

The problem statement is split into two parts related to this problem area. The first part is a comparison of constructing and using a DSL versus a framework for making a general solution in the same domain. The second part looks at how the implementation of static semantic analysis is different for frameworks than for languages.

Building DSLs typically involves taking a domain model and make a custom syntax and semantics. Users of the DSL write source code either in separate files or embedded in other languages. In order to execute the code a standard compiler pipeline is usually implemented. The DSL code is tokenized, parsed, its static semantics are checked and then code is generated into some lower level language. This lower level code is then either executed or compiled together with other code in the lower level language.

Building object-oriented frameworks involve making classes. Users of a framework instantiates some of these classes according to how the framework is supposed to be used. Framework classes are often used on their own, but also often control the flow themselves,

calling user code when needed. The framework and the user defined-classes are commonly implemented in the same language. Hence multiple frameworks can and are often used in the same application.

It is clear that either technique is powerful enough to create general software solutions. A comparison of the techniques will help developers decide which technique to use for their general solution. Software solutions have requirements which can be judged with respect to discovered differences between the two. Developers can also be explicit about drawbacks and trade-offs they accept.

Static semantic analysis is usually done for languages. After source code has been tokenized and parsed, the code is in the form of an abstract syntax tree (AST). What could not be checked by the tokenizer and the parser is checked by the static semantic analyzer, which is commonly programmed with a general purpose language operating on the AST. Typical checks are for making sure variables are declared, and that the types are correct for expressions. There are additional static checks for every domain.

When performing static checks before run-time, one knows that the checked problem does not occur at run-time. Therefore, a static semantic analyzer for frameworks can reduce the number of run-time exceptions in the framework code.

The number of existing frameworks in use today is a reason in itself why static semantic analysis for frameworks is worth considering. Some of these might be eligible for static semantic analysis or might become eligible with minor modifications. If a framework has many users and a large amount of user-code, a static semantic analyzer gives the users the ability to get feedback not only at run-time. The feedback is not only on what is wrong, but also warnings about what is dubious. Static semantic analysis is also a benefit for a framework even if it only applies to parts of the framework.

1.2 Method

The method used to evaluate the first part of the problem statement is to design and implement the same application as a framework and as a DSL. Doing a complex example with the two approaches, while keeping in mind the current state of the art, should help give insight into the problem statement questions.

The method is hence to perform an experiment. The experiment is constructed as follows. Take a complex domain where there exist a number of general solutions and perform a domain analysis on this domain. The existing solutions can be used in order to advance past the domain analysis quicker. Make two implementations within this domain: A DSL and a framework. In order to construct the DSL for the domain, describe the syntax of the language using EBNF grammars and the semantics in English. Implement the lexer, the parser and the static semantic analyzer for the language. The idea is to generate code from the DSL to the developed framework when it is done. Having the complete DSL-specification, this is therefore a good time to start on the framework. Select an object-oriented programming language with support for reflection. Construct a framework which is as similar to the DSL as possible and which retains the features of the DSL-implementation to as high a degree as possible. This should give insight into either why it is not possible to achieve similar features, or why it is possible to actually have similar features for the framework.

The order of implementation is relevant. DSLs are expected to be the most powerful, while frameworks are expected to be more flexible. The reason for this is that a DSL can have custom syntax and semantics, which places very few limitations on how a DSL is

constructed. A framework is limited by the constructs available in the implementation language. This also provides the framework with flexibility in interacting with other software systems. It is expected that what can be constructed with the framework, probably can be constructed with the DSL. The same thing cannot be said about the flexibility of the DSL. The flexibility of a framework seems simply to be different than the flexibility of a DSL. It is therefore a point to start with the DSL-specification. Then later move on to see if a comparable framework may be constructed.

The method used to investigate the second part of the problem statement is as follows: Given the completed DSL and framework implementations, modify the framework in an attempt to support static semantic analysis. The framework is modified so that reflection can be used to extract information from instantiations of the framework. A static-semantic analyzer works on an AST. Therefore, if something equivalent to an AST can be constructed from framework instantiations, its static-semantics might also be checked.

The concrete experiment. The experiment described in this thesis is in the domain of electronic commerce (e-commerce). Domain analysis is chosen as the method for understanding the domain and to identify the concepts within it. The programming language used for the implementation is the Java Programming Language 1.6 (Java) specified in [GJSB05]. Java is currently an actively used production language which is object-oriented and with good tool and IDE support. The DSL syntax is described using regular expressions (RE) and Extended Backus-Naur Form (EBNF). The parser is implemented using ANTLR 3.0.1. The semantics are described in English and implemented in Java[1]. Java is used for static semantic analysis. Java and Velocity 1.6.2 are used for code generation from the abstract syntax tree produced by the parser. The framework architecture is described in OMG UML 2.0 [BRJ05] models and with Eclipse Ecore class diagrams. UML and Java terms are used throughout the experiment report, as they are the two major technologies used.

Experiment implementation. The complete experiment has been implemented for this thesis as described in this method description. The most important parts of the implementation are shown in the experiment reports; chapter 3 and 4. Examples from the experiment implementation are used in the discussion of the results in both chapter 3 and 4. The complete project source code and where to get it is described in Appendix A.

Note that some of the tools used are experimental at the point of writing this thesis. Therefore, some of the implementation models and details might contain names which are slightly different than the ones used in the thesis. The reason is that these experimental tools are not mature enough to allow for name changes at all levels. Therefore, some of the names and terms are outdated. The deviations are noted in the description text.

Example domain. The example domain is e-commerce. There are many parts of an e-commerce system. The domain used in this experiment is the part which facilitates interaction with the customer to allow him or her to purchase commodities. This part of an e-commerce system is often called a webshop. A webshop lets the customers browse through a catalog of commodities and to purchase a selection in an orderly manner. The webshop handles this task as if it is a salesclerk. The complete domain analysis of

[1]Note that not absolutely all semantics described are implemented.

this part of an e-commerce system is described in detail in the domain analysis, section 3.1. The domain was chosen because of the authors experience with working with an e-commerce system professionally and due to the abundance of open-source solutions available. The domain analysis of the e-commerce domain resulted in a domain model and a (requirements) specification for e-commerce systems. The domain model and the specification form the basis for both the DSL and the framework. The domain analysis was performed by considering the author's experience, the domain itself and nine different e-commerce systems, resulting in a list of 60 distinct requirements from the system-user, the administrator and the customer perspective[2]. These requirements were all used to create the specification for the DSL and the framework presented in this thesis.

DSLs promise flexibility in how to solve problems. It is a goal to produce a domain model unrestricted by existing language concepts. To enjoy the full range of possibilities enabled by custom syntax and semantics, the domain model is created without special care for existing language concepts. How this is achieved for the experiment presented in this thesis is considered next.

Design exploiting DSL possibilities. The solution presented in the e-commerce domain solves two of the major problems which caused repetitive work and a muddy design in the author's experience with working with three large Norwegian webshops[3]. The solutions are the construction of a catalog and the usage of statecharts to describe behavior. The usage of statecharts is considered in the two articles [SB04] and [Gli95]. The former considers statecharts especially for webshops. One approach which neither of the articles considered is approaching the concrete design-choices of the statecharts from a game/simulator-developers perspective: Model the objects and their behavior. Instead of modeling the webshop as a monolithic system, model the customers, administrators, orders and products. Then combine these into a shop-simulation which can be given a website view. This is the way statecharts are used with great success in the game framework and language [Swe08]. It uses statecharts to describe the behavior of entities in the game world. Hopefully this will be a better choice for a DSL than the approaches taken to statecharts in the two articles. The use of statecharts is elaborated in the domain analysis, section 3.1.

1.3 Document structure

This document is organized as follows. In chapter 2 we look at background knowledge relevant for this thesis. In chapter 3 we look at the DSLs vs. frameworks experiment. It starts with a domain analysis and continues with the DSL and the Framework designs and implementations. Then the results are analyzed. In chapter 4 we look at the static semantic analysis for the framework experiment. First the modifications of the framework are described, and then there is an analysis of the results. Finally, chapter 5 concludes the thesis.

[2]See Appendix B for the nine systems and the complete list of requirements.

[3]Dangaard B2B webshop, Netcom B2C webshop, Tele2 B2C webshop in 2005 and 2006

Chapter 2

Background information

The most important terms used in this thesis are DSLs, frameworks, static semantic analysis and statecharts. These terms are considered in turn in this chapter.

2.1 DSLs

The definition of DSL used in this thesis is from [Spi01]: "A domain-specific language is a programming language tailored specifically to an application domain: rather than being general purpose it captures precisely the domain's semantics." As with all programming languages, a DSL consist of a custom syntax and semantics. The syntax, defined by grammar, specifies what well formed sentences within the language are. The semantics describe the meaning of terms and sentences within the language.

Example. The following is a grammar for arithmetic expressions using only addition.

Listing 2.1: Simple example grammar

```
expression ::= number | number "+" expression
number ::= [1-9][0-9]*
```

For simple arithmetic, $1 + 2 + 3$ is according to grammar, while $1 + 2+$ is not.

The vocabulary of this language is: expression, number and addition.

The semantics of this simple language is: The first and second operands of an expression are added. □

The vocabulary of a DSL is taken from its domain. The syntax should be created to fit the domain, and preferably also the conventions within the domain. The domain model is created by a domain analysis, preferably by a computer scientist together with a domain expert.

2.1.1 External, internal and embedded DSLs

External DSL is defined in [Fow05]: "I define an external DSL as one that's written in a separate language to the main language of an application [...]. Unix little languages and XML configuration files are good examples of this style." Internal DSLs are also defined in the same paper: "Internal DSLs morph the host language into a DSL itself - the Lisp tradition is the best example of this." The author of the paper coined the terms internal and external DSL. He explains why he does not use the term "embedded DSLs":

> Internal DSLs are often called 'embedded DSLs' but I've avoided the 'embedded' term because it gets confused with embedded languages in applications (such as VBA embedded into Word which if anything is an external DSL.) However you'll probably come across the embedded term if you look around at more writing on DSLs.

The DSL created in this thesis is an external DSL.

2.1.2 Specification- and implementation-languages

A specification language is a language used to specify a system and not for execution primarily. An implementation language is a language used to implement a running system. The DSL presented in this thesis aims to be a language which specifies a system as well as being executable. Since a DSL has a custom syntax and semantics, the information required in order to have an executable system might be reduced to a minimum. Hence the gap between the specification and the implementation of the system is also lessened as a consequence.

2.1.3 DSL vs. DSL code vs. another application

It is important when reading this thesis to keep in mind that there are three different programs involved in DSL development. The first kind is the compiler and run-time system of the DSL. The second kind is the DSL code written in the DSL language. The third kind is an application using the constructs created in the DSL. These three kinds of programs are all considered throughout this thesis.

2.2 Frameworks

Frameworks are considered in this thesis as explained in [Bos98].

> [...] an object-oriented framework is a kind of reusable software architecture comprising both design and code. [JF88] defines a framework [as] *a set of classes that embodies an abstract design for solutions to a family of related problems.* In other words, a framework is a partial design and implementation for an application in a given problem domain. The central part of the framework design comprises both abstract and concrete classes in the domain.

2.2.1 Frozen- and hot-spots

Further, the notions of frozen and hot spots in a framework are important. This is explained in [BMR+96]:

> According to [Pre94] an application framework consists of frozen spots and hot spots. Frozen spots define the overall architecture of a system, its basic components and the relations between them. These remain unchanged in any instantiation of the application framework. Hot spots represent those parts of the framework that are specific to individual systems.

Example. For the example E-Commerce Framework, the hotspots are the contents of a catalog, and the definition of possible user and commodity behaviors. These hot spots let the developer specify what is specific to his or her e-commerce system, be it a webshop selling aquarium fish or electronic equipment. The frozen spots are the parts of the system which are common for all implementations of the hot spots. The notions used from the domain in this example are discussed in detail in the domain analysis, section 3.1. □

2.2.2 Black box and white box frameworks

According to [FS97] "frameworks can also be classified by the techniques used to extend them, which range along a continuum from white box frameworks to black box frameworks." The difference between the two is explained in the same paper.

> White box frameworks rely heavily on OO language features like inheritance and dynamic binding to achieve extensibility. Existing functionality is reused and extended by (1) inheriting from framework base classes and (2) overriding pre-defined hook methods using patterns like Template Method [GHJV95].
>
> Black box frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by (1) defining components that conform to a particular interface and (2) integrating these components into the framework using patterns like Strategy [GHJV95] and Functor.

The framework designs and implementations in this thesis are mostly white box.

2.2.3 Framework vs. framework instantiation vs. another application

As for DSL, frameworks development also involves three kinds of programs. The first is the framework itself. The second kind is an application created using the framework as a basis. This kind of program is a framework instantiation. The code is called the framework user-code. The third kind of program is another application using the instantiated framework in some way.

2.2.4 Static constraints and object-orientation

For the static constraints found during the domain analysis, why not lay out relations and structures in a class-diagram which ensures the static constraints? The reason is that it is hard or impossible according to [WK05]:

> The information conveyed by such a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. Many of the flaws in the model are caused by the limitations of the diagrams being used. A diagram simply cannot express the statements that should be part of a thorough specification.

This is therefore a problem for the framework. OCL is the object constraint language introduced into UML in order to be able to express constraints which cannot be expressed

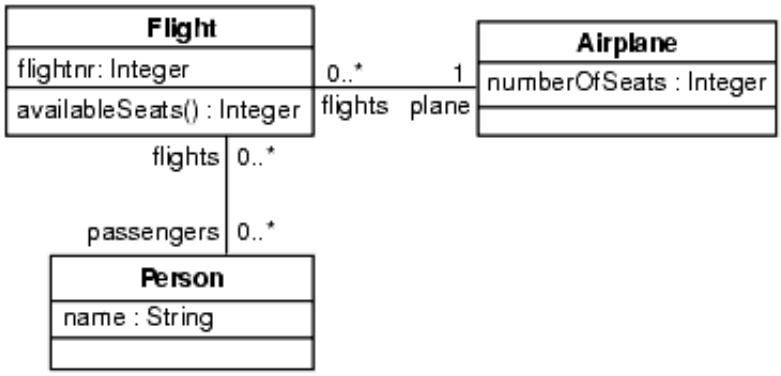

Figure 2.1: Example where OCL is needed to express a constraint

by a class diagram. When programming a framework, one has to write OCL expressions to express the constraints identified in the domain model which cannot be expressed in the class diagram otherwise. OCL is for creating run-time checks, and hence do not solve the problem of static constraints for frameworks. The following example is taken from [WK05].

> In the UML model shown in Figure [2.1], an association between class Flight and class Person, indicating that a certain group of persons are the passengers on a flight, will have multiplicity many (0..*) on the side of the Person class. This means that the number of passengers is unlimited. In reality, the number of passengers will be restricted to the number of seats on the airplane that is associated with the flight. It is impossible to express this restriction in the diagram.

For a DSL, static constraints are a part of the compilation process. It is therefore a natural part of DSL construction to program the static constraints. Hence this is a drawback for framework construction. Someone using a framework is not informed at compile time. Breaking the static constraints of the domain model is not expressed in the class-model in such a way that an object-oriented compiler will catch it. Therefore, class models are not enough to express static constraints.

2.3 Static-semantic analysis

Semantics and static-semantic analysis is described in [WG84, p. 12 and p. 183] as follows.

> Semantics include properties that can be deduced without executing the program as well as those only recognizable during execution. Following GRIFFITHS [1973], we denote these properties static and dynamic semantics respectively.
>
> Semantic analysis determines the properties of a program that are classed as static semantics, and verified the corresponding context conditions the consistency of these properties.

Static-semantic analysis is the compilation phase which commonly follows lexical analysis and parsing. According to [WG84] the semantic analyzer typically performs "name analysis, finding the definition valid at each use of an identifier. Based upon this information operator identification and type checking determine the operand types and verify that they are allowable for the given operator." Note that not all DSLs require these example semantic checks necessarily.

When and how can static-semantic analysis be performed? The following are a few examples.

Example 1 If a program takes no input and does not interact with anything except for printing a number to the console, then the program need only be evaluated once and for all at compile-time, because it cannot change.

Example 2 In "dynamic"/un-typed languages there is only one type, this type is very general and have uninteresting static semantics. An example is a program which takes input from the user console. A console typically allow string-input from the keyboard. Therefore the input type of the program is string, even though the type of the data actually entered is unknown. The program then typically checks that the input belongs to a certain type, or else a run-time exception is thrown. If the input does belong to a type, then the data can be submitted to a typed program. This typed program's static semantics have been guaranteed by the static semantic analysis.

Example 3 If a language uses types, then the types can be reasoned about at compile-time. Operations have static semantics such as:

- $78474 + 746385 =?$, but we know that $Integer + Integer = Integer$
- $23761232 * a =?$, but we know that $EvenInteger * Integer = EvenInteger$
- $a * 1.1101 =?$, but we know that $Integer * Double = Double$[1]

As we can see from example 3, the domain of arithmetic has some static semantics. By doing the same conversion for the domain in this thesis, domain concepts might be converted into a form similar to the examples in example 3. That is, reason about what is known about the program as it is written. Static-semantic analysis does not have to be done at compile-time, it can be done on any data at any time. Static-semantic analysis is an evaluation of a program without actually running it. The result is an evaluation of the consistency of a part of a system.

2.4 Statecharts

Statecharts are based on normal finite state machines (FSMs). They were invented by David Harel and are described in his paper [Har87]. Statecharts are a collection of practical additions to FSMs, which do not increase their expressiveness. The additions are mainly to avoid the state explosion problem which one encounters when trying to use normal FSMs for practical problems. Statecharts resolve the duplication by adding some handy abstractions to the FSM formalism.

[1] dynamic languages will change to Integer if that is the run-time result

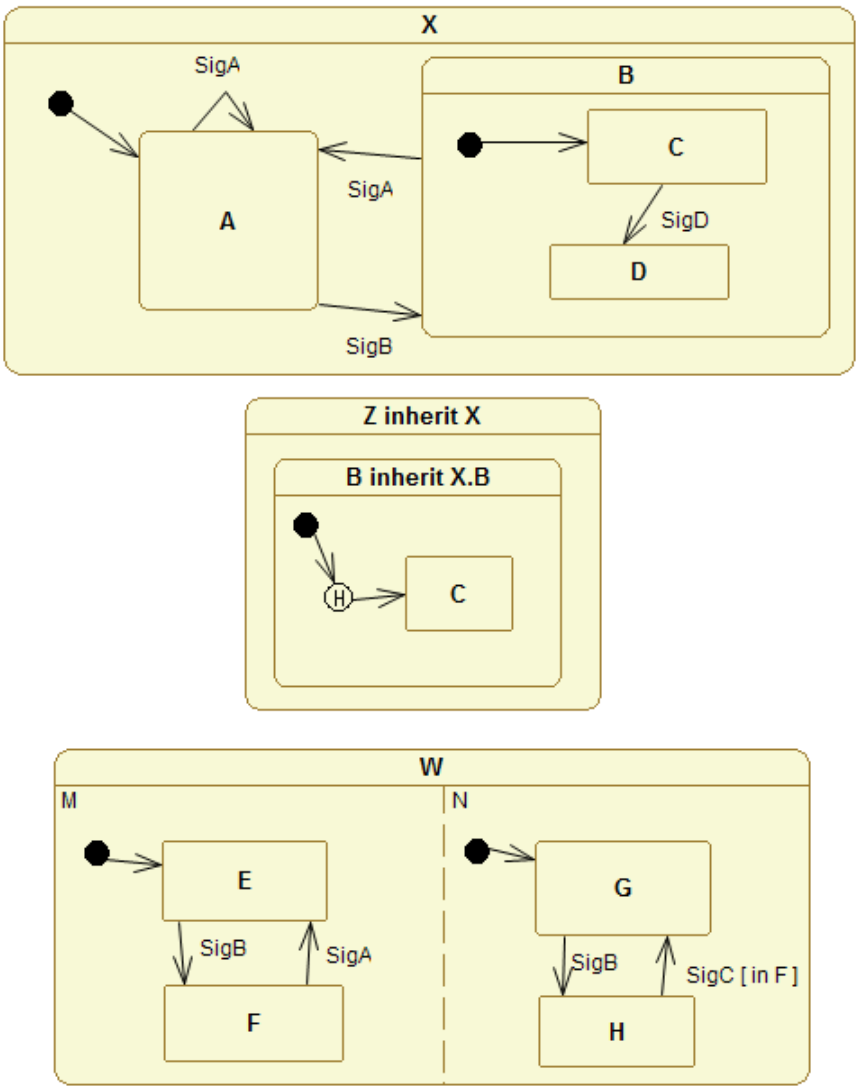

Figure 2.2: Statechart example

Figure 2.2 shows an example statechart. The following subsections describe the additions used in this thesis.

2.4.1 Composite states

Composite states are states having a state-machine inside of it. Composite states are either xor-states or and-states. The additions used in this thesis are described in the following subsections.

XOR states

An xor-state, both states X and B in 2.2, is a state where the system is in only one of the sub-states at a time. This type of composite state is normally used to describe a state machine.

AND states

An and-state, for example state W in 2.2, is a composite state where the system is in all the sub-states simultaneously. For example, W is in both state M and state N simultaneously. The sub-states of an and-state are xor-states, such as M and N in 2.2.

2.4.2 Inheritance

Statechart inheritance, for example as for state Z in 2.2, reuses the composition of existing states in order to construct new ones. The new states can override existing states, by using the same name, or transitions, by using the same source and trigger. In figure 2.2 Z overrides X in which a new B overrides the old B. In the new B, both the initial state and the initial transition are overridden. The state C is the same as in the old B. Hence, the new B adds a history state as the initial state of B.

2.4.3 History states

If one leaves a substate such as B, when one returns, one is back at the initial state. Sometimes it is practical to have a state remember which state it was in. This is what the history state is for. A transition to a history state is a transition to the state one was previously in. Hence it only makes sense to have the history state as the initial state.

2.4.4 Constraints

A constraint is seen in figure 2.2, xor-state N. The notation used is square brackets with a constraint written inside. The two types of constraints used in this thesis are in-constraints and not-in-constraints. An in-constraint requires that the current statechart is in some particular state in order for the transition to occur. The not-in state requires that the current statechart is not in a particular state. In example figure 2.2 the transition from H to G can only happen if M is in F.

2.5 The class diagrams in this thesis

The class diagrams used in this thesis is drawn using the Eclipse Ecore diagram editor. Figure 2.3 shows as example class diagram. Classes have names written in normal type font and have a box with three compartments as an icon. Interfaces have names written in italics and a blue circle with a capital I in it as an icon. Operations for both classes and interfaces are listed in the third compartment. Only public operations are shown. Neither parameters names nor generics are shown in the diagram. Classes and interfaces shown with only one compartment is an elided view of the class. In other words, its operations are irrelevant for the diagram. The contents are then probably shown in a diagram where it is actually is relevant. The diagrams should follow the UML 2.0 standard for class diagrams.

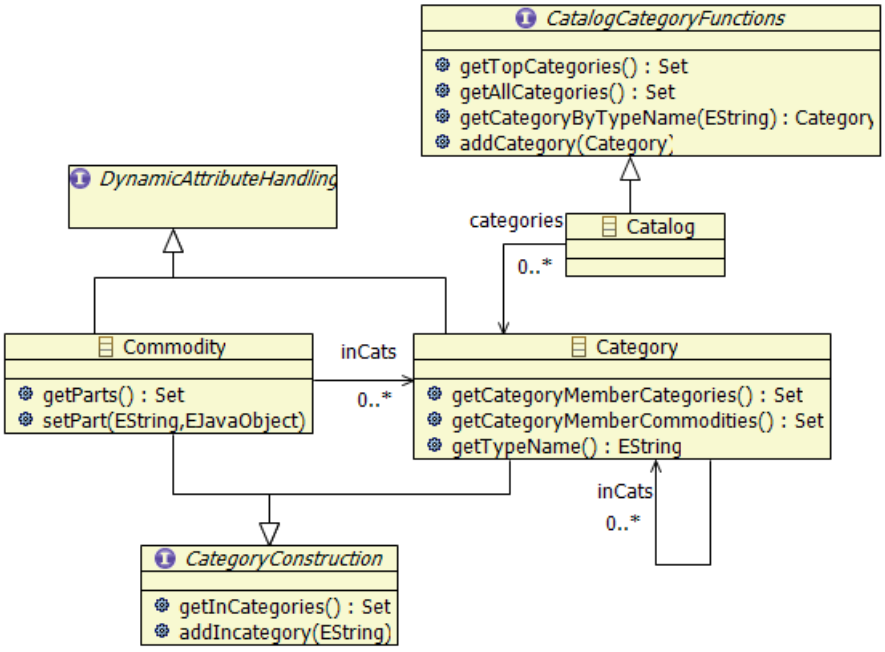

Figure 2.3: An example class diagram

Chapter 3

DSL vs. Framework

This chapter covers the first part of the problem statement. It studies the differences between DSLs and frameworks by implementing the same example with both techniques. First a domain analysis is done followed by the design and implementation of the DSL and of the framework. The experiences are discussed, including a consideration of some relevant issues regarding the comparison in light of these experiences. The chapter ends with a look at related work.

3.1 Domain analysis

In this section a domain analysis of the electronic commerce domain is presented. This domain analysis is a part of the experiment to compare DSLs and frameworks by programming the same solution as a DSL and as a framework. The domain analysis gives insight into how the systems should be designed, which concepts it should include and how they relate.

The domain analysis is not done according to a specific method. It is simply a systematic look at the domain.

3.1.1 The commerce domain

The purpose of this domain analysis is to gain insight into the domain of electronic commerce. Therefore we should start by examining the domain which is to be automated electronically, namely *commerce*.

Commerce is defined as "transactions (sales and purchases) having the objective of supplying commodities (goods and services)"[1]. In other words, commodities are offered by a party for which a proposal of trade is made by other parties. When the parties have come to an agreement, they produce a contract. This contract establishes such things as the requirements of each party to deliver the commodities to the other party, the deadline of the delivery, the quality and integrity of the delivered product and the warranty. This contract is then enforced by the government, so the trade is secured by this mechanism.

The concepts used so far in the domain are *commodity*, *party* and *contract*. A commodity is a good or a service. "A party is a person or group of persons that compose a single entity which can be identified as one for the purposes of the law."[2] A contract is

[1] WordNet

[2] http://en.wikipedia.org/w/index.php?title=Party_(law)&oldid=238728844

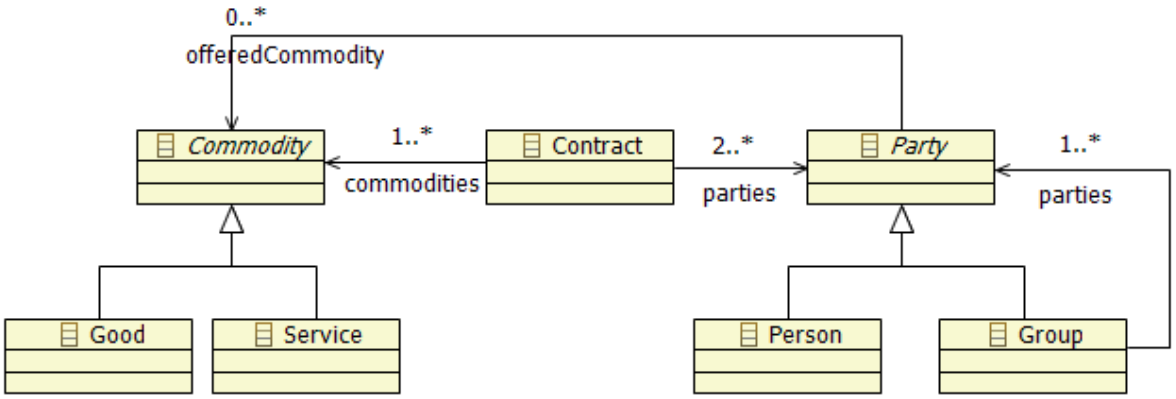

Figure 3.1: An overview of the commerce domain in an EMOF class diagram

"a binding agreement between two or more persons that is enforceable by law."[3]

These concepts and their relations are modeled with a class diagram in figure 3.1. In the diagram we see the domain concepts discussed so far with the relevant relations mentioned. A contract must always have at least two parties involved and the contract is for one or more commodities. A party is either a person or a group. The different parties are eventually reducible to persons, as one would expect.

The offering of commodities by a party is done by organizing the offered commodities in a catalog. A catalog is "a complete list of things; usually arranged systematically."[4] The information in a catalog is redundant with respect to the actual physical commodities. The catalog is updated by the party which offers the commodities listed inside it, who makes sure the integrity of the information inside is valid. The catalog is browsed by other parties in search for possible purchases. A buyer makes an order of commodities which he may want to buy. He then negotiates a contract with the seller based on the commodities in the order.

Example. For a physical retail store, like the grocery store around the corner, the goods offered are put out in display physically in the shop. In this case the store does not have to maintain a catalog. The buyer takes the goods off the shelves and places them into a basket which he holds. This way, the goods are reserved by the buyer who may want to buy them. The negotiation of a contract is also simple. The price is in money and is written on each good, the buyers gets his goods now, and the store has a standard contract which is valid for all goods purchased. There are normally special conditions for buying certain goods such as cigarettes. Such things are managed by the salesclerk who handles the customer. □

Example. A physical computer store might not have all commodities available in the store at a single point in time, such as the common retail store in the previous example. In this case the buyer and the seller discuss the order based on a catalog and make the contract together. For example, a contract contains "Take my computer; replace the malfunctioned disk; install double the RAM." □

[3] WordNet
[4] WordNet

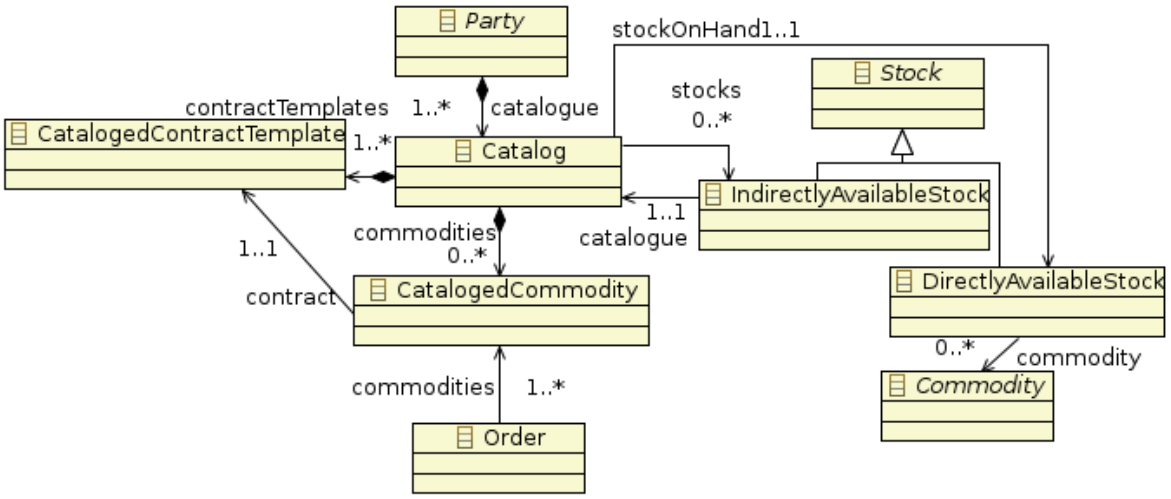

Figure 3.2: Extension of the commerce model with catalogs, orders and contracts

The new concepts introduced here are catalog and order. A catalog is browsed by someone interested in buying something. An order is a list of commodities which two parties, the buyer and seller, will use as a basis of contract negotiation.

These concepts and their relations are modeled in figure 3.2. This diagram introduces the catalog and its relations to the concepts in figure 3.1, which it extends. This diagram requires a bit of explanation, so let us walk through it step by step. First of all a party can have several catalogs available for browsing. A catalog can contain several cataloged commodities. The reason a catalog does not contain commodities is that commodities are physical things or actual services. What a catalog can contain is simply a representation of the commodities, which presents it somehow, for example with some text and a picture. Commodity objects represent actual commodities, while cataloged commodities are a representation of the commodity. The duplication of information in the catalog discussed earlier is hence modeled this way. An order contains a reference to one or more cataloged commodities, which the customer may want to buy. A catalog also contains one or more cataloged contract templates. The catalog qualifier is used on contract here for the same reason as for cataloged commodities. Further, it is a template because there are blanks in the contract which must be specified before it is a completed and valid contract. A cataloged commodity is related to a contract. For example, cigarettes require a customer to be over the age of majority. Hence cigarettes reference a different contract than do a can of soda. On the right side of the diagram we find the stock. A stock is an actual storage of commodities. Commodities are directly available if they are at the catalog owners themselves. If the party has to order the commodity from another party, then they have to browse their catalog. Hence other party's catalogs references the indirect stock. This hierarchy of stocks finally resolves to an array of actual commodity-representations. The cataloged commodities presented in a catalog are intended to collect and order all these commodities in a structured fashion.

Example. Ole sets up a business for selling books. He needs commodities to offer, which he decides are goods. He makes deals with several publishers using their catalogs and collects the goods he would like to offer in his own catalog, which he maintains. He then makes a standard contract. The contract, he finds out, requires two parties: The seller

and the buyer. He registers his firm with the local authorities and gets an organization id which is used in the contract to identify him as the seller, and he leaves a blank spot for the id of the buyer. He publishes his catalog as a static document on the Internet together with the contract template he created. In order to receive orders he publishes his email address and postal address. A potential buyer then browses his online catalog, writes his or her order as a text-document and sends it to Ole. Ole then process the order, signs the contract and sends the bought books together with the contract to the buyer. □

In general terms commerce is typically done as follows. In order to set up a catalog of commodities to sell, the selling party takes into account its stock and the stocks of other parties, i.e. their indirect stock, specified in their catalogs. After having compiled a catalog, it is presented to other parties. They browse the catalog, compile an order and send it to the selling party. The seller verifies the order against the catalog and the contract template. A contract is produced. If there are missing required information in the order, or some of the information is incoherent or ambiguous, the buying party is contacted for clarification. The goods are packaged with a copy of the actual contract and sent to the buying party.

3.1.2 Automation of commerce

Let us look at possible automations using the bookstore example as a guide.

Ole's sales are going very good, and he soon has to work overtime every day. He has two options: He can hire more people to do the work, which prior to computers indeed was the only option, or he can automate some of the tasks using computer systems.

Which tasks does he have? (1) He receives updated catalogs from the publishers every 4 weeks, from which he updates his own catalog and publishes it. He also updates which books he has in stock. (2) He has to contact some of the customers because of incomplete, incoherent or ambiguous orders, in order to complete the order. (3) He receives orders which he simply packages and sends. If an order requires him to get books from another seller, he orders missing books from the publisher and waits for the books in order to package and send the order later. (4) He has to handle returned books, which did not meet the contractual agreement or where the customer used his 14 day return policy right. (5) If the books Ole has ordered from another seller violates a contractual agreement with them, e.g. the book is misprinted, he has to return it and handle the case with them.

What is it possible to automate here? The possible automations are marked with numbers in the previous paragraph. Each is discussed in turn in the following paragraphs.

(1) Compilation of the catalog and keeping it up to date. Ole contacts one of the publishers and learns about their web service for accessing their catalog. He contacts a second and smaller publisher and learns that their only catalog is a printed one. He contacts a third and medium sized publisher, and learns they post updates as news on their website. The important point here is that information which can be handled by a computer, can also be automated using computers. Hence, information in a catalog should be processable by a computer. If other catalogs are also processable, the computer system can keep different catalogs up to date. A computer can use a structured catalog to render a browse-able catalog online, and also other vendors can use Ole's catalog to resell books in another business.

(2) Handling the orders and ensuring their correctness. Ole requires customers to manually set up the order. He observes the relationship between the catalog, the contracts and the order. The order is correct if the books ordered are available in the up to date catalog, and the information missing in the contract is specified, such as delivery-address, personal id and payment details. A customer writing the order manually might not get all the details right. If the catalog and the requirements of the contract are available in a computer-processable model, then the customer builds the order with the system, which ensures its validity as its being built. The customer might still produce an order which is different from his or her intention and with a misspelled address, but at least the system can guarantee that the order is consistent and complete with respect to what is in it.

(3) Processing the order. Having allowed the user to only enter valid orders, the incoming orders are simply stored in the system and marked for packaging. Automation of the packaging and sending the order is outside the scope of this domain analysis. As the order is valid with respect to the catalog, concrete commodities can now be assigned to the order. For example, the customer might have ordered a book. When the order reaches the vendor it is assigned to one concrete book which matches the criteria of the order. If a commodity is not available in the stock, then an order is set up and sent to another vendor which has the commodity in its catalog. When all the commodities are in stock, the order is ready for shipping.

(4) Contract violations. A contract violation on the seller's part is reported by the buyer. Having the contract and order available in a computer systems archive for the buyer to access, he or she can report the violation in the system's interface, such as a web-interface. The order is now marked as problematic. This generally requires human intervention. Certain classes of problems may be handled automatically. This problem is, however, outside the scope of this domain analysis.

(5) Returning purchases. As the contents of contracts are handled by the system, the buying party specifies his desire to return an order using an interface to the system, such as a web-interface. The complaint is then either handled by the computer, if the task has a standard solution, or it is handled by human intervention. The human handling is, however, outside the domain of this domain analysis.

Summary. The tasks which automation is considered further in this domain analysis are: (1) The compilation of the catalog and keeping it up to date, (2) handling the orders and ensuring their correctness and (3) processing the orders. (4) Contract violations and (5) returning purchases are not considered in detail. The analysis of the commerce domain resulted in this list of possible electronic automations. Let us now analyze the automations further.

3.1.3 Internal structure of an electronic catalog

How a party wants to present its commodities is dependent on its business profile. There are many valid models which are equally computer processable. Since this document is concerned with a generic e-commerce system, a generic way of presenting commodities is considered.

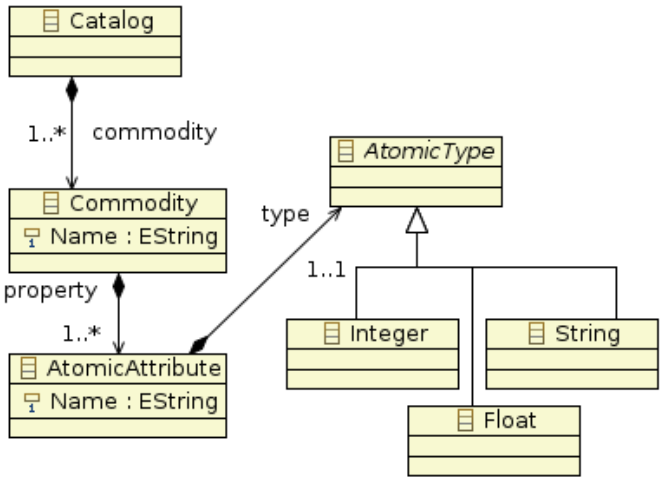

Figure 3.3: How commodity-types may be constructed

Another dimension of variation is the level of refinement. This design document considers a refinement level which is similar to the level chosen by available open source e-commerce systems[5], as they are available for study.

Commodities

Most e-commerce solutions specify what information commodities, often represented by rows in a database, can contain. The problem with this is that all commodities must contain a union of all interesting attributes, or some of the attributes serve several purposes. When having specific attributes in a commodity, adding a new attribute is a painful operation as the administration-interface of the commodities must be updated to support the new attribute.

An exception to this is the e-commerce framework Magento[6] which let users add and remove attributes of commodity-types. The domain analysis presented here suggests a similar solution: To move up a level and specify how the commodities and their attributes may be constructed. This is modeled in figure 3.3. In this figure we see the catalog and commodity concepts from earlier in this domain analysis. The commodity concept is now extended by being able to have several attributes. Each of these attributes is one of a set of types. The idea is that each type has an editor associated with it. Hence, by adding a new attribute to a commodity, the administration view of it can be updated accordingly. The attributes are qualified as atomic. The reason is that structured types are reserved for domain concepts. The purpose of the attributes is to add information to the commodity, not structure.

Categories

As the number of commodities in a catalog grows large, it is helpful with some sort of hierarchical structuring of the different commodities.

[5]Magento is the most feature-rich. But, systems with essentially a subset of the features of Magento have also been studied: Interchange, osCommerce, Zen Cart, VirtueMart

[6]http://www.magentocommerce.com/media/tour/magento-tour-admin/view

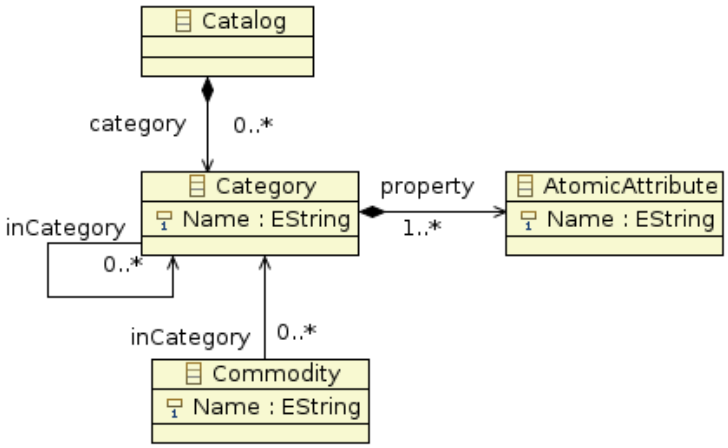

Figure 3.4: Categories

In many stores commodities are grouped into categories. A category is "a collection of things sharing a common attribute."[7] This is a loose requirement, which allows for many different categories and allows commodities to belong to several categories. It also allows a category to be in another category. In computer terms, the structuring into a hierarchy of categories allows the user of a catalog to search through a tree instead of a list. The whole structure of categories is a directed acyclic graph which topological sort is a tree.

An example of a traversal for someone looking to buy a couple of earplugs for his or her Sony Ericsson W800i mobile phone is: Audio&Pictures > Headphones > earplugs. This gives a wide array of different earplugs for many different mobile phones. A more productive search is: "Telephony > Mobile Sony Ericsson > Phones > W800i > W800i accessories". It leads the user to a narrower search, and the earplugs are found faster.

How then to integrate this new concept into the domain model? The same reasoning goes for categories as for commodities. Let the user configure the attributes of a category, then have an administration view automatically generated for this particular category.

Figure 3.4 shows the extension of the domain model with categories. Again we have the catalog concept first in the diagram. The catalog can contain a number of categories. The categories can each have their own set of attributes, just like the commodities. We also find the commodity concept here. As we can see, a commodity can be in a category and a category can be in another category.

Composites

Many commodities consist of other commodities. Computers consist of a screen, some memory, a CPU etc. Popular mobile phone-offers are bundles with a mobile phone and a subscription. The concept which underlies this is *composition*.[8]

A composite is something "consisting of separate interconnected parts."[9] The parts may or may not be sold separately. A composite is also composable from other composites. I.e. a composite may be a part itself.[10]

[7] WordNet

[8] The system does not support for example selling a t-shirt with configurable color, size, design and printing.

[9] WordNet

[10] See the Whole-Part pattern in [BMR+96].

It is very common to have several choices for one of the parts of a commodity. For example, one might offer a choice between several CPUs for a computer. One approach is to have the choices simply listed for each part. This is quite restrictive and cumbersome. If several composites have several alternatives, adding a new choice is a repetitive task. A solution within the domain concepts discussed this far is to point to a category, and let the user choose any member of this category. If this is possible, then the model is prone to allow you to set up composites where the parts do not fit together. What is needed is to ensure interconnection. Interconnection means that the parts of a composite work together as a whole.

General commodities. In order to secure this, a more restrictive form of category is needed; a generalization. A general commodity is a commodity definition which covers many concrete commodities according to what characterizes them. For example, there are countless mp3-players, but all share a common thing of being mp3-players. Therefore a general commodity called mp3-player can be defined. All other the mp3-players can then have this general commodity as their super-type. One important characterization of a general commodity is its interfaces. In order to be a subtype of a general commodity, a commodity must support all its interfaces. Hence, if a part of a composite is any commodity fitting a generalization and the generalizations are all interconnectible through their interfaces, then the model does not allow the buyer to combine in-combinable parts. The notion of type and subtype used here is according to the Liskov substitution principle [LW01].

Bundles. A bundle is "a collection of things wrapped or boxed together."[11] Bundles are different from general commodities in that they do not have to consist of related parts. It is often the case for webshops that certain products are sold together for different reasons than their interoperability. Bundles can also contain a choice of any member of a category, or a choice between one of a set of commodities. Neither of these sets requires that the commodity-alternatives are related to each other by any important reason. For a deeper look at which requirements requires the differentiation between bundles and general commodities, see Appendix B.

Interfaces. Inter-connectivity is modeled by interfaces. The notion of interface is taken from hardware where two pieces of hardware usually is connected by supporting the same interface. For example, a couple of headphones have an interface called a male mini-jack. This interface is supported by most music equipment which has a female mini-jack. To model this, a catalog contains the two interfaces, the male and female mini-jacks and commodities such as an mp3-player and a couple of headphones have one of the two interfaces. The two interfaces are modeled as being inter-connectible.

Figure 3.5 contains this part of the domain model. In the center of this diagram we find the commodity concept. It now has two new sub-types. A general commodity means a non-concrete commodity. For example, an mp3-player is a general type while an iPod is a specific type of mp3-player.

Having these concepts, one can define such things as a desktop computer. A desktop computer consists of a computer-box, a keyboard, a mouse and a monitor. All these

[11] WordNet

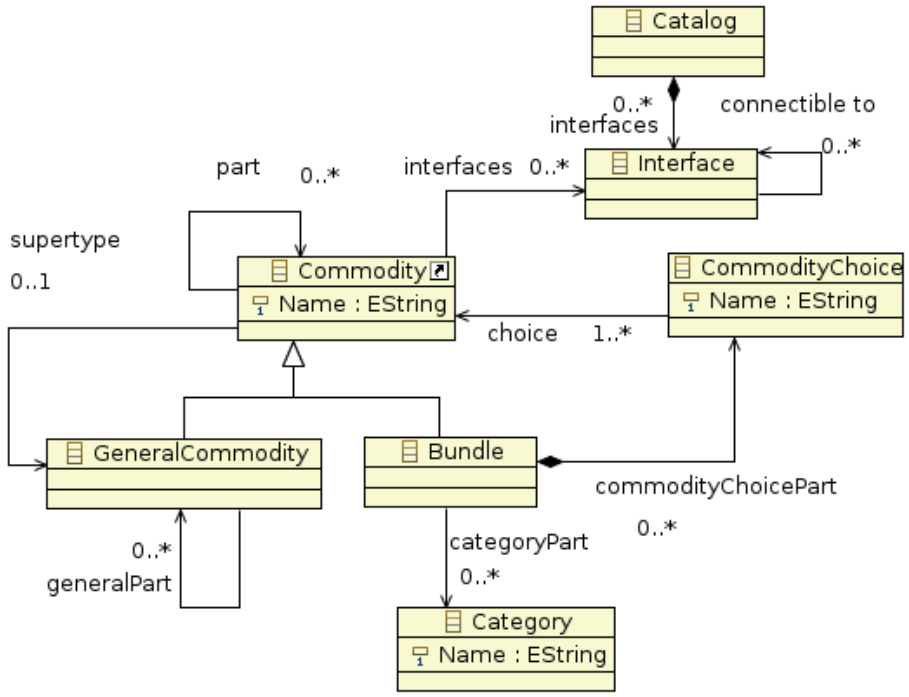

Figure 3.5: Composites

parts are general parts. If what a customer wants is a desktop computer he can start out with selecting a desktop computer, then the system will guide him or her in configuring it with sub-types of the general types. They are guaranteed to inter-operate because the relationship between general commodities and commodities is a sub-type relation.

3.1.4 Static semantics of the catalog

The static semantics identified for the concept of a catalog are as follows. (1) A commodity is general if at least one of its parts is general, and all parts are inter-connectible with their interfaces. (2) Bundles must have at least one part, if not it is nothing. (3) Cycles in the generalization hierarchy are not allowed. (4) Parts of a commodity, where a selection between alternatives is possible, must have the same or a subset of the alternatives of the same part in its generalization. (5) The choices of a choice-set must all be different. (6) A commodity must have the same or more general interfaces than its generalization. (7) The parts of a non-general commodity and of a bundle cannot be a general commodity. (8) A bundle cannot have interfaces. (9) A cycle in the category membership is not allowed.

3.1.5 Electronic ordering

Since the purpose of this section is to analyze the automatization of the well known process of buying something, let us think about our e-commerce system as a simulation of the actual procedure. We have our commodities and our buyer and seller. They interact in a specific way.

Modeling such dynamic behavior is a well-known problem in the gaming industry. The well known solution is using finite state machines (FSMs). A popular framework for game-development is the Unreal Engine. It indeed solves this problem with its own DSL,

namely UnrealScript [Swe08]. The usage of FSMs is explained in the system references [Swe08] as "a natural way of making complex object behavior manageable." Hence FSMs are chosen as an interesting way to model the behavior of commodities, customers and salespersons.

FSMs are a bit limited and suffer from the state explosion problem. In order to remedy this statecharts [Har87, BRJ05] are used instead, extended with some domain specific constructs. UML state diagrams implement most of the concepts found in statecharts [BRJ05]. Statecharts were chosen because they are good for modeling reactive systems[12] and because they are used successfully in modeling the behavior of entities in Unreal Engine 1, 2 and 3's Unreal script [Swe08], which is used for many sophisticated modern computer-games.

Surely, e-commerce systems require more complex implementation and more expressive constructs than those found in statecharts. The idea is to have statecharts on top of operations and other general purpose code-blocks. The statecharts dictates when and how these are called. What is called and used by the statecharts cannot alter the overlying behavior implemented in the statechart, so the integrity of what is specified on top of the general purpose constructs are preserved. Statecharts can use classes for expressing signals and operations to express actions.

In UnrealScript each entity in the simulation has its own statechart. In e-commerce, this means that the customer, the salesperson and the different commodity types get their own statechart to describe their behavior.

Example. The following is an example of how statecharts can be used to model a trade. When a buyer first enters a store, the salesperson knows nothing about him. The customer is in the state of an anonymous buyer. If the buyer requests to order a bottle of wine, the contract of wine requires the buyer to be over the age of majority (at least in Norway). Since the seller does not know the age of the customer, age verification is required before the buyer may purchase the good. When the age is verified the seller knows that the buyer is over the age of majority. This is a transition into the state "known buyer, over the age of majority". The transition is triggered by showing an ID. In this new state the buyer can buy wine in this store. Hence the domain concept of an *ability* can be attached to a state. In this case the ability to purchase wine. As the customer purchases the wine, the wine transitions from the state "in stock" to "owned". This transition is triggered by the purchase.

Later that week the buyer comes back to the store. He had opened the bottle of wine and found it to be refermented. He demands a replacement bottle. The seller looks up the purchase in the system, finds the wine to indeed be purchased by this person from his store. He knows this since the wine is in the state of owned in his systems. He registers the wine as refermented and provides a new bottle. □

Why using statecharts is indeed a good idea? A customer has a certain finite number of classifications the seller can put on him or her. The same can be said about a bottle of wine. Gaining a new classification requires a certain sequence of events to happen. The classifications are the states and the event-sequences are the transitions. The state of an entity is a unique configuration of its attributes. There might be a vast amount of states and transitions for some entity. Many states are essentially the same. Whether a good is sold to x or y, the good is still sold. This is the difference between a concrete and an abstract state. The transitions between abstract states are the abstract transitions.

[12] [BRJ05] page 339

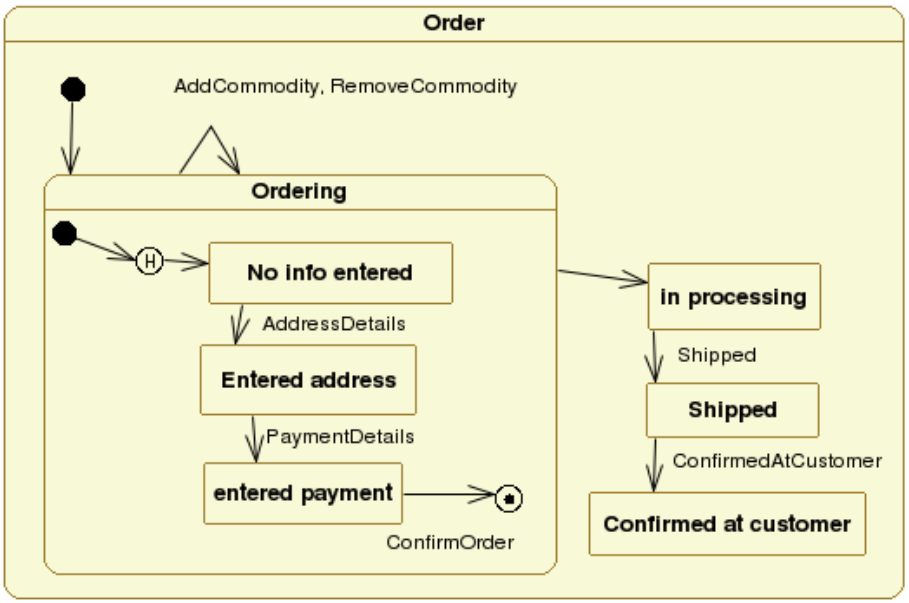

Figure 3.6: Order Behavior

Abstract states are equivalence classes over the concrete state space, i.e. they do not overlap.

Example. Figure 3.6 is a possible concrete model of the behavior of an ordering process. It shows the statechart for a simple order. When an order is created, it enters the state of "Ordering". In this state the customer can add and remove commodities from the order. When the customer is ready, he or she can perform a checkout. The checkout consists of adding address information, payment information and confirming the order. During this process the customer may add and remove commodities. After the customer has confirmed the order, it is no longer possible to add or remove commodities. Signals for when the order has been shipped and confirmed at the customer are gotten by the system from external system such as packaging and shipping systems.

After the purchase is confirmed the buyer might have some further requests of complaints. This is expressible by extending the ordering-statechart. □

The customer statechart specifies what a customer is allowed to do. Within the transitions is general purpose code for doing the actual actions. But the statechart is the top-level architectural element which determines what a customer is allowed to do. Let us look at an example of a customer statechart.

Example. Figure 3.7 shows an example of a customer statechart. Initially the customer is virtually outside the electronic shop. Entering the shop is typically visiting a web-site for the first time. If a certain customer is not welcome to the site, then he or she can be denied entrance to the site here. Once inside a shop, the user has access to the catalog, is allowed to order and allowed to register for some special privileges such as administrator rights and for example one click shopping. Transitions in the customer statechart may send signals to other statecharts in the system. This is the way which the internal statecharts of the system interact with each other. For example, on a request from a customer, a salesclerk must do various things including interacting with various commodities and the

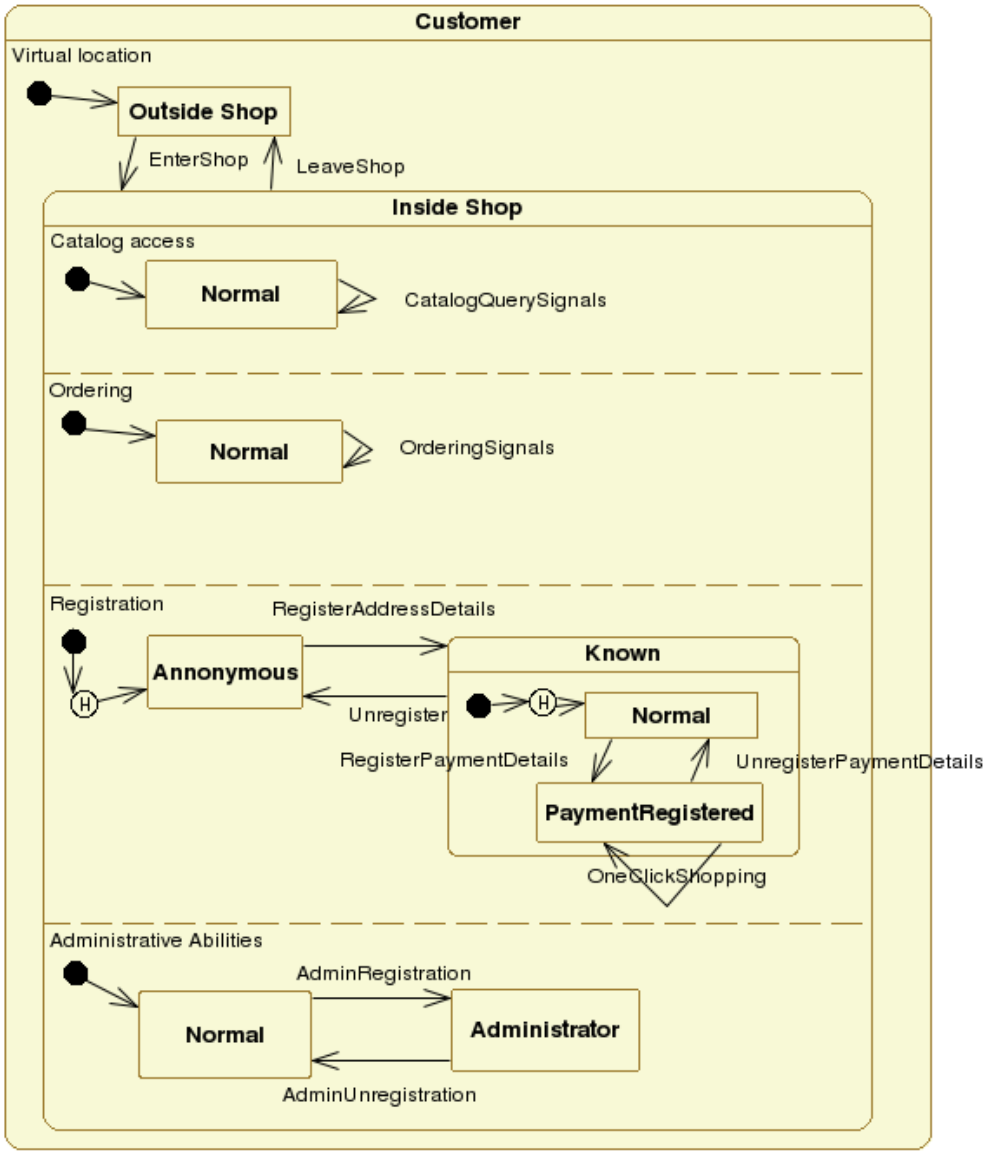

Figure 3.7: Customer Behavior

ordering system. The salesclerk does this by sending signals to the behaviors of these other entities. □

More detailed semantics

Varying interfaces. A Java interface consists of several methods. The list of methods is always fixed. If one looks at the signals which will trigger a transition in a statechart, this is a list of signals. As the state of a statechart changes, this list of signals which can cause a transition also changes. A signal is constructed by giving it a list of parameters. If the signals are seen as method calls, a statechart can be seen as a variable interface, allowing the list of methods to change according to which state the statechart is in. A statechart interface requires two methods defined in a Java interface: One method for returning a list of signals which will trigger a transition and a method for sending a signal to the statechart. All the entities described with statecharts get behavior defined with such a variable interface.

Conditional transitions. A statechart transition should be atomic. If a transition is implemented as a transaction, a rollback is possible if it fails. But certain transitions are too long. They should return to the customer and inform him or her that the action is in progress. Hence wait then report the error on return or inform that the transition is in progress and that the error is reported when done. This is important to know when designing the statecharts. One does not need a choice point if something has failed. If something does fail, the transition is not performed, the action is rolled back and the user is informed of the error. On the other hand, if a transition takes really long it should have a state in which the result is awaited. The user can then use the rest of the system until the result is received.

3.1.6 Structure and behavior integration

The following is a discussion of the semantics and binding between the static commodity model and the dynamic statechart model.

In the e-commerce system described here there is one user-object per user of the system. This user object's life cycle is described by a statechart. The system also contains one object per commodity, whose life cycle is also described by a statechart. When a request is made to the system, the system builds a signal from the request and checks if the signal is accepted by the user-statechart of the user making the request. If it is rejected, the signal is thrown away. If it is accepted, the user-object passes the signal to the commodities for which the request was intended. This construction is meant to provide authentication and authorization. The authentications are transitions in the user-statecharts. The user is then authorized to perform certain actions on certain commodities.

Using statecharts with signals as input makes sure the input is correct. [Chr09] calls improper input validation "the number one killer of healthy software". The lifetime of user objects is modeled using a domain specific version of statecharts. If the signal received by the user object results on a transition in its statechart, the signal is passed on to the current view's controller based on a view ID supplied with the request. The specification of which signals can be passed on is called the *abilities* of the user. This is the authorization mechanism of the framework. A user gains abilities by sending a valid authorization signal to the user object.

The signals may come to the state machine of some commodity, if the request requires the change of state of some commodity. Since there might be millions of commodities in a web-shop, these commodities are accessible through a catalog. Whether the commodity representations actually exist within the catalog or are created dynamically from a database, is up to the catalog implementation.

This document does not prove that the statechart formalism is the best abstraction to use for the dynamic behavior of commodities nor salespersons. The statechart formalism is used here as a handy tool for dealing with complexity.

One object per entity. The idea is that an actual commodity is represented in a system with an object. This object contains an instance of the statechart which describes its behavior. One object per commodity makes sense for goods, but what about services? Do services such as a haircut exist in a limited amount? Yes they do. Even if a haircut is performable an unlimited amount of times, each actual haircut must allocate a person to perform the job and a time slot. The person and the time slot are the existent which the object refers to.

3.1.7 What is general and what is specific

The purpose of the experiment described here is to make a solution to a number of related problems. The problem here is the automation of some aspects of commerce. The question is therefore, which parts of an e-commerce system can be made general, and what is specific for each concrete solution?

The first part, which certainly fits into the general part, is the mechanisms for setting up a catalog. Such a catalog contains commodities which are structured in a certain way. Let us call this part the Catalog-Composition part. The user of this subsystem fills in his or her specific commodities and their structure. Then such a system can generate a browse-able catalog, both for customers and other computer systems. The structure must adhere to the domain model, and to the static semantics of the model.

The second part is the description of the trade-process. Different stores have different policies. But all stores have a policy. It is never OK for a customer to come and pick up the commodities for free. In order to ensure a proper procedure in a physical retail store, a salesperson ensures that the concrete policies of the store are followed by interacting with the customer. What are general here are the facilities to program the policies as statecharts for each entity acting in a trade.

The statecharts are used to describe the behavior of orders, customers, salesclerks and commodities of all types. Since there is such a wide range of things this behavior-model describes, this part of the system is called the Entity-Behavior part. When using it, one defines the concrete behaviors of the entities in the system, such as the customer, the salesclerks and the different commodities. This enables a customer to compile an order and to perform a checkout.

Actions can be defined for statechart transitions. The nature of these actions was not studied in the domain analysis. Therefore, actions are implemented, not using domain concepts, but using a general purpose language. Only the high-level structure is specified using Entity-Behavior, and not the details of the trade process. The details are described using a general purpose language in the actions of the transitions.

3.2 DSL design

Having described the domain analysis, this section aims to show the design of a DSL for the e-commerce domain. The DSL is simply called the E-Commerce Language. The point about the DSL is to allow the user of the DSL to create a concrete e-commerce solution for his or her company. A DSL for the two parts, Catalog-Composition and Entity-Behavior, are described separately. Their integration into one language, the E-Commerce Language, is described in the final subsection of this section.

3.2.1 Catalog-Composition Language

The DSL for the Catalog-Composition part is called the Catalog-Composition Language (CCL). The DSL is implemented in this experiment with textual syntax described with EBNF and regular expressions. Listing 3.1 shows the grammar of identifiers and values in the Catalog-Composition Language.

Listing 3.1: CCL values and identifiers

```
Identifier : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
Value : '"' .* '"';
```

Catalogs

The top-level constructs in the language are catalogs. A catalog contains four things: Commodities (figure 3.3), which may also be general commodities and bundles (figure 3.5), categories (figure 3.4), interfaces (figure 3.5) and commodity choice sets. The grammar in listing 3.2 describes the syntax for creating a new catalog. To begin specifying a catalog, enter the keyword Catalog followed by the catalog name. Everything following the catalog declaration is in the catalog. Only one catalog is supposed to be defined per source code file. Therefore, the catalog does not require a keyword for ending the definition.

Listing 3.2: CCL catalogs and commodities

```
catalog : 'Catalog' Identifier catalogcontent*;
catalogcontent : commodity | category | choiceset | interfaceDef;
commodity : concretecommodity | generalcommodity | bundle;
```

Semantics. With this declaration, the user gets an electronically browse-able catalog. The browsing experience is determined by the amount of internal structure given to the commodities by the user when specifying the contents. Catalog administration is also possible. The user can add commodities of the types specified, and assign different values to the attributes. The user can also execute DSL-code to modify the catalog. This is done by overwriting an identifier with a new definition. At any point during run-time, it is possible for the user to choose to export the catalog specification to DSL-code.

Example. An example of a computer-webshop is explained as an example throughout this specification. Listing 3.3 shows the code which creates a catalog for the webshop. The following subsections goes through each of the catalog items in turn.

Listing 3.3: CCL definition of a catalog

```
Catalog ComputerShop
```

□

Simple commodities

Commodities contain attributes (figure 3.3), can be a member of categories (figure 3.4) and can have a super type, parts and interfaces (figure 3.5). Listing 3.4 shows the grammar for defining commodities and their contents.

Listing 3.4: CCL grammar for defining commodities

```
concretecommodity : 'Commodity' Identifier supertype? incategories?
    interfaces? content*;
incategories : 'in categories' ':' Identifier (',' Identifier)*;
supertype : 'supertype' ':' Identifier;
content : part | attribute;
attribute : Identifier ':' atomictype ('=' Value)?;
atomictype : 'Integer' | 'String' | 'Date' | 'Float' | 'Double';
part : Identifier ':' Identifier;
interfaces : 'interfaces' ':' Identifier (',' Identifier)*;
```

Semantics. What the user of the DSL creates using the grammar in listing 3.4 is a commodity type, such as an iPod nano. A specific iPod nano is an instantiation of this created type. The instantiation assigns values to the unassigned parts of the type. Actual commodities are usually added at run-time to the system. The number of actual products available for each type usually shows up in the catalog as the number of products in stock. When a customer buys a commodity, the commodity object's statechart is put into the state of ownership, and the number of commodities in stock is thereby reduced by one.

Example. Listing 3.5 shows the DSL code for adding four commodities to the catalog of the running example. The super types and the categories are defined later.

Listing 3.5: CCL code example defining commodities

```
Commodity Inspiron530
  supertype : ComputerBox
  in categories : Dell
  interfaces: RgbFemale, UsbFemale, UsbFemale, UsbFemale
  price : Integer = "3490"
Commodity G900HDA
  supertype : RgbMonitor
  in categories : BenQ
  interfaces: RgbMale
  price : Integer = "1095"
Commodity KB1400SNO
  supertype : UsbKeyboard
  in categories : Trust
  interfaces: UsbMale
  price : Integer = "79"
Commodity RX250OpticalMouse
  supertype : UsbMouse
  in categories : Logitech
  interfaces: UsbMale
  price : Integer
  image : String = "http://www.komplett.no/images/121313.jpg"
```

□

Categories

Categories can reference other categories and contain attributes (figure 3.4). To define a category, start with the keyword Category followed by the name. Then write a list of categories in which the category is a member, followed by the attributes with data about the category.

Listing 3.6: CCL grammar for defining a category

```
category : 'Category' Identifier incategories? attribute*;
```

Semantics. A category is a structural building block as well as an informative structure. Categories are typically used to build a tree which can be browsed by a customer in search for some particular commodity. The user is, for example, presented the top-level categories, which are not members of any category. He or she can then expand the category of choice to get the member categories and optionally see a selection of the commodities in the category or sub-categories. The information in a category can also be displayed to the customer.

Example. The computer store sells products from the manufacturers Dell, BenQ, Trust and Logitech. In addition, the store wants to show certain products on the front-page, therefore a category is created for these products in listing 3.7.

Listing 3.7: CCL code example defining categories

```
Category Manufacturer
Category Dell
  in categories : Manufacturer
Category BenQ
  in categories : Manufacturer
Category Trust
  in categories : Manufacturer
Category Logitech
  in categories : Manufacturer
Category FrontPage
```

□

General commodities

General commodities are modeled in figure 3.5. They are a subtype of commodity where some parts are general. They may also be super types of other commodities. The parts of a general commodity must be inter-connectible.

Semantics. General commodities serve a similar function to categories. They can serve as parts to composites, but also be a guide to the customer. Sometimes, a customer simply wants the cheapest mp3-player. He or she can then select the general commodity of an mp3-player and get a sorted list of commodities which indeed are functional mp3-players. It can also serve as a part. Parts are a guide to subtypes. Subtypes must contain the same number of types with an equal or subtype type. The attributes of a general commodity are inherited for a subtype. When a buyer selects a general commodity, say a desktop computer (figure 3.9), he or she can choose to specify the parts themselves among the subtypes, choose to have the system select the cheapest parts or let the system choose recommended commodities among the subtypes parts given a certain price.

Listing 3.8: CCL grammar for defining a general commodity

```
generalcommodity : 'GeneralCommodity' Identifier supertype? incategories?
    interfaces? content*;
```

Example. Listing 3.9 shows a definition of a desktop computer. It is a general commodity which consists of four parts: A monitor, a keyboard, a mouse and a box which contains all the components of a computer. Defining a general commodity type is done by typing GeneralCommodity followed by the name of the type. Inside the type, we have four lines specifying parts. Each part has a part name, followed by a colon and the type of the part. Each part type is a general commodity. In practice, each of the general commodities defined in listing 3.9 would specify more details. But the example shown only specifies the essentials for the example.

A concrete desktop computer can now be defined. Listing 3.10 shows the definition of it where the parts of the DesktopComputer type are refined to more specialized types.

Listing 3.9: CCL implementation of Desktop Computer and its parts

```
GeneralCommodity DesktopComputer
  box : ComputerBox
  rgbMonitor : RgbMonitor
  usbKeyboard : UsbKeyboard
  usbMouse : UsbMouse

  price : Integer

GeneralCommodity ComputerBox
GeneralCommodity RgbMonitor
GeneralCommodity UsbKeyboard
GeneralCommodity UsbMouse
```

Listing 3.10: CCL code for defining a concrete subtype of desktop computer

```
Commodity DX2400
  supertype : DesktopComputer
  in categories : Dell, FrontPage

  box : Inspiron530
  rgbMonitor : G900HDA
  usbKeyboard : KB1400SNO
  usbMouse : RX250OpticalMouse
```

□

Bundles

Bundles are modeled in figure 3.5. Bundle is a subtype of commodity. Bundles can have parts which are not inter-connectible. Subtyping hence does not make sense for bundles. Bundles may contain choice-sets. These are simply lists of commodities where the customer must choose one. Bundles can also have parts were the user chooses one of the members of a category. Listing 3.11 shows the grammar for defining a bundle.

Listing 3.11: CCL grammar for defining a bundle and a choice-set

```
bundle : 'Bundle' Identifier incategories? content*;
choiceset : 'ChoiceSet' Identifier part*;
```

Semantics. A bundle is presented to the user as a collection of different commodities sold together, usually at a lower price than the sum of the parts. If some of the parts are non-specified such as a category part or a choice set, then the customer, browsing the catalog in a webshop, typically get radio-buttons to select the desired commodity, or some other fancy AJAX mechanism for selecting the desired commodity among the members of a category.

Example. Listing 3.12 shows an example of a common commodity found in computer stores. A keyboard sold together with a mouse. The owner decides to advertise the product on the front page. In addition, the owner decides that the customers can choose between some freebies.

Listing 3.12: CCL example of a bundle definition

```
Bundle KeyboardAndMousePack
  in categories : FrontPage
  usbKeyboard : KB1400SNO
  usbMouse : RX250OpticalMouse
  freebie: Freebies
  price : Integer = "150"
```

□

Interfaces

Interfaces are modeled in figure 3.5. An interface is a point of connection between two commodities. For example, a cell phone and a subscription have a point of connection, for example the GSM net. A cell phone is a good while a subscription is a service; interfaces are not limited to physical products. An interface consists of a name and a specification of which interfaces can be connected to it. Listing 3.13 shows the grammar for defining an interface.

Listing 3.13: CCL grammar for defining an interface

```
interfaceDef : 'Interface' Identifier connectible? attribute*;
connectible : 'connectible with' ':' Identifier (',' Identifier)*;
```

Semantics. Interfaces are a good way for a customer to learn which other commodities he or she can use with a certain other commodity. Usually, when buying a product the customer want to see accessories. Accessories must be usable in relation with the product somehow. This is the role of interfaces. The user can also ask the system how to connect two certain products, and the system can do a search and come up with a proper converter.

Example. Listing 3.14 shows the interfaces for USB and RGB signals which connect the computer box, mouse, keyboard and the monitor from the previous examples.

Listing 3.14: CCL example of interface definition

```
Interface UsbMale
Interface UsbFemale
  connectible with: UsbMale
Interface RgbMale
Interface RgbFemale
  connectible with: RgbMale
```

□

3.2.2 Entity-Behavior Language

The DSL for the Entity-Behavior part is called the Entity-Behavior Language (EBL). This DSL is implemented in this experiment with textual syntax described with EBNF and regular expressions. A statechart-like language is probably nice to have as a graphical DSL. The problem is, however, the complexity of making one and the immature tools for programming with graphical syntax. In this experiment, therefore, a textual concrete syntax is presented.

The language is explained using an example followed by an explanation of the language grammar. The example is the behavior of an order as shown in figure 3.6. Listing 3.15

shows the DSL-code for the Order state. The xor-state is defined using the keyword XorState followed by the name of the state. What follows are substates which are defined by writing the name of the substate, followed by a colon, followed by the state-type name. Ordering, InProcessing, Shipped, ConfirmedAtCustomer are the substates of Order. Each substate can have outgoing transitions which are defined using the $->$ symbol followed by the target state name, followed by a forward slash and the trigger signal. A special type of signal is the epsilon signal, which is triggered if the source state transitions to a final state. It is also needed if a history state has never been entered before to define the default target state. An ability belongs to a state. It is defined using the * symbol followed by a trigger signal. It can be seen as a self-transition which leaves and re-enters the same state, which gets a transition-action executed.

The behavior of listing 3.15 is what is commonly known as the checkout procedure of a webshop. A customer is allowed to add and remove commodities from the order until he or she leaves the Ordering state. Once he or she has left this state, adding and removing commodities are no longer possible. The Shipped signal is got by the Order once the commodities have been shipped to the customer. And finally, once the shipping-company confirms that the user has picked up the commodities, the ConfirmedAtCustomer signal is received.

Listing 3.15: EBL example behavior: Order state part 1

```
XorState Order
  Ordering : OrderingState
    -> InProcessing / epsilon
    * AddCommodity
    * RemoveCommodity
  InProcessing : InProcessingState
    -> Shipped / Shipped
  Shipped : ShippedState
    -> ConfirmedAtCustomer / ConfirmedAtCustomer
  ConfirmedAtCustomer : ConfirmedAtCustomerState
```

There is behavior within the OrderingState state is shown in listing 3.16. Notice that both the abilities AddCommodity and RemoveCommodity are available while one is in this state. ConfirmOrder is the signal which finishes off the ordering state, and then its abilities are no longer usable. The OrderingState elaborates on the checkout process. The user can here enter all his or her address details and payment details. If the action processing these signals fails, an exception is thrown, and the transition is not performed. This typically happens if the payment details do not have sufficient money available for the purchase. Defining behavior of the ordering process such as this makes the checkout procedure clear and accessible for the programmers and for the owners of the webshop system.

Listing 3.16: EBL example behavior: Order state part 2

```
XorState OrderingState
  HistoryState : ShallowHistoryState
    -> noInfoEntered / epsilon
  noInfoEntered : NoInfoEntered
    -> enteredAddress / AddressDetails
  enteredAddress : EnteredAddress
    -> enteredPayment / PaymentDetails
  enteredPayment : EnteredPayment
    -> final / ConfirmOrder
  final : FinalState
```

Since the DSL only specifies the behavior in terms of states and transitions between these, some general purpose language must be used to program the actions of the transitions. Since Java is the target code, the actions are also specified in Java. Signals have corresponding classes with the same names as the signals. Transitions have associated Command objects[13] which are executed on transition. They have a name consisting of the from-state and the trigger-name, since this is unique for every transition.

3.2.3 Entity-Behavior Language Grammar

Listing 3.17 shows the grammar of the Entity-Behavior Language. A behavior definition, called a statechart in the grammar, consists of several states. These states can be xor-states and and-states. Both state-types can inherit from other states and have substates. Each substate can define outgoing transitions and have assigned abilities. The transitions can have constraints. An in-constrain means that it is a requirement that one is in the mentioned state of the transition to occur.

Listing 3.17: Complete grammar of the Entity-Behavior Language

```
statechart : state+;
state : xorState | andState ;

xorState : 'XorState' Identifier inherit? subState+;
andState : 'AndState' Identifier inherit? subState+;
inherit : 'extends' Identifier;
subState : Identifier ':' Identifier option*;

option: transition | ability;
transition : '->' Identifier '/' Identifier inconstraint? notinconstraint?;
inconstraint : 'in' Identifier;
notinconstraint : 'notin' Identifier;
ability : '*' Identifier;

Identifier : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
Whilespace : ( '\t' | '\n' | ' ' | '\r' | '\u000C' )+;
```

3.2.4 Integration into the E-Commerce Language

The E-Commerce Language consist of programming a catalog using the Catalog-Composition Language, and then program behaviors of different entities in the system using the Entity-Behavior Language, such as commodities, customers, salesclerks and orders. In order to get a running shop, various other small things not covered by the E-Commerce Language must be programmed. These aspects are not discussed in detail in this thesis as the two described parts were considered adequate to analyze the problem statements.

The framework design in section 3.3.3 specifies the run-time system of the E-Commerce Language in more detail.

3.2.5 Implementation

The implementation of the E-Commerce Language consists of a combined lexer and parser written using ANTLR, a code-generator written in Java and a static-semantic analyzer.

[13] An application of the command pattern of [GHJV95]

The parser

Listing 3.18 shows the implementation of the lexer and parser for the Catalog-Composition Language. This is the same grammar as the one shown bit-by-bit in the description of the DSL. The first line gives a name to the grammar. The options parts are used to tell ANTLR it should generate an AST automatically. Information required to generate an AST is within the grammar. The circumflex symbol is used to define a tree node. The $->$ symbol is used to describe which nodes should be included in the AST tree node.

Listing 3.18: Ccl.g: ANTLR parser implementation for CCL

```
grammar Ccl;

options{ output = AST; }

Part : 'part';
Attribute : 'attr';

catalog : 'Catalog'^ Identifier catalogcontent*;
catalogcontent : commodity | category | choiceset | interfaceDef;
commodity : concretecommodity | generalcommodity | bundle;

concretecommodity : 'Commodity'^ Identifier supertype? incategories?
    interfaces? content*;
category : 'Category'^ Identifier incategories? attribute*;
generalcommodity : 'GeneralCommodity'^ Identifier supertype? incategories?
    interfaces? content*;
bundle : 'Bundle'^ Identifier incategories? content*;

choiceset : 'ChoiceSet' Identifier part*
  -> ^('ChoiceSet' Identifier part*);

interfaceDef : 'Interface'^ Identifier connectible? attribute*;
connectible : 'connectible with' ':' Identifier (',' Identifier)*
  -> ^('connectible with' Identifier Identifier*);

content : part | attribute;
incategories : 'in categories' ':' Identifier (',' Identifier)*
  -> ^('in categories' Identifier Identifier*);
interfaces : 'interfaces' ':' Identifier (',' Identifier)*
  -> ^('interfaces' Identifier Identifier*);
supertype : 'supertype' ':' Identifier
  -> ^('supertype' Identifier);
attribute : Identifier ':' atomictype ('=' Value)?
  -> ^(Attribute Identifier atomictype Value?);
atomictype : 'Integer' | 'String' | 'Date' | 'Float' | 'Double';
part : Identifier ':' Identifier
  -> ^(Part Identifier Identifier);

Identifier : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
Value : '"' .* '"';

Whilespace : ( '\t' | '\n' | ' ' | '\r' | '\u000C' )+ { $channel = HIDDEN;
    };
```

The implementation of the lexer and parser for the Entity-Behavior Language is very similar to the implementation of the Catalog-Composition Language. Consult Appendix A for the implementation source code.

The code generator

Since the E-Commerce Language AST is so simple, the code generator is implemented using Java and the Velocity Template engine. The target of the code generation is an instantiation of the E-Commerce Framework described in subsection 4.3.2 and subsection 4.4.2. The E-Commerce Framework is an implementation of the run-time system for the E-Commerce Language.

Parts of the code-generator are shown in listing 3.19 and in listing 3.20.[14] The generator uses the AST produced by the parser shown in listings 3.18, which is given as a parameter to the generateCode operation. The generated code is returned as a list of source code files. The code generator works by filling in a tree-structure which is used by the template engine. In the listing, the code for generating commodities and their attributes are shown. The rest of the code generator is implemented in a similar fashion.

One operation requires some explanation. findNamedElements searches a tree node for children of a particular type. For example, a commodity has attribute-children which are used to generate the code for the attributes. This operation helps with searching through the children in search for, for example, attributes.

Listing 3.19: Parts of the code generator of CCL

```
...
public class CclCodeGenerator implements CodeGenerator {
  ...
  public List<SourceCode> generateCode(CommonTree ast) {
    ...
    VelocityContext context = new VelocityContext();

    // Catalog
    context.put("CatalogName", CatalogName);

    // Commodities and General Commodities
    l = new ArrayList<Map>();
    for(Tree c : findNamedElements(ast, "Commodity"))
      l.add(commodity(c));
    for(Tree c : findNamedElements(ast, "GeneralCommodity"))
      l.add(generalCommodity(c));
    context.put("commodities", l);

    // Categories
    l = new ArrayList<Map>();
    for(Tree c : findNamedElements(ast, "Category"))
      l.add(category(c));
    context.put("categories", l);

    ...
    return sources;
  }
  private List<Map> attributes(Tree c){
    List<Map> attrs = new ArrayList<Map>();
    List<Tree> as = findNamedElements(c, "Attribute");
    for(Tree a : as){
      Map ad = new HashMap();
      ad.put("name", a.getChild(0).getText());
      ad.put("type", a.getChild(1).getText());
```

[14] The code generator for EBL together with the rest of the code is found through Appendix A

```
        ad.put("init", "");
        if(a.getChildCount()>2)
          ad.put("init", a.getChild(2).getText());
        attrs.add(ad);
      }
      return attrs;
    }
    private Map commodity(Tree c){
      Map m = new HashMap();

      m.put("name", c.getChild(0).getText());
      m.put("supertype", "CommodityImpl");
      if(supertype(c) != null)
        m.put("supertype", supertype(c));
      m.put("incategories", incategories(c));
      m.put("interfaces", interfaces(c));
      m.put("attributes", attributes(c));
      m.put("parts", parts(c));

      return m;
    }
}
```

Listing 3.20: Parts of the velocity template for the target code of CCL

```
...
public class $CatalogName extends CatalogueImpl {
...
#foreach($commodity in $commodities)
  static public class $commodity.name extends $commodity.supertype{
  #foreach($attribute in $commodity.attributes)
    $attribute.type $attribute.name
    #if($attribute.init != "")
      = new $attribute.type ( $attribute.init )
    #end
  ;
  #end
  ...
  }
...
#end
}
```

Static-semantic analyzer

Static-semantic analysis is commonly done before code generation. But, in the case of this experiment, the target framework has a static-semantic analyzer which performs the static checking on the generated code. This is described in detail in chapter 4.

How to use the language in practice

In order to build an application using the E-Commerce Language, write the catalog using the Catalog-Composition Language in a .ccl-file, specify the behavior in an .ebl.file of the different commodities with the same name as the commodities or the general commodities as defined in the .ccl-file. In addition define a statechart called User, Order and

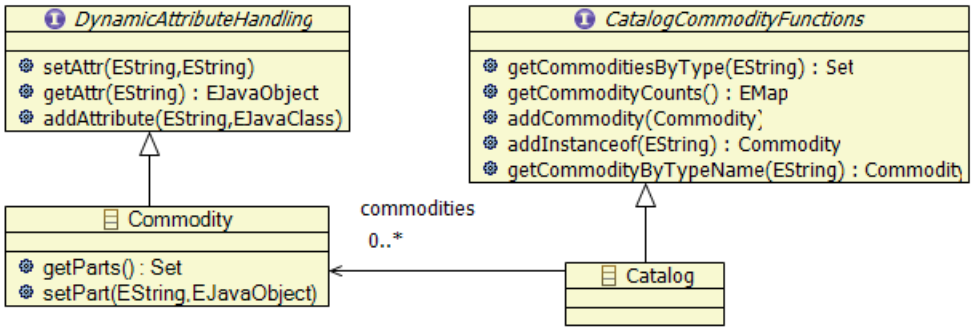

Figure 3.8: Catalog and commodity framework classes

SalesClerk. The target code contains references to classes for the statechart signal-classes and transition-command classes. These classes must be programmed with actions and contents in addition to the DSL-code. Finally, the view component of the system must be designed to fit the behavior. This will result in an operational webshop system.

Note that the view component could have been implemented such that fewer manual changes were needed after writing the .ccl- and .ebl-files. But, this is an experimental webshop system. The DSL would have to be extended in order to build a production grade webshop system. The system is operational enough for the purpose of the experiment in this thesis.

See Appendix A, subsection A.2.2 for a quick demo of a running webshop system.

3.3 Framework design

Having described the design for the DSL, it is now time to look at the corresponding framework design. The framework is simply called the E-Commerce Framework. The point about the framework is to allow the user to create a concrete e-commerce solution for his or her company.

Frameworks for the two parts, Catalog-Composition and Entity-Behavior, are described separately. Then the integration into one framework, the E-Commerce Framework is described. The E-Commerce Framework contains additional domain specific code in addition to Catalog-Composition and Entity-Behavior. The implementation of the framework is done in Eclipse using Java, UML and the Eclipse Modeling Framework (EMF).

3.3.1 The Catalog-Composition Framework

Catalogs and simple commodities

In figure 3.8 we see the commodity class. This is the class of all products and services in the E-Commerce Framework. Commodities are stored within the catalog class, which serves as facade to access commodities. The reasoning behind having a facade such as this is being able to support millions of commodities without having to have them all in memory. The catalog hides the implementation and whether the commodities are in fact objects all the time.

The catalog and the commodity classes implement several Java interfaces. The reason to place the functionality in interfaces is to have the methods related to the same aspect

of the framework in one diagram.

For the commodities we find the DynamicAttributeHandling interface. This interface contains the methods for adding and removing attributes from a commodity type. In the interface CatalogCommodityFunctions we find the methods used to query the catalog for commodities and to add commodity types to the catalog. They are intended to be used by the view of the catalog and during the construction and modification of the catalog.

Example. During the description of the framework, in this section, an example catalog is set up as an example. In order to set up a new catalog, create a subclass of the Catalog class. In the constructor of the catalog, the catalog's initial contents are constructed. Listing 3.21 shows the framework user-code for defining a catalog.

Listing 3.21: Framework definition of a catalog

```
public class ComputerShopCatalog extends CatalogImpl{
  public ComputerShopCatalog(){
    // Construct the initial catalog setup here
  }
}
```

Listing 3.22 shows the declaration of a commodity type within the constructor of the catalog. First the commodity is constructed. The parameters to the constructor are the owning catalog, the name of the commodity, and the super-type respectively. Note that the named structures, such as the super-type in this example must be defined before use. The declaration code of these things is described later. After the commodity is constructed, its internals can be set up. First add the category-memberships. Then set the attribute-values. These attributes are inherited from the super-type ComputerBox, defined later. Finally the commodity is added as a part of the catalog. The commodity is stored by the internal mechanisms of the catalog with the operation addCommodity.

Listing 3.22: Framework definition of commodities

```
Commodity inspiron = new CommodityImpl(this, "Inspiron530", "ComputerBox");
inspiron.addInCategory("Dell");
inspiron.addInCategory("Frontpage");
inspiron.setAttr("price", "3490");
inspiron.setAttr("modeltype", "Inspiron");
inspiron.setAttr("modelnumber", "530");
inspiron.setAttr("image", "insp_530_right_155x160.jpg");
addCommodity(inspiron);
```

In order to add an instance of the commodity, invoke the addInstanceof operation of the catalog as seen in listing 3.23. The catalog then sets up an instance of the commodity type. The number of instances of a commodity reflects the number of this commodity available for sale by the vendor.

Listing 3.23: Framework code for adding an instance of a commodity type

```
addInstanceof("Inspiron530");
```

□

Categories

In figure 3.9 we see the category-related class-diagram of the Catalog-Composition-model. In this diagram the Category class is introduced into the model with several interfaces used to show the relevant methods for this diagram.

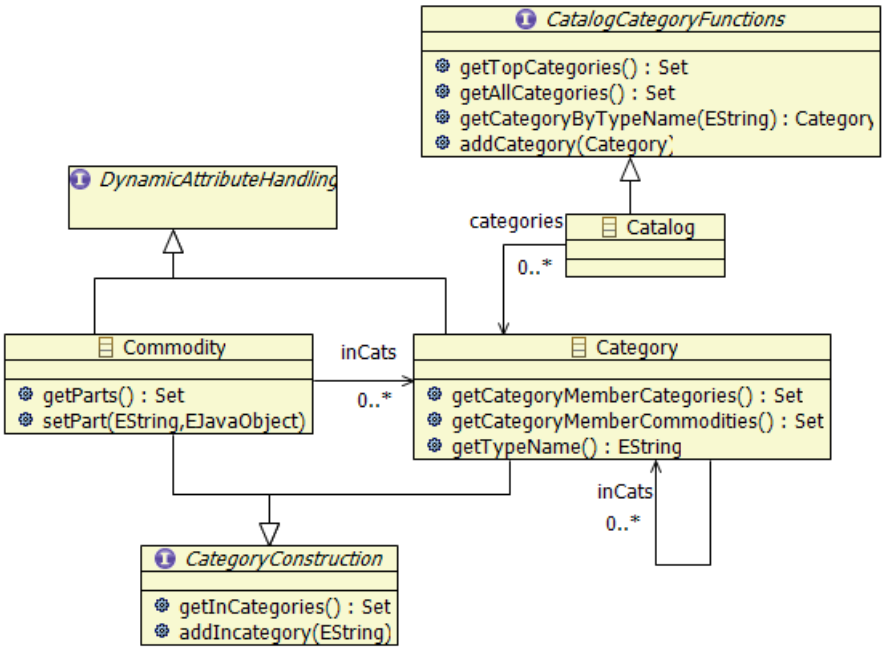

Figure 3.9: The Category-related framework class diagram

The two associations named inCats refer to the in-category relation discussed in the domain analysis. A category is stored within the catalog and also has dynamic attributes, just as commodities. The catalog gets some additional functionality related to categories and both Commodity and Category gets operations for constructing category relationships.

Example. The construction of categories is quite simple. Just create a new instance of Category and specify category name and in-categories as the two parameters as seen in listing 3.24.

Listing 3.24: Framework user code example of adding categories

```
addCategory(new CategoryImpl("Manufacturer"));
addCategory(new CategoryImpl(this, Dell, new String [] {"Manufacturer"}));
addCategory(new CategoryImpl(this, BenQ, new String [] {"Manufacturer"}));
addCategory(new CategoryImpl("Front page"));
```

□

Composites

In figure 3.10 general commodities, bundles and commodity choice sets are introduced. The commodity class has a super-type association to general commodity. Commodities can have parts which are themselves commodities. Bundles can both have category parts and commodity choice sets. These are the compositional aspects. The catalog gets additional operations to handle the construction of composites.

Example. Listing 3.25 shows the construction of the commodity good. In this case the good gets some attributes which its subtypes will be required to specify. The name followed by the type of the attribute is passed to the addAttribute operation. The type

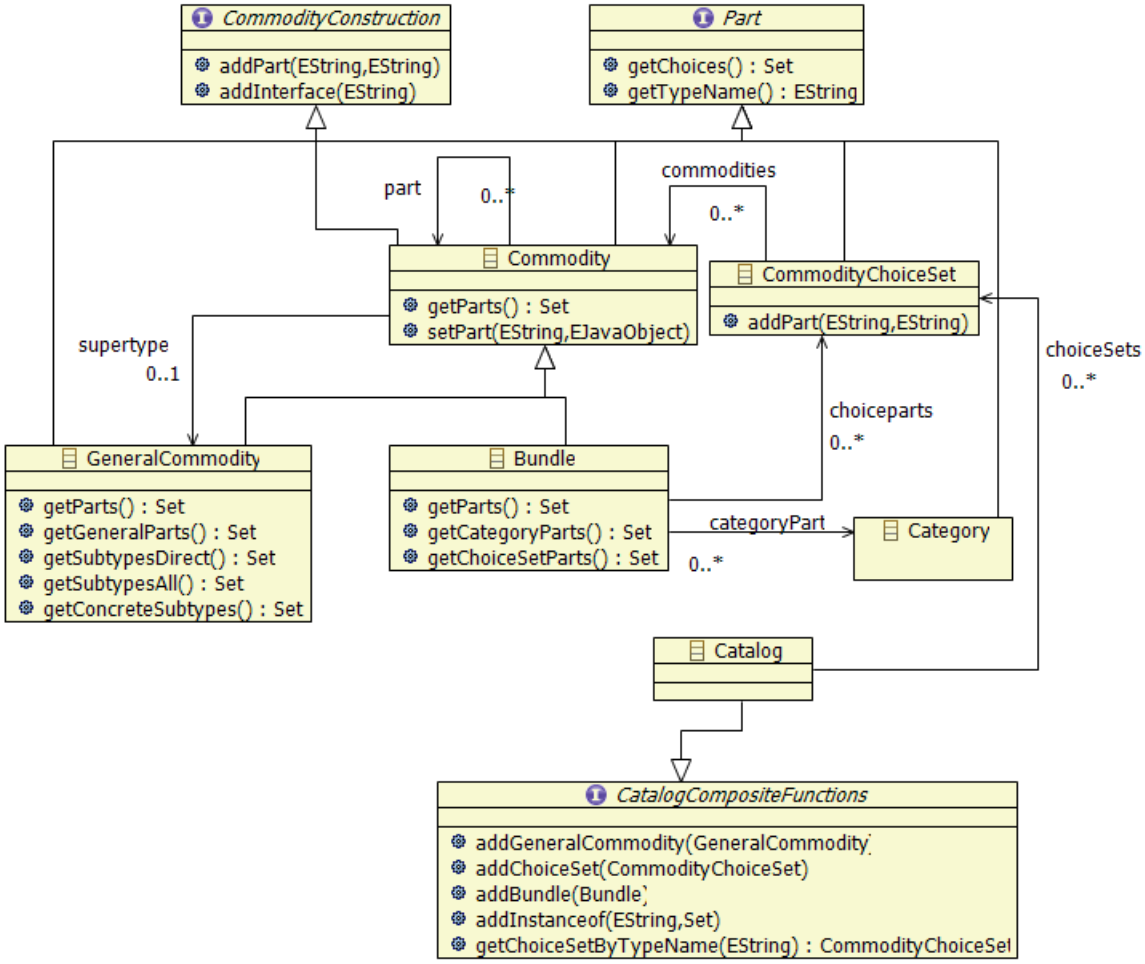

Figure 3.10: Catalog-Composition Framework composite-related class diagram

is a java class with the following requirements. It must have a constructor which takes one string. It must have a toString operation which returns the string representation of that object. The value returned by the toString operation must be compatible with the constructor of the class.

Listing 3.25: Framework example of defining a general commodity

```
GeneralCommodity good = new GeneralCommodityImpl("Good");
good.addAttribute("price", Integer.class);
good.addAttribute("modeltype", String.class);
good.addAttribute("modelnumber", String.class);
good.addAttribute("image", String.class);
addGeneralCommodity(good);
```

Parts are added in the same fashion. In listing 3.26 a computer is specified to consist of a box, a monitor with an RGB display port and a USB keyboard and mouse. It is a subtype of good defined above.

Listing 3.26: Framework implementation of Desktop Computer

```
GeneralCommodity computer = new GeneralCommodityImpl(this, "Computer", "
    Good");
computer.addPart("box", "ComputerBox");
computer.addPart("rgbMonitor", "RgbMonitor");
computer.addPart("usbKeyboard", "UsbKeyboard");
computer.addPart("usbMouse", "UsbMouse");
addGeneralCommodity(computer);
```

Listing 3.27 shows an example of a choice-set. The choices are simply added as individual parts to the choice set object. In the example a customer can choose between two different mobile phones for a bundle.

Listing 3.27: Example choice set definition

```
CommodityChoiceSet cs = new CommodityChoiceSetImpl("TwoMobiles");
cs.addPart("w800i", "W800i");
cs.addPart("w200i", "W200i");
addChoiceSet(cs);
```

Listing 3.28 shows the declaration of a bundle. The user can for this example choose between the two mobile-phones, then select a subscription and choose any member of the category freebie. This should illustrate the flexibility of the composition setup described.

Listing 3.28: Framework user-code example for defining a bundle

```
Bundle mb = new BundleImpl("MobileBundle");
addBundle(mb);
mb.addPart("aMobile", "TwoMobiles");
mb.addPart("subs", "Subscription");
mb.addPart("freebie", "FreeBees");
```

□

Interfaces

In figure 3.11 the Interface class is introduced. Interfaces are stored in a catalog. Commodities can have interfaces and interfaces can connect to other interfaces.

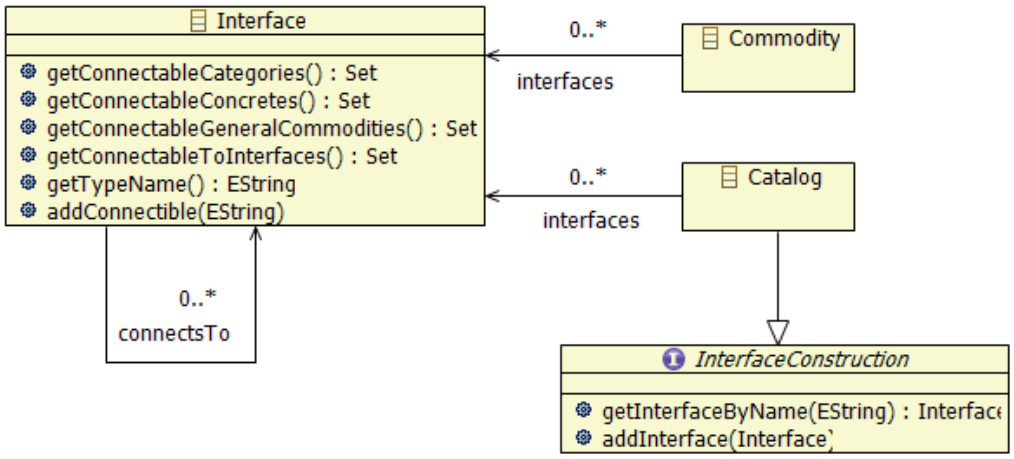

Figure 3.11: Catalog-Composition Framework interface-related class diagram

Example. USB is a well known interface found on most computers and peripherals. The interfaces on computers are the female version of the interface, while the peripherals have a male version. The USB interfaces are defined and added to a commodity in listing 3.29.

Listing 3.29: Framework user-code examples for defining and assigning interfaces to commodities

```
addInterface(new InterfaceImpl("UsbFemale"));
addInterface(new InterfaceImpl("UsbMale", "UsbFemale"));

usbkeyboard.addInterface("UsbMale");
```

□

3.3.2 The Entity-Behavior Framework

The Entity-Behavior Framework is described using the same example as was used for the description of the Entity-Behavior Language. This is done to make the contrast between the two systems as clear as possible.[15]

Listing 3.30 is the implementation of the Order behavior of figure 3.6. It is the equivalent implementation shown for the DSL in listing 3.15. The behavior is here set up in the constructor of a class. The class name is the name of the state and it extends the framework class XorStateImpl. This extension makes Order an xor-state. Inside the constructor the substates are added. The first parameter is the state instance name. The second parameter is the state object. If the substate is a simple state, then an object of the framework class SimpleStateImpl can be made. If the substate is a custom one such as for the OrderingState state, then this class is instantiated instead.

Then the initial state is set. The second parameter to this call is the Command[16]-object executed when the Order state is first entered. In other words, it is the action of the initial transition.

Transitions are added using the addTransition operation. This operation takes the from-state name, the target state name, the transition action Command-object and the

[15] For additional examples see Appendix D.
[16] As in the Command pattern of [GHJV95]

trigger signal class as parameters. The framework does not have a special operation for adding abilities. This must hence be implemented as a self-transition instead. Also the EpsilonSignalImpl class is a framework class for the epsilon signal.

Listing 3.30: EBF example behavior: Order state part 1

```
public class Order extends XorStateImpl{
  public Order(){
    addSubstate("Ordering", new OrderingState());
    addSubstate("InProcessing", new SimpleStateImpl());
    addSubstate("Shipped", new SimpleStateImpl());
    addSubstate("ConfirmedAtCustomer", new SimpleStateImpl());

    setInitialState("Ordering", new OrderOrderingepsilon());

    addTransition("Ordering", "Ordering", new OrderAddCommodity(),
        AddCommodity.class);
    addTransition("Ordering", "Ordering", new OrderRemoveCommodity(),
        RemoveCommodity.class);
    addTransition("Ordering", "InProcessing", new OrderOrderingepsilon(),
        EpsilonSignalImpl.class);
    addTransition("InProcessing", "Shipped", new OrderShipped(), Shipped.
        class);
    addTransition("Shipped", "ConfirmedAtCustomer", new
        OrderConfirmedAtCustomer(), ConfirmedAtCustomer.class);
  }
}
```

The OrderingState is implemented in listing 3.31. This state is constructed very similarly to the Order state. There are two special framework classes used. The first is ShallowHistoryStateImpl which is a history state. The second is the FinalStateImpl which implements a final state.

Listing 3.31: EBF example behavior: Order state part 2

```
class OrderingState extends XorStateImpl{
  public OrderingState(){
    addSubstate("HistoryState", new ShallowHistoryStateImpl());
    addSubstate("noInfoEntered", new SimpleStateImpl());
    addSubstate("enteredAddress", new SimpleStateImpl());
    addSubstate("enteredPayment", new SimpleStateImpl());
    addSubstate("final", new FinalStateImpl());

    setInitialState("HistoryState", new InitialOrderingState());

    addTransition("HistoryState", "noInfoEntered", new OrderingStateepsilon
        (), EpsilonSignalImpl.class);
    addTransition("noInfoEntered", "enteredAddress", new
        OrderingStateAddressDetails(), AddressDetails.class);
    addTransition("enteredAddress", "enteredPayment", new
        OrderingStatePaymentDetails(), PaymentDetails.class);
    addTransition("enteredPayment", "final", new OrderingStateConfirmOrder
        (), ConfirmOrder.class);
  }
}
```

The Command-classes must implement the interface in listing 3.32. The execute operation is executed during the transition. The signal object which triggered the transition is passed to it. The signal object contains information used in the transition action. The

object must be casted to a more specialized signal in order to retrieve the information in it. This is done within the execute operation.

Listing 3.32: Transition action interface used for the Command-classes

```
public interface Transition{
  void execute(Signal s);
}
```

The Entity-Behavior Framework design

Figure 3.12 shows the class diagram with all the classes in the Entity-Behavior Framework. The two operations which are mostly used by programs using the Entity-Behavior Framework are the Set getPossibleSignals() and void acceptSignal(Signal s). The first operation returns all signal-classes which will trigger a transition in the statechart. This is typically used for generating alternatives for a user with regard to, for example, an order. The second operation is used to send a signal-object to the statechart. The signal objects are typically initialized with information which is used by the transition action.

3.3.3 Integration into the E-Commerce Framework

The components in the E-Commerce framework are modeled in figure 3.13. Inside the shop component we find the uses of the Catalog-Composition Framework and the Entity-Behavior Framework. The Shop component provides an interface which is the unrestricted access to the shop. The Shop component is combined with a SalesClerk component to build a larger Shop component which is managed by a SalesClerk component. This is what the View component uses, which is in turn what the customer interacts with in the view. Hence customers can only ask the salesclerks about things. The salesclerk will perform the action on the shop if it finds it permissible. There is a salesclerk and a view per session, but only one shop. The shop and the salesclerks are situated on a server node, while the view and the customers are at the client.

The catalog, the commodity behaviors and the ordering behavior are all located in the Shop component. The salesclerk behavior is in the SalesClerk component. This way, the system is built as a simulation of a real shop. A real shop is just an area where there are commodities and apparatus for carrying out purchases and support. The Shop component represents a real shop. Every shop needs to have a salesclerk. The salesclerk has access to everything in the shop, and his or her job is to do this with care. A customer only has access to the things in the store through the salesclerk. The salesclerk serves as an abstraction on top of the detailed functioning of the shop. He or she provide a more pleasant and secure way to purchase commodities. The View component is a virtual customer interacting with a virtual salesclerk in order to buy things from the shop.

The interaction involved when a user submits a form using a web-browser, is modeled in an interaction diagram in figure 3.14. Using a view, a customer submits a form. The controller takes the request and handles it by sending signals to the personal salesclerk on the server. The signals reach the salesclerk statechart, for example a similar one to figure 3.7. If the signal is accepted, the action of the transition sends a number of requests to the shop component. The shop component contains commodities and orders. If a change is made to the model, the views are notified. The views request changes from the salesclerk, who is responsible for only giving the view information it is authorized to get.

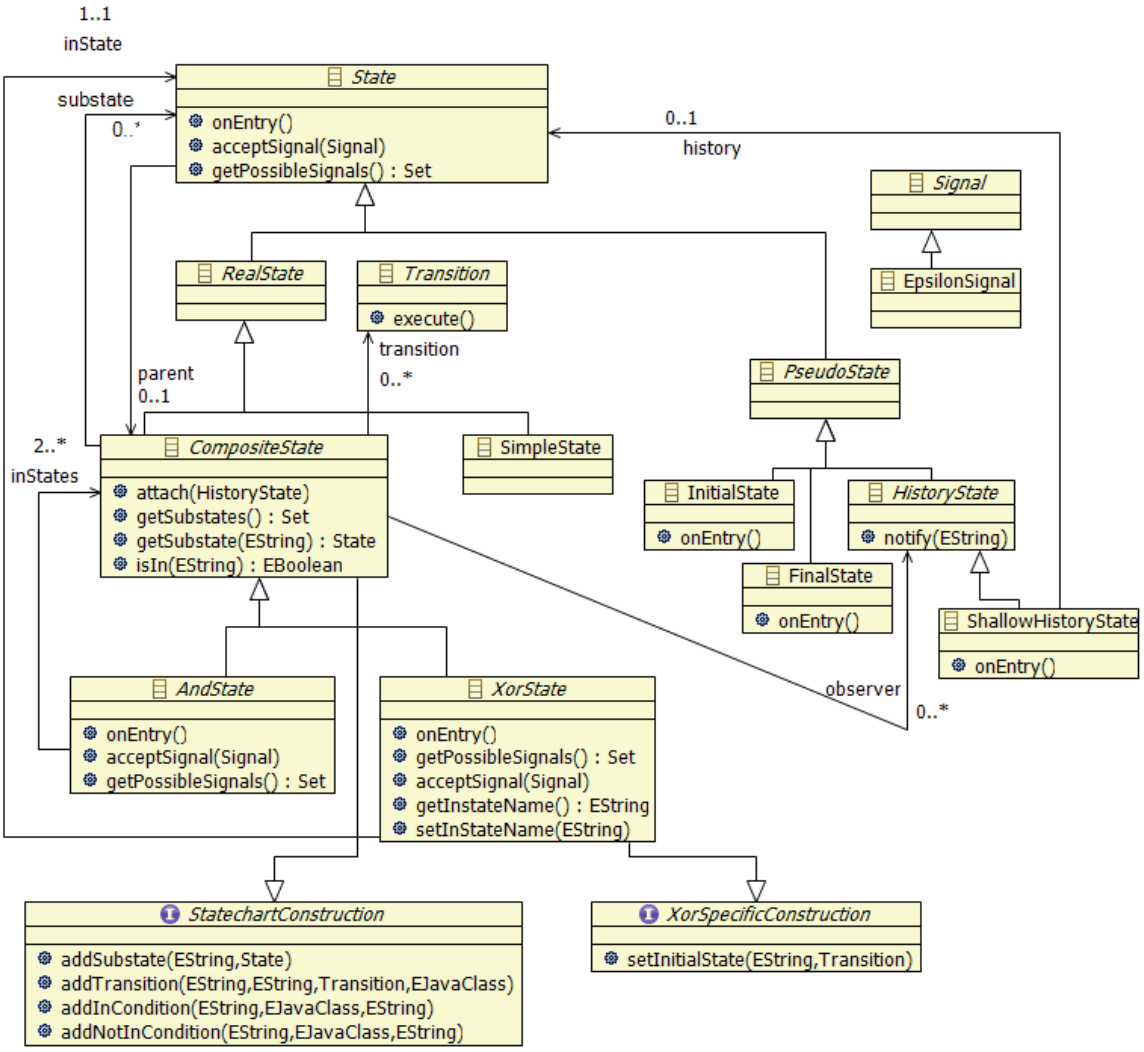

Figure 3.12: Entity-Behavior Framework class-diagram

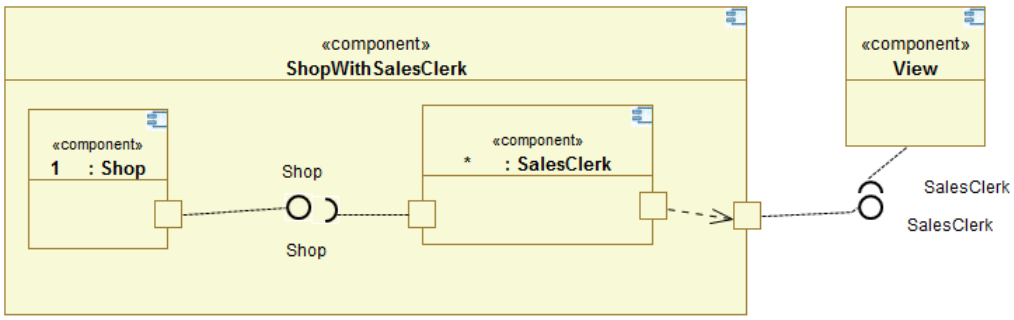

Figure 3.13: E-Commerce Framework components

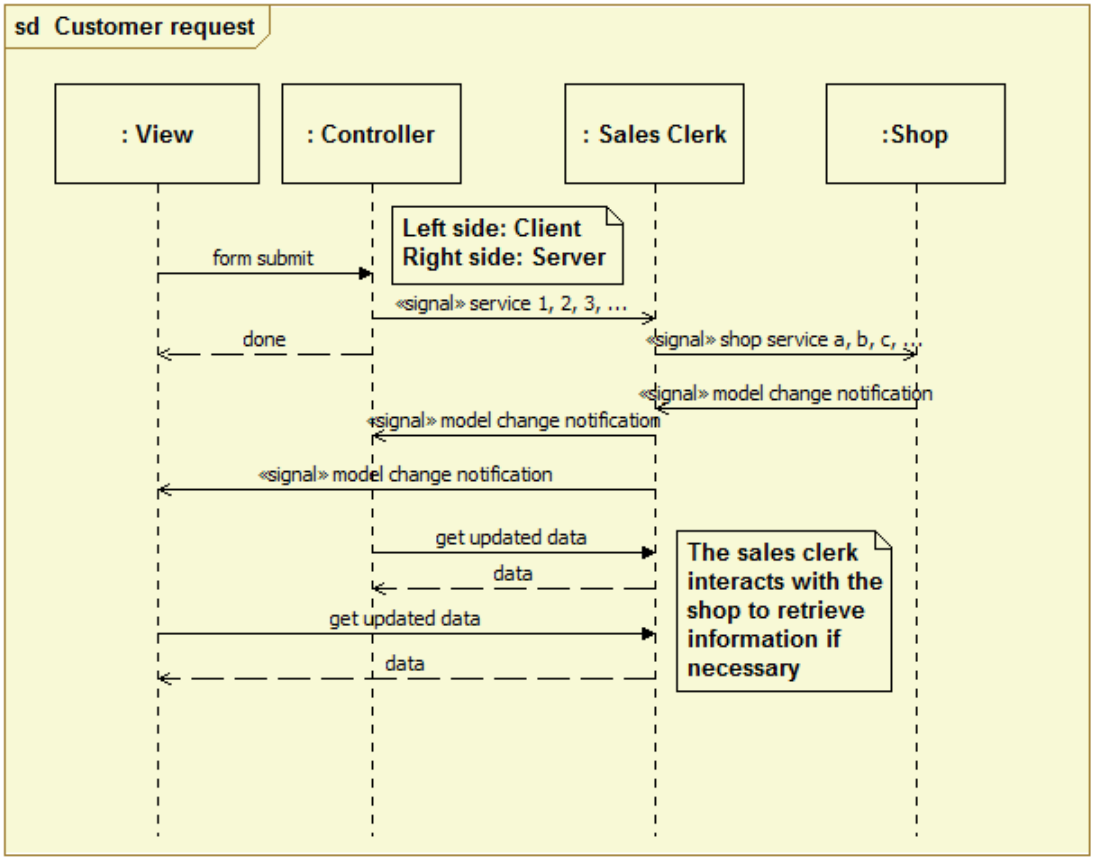

Figure 3.14: Interaction view of the E-Commerce Framework

Example. An example of a request is adding a commodity to an order. The user clicks a link called, for example, add to order which is found in a commodity presentation on a website[17]. The controller gets this request and constructs a signal containing a request to add an instance of the commodity-type to the order. The salesclerk gets the request to add the commodity to the current order. The salesclerk requests the shop component to add the commodity to the current order. Once the commodity is added, the Shop component notifies the View component that something has changed. The View component responsible for showing orders, requests the new order-contents from the Salesclerk component, which in turn requests it from the shop. The view is finally updated with the commodity added to the current order.

If an error occurs during this process, the user will be notified by the salesclerk who is in turn notified by the Shop component. □

3.3.4 Implementation

Method

The method used to implement the frameworks is as follows. EMF is the Eclipse modeling framework. This framework comes with a class model- and diagram-editor for creating class models. EMF can then generate code from this model. This generated code is annotated with information about which operations are generated and which are not in order to be synchronize-able with the model. Implemented operations are marked as not generated and are not modified during synchronization.

Using this generated code, tests are written using the code generated, particularly the interfaces. The tests and the class models are, together with the written specification of the framework, the complete specification. This forms a 'framework interface', which may be implemented according to which needs the concrete usage of the framework has. For the experiment a simple implementation was sufficient to see that the system worked. For a production ready system, a more sophisticated implementation using a database is suggested.

Generated and unimplemented methods throw an exception. During the implementation phase, these exceptions show up in the test-results indicating which method to implement next. This is basic EMF usage, and is described in detail in [Ecl08]. Note that some of the boilerplate generated by EMF is omitted from the examples in this thesis, since it is irrelevant to the functioning of the framework.

Implementation of the Catalog-Composition Framework

The model used for generating the classes of the framework is presented in the diagrams 3.8, 3.9, 3.10 and 3.11. Note that the diagrams do not include all overriding methods.

In figure 3.15 the project overview of the Catalog-Composition Framework is shown. The diagrams for the Catalog-Composition Framework class model located in CCF.ecore are found in CCF.ecorediag. The pom.xml file contains the project configuration and dependencies which maven handles.

src/main/java contains the implementation. There are three packages: The base package no.uio.ifi.heim.martifag.ccm contains the project interfaces, the impl package contains an implementation of the interfaces and the util package is only generated as a part of the EMF-code generation and is not used.

[17] See figure A.1

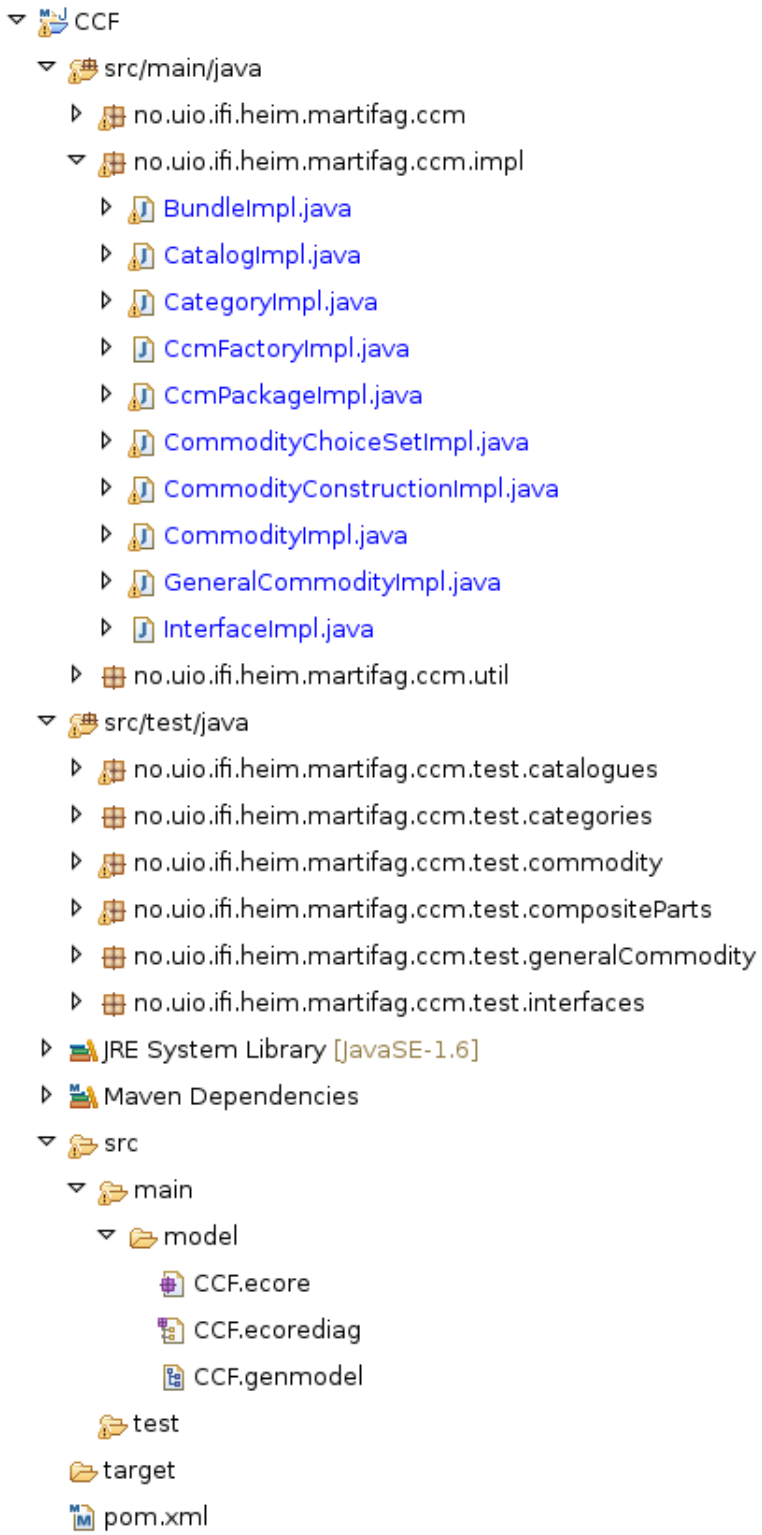

Figure 3.15: Project overview of the Catalog-Composition Framework

Listing 3.33 shows some interesting parts of the source file for the catalog implementation. The contents of the catalog are simply stored as elements of several lists: commodities, categories, interfaces, choiceSets and instances. The instances variable contains instances of the commodity definitions found in the commodities attribute.

Most of the operations perform set and list-based operations on the attributes containing the catalog contents. SQL (in the context of databases) is therefore suited for implementing a production version of the Catalog interface. For simplicity reasons, this was not done for the implementation used in the experiment.

The following are descriptions of the most important of the operations shown in listing 3.33.

void addCommodity(Commodity c). This method adds a new commodity-definition to the catalog. Similar operations are used to add other types of catalog content. The commodity gets the catalog object of the catalog which owns it. The commodity must query the catalog in order to implement its own operations.

Commodity addInstanceof(String typeName). This method adds a new instance of a commodity. The operation finds the commodity of the type-name given as a parameter. If the commodity type is found, a cloned object is placed in the instance list. The objects in the instance list then live their own lives as representations of actual physical products, or services bound to a particular resource allocation.

Set<Category> getTopCategories(). This one returns the set of categories which are not members of another category. These are typically used as the top of a tree-view of the categories. The operation simply collects the categories which are members of zero other categories.

Set<Commodity> getCommoditiesByType(String name). This method collects the instances which are subtypes of the parameter type name. If the name is the name of a general commodity, then the operation must recursively search for instances of subtypes of the general commodity. This explains the two branches on the major if-statement.

Map<String, Integer> getCommodityCounts(). It returns a map of commodity types and the number of instances. This is used to generate the number of commodities available in stock for a customer to buy.

Listing 3.33: CatalogImpl.java

```
...
public class CatalogImpl extends EObjectImpl implements Catalog {
  protected EList<Commodity> commodities;
  protected EList<Category> categories;
  protected EList<Interface> interfaces;
  protected EList<CommodityChoiceSet> choiceSets;
  protected Map<String, List<Commodity>> instances;
  ...
  public void addCommodity(Commodity c) {
    c.setOwningCatalogue(this);
    commodities.add(c);
  }
  public Commodity addInstanceof(String typeName) {
```

```
    for(Commodity c : commodities){
      if(typeName.equals(c.getTypeName())){
        try {
          Commodity newc = (Commodity) ((CommodityImpl)c).clone();
          List<Commodity> list = instances.get(typeName);
          if(list == null)
            list = new ArrayList<Commodity>();
          list.add(newc);
          instances.put(typeName, list);
          return newc;
        } catch (CloneNotSupportedException e) {
        }
      }
    }
    return null;
  }
  public Set<Category> getTopCategories() {
    Set<Category> res = new HashSet<Category>();
    for(Category c : categories){
      if(c.getInCategories().size() == 0)
        res.add(c);
    }
    return res;
  }
  public Set<Commodity> getCommoditiesByType(String name) {
    Set<Commodity> res = new HashSet<Commodity>();

    Commodity c = getCommodityByTypeName(name);

    if(c instanceof GeneralCommodity){
      GeneralCommodity gc = (GeneralCommodity)c;
      EList<Commodity> coms = getCommodities();
      for(Commodity com : coms){
        Commodity orgcom = com;
        while(com != null){
          if(com.getTypeName().equals(name)) if(!(orgcom instanceof
              GeneralCommodity)){
            res.addAll(getCommoditiesByType(orgcom.getTypeName()));
          }
          com = com.getSupertype();
        }
      }
    }else{
      List<Commodity> x = instances.get(name);
      if(x != null)
        res.addAll(x);
    }

    return res;
  }
  public Map<String, Integer> getCommodityCounts() {
    Map<String, Integer> res = new HashMap<String, Integer>();
    for(Entry<String, List<Commodity>> x : instances.entrySet()){
      res.put(x.getKey(), x.getValue().size());
    }
    return res;
  }
}
```

Implementation of the Entity-Behavior Framework

The class model of the Entity-Behavior Framework is seen in figure 3.12. One of the interfaces in this model is XorState which is implemented in XorStateImpl. Listing 3.34 shows parts of the implementation of the XorState interface. This class is the most important class for the statechart framework. Therefore a deeper look at this class is presented. For the rest of the source, see Appendix A. The two most interesting operations in XorStateImpl are acceptSignal and getPossibleSignals.

void acceptSignal(Signal s). The purpose of this method is to send a signal to the statechart. This method first checks if the signal will trigger a transition in the xor-state itself. If not, the signal is passed on to the current sub-state. Next the operation gets the transition target and action. The action is executed and the current state is changed. If the action throws a run-time exception, the state is not changed into the new state, since that happens after the successful execution of the transition action. Then the history states are notified of the state change. Finally the new state's onEntry is called for the new sub-state to perform its action on entry. This is typically where history states and final states implement their special behavior.

Set<Class<? extends Signal>> getPossibleSignals(). The purpose of this method is to return the signals which will cause a transition to occur. It calls the substates recursively to get all the possible signals which will trigger a transition. It fetches the signals for the current state and for the current sub-state. It then removes the epsilon signal, as it might never be used by anything else than the framework itself. To get the signals for the current state, the framework check which signals are possible and takes into account in-constraints.

Listing 3.34: XorStateImpl.java

```
...
public abstract class XorStateImpl extends CompositeStateImpl implements
    XorState {
  protected String inStateName;
  private String initialState;
  private Transition initialTransition;
  ...
  public void setInitialState(String stateName, Transition transition) {
      ... }
  public void onEntry() { ... }

  public Set<Class<? extends Signal>> getPossibleSignals() {
    Set result = new HashSet();
    result.addAll(getPossibleSignalsLocally());
    result.addAll(getSubstate(inStateName).getPossibleSignals());
    result.remove(EpsilonSignalImpl.class);
    return result;
  }
  protected Set<Class<? extends Signal>> getPossibleSignalsLocally(){
    Set result = new HashSet();
    if(transitions.get(inStateName) != null){
      for(Entry<Class, TransitionData> x : transitions.get(inStateName).
          entrySet()){
        CompositeState s = this;
        while(s.getParent()!=null)
```

```
            s = s.getParent();
          if(x.getValue().inCondition!=null) if(!s.isIn(x.getValue().
              inCondition)) continue;
          if(x.getValue().notInCondition!=null) if(s.isIn(x.getValue().
              notInCondition)) continue;
          result.add(x.getKey());
        }
      }
      return result;
    }
    public void acceptSignal(Signal s) {
      // Check if signal is available
      if(!getPossibleSignalsLocally().contains(s.getClass())){
        inState.acceptSignal(s);
        return;
      }

      // Get information
      Map<Class, TransitionData> map = transitions.get(inStateName);
      TransitionData data = map.get(s.getClass());

      // Perform transition
      data.transition.execute();
      inStateName = data.to;
      inState = getSubstate(inStateName);

      for(HistoryState o : observer){
        o.notify(inStateName);
      }

      // If entering composite state
      if(inState instanceof CompositeState || inState instanceof PseudoState)
          {
        inState.onEntry();
      }
    }
}
```

Implementation of the E-Commerce Framework

The top-most view of the E-Commerce Framework is seen in figure 3.13. In this figure we see the components that the complete system consists of. The system has a model-view-controller (MVC) architecture as described in [BMR+96]. The class diagram of this pattern is seen in figure 3.16. The ShopWithSalesClerk component is the Model of the MVC pattern. Both the View and the Controller of the MVC pattern is within the View component in figure 3.13. Hence, the ShopWithSalesClerk component is on the server, while the View component is on the client. An example interaction of these three components is seen in figure 3.14. On the server we find the Shop and the SalesClerk components. On the client we find the controller and the view of the MVC-pattern.

Figure 3.17 shows how the View and the Controller can access the Model differently. The Controller can perform changes on the Model, while the View can query the Model for information.

The SalesClerk component is shown in figure 3.18. It is called ShopManager in the diagram. The salesclerk can do whatever he or she want in the shop. Therefore, the

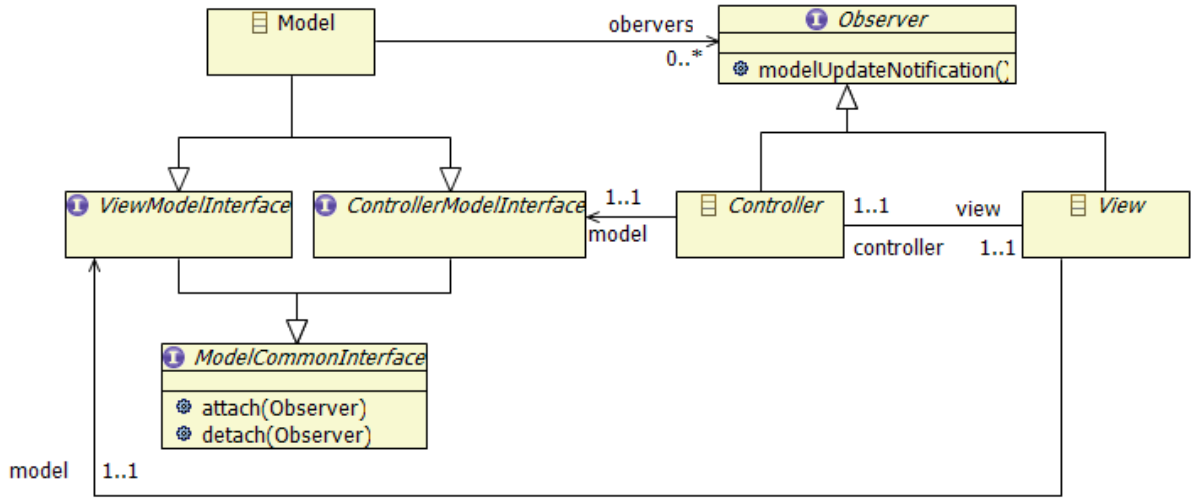

Figure 3.16: MVC architecture of the E-Commerce Framework

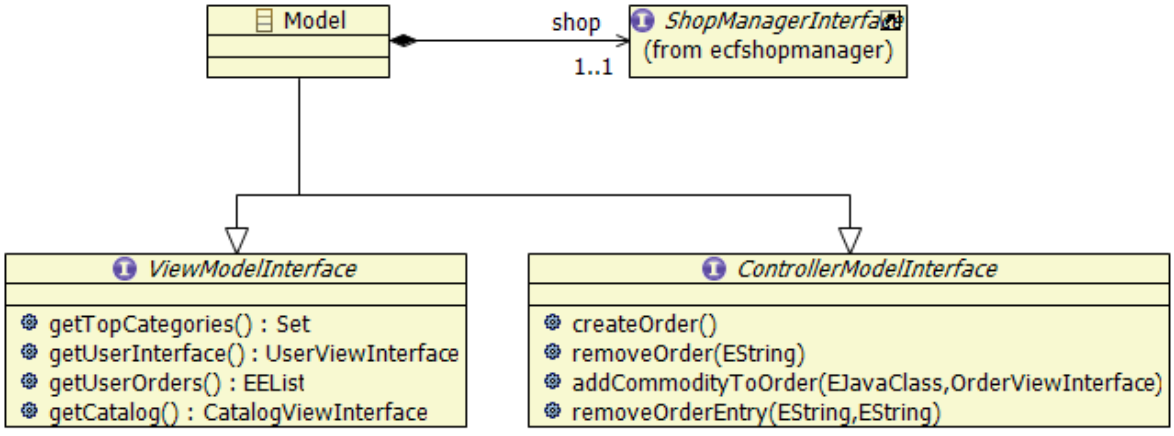

Figure 3.17: The controller-model and view-model interfaces

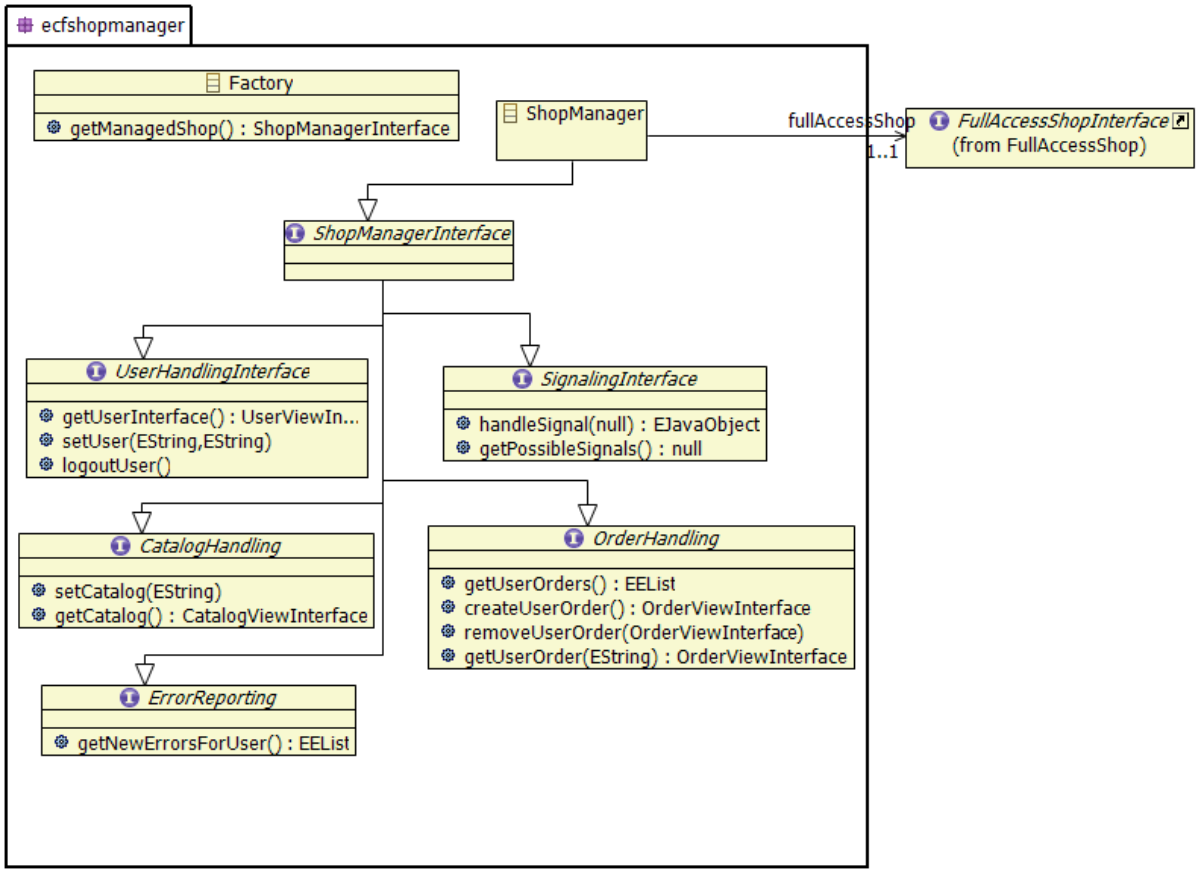

Figure 3.18: The E-Commerce Framework salesclerk

salesclerk has full access to the shop, hence the name FullAccessShopInterface. Inside the full access shop we find the catalog and the statecharts.

The SalesClerk component is an integrating component in the system. Listing 3.35 shows some parts of the implementation of the ShopManager interface as modeled in figure 3.18. This component stands between the customer on the one hand and the catalog and statecharts on the other. It handles the following things.

The salesclerk. The salesclerk behavior is implemented using the Entity-Behavior Framework and is found in the attribute manager. Though not completely implemented for all operations, the SalesClerk's operations are supposed to check with the statechart in the manager attribute before doing anything. Or better, it is supposed to perform the actions as a part of a transition. Due to time constraints, this was realized to a certain degree. For example of this, see operations such as createUserOrder(), where the statechart is asked before performing any actions.

Users. Users are handled by another attribute called userManagerStore. The two operations setUser and logOutUser logs users in and out. When a user is set, the user behavior of this user is found in the user attribute. The user is also programmed using the Entity-Behavior Framework. The user behavior is implemented as a statechart and provides authorization and authentication for all users of the system. See figure 3.7 for an example of how the user behavior might be implemented.

Orders. Orders are also implemented using statecharts. Each user might have several orders. Each order has its own statechart. Typical order behavior is seen in figure 3.6.

Error reporting. If an error occur during one of the transitions or otherwise, the error message is taken care of by the salesclerk which shows it to the user when appropriate.

Listing 3.35: ShopManagerImpl.java: An implementation of a salesclerk

```
...
public class ShopManagerImpl extends EObjectImpl implements ShopManager {
  protected FullAccessShopInterface fullAccessShop;
  protected Manager manager;
  protected User user;
  protected UserManagerStore userManagerStore;
  protected List<OperationAbortReport> userErrors;
  protected List<Order> userOrders;
  ...
  public ShopManagerImpl(FullAccessShopInterface fullShopControls) {
    // Shop component link
    fullAccessShop = fullShopControls;

    // User
    userManagerStore = new UserManagerStoreImpl(this);
    Pair<User, Manager> e = userManagerStore.getCurrentUser();
    user = e.getUser();
    manager = e.getManager();
    ...
  }
  public OrderViewInterface createUserOrder() {
    Signal co = new CreateOrderSignal();

    if(!manager.getPossibleSignals().contains(CreateOrderSignal.class))
      return null;

    Order o = fullAccessShop.createOrder();
    userOrders.add(o);

    return (OrderViewInterface) new OrderViewInterfaceImpl(this, o);
  }
  public void setUser(String username, String password) { ... }
  public void logoutUser(){ ... }
}
```

3.4 Comparison

The comparison starts with a short discussion of some well known advantages and disadvantages of DSL and Frameworks also experienced in the experiment of this thesis. The more interesting questions are discussed in subsequent subsections.

3.4.1 DSL advantages

Domain specific concrete syntax

A DSL can have its own concrete syntax. A framework cannot have its own syntax since it is used in the same language in which it was created. Concrete syntax is defined by

grammar and is used to create a parser which converts code into an abstract syntax tree. The syntax can be domain-specific in that it uses terms and concepts from the domain. Many keywords can be from the domain. The way a language construct is composed can be from the domain and follow standard domain-specific ways of doing things.

Example. In the Catalog-Composition Language grammar there were several domain specific keywords, including Catalog, Commodity, Category and GeneralCommodity. Working with such keywords is more natural than generic constructs such as objects and classes. □

The syntax is not limited to textual syntax. The syntax can be graphical, such as is used for class diagrams. The GUI for working with the syntax can also be very elaborate. Indeed working with a file in a DSL can be a full GUI using the wide range of windowing widgets. Such a GUI can be generated from the grammar or the meta-model of the language.

Example. In Eclipse[18] there are several plug-ins for working with XML- and text-files which programs certain applications. The Maven POM-editor[19] for eclipse uses an elaborate GUI for configuration of the XML-file. □

It should be noted that concrete syntax and other supporting tools for a DSL have to be created and maintained with the DSL. This involves a considerable amount of work.

Domain specific static semantic analysis

Static-semantic analysis is commonly performed after the parsing of source code. The static-semantic analysis phase of the compilation process is when the compiler performs static checks which were not simple or possible to guarantee with the grammar itself. Static-semantic analysis is beneficial because errors detected at compile-time are not a problem at run-time. Finding errors in code at run-time is a notorious problem, since it is unpredictable when they occur. It is, however, a limit to what can be checked at compile-time. Note that DSLs have more possible static checks than general purpose languages. The reason is that something general, by definition, has fewer constraints than something specific.

Example. Consider the construction of a Commodity with the Catalog-Composition Language. There are several rules on how a commodity might be put together and several rules of how to classify a commodity from how it has been put together. For example, a commodity containing general parts is it-self general.[20] If the parts of a commodity are totally unrelated, then its static semantics is a bundle. Static semantics such as this is found by domain experts during the domain analysis phase of the system design. □

3.4.2 Framework advantages

Easier to make

The requirements for making the E-Commerce Framework are knowledge of an object-oriented language[21]. Most people have knowledge of Java, C# or C++ today. The

[18] http://eclipse.org

[19] POM stands for the Project Object Model, and is used by the Apache Maven build system to configure projects.

[20] this is similar to abstract methods in a class. Then the class is also abstract.

[21] See the definition of a framework in section 2.2

E-Commerce Language, on the other hand, requires knowledge of compiler and tool construction in addition to knowledge of a general purpose language. This includes knowledge on lexical analysis, grammars, static-semantic analysis and code-generation techniques. The two latter are normally programmed in a general purpose language. If something went slightly wrong in the construction of the framework, one can probably work around the problem. This work around is possible because the general purpose language in which one uses the framework is not limited to what is defined by the framework. Such workarounds are less probable within DSL-code. This is an argument for why a framework which works is easier to make than a DSL that works. It may be the case, however, that a properly trained person might be equally efficient for DSLs and for Frameworks.

Constructs based on the framework can be used in the host language

Constructs that are built by using the E-Commerce Framework are easily usable in another Java program. This is very natural since the framework, the framework instantiation and the programs interacting with the framework instantiation are all written in the same language. For the E-Commerce Language there is a mismatch here. The constructs programmed are domain specific. The code is generated towards the framework later, but the DSL constructs are not available in Java unless passing through code generation first. Then they are available as classes, which are a lot more general than the original domain specific constructs of the E-Commerce Language.

Existing tools can be used

The fact that Java-tools can be used when programming towards a Java-framework is not surprising. The tool support available for Java is a benefit the E-Commerce Framework. There might be advances of automatically generated tooling in the future. For example, the Eclipse Graphical Editing Framework which "allows developers to take an existing application model and quickly create a rich graphical editor." [22]

There is an important distinction which must be made here. When talking about tooling for a DSL it is specifically the domain specific tooling one is talking about. The tools that can be reused for the general purpose languages behind frameworks are not domain specific. But, indeed the general purpose tooling is often not available for DSLs. So this is where the distinction has to be made.

3.4.3 One can base an application on several general solutions simultaneously

Using several frameworks to build an application is a common practice. The E-Commerce Framework can be constructed using the frameworks in the following example.

Example. A common structure of Java programs today is using the following frameworks and systems.

- Spring[23] for MVC and dependency injection
- Hibernate[24] for OR-mapping towards a relational DBMS

[22] http://eclipse.org/gef/
[23] http://springsource.org
[24] http://hibernate.org

- Maven[25] for dependency management and project layout
- JUnit[26], for unit testing java

The different frameworks serve the following purposes. Spring dependency injection facilities and the MVC frameworks get the user to produce components and classes in a certain fashion. Hibernate maps objects into relational concepts of a database. All these classes are tested using JUnit. The dependencies of the different classes are managed with maven. □

Basing an application on several frameworks is possible since the frameworks are all programmed in the same language. They produce constructs available and composable in the language they were created with.

This is not as trivial for DSLs. Programs in a DSL produce domain-specific constructs. If two DSLs are to inter-operate, they have to agree on which common notions to use in their interaction with each other. For frameworks, choosing the common ground is the obvious choice of the common implementation language.

The E-Commerce Language cannot use the four frameworks mentioned without programming support for it into the grammars of the DSL. What one can program is strictly limited by the grammars. The grammar would have to contain some grammar elements which serve as a connection point for other systems in general. No such standard was found to exist for the four mentioned frameworks.

3.4.4 Working at the right abstraction-level

When programming at a too low abstraction-level relative to a domain, one is at risk of creating a different *implicit architecture* than the intended *legitimate architecture*. [Lau01, 4.4, page 34] differentiates between these two architectures:

> Legitimate architectures are architectures that are deliberately constructed with a specific purpose in mind, whereas accidental architectures are a coincidental (i.e. accidental) side effect of the construction process.

[ND04] also notes the difference:

> Implicit architecture: One of the biggest hurdles in most programming languages, and especially in object-oriented languages, is that the run-time architecture is not apparent from the source code. This makes many kinds of changes difficult to understand, assess and realize.

What is meant by the term 'run-time architecture' by [ND04] is the same as the term 'implicit architecture': The architecture which is implied by the code itself. Even if the architecture was intended to be different, it is the implied architecture which is important. A change to the system must take the actual, implied architecture into account. A change to the system based on the wrong understanding of the architecture is at a severe risk of being wrong.

[25] http://maven.apache.org
[26] http://junit.org

Example. Encapsulation is a useful abstraction. But, if the programming language one is working with allows object-access outside of an object's interface, encapsulation is not ensured automatically. If encapsulation is supported in the programming environment, one is not allowed to break it. One way to break encapsulation is having access to the lower level abstractions used to implement the encapsulation rules. Then be able to apply them on the objects which are not seen as encapsulated at this lower level of abstraction.

One way to ensure encapsulation automatically is working in an environment where access to encapsulated parts of an object is impossible, where lower levels of abstraction, such as access to the byte structure of the object in memory, are not available.

If, in a large program, there is one violation of encapsulation rules, the implicit architecture is not one containing the notion of encapsulation. If one assumes that encapsulation applies, a change to the hidden part of the encapsulated class will render the exception to the encapsulation rules wrong. □

When working with DSL code, one is restricted to just using the concepts available in the DSL. If one wants to do something which violates the legitimate architecture, the legitimate architecture inherent in the DSL-design has to be changed. Hence the danger of hard-to-see implicit architectural changes is lessened for someone working with DSL-code.

The ability to do quick-fixes or hacks on a system in order to make it work is a known phenomenon. DSLs may be experienced as rigid, strict or inflexible because of the required use of the DSL concepts and nothing more. Even though a quick-fix might please the boss in the short term, it is a disaster in the long term, as the implicit architecture spins out of control and becomes incomprehensible as a result of many quick-fixes.

When fixing a program it is important to ensure that the architecture is updated to accommodate for the fix. In DSL-code, one has less options of creating a breach between the implicit and legitimate architectures. If there is a separation between the DSL-developers and the DSL-users, the DSL-users must contact the DSL-developers and request a change in the DSL. The DSL-developers together with domain experts can then do a thorough evaluation of whether to incorporate the change or figure out why this is not desired or how to work around the problem using the existing architecture.

Whether a framework can protect against the user altering or circumventing the internal behavior is considered in subsection 3.4.5.

3.4.5 Protection of the system internal functioning from user-code

One of the major benefits of DSLs is that there is strict control of what the user is allowed to program. This control is significantly lower for a common framework where the user inputs general purpose code into the framework. What is interesting is why this is a problem for frameworks and how the problem might be lessened.

Example. Listing 3.36 shows an example from the Entity-Behavior Framework. Within the operation aToB, which is a transition action operation, the user has access to all protected and public operations of the framework super-class XorStateImpl, but also other transition operations of the xor-state X. The user can alter the in-state of the xor-state by modifying the variable 'instate', this would cause the statechart to enter the new state outside of the framework mechanisms. This essentially breaks the statechart framework.

Listing 3.36: An example from the Entity-Behavior Framework

```
public class X extends XorStateImpl{
  ...
  void aToB(A from, SignalB s){
    // Here
  }
}
```

□

There are several ways a user might damage the functional-integrity of a framework in an object-oriented language:

- Modify inherited attributes of framework classes
- Call inherited operations of framework classes
- Call other operations of the class itself, public, protected or private
- Use classes available in the same scope, even if these classes are not supposed to be used here
- There might be vulnerabilities related to generic instantiations of classes
- Late-binding makes it unpredictable from the framework what code will execute within a class
- Annotations are for example commonly used to generate boilerplate code. This boilerplate code might interfere with the functioning of the framework.

There are several ways to reduce the damage a user can do to the functional-integrity of a framework.

Advice against it. The most obvious one is warning against it in the framework documentation with a list of things a user of a framework should not do.

Use patterns or other design techniques One can for example use the command pattern from [GHJV95] to specifically solve the problem of having a user access the protected and private elements of a class.

Hide information. One can store away vulnerable information before calling a user-created operation, and then restore the data after the operation returns. Then whatever the user modifies is restored before continuing execution. There are concurrency issues with this approach which must be addressed for concurrent programs.

Disable methods. There can be a guard in the beginning of all the operations which simply returns at once or throws an exception if the execution is within a user-created operation. There are concurrency issues with this approach which must be addressed for concurrent programs.

Reflection on operation contents. One could utilize reflection on operation contents such as on statements and expressions. This reflection could at compile-time check that the user does not call or access operations he is not allowed to. It could also do the opposite: just allow certain constructs in the host language to be present in an operation. This kind of reflection is not offered in Java as of version 1.6.

When programming a framework, all the features of the host language must be carefully considered. Their implication for the framework's functional-integrity must be checked.

3.4.6 Extensibility

Extensibility is about expanding the code base of the framework itself or of the DSL language itself. Specialization is different; it is discussed in subsection 3.4.7.

General solutions, such as the E-Commerce Language and Framework, are held separate from the specific instantiations of it. In case of the framework, this is an instantiation of framework hot-spots. In case of the DSL, this is DSL-programs. The benefit of this separation is that bugs in the general part of the system may be fixed once for the general part, and then updated for all the concrete instantiations. Another aspect is the extension of the general part. It is an interesting question whether it is possible to extend the general part without breaking existing instantiations.

Both the DSL and the Framework, which is an implementation of the e-commerce domain model, depends on the domain model. An extension of either is therefore reflected in the domain model. If not, the extension is not of something related to the domain. The question is how an extension manifests itself in the general solution and how this affects the instantiation of the framework or the source code of the DSL.

Source code in the DSL has a one-to-one mapping to an AST. An AST is a prerequisite of the code generation stage of a compiler. But for frameworks, hot-spot instantiations are not converted into an AST or any other processable representation. The framework source code is simply compiled using the general purpose language compiler. A complicating factor is that a program using one framework might also be using several other frameworks. Separating out the parts which instantiates and interacts with the E-Commerce Framework is not easy if the application is based on and uses several other frameworks simultaneously. What differentiates the DSL code from a framework instantiation is that the DSL code is processable by its nature.

Having a processable representation means that one can use model-to-model transformations. A transformation could be supplied with an update of the E-Commerce Language. Developers can then use this transformation on their source code files. This is, for example, done within the Eclipse project in order to upgrade their implementation of the meta-model for UML[27]. Such an update is of course limited, but is of great help to someone wanting to upgrade their DSL code.

Let us look into the different changes one can make on a domain model and which consequences this has for a program in a DSL or one using a framework. The following list contains classifications of extensions of a domain model. The classifications are based on what kind of transformation is required to transform source code of one DSL implementation to source code of a new implementation.

[27] http://www.eclipse.org/modeling/mdt/uml2/docs/guides/UML2_2.0_Migration_Guide/guide.html

1. No transformation
2. A one-to-one transformation
3. A one-way transformation including user additions after the transition
4. A guided one-way transformation. The user must help the transformation
5. A completely manual transformation

The first category includes bug-fixes, refactorings and updates which extends a system without altering or influencing the existing code.

The second category contains transformations where the user code is guaranteed to contain the information required to perform the transition, but where a transition is required.

The third category contains transitions where there are new additions or the existing concepts are refined somehow. The resulting code, after the transformation, will likely contain holes where the user must fill in new or more detailed code.

The fourth category is when the user must supply information in order for the transformation to succeed. An example is creating a more general domain concept. It requires the old specific instantiations to be converted into the more general concepts. This is a known problem of reverse engineering something specific into something more general.[28]

The fifth category contains all other transformations. These are the transformations where relatively little transformation is possible, and most has to be done manually.

Updating over several versions is possible by combining transformations. When combining several transformations, the type of the new transformation is the type of the highest number on the classification list.

For the E-Commerce Language, the AST of the source code is always available. This makes transition-categories 1 to 4 feasible using a model-to-model transformation. The reason is that the AST of the language is a processable representation which is commonly used for model-to-model transformations, using languages such as MOFScript[29] and ATL[30].

For instantiations of the E-Commerce Framework, however, it is not as simple to make a transformation. Framework instantiations are not processable. Of course, category 1 is not a problem for the framework. For category 2 extensions, the framework does not have to be updated. One can supply a wrapper in the framework which accepts instantiations and interactions of the old type and translate them to the new version. For category 3 extensions and up, the framework must be modified by hand.

These considerations were on extensions to the E-Commerce Language and Framework. What about the programs using the constructs created in the framework instantiations and in the DSL programs? For example, a catalog can be programmed with the Catalog-Composition Language. But, a third program might interact with this catalog. This third program is the kind of programs in question now. The framework might be changed in order to improve the way these third kinds of programs interact with the domain applications. This requires a change in the system using, for example, the catalog.

[28] This is explained in [SV06]

[29] http://eclipse.org/gmt/mofscript/

[30] http://eclipse.org/m2m/atl/

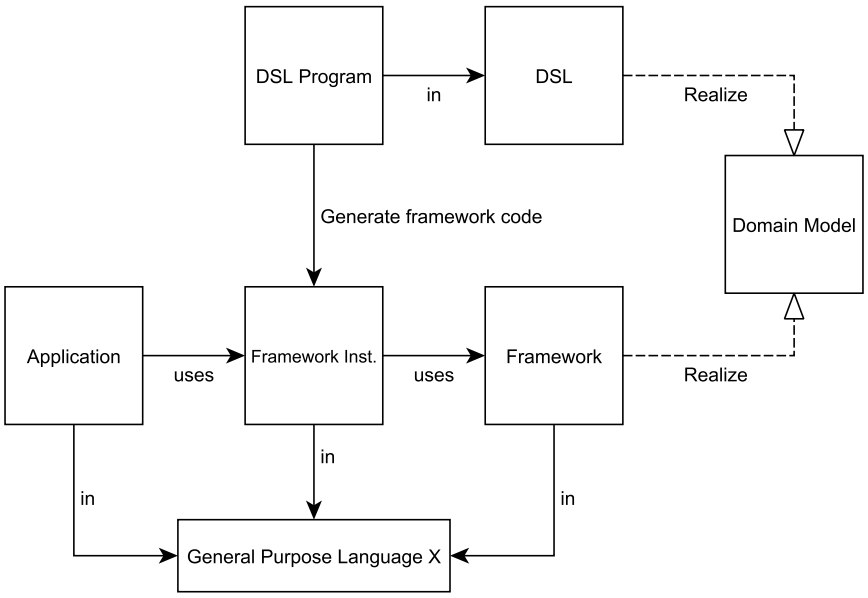

Figure 3.19: Non-UML diagram showing the relationship between frameworks, DSLs and domain models

This is also a problem for a program using a catalog programmed with the Catalog-Composition Language. The solution to this problem is probably manual conversion of the programs. Consider this example.

Example. The Catalog-Composition Framework stops using a catalog-interface to catalogs and decides navigating from one commodity or category to the other by association instead. A program has to start, instead of using a catalog interface, with an all encompassing category. Each commodity is, in addition, required to be within some category and the category graph is required to be connected. These modifications are, as a whole, a one-way transformation where additional user input is required, i.e. category 3. Hence the conversion of the code using the framework or the DSL code is not that difficult.

There are three programs in this example. We have the Catalog-Composition Language or the Catalog-Composition Framework. A second program is written to specify what is in a catalog. The third program uses this catalog in order to show it to a customer.

On update, the framework or the DSL compiler is replaced. A transformation helps the developer to transform the second program into supporting the new framework or DSL. When it comes to the third program, it is the architecture of this system which must change. The DSL or framework creator has no chance of knowing what the user of a program for a DSL or a framework does. Therefore automatic transformations are less feasible for this kind of program. □

This example shows that one must consider all three types of programs when updating a framework or a DSL. First one must replace (1) the DSL-compiler or the framework code. Then one must transform or manually handle (2) programs which use the DSL-compiler or the framework. Third one must manually update (3) the code which uses the resulting structures after having used the DSL or framework. The relationship between the different types of programs is illustrated in figure 3.19.

Other things which might be problematic on an update are other artifacts than source code. Even though the source code of a DSL program is converted, or the code using a

framework is correctly changed, there might still be database contents and files which are outdated by the change. These problems are not discussed further in this thesis.

3.4.7 DSL versus Framework specialization

In this subsection, what it means to specialize the E-Commerce Framework and whether it makes sense to specialize the E-Commerce Language are discussed. To *specialize* is defined as "become more focused on an area of activity or field of study"[31].

Example. The following is a list of example specializations outside the domain of software engineering.

- Laptop computers are specializations of computers for increased mobility
- Sport cars are specializations of cars for e.g. higher acceleration and better handling
- Computer monitors are specializations of screens for seeing details clearly
- Desks are specializations of tables for the purpose of e.g. paper- or computer-based work
- Easy-chairs are special chairs used for relaxation

□

Subtyping is the specialization of some type. Inheritance is seen as subtyping in this thesis. A framework consists of many classes. These classes may be sub-classed to create a new specialized framework for a special-domain of the framework's domain. For example, a specialization of the domain of e-commerce is selling cars online.

Framework specialization is different from framework instantiation. The latter is the creation of a concrete application based upon the framework. Extension is also different from specialization, and it has been discussed in subsection 3.4.6. Since specialization is not instantiation and not extension, it is simply making a system which is more focused on a certain special-domain.

The importance of an accurate representation

A special-domain is by fact a specialization of a domain. Whether or not this specialization is easy to implement depends on how accurately the representation of the domain is in an implementation of it. The term *accurate* is used to mean "without deviations from the actual facts". If an implementation is accurate, the specializations done within the domain is naturally reflected in the implementation, making it natural to specialize.

Example. Catalogs exist in the e-commerce domain. Within the special-domain of car commerce there exist specialized catalogs for presenting cars. If there does not exist a notion of a catalog in an e-commerce system, what is one then to specialize? It is no longer clear from the domain knowledge itself how to perform the specialization nor if it is possible in the software system. □

[31] wordnet

If a framework is an accurate implementation of a domain using a class model with OCL[32]-code, then the specialization cannot only concern itself with the classes; the OCL needs to be specialized as well. If other techniques other than these are used, then those have to be considered for specialization as well.

Given an accurate representation

Given that we actually have an accurate implementation of a domain, let us look at some aspects of how to perform the specialization. Both the E-Commerce Language and Framework are derived from the domain model. The specialization of these systems happens on the domain model, unless the specialization is to something not related to the domain. The question is how to implement the specialization for the DSL and for the framework, given a certain specialization of the domain model.

When a something is modified to become more specialized, such as for the example specializations listed above, it gets features which are improved with respect to the new special domain. For example, a sports car is improved with respect to fast driving. For software systems, the software probably gets more features which targets special problems within the special domain which were not applicable to the more general domain. The code using the specialized program is probably also specialized itself.

In case of frameworks this might mean more methods in the subclasses. Applications using the specialized system will probably use the new subclasses and not their unspecialized superclasses.

In case of DSLs, it might mean an extension of its grammar. Programs for the specialized DSL will use this specialized grammar. Applications using the constructs constructed by the DSL program will probably also use the extended programming-interface given to it. For example, a catalog specialized for cars will have special queries which are specialized for looking up cars.

The following is a list of things that may be done to specialize the E-Commerce Language.

- Add grammar elements
- Specialize semantics of existing constructs
- Have stricter static semantics

Example. Take the Catalog-Composition Language, which is a DSL for programming an online browse-able catalog described in section 3.3.1. The following is a specialization of the Catalog-Composition Language into a Car-Catalog-Composition Language.

Listing 3.37 shows the specialized semantics of some existing constructs of the Catalog-Composition Language. "->" denotes a specialization relation the way the arrow points. For example, a CarCatalog is a specialized Catalog which has features more focused on presenting cars. It is at the same time is less suited for selling, for example, aquarium fish.

[32]The Object Constraint Language is a part of the UML standard.

Listing 3.37: A specialization of the Catalog-Composition Language vocabulary

```
CarCatalog -> Catalog
CarType -> GeneralCommodity
Manufacturer -> Category
Car -> Commodity
CarPart -> Part
```

Listing 3.38 shows the additional grammar elements. "+:" denotes addition of grammar elements. For example, a car type gets additional grammar elements for describing different functional parts of a car.

Listing 3.38: A specialization of the Catalog-Composition Language grammar

```
Car +: CarType? Manufacturer?
CarType : "CarClass" "="
  ("SUV" | "MPV" | "Sedan" | "Cabriolet"
  | "Coupe" | "Pick-up" | ...)
  "RequiredFunctionalParts" ":"
  FunctionalPart ("," FunctionalPart)*
Manufacturer : "Manufacturer" "=" Value
CarPart +: "FunctionalParts" : FunctionalPart ("," FunctionalPart)*
FunctionalPart :
  "Entertainment" | "Comfort" | "Handling" | "Security" | ...
```

The following are some examples of stricter static semantics. They are related to the specialized grammar and vocabulary just mentioned. For example, the parts of a car-type, e.g. its wheels, windows, spoiler, steering wheel etc, must be assigned to some functional part of the car.

- Car types and its parts must be assigned to some defined functional part of the car.
- The functional parts of a car type must be matched in a concrete car.

□

3.4.8 Static-semantic analysis for frameworks

Static semantic analysis is one of the benefits of DSLs. Static-semantic analysis is in a language context performed after a DSL program is tokenized and parsed into an abstract syntax tree (AST). The parts of this syntax tree can then be verified according to several semantic constraints of the language. The semantics of a DSL is of course often domain-specific, so there are an increased number of checks possible for DSLs compared to a general purpose language. A general purpose language cannot have domain-specific checks, since it is general purpose.

The question is: Why does static-semantic analysis make less sense for frameworks than for DSLs?

If it is the case that static-semantic analysis is impractical for frameworks, then a study of why will probably give interesting difference between DSLs and frameworks. These aspects are worth an analysis of why this is the case. This would also contribute to guidelines on choosing between DSL and framework-based solutions. This question is the second part of the problem statement of this thesis, which is covered in chapter 4.

3.5 Summary

Using several general solutions. When building an application, it is quite common to base it on several frameworks. It is easy to use several frameworks if they are all implemented in the same language. External DSLs, such as the E-Commerce Language, do not have this de facto common ground. The E-Commerce Language has no support for a standard basis for using it together with other DSLs. The E-Commerce Framework was not designed with this in mind either, but it is anyway usable in Java programs.

Implicit architecture. The implicit architecture of a software system is the architecture implied by the source code, as opposed to the legitimate architecture of the system design. Due to the lack of static checking possible for frameworks discussed in chapter 2, the E-Commerce Framework cannot fully protect itself from code thwarting its internal behavior. Code written in the E-Commerce Language, however, only allows code which passes the parser and the static-semantic analyzer. Hence, the DSL can provide better protection of its internal behavior and better ensures that the implicit architecture does not deviate from the explicit legitimate architecture of a system.

Processability. Something is processable if it can be loaded into a data structure where all the relevant information is available and properly structured. An AST of the E-Commerce Language is processable. The instantiation code of the E-Commerce Framework is Java code. The Java code can be loaded into a Java AST, but this might contain bits and pieces from code not involved with the E-Commerce Framework, such as instantiation code for another framework. There is no built-in support in the E-Commerce Framework for loading the framework instantiation code into a processable structure. Hence, the DSL code is more processable than the framework instantiations.

Implementation accuracy. The implementation accuracy is a measure of how little deviation from the domain is allowed by the general solution, in applications based on it. Code written for the E-Commerce Language is checked by a parser and a static-semantic analyzer. The DSL allowed for more accurate restriction of which programs are allowed, than the framework. The primary reason for this is the lack of abstractions in Java to implement the e-commerce domain. Object-orientation is quite limited in expressing constraints as explained in chapter 2.

Extensions of general solutions. An extension of a general solution is to add extra functionality to it. This commonly happens to most software systems. The E-Commerce Language is more processable than an instantiation of the E-Commerce Framework. By utilizing this processability, one can implement a model-to-model transformation for an extension to the general solution. The users of the DSL can run this transformation on their DSL source code. The source code then supports the newer version. This transformation is of help even if only partial. There are three different applications affected when updating the E-Commerce Language: (1) The compiler and run-time system, (2) programs written using the DSL and (3) applications using constructs created in the DSL code. (1) and (2) can have partial automatic updates, while (3) is less predictable.

General solution specialization. Specialization of a general solution is making the solution less general, but better suited to a specialized domain. The implementation

accuracy of the E-Commerce Language makes it better suited for specialization than the E-Commerce Language. The reason is that observing the specialized domain itself is better reflected in the more accurate implementation of the domain. Three areas of specialization for the E-Commerce Language were identified. One can (1) add grammar elements to the grammar, (2) specialize the semantics of existing DSL constructs and (3) refine the static semantics.

Static-semantic analysis. When programming the E-Commerce Framework, the placement and the nature of the hot-spots were determined by the e-commerce domain. Hence, one is also programming in a domain specific way when instantiating the E-Commerce Framework. Why then, is not static-semantic analysis as simple to implement for the framework? This question is considered in depth in chapter 4.

3.6 Related work

3.6.1 Evaluation of the evaluation in [Deu97]

Another work which does a comparison between DSLs and frameworks is [Deu97]. This work is based on experiences from a financial engineering case study. The evaluation of the difference is presented as six points, about five lines each. Each of these evaluations is considered in the light of the experiences from this thesis.

Expressiveness

According to [Deu97]:

> A DSL provides a natural way to express the non-technical essence of a particular domain, increasing readability and portability. On the other hand, using a full general purpose language as in a framework provides more flexibility in adapting it to specific needs.

The increase in readability and portability is also experienced in this thesis. Listing 3.9 is more readable than if it is implemented with a framework such as in listing 3.26. It is also more portable in that it is not directly bound to any implementation language.

When it comes to the flexibility of adapting the framework to specific needs, this is experienced by the E-Commerce Framework being implemented in a general purpose language. It is hence possible to use several frameworks for an application and even in one sequence of statements, or start developing something completely different and still be able to use the framework directly. For DSLs there is less flexibility in this matter. The reason is that there is no de facto common ground between different DSLs.

Readability and portability are good features. Processability is a powerful feature of DSLs which is not mentioned in the article. The benefits of processability of the DSL comes from the fact that DSL code can be represented as an AST, where as no such representation is common for frameworks.

Legacy Libraries

According to [Deu97]:

> RISLA is an example of a DSL that was used to access a given existing library. This library was functionally entirely adequate, but written in a legacy language. A DSL can be used to provide access to such legacy libraries. In principle, wrapping could be used to achieve the same effect using an object-oriented language, thus basing a framework on an existing legacy library.

The experiences from this experiment agree that both a DSL and a framework might serve as a wrapper for a legacy library. The DSL implemented in the experiment is loosely coupled to the target language. This looser coupling gives a DSL a slight advantage as the legacy systems can be refactored more easily. A framework could be implemented in the same language as the legacy system or in a different language with a programming-interface to the old language. In either case, there would be tighter coupling than for a DSL.

The calling framework

According to [Deu97]:

> A framework often is an active entity: It does not get called (as a library), but it calls functions provided by the application developer. The same situation is easily obtained in a DSL setting: A RISLA product description defines functions for computing cash flows, which are called by other systems running at the bank. The fact that COBOL is generated from RISLA makes it easy for legacy systems to connect to information defined in RISLA.

Both DSLs and Frameworks can be used in an inverse manner. Instead of having DSL code and general purpose code calling the framework, the system can call and use things in the DSL or hot-spots. This inversion cannot differentiate between whether it is framework code or DSL code in the other end of the call.

Overriding Default Behavior

According to [Deu97]:

> So-called white-box frameworks allow the application developer to override default behavior using inheritance. Although this could be encoded in a DSL as well, this does not seem as natural as in a framework setting.

Overriding operations in a subclass is a hot-spot type of frameworks. One must answer why one has chosen subclassing. If it is a domain-specific reason for this, then there is no reason this cannot be done in a DSL.

Language Technology

According to [Deu97]:

> Interactive DSL development requires tools supporting the rapid prototyping of scanners, parsers, type checkers, interpreters, compilers, etc. [...]

Tool support is indeed an overhead for DSLs as opposed to frameworks. One could ask however, how useful it would be to get domain-specific tool support. This is usually what one programs in relation to a DSL, but not what one gets when programming a

framework. This was discussed in section 3.4.2. For frameworks, one can use the existing tools for the implementation language, but these will not contribute with domain specific tooling.

Mutual Benefits

According to [Deu97]:

> When developing a DSL from scratch (rather than developing it to access legacy systems), with freedom of choice for the target language, it is most natural to base the DSL implementation on a domain-specific object-oriented framework. When developing a domain specific framework, extending it with a DSL to access its functionality has a number of advantages:
>
> - It is a guide to the design of the framework. If there is no way to express a certain class or method as a language construct, it is likely that this class or method does not correspond to a natural concept of the domain.
> - It encourages the development of black-box frameworks (based on composition) rather than white-box frameworks (based on inheritance).
> - It gives more abstract access to the framework, hiding (encapsulating) what language is used to implement the framework.

The DSL presented in this thesis is compiler to code using a framework. The DSL was also implemented before the framework and did indeed guide the design of the framework. This is apparent by looking at the many similarities between the DSL and the framework. The code-generation step is also easier when the DSL and the framework are similar.

If one wants to use the E-Commerce Framework instead of the Language to build an e-commerce system, one does not exclude the DSL developed in this thesis. This is in agreement with [Deu97]. One can for example use the DSL for one part of the system, and generate code for that particular part. IDEs such as Eclipse support a distinction between generated and manually written code. Hence, the DSL is seen as a layer on top of the framework, giving more abstract access to it.

3.6.2 Minor work on the comparison

Evaluation of comparison in [HJE95]

[Deu97] mentions another paper [HJE95] which also did a comparison of using DSLs and frameworks for the same application in the domain of network protocol software. From [Deu97]:

> Network protocol software is another area where both a DSL (the Morpheus language [AP93]) and a framework (Conduits+ [HJE95]) have been developed. [HJE95] includes a short comparison, in which it is concluded that "although the framework is unlikely to achieve the same execution efficiency as a special-purpose programming language, it offers similar, but more easily extensible, composition facilities."

Frameworks were in this thesis experienced to have better composition facilities. The reason is that the framework is in a general purpose language which allows for composition with other frameworks produced in the same language.

When and how to develop domain-specific languages

[MHS05] argues that frameworks are a stage in the development of a DSL, and that most potential DSLs never progress further than frameworks. This was not experienced in this thesis. The reason is probably that the DSL was developed before the framework and that the DSL was planned from the requirements stage. It is questionable if this is a common practice for the system designers who developed the systems used in the statistics.

Chapter 4

Static-semantic analysis for a framework

4.1 Introduction

In the comparison of DSLs and frameworks in chapter 3 of this thesis, a question of why static-semantic analysis makes less sense for frameworks than for DSLs was raised. The purpose of this part is to do an experiment and to analyze this question.

If it is the case that static-semantic analysis is impractical for frameworks, then a study of why this is the case will probably give interesting differences between DSLs and frameworks. This would also contribute to guidelines on choosing between DSL- and framework-based solutions and also give insight into how to change an existing framework in order to support static-semantic analysis.

It is important to note that the purpose of the experiment is to gain insight into the problem in question. The analysis will take a more detailed look at the problem, and in fact concludes that the method used in the experiment is unnecessarily restrictive. The experiment helped reach this conclusion.

This chapter is organized as follows. First we take a deeper look at the question of static-semantic analysis for frameworks, and then we look at the experiment which was performed, finally an analysis of the problem and a look at related work.

4.2 The idea behind the experiment

All DSLs are not defined thoroughly (many are just providing domain specific syntax and code generation), but because DSLs are languages, they belong to a field with well-known concepts. Language implementations usually have a static semantic analyzer, and it is usually performed on an abstract syntax tree (AST). Meta models, by which some DSLs are defined, form the basis for ASTs in a more extensive way than grammars.

Using a framework is quite different from basic object-oriented programming. Instead of determining which classes to make, one implements the hot-spots. The placements and the natures of the hot spots are specific to the domain and therefore part of the framework. One has to follow the requirements of these. In a sense one is programming in a domain specific way, which is in a DSL.

Given this similarity, why not perform static-semantic analysis on the code which instantiates frameworks? One reason is that the information required for static-semantic

analysis is not available in some processable form such as an AST. The static-semantic analyzer works at compile-time and cannot instantiate classes to retrieve information. At least not in the way the experiment in this chapter is set up. The reason is that the instantiation process is programmed in general purpose code which might involve an arbitrary amount of work and side-effects and hence does not constitute compile-time.

Given that a hot-spot is implemented in the same language as the framework, the AST of the implementation language, such as Java, is available at compile-time. It should be possible to use this AST for static-semantic analysis. Reflection can be used to get access to this AST. A problem with the use of the AST is that there are features that are hard or impossible to determine at compile time. Examples of such features are the contents of a variable after the execution of a number of statements involving arbitrarily complex operations. Therefore, in order to access the AST to get information without instantiation, the information should be present in a form not too difficult to infer.

Examples of easy availability is, for example, a static final variable with a certain name, attributes with a certain super-type and methods with certain names, parameters or return types. The way these things can be accessed by the framework is through reflective code. This reflective code can anticipate where to find information in framework instantiations. For example, a framework class can anticipate that a subclass must have an attribute called `x`. Hence, it has some freedom in choosing where the information should be placed. The requirements of which attributes and methods the user of the framework should supply, should be constructed in such a way that both the domain concepts and the framework are natural to use.

The code in the framework that expects to find certain information in particular places in the subclasses, are examples of hot-spots. In fact it is a mechanism which looks like the abstract-notion in Java. If a class has an abstract method, the subclass must either specify a method or an operation for this abstract method. In the same way, if reflective code expects a certain constant variable, of a certain type to be present, this is equivalent to an abstract constant attribute. These meta-programming characteristics of reflections are used in order to program other types of hot-spots, other than those Java offers without reflection.

The static-semantic analyzer for the framework in this experiment is a program which uses reflection to access the information in framework instantiations. It construct the AST from the framework instantiations which is equivalent to the one produced by a DSL program with the equivalent behavior. The implementations of the concrete static checks are therefore the same for both the framework and for the corresponding DSL. This is also an argument for why the static semantics verifiable in a DSL is probably also verifiable for a framework if it is constructed by the technique of using reflection, as described for this experiment.

The resulting framework is similar to an internal DSL according to the definition of [Fow05]: "Internal DSLs morph the host language into a DSL itself." The framework is modified into something which resembles an internal DSL, but is still a framework.

Let us now look at the concrete experiment performed. The implementation of static-semantic analysis for frameworks is explained using an example of a computer shop. The modifications performed on the framework are contrasted with the original Catalog-Composition Framework.

4.3 The modified Catalog-Composition Framework

The Catalog-Composition Framework is modified in order to support static-semantic analysis. The description of the modified framework uses the same example as used in the description of the original framework in section 3.3.1. The first thing to look at it the definition of a new catalog seen in listing 4.1. The catalog class is identified properly as a catalog since it subclasses the class CatalogImpl.

Listing 4.1: Defining a catalog for the modified framework

```
public class ComputerShop extends CatalogImpl{
```

In the original Catalog-Composition Framework, the initial content of the catalog was set up in the constructor as seen in listing 3.21. Since the class is not to be instantiated when the static-semantic analyzer checks it, the catalog contents must be set up without doing it in the constructor. The solution was to construct the initial contents as static inner-classes[1]. Then, the constructor of CatalogImpl can search its subclasses on instantiation. For the example catalog in listing 4.1, it can search the static inner classes of ComputerShop. These type-declarations are then added to the catalog automatically, instead of being placed there manually as in the original Catalog-Composition Framework. The actual implementation of this feature can be seen in listing 4.8, in the constructor of the CatalogImpl class.

It is important to note that the catalog can still be modified at run-time as in the original Catalog-Composition Framework. The static-semantic analyzer is only available at compile-time for this experiment, however.

Listing 4.2 shows inner-classes which defines a general commodity called DesktopComputer by sub-classing instead of instantiating GeneralCommodityImpl as in the original framework. The Java class name is now used as the commodity type name.

Listing 4.2: Defining general commodities using the modified framework

```
static public class DesktopComputer extends GeneralCommodityImpl{
  ComputerBox box;
  RgbMonitor rgbMonitor;
  UsbKeyboard usbKeyboard;
  UsbMouse usbMouse;
}

static public class Good extends GeneralCommodityImpl{
  static Integer price;
  static String modeltype;
  static String modelnumber;
  static String image;
}
```

Since the commodity-types are now classes instead of objects, the Java classes can be defined as attributes to the type instead of adding attributes by type name. Java will hence ensure that types are defined before use.

The framework identifies the attributes as general parts because they are subclasses of GeneralCommodityImpl. The framework would recognize the Java attributes as framework attributes if their types are not defined as subtypes of one of the framework classes. Framework attributes also have to meet the string-constructor requirements as discussed

[1] A static inner-class in Java is a class defined within another class which can be instantiated without instantiating the parent class.

for the original Catalog-Composition Framework section 3.3.1. This is the case for the attributes of the Good class.

Since Good is a general commodity, the attributes are not required to be initialized. They must, however, be initialized for all concrete commodities, such as the ones in listing 4.4.

The general commodities used as attributes in 4.2 are defined in the same manner in listing 4.3. Inside the general commodity class, we see the interfaces are specified using a constant array.

Listing 4.3: Defining more general commodities using the modified framework

```
static public class ComputerBox extends Good{
  static final Class interfaces [] = new Class [] {
    RgbMale.class,
    UsbFemale.class, UsbFemale.class, UsbFemale.class, UsbFemale.class,
    Ethernet100mbsFemale.class,
    HdmiOutput.class,
    PhonoFemale.class,
  };
}
static public class RgbMonitor extends Good{
  static final Class interfaces [] = new Class [] {RgbMale.class};
}
static public class UsbKeyboard extends Good{
  static final Class interfaces [] = new Class [] {UsbMale.class};
}
static public class UsbMouse extends Good{
  static final Class interfaces [] = new Class [] {UsbMale.class};
}
```

Commodities which are not general are either defined as subtypes of the CommodityImpl class or as a final class which subclasses another general commodity. The final modifier is recognized by the framework as a sign that it is non-general.

The attributes of the Good super-class now has to be initialized. The attributes are re-defined, with the same name and type, and initialized. The fact that this is done for each subtype is checked by the static-semantic analyzer.

Listing 4.4: Defining more commodities using the modified framework

```
static final public class Inspiron530 extends ComputerBox{
  static final Class inCategories[]
    = new Class[] {Dell.class};

  static Integer price = new Integer("3490");
  static String modeltype = new String("Inspiron");
  static String modelnumber = new String("530");
  static String image = new String("insp_530_right_155x160.jpg");
}
static final public class G900HDA extends RgbMonitor{
  static final Class inCategories[]
    = new Class[] {BenQ.class};

  static Integer price = new Integer("1095");
  static String modeltype = new String("");
  static String modelnumber = new String("G900HDA");
  static String image = new String("377683.jpg");
}
```

In listing 4.4 an example for declaring concrete subtypes of the general commodities is shown. The in-category relationship is initialized with a constant array using the type-classes as specifiers of the relationship. For example, RX250OpticalMouse is now in the category Logitech. The type system of Java does not guarantee that each entry is a category. This is ensured by the static-semantic analyzer.

Categories are defined as in listing 4.5. Categories are also inner-classes, use sub-classing for declaration and have a constant array for in-category relationship specification.

Listing 4.5: Defining categories using the modified framework

```
static public class Manufacturer extends CategoryImpl{}
static public class Dell extends CategoryImpl{
  static final Class inCategories []
    = new Class [] {Manufacturer.class};
}
static public class BenQ extends CategoryImpl{
  static final Class inCategories []
    = new Class [] {Manufacturer.class};
}
static public class Trust extends CategoryImpl{
  static final Class inCategories []
    = new Class [] {Manufacturer.class};
}
static public class Logitech extends CategoryImpl{
  static final Class inCategories []
    = new Class [] {Manufacturer.class};
}
static public class FrontPage extends CategoryImpl{}
```

Listing 4.6 shows how a concrete desktop computer is defined. The part-types are refined to a subtype and identified as such by having the same name as an attribute in the super-type defined above. For example, box is refined from ComputerBox to Inspiron530 which is possible since it subclasses ComputerBox. This is checked by the static-semantic analyzer for the framework.

Listing 4.6: Defining a concrete subtype of a general commodity using the modified framework

```
static final public class DX2400 extends DesktopComputer{
  static final Class inCategories []
    = new Class [] {Dell.class, FrontPage.class};
  Inspiron530 box;
  G900HDA rgbMonitor;
  KB_1400S usbKeyboard;
  RX250OpticalMouse usbMouse;
}
```

Interfaces and their connectivity are specified in the same manner as the other constructs described this far and is seen in listing 4.7.

Listing 4.7: Defining interfaces using the modified framework

```
static public final class UsbFemale extends InterfaceImpl{}
static public final class UsbMale extends InterfaceImpl{
  static final Class connectableWith [] = new Class [] {UsbFemale.class};
}
```

4.3.1 Extracting the AST from Catalog-Composition Framework hot-spots

The following explains how the DSL-equivalent AST is extracted from the framework user-code without instantiating the defined classes.

A catalog is defined as a subtype of CatalogImpl. Every class in Java has a corresponding instantiation of the class java.lang.Class. One can work with this object without instantiating the class it refers to. This object allows us to access the inner classes of the catalog using Class[] getClasses(). The inner classes are the commodity-, category-, bundle- and interface-types. This corresponds to the grammar in listing 3.2.

The different types of commodities all have some of the following features: attributes, parts, a possibility of being in categories, have super types and have interfaces.

Parts. Parts are recognized as java attributes which types are subclasses one of the framework classes CommodityImpl, CategoryImpl, BundleImpl, InterfaceImpl. < U > Class < ? extends U > asSubclass (Class <U> clazz) is used to check whether a class is a subclass of one of these classes.

Attributes. If an attribute is not a part, it is an attribute. The type is checked for a string-only constructor using the Constructor getDeclaredConstructor (Class [] parameterTypes) method of the type's Class-object.

In categories. An attribute named inCategories is chosen as the place where in-category relationships are specified. The categories are simply listed in an array. getField ("inCategories") is used to fetch the Field-class of the attribute. Then get(null) is called on the field to get the static initialization. The entries into the array are then checked to be subclasses of CategoryImpl.

Supertypes. Super-types are specified as super-classes in Java. The Class getSuperclass() is used to fetch the super-class.

Interfaces. This is implemented similarly to category relationships. Instead the attribute named interfaces is used and the entries must be subclasses of InterfaceImpl.

The things explained in the previous paragraphs are used to construct AST parts for the DSL grammar listed in listings 3.4, 3.6, 3.8, 3.11 and 3.13.

4.3.2 Implementation

For the implementation of the original Catalog-Composition Framework the class CatalogImpl was shown in listing 3.33. Listing 4.8 shows some interesting things in the implementation of this modified class. The following is a description of some of these.

Storage. Catalog contents are stored a bit differently in the modified version. Commodity types, along the other types, are stored as a list of Class objects. The commodity instances are stored as a list of Commodity objects in commodities. The getObjectFromClass operation simply creates an object from a class using the no-parameter constructor.

The object created can be used to query, for example, a category for its member's commodities.

The constructor. The constructor uses the Class[] getDeclaredClasses() method to retrieve all the static inner-classes defined for the concrete catalog, that is the subclass. They are then stored in the catalog attributes.

getCommoditiesByType(Class<? extends Commodity> type). This method returns Set<Commodity>. In contrast to the same method in listing 3.33, this method takes a Class as a parameter. Since the class must be a subclass of Commodity, as defined by the generics, we know the type is in fact a commodity. Since Java classes are used, the sub-classing mechanisms of Java can be reused to check if commodities are subtypes. The cast operation tries to cast the object. If it fails, we also know that the commodity is not a subtype.

void add(Commodity commodity). This method works the same way as the one in listing 3.33 called addInstanceof.

Listing 4.8: The modified CatalogImpl.java for static-semantic analysis support

```
...
public class CatalogueImpl extends EObjectImpl implements Catalogue {
  // Types
  Set<Class<? extends Object>> classes;
  Map<Class<? extends Object>, Object> classToObjectMap;
  Map<Object, Class<? extends Object>> objectToClassMap;

  // Instances
  protected EList<Commodity> commodities;
  ...
  protected CatalogueImpl(){
    super();
    commodities = new BasicEList<Commodity>();
    classes = new HashSet<Class<? extends Object>>();
    classToObjectMap = new HashMap<Class<? extends Object>, Object>();
    objectToClassMap = new HashMap<Object, Class<? extends Object>>();

    // Add classes which are specified as inner classes
    Class [] classesArray = getClass().getDeclaredClasses();
    for(Class c : classesArray){
      if(
        instanceOf(c, Commodity.class) ||
        instanceOf(c, Category.class) ||
        instanceOf(c, Interface.class) ||
        instanceOf(c, CommodityChoiceSet.class)
      )
        add(c);
    }
  }
  public void add(Class<? extends Object> c) {
    if(classes.contains(c)) return;

    // Construct
    Object o = getObjectFromClass(c);
```

```
    // Set owning catalogue
    if(o instanceof Commodity) ((Commodity)o).setOwningCatalogue(this);
    if(o instanceof Category) ((Category)o).setOwningCatalogue(this);
    if(o instanceof Interface) ((Interface)o).setOwningCatalogue(this);
    if(o instanceof CommodityChoiceSet) ((CommodityChoiceSet)o).
       setOwningCatalogue(this);

    // Add
    classes.add(c);
    classToObjectMap.put(c, o);
    objectToClassMap.put(o, c);
  }
  public Set<Commodity> getCommoditiesByType(Class<? extends Commodity>
     type) {
    Set<Commodity> s = new HashSet<Commodity>();
    for(Commodity c : commodities){
      try{
        type.cast(c);
        s.add(c);
      }catch(ClassCastException e){}
    }
    return s;
  }
  public void add(Commodity commodity) {
    commodity.setOwningCatalogue(this);
    commodities.add(commodity);
  }
}
```

The static-semantic analysis for the Catalog-Composition Framework

Listing 4.9 contains some parts of the static-semantic analysis for the Catalog-Composition Framework. The following is a description of how it works.

The constructor. The constructor of the static-semantic analyzer takes every class in the catalog and places them in several lists. These lists are checked in many different checks executed at the end of the constructor.

void testCycleInCategoryMembership(). A bottom-first search is done on each category to see if there is a cycle in the graph of category members. If a cycle is found, an error-message is added to the list of errors. This list of errors is used by a client to inform the users about the static checks, for example, by printing the errors to the console during the building phase of the system.

void testSubTypeMustHaveSubtypeParts(). This method matches the parts of a type with the parts of its subtype. The static semantics demand that for each general part, the corresponding part in a subtype is also a subtype. This is checked by testing whether the corresponding parts can be cast the class in the super-type.

Listing 4.9: Static-semantic analyzer implementation for the modified Catalog-Composition Framework

```
...
public class TestConstraints {
  List<AnalysisMessage> messages = new ArrayList<AnalysisMessage>();
  List<String> errors = new ArrayList<String>();
  ...
  private void addMessage(Type type, String message, ErrorCodes ec, Class
      state){
    AnalysisMessage am = new AnalysisMessage();
    ...
    messages.add(am);
  }
  public List<AnalysisMessage> getMessages(){ return messages; }
  public int getErrorCount(){ return errors.size(); }
  ...
  private Set<Class> testCatalogues = new HashSet<Class>();
  private Set<Class> testCommodities = new HashSet<Class>();
  ...
  public TestConstraints(Class [] cs){
    testCatalogues.addAll(Arrays.asList(cs));
    for(Class testCatalogue : testCatalogues){
      for(Class c : testCatalogue.getClasses()){
        if(instanceOf(c, Commodity.class)) testCommodities.add(c);
        if(instanceOf(c, Bundle.class)) testBundles.add(c);
        ...
      }
    }
    // Perform tests
    testBasics();
    testBundleCannotHaveInterfaces();
    testCycleInCategoryMembership();
    testSubTypeMustHaveSubtypeParts();
    ...
  }
  private boolean categoryCycleSearch(Class cat){
    return categoryCycleSearch(cat, new HashSet());
  }
  private boolean categoryCycleSearch(Class cat, Set v){
    Field f = getFieldInHierarchy(cat, "inCategories");
    if(f == null) return false;
    Set<Class> cs = getFieldValuesInHierarchy(cat, "inCategories");
    for(Class c : cs){
      if(v.contains(c)) return true;
      v.add(c);
      boolean r = categoryCycleSearch(c, v);
      if(r == true) return true;
      v.remove(c);
    }
    return false;
  }
  public void testSubTypeMustHaveSubtypeParts(){
    for(Class g : testGeneralCommodities){
      ...
      for(Field tp : typeParts) // Parts in the type
        for(Field stp : superTypeParts) // Parts in all supertypes
          if(tp.getName().equals(stp.getName())) // With equal names
            if(!instanceOf(tp.getType(), stp.getType())) // And not
```

```
                subtyped parts
                addMessage(Type.Error, "Subtype must have corresponding
                    subtype parts to its supertype", ErrorCodes.
                    SubTypeMustHaveSubtypeParts, g);
    }
}
```

Unit test checker

The static-semantic analyzer can be wrapped into a unit test in order to integrate the analysis with the build-process of the framework instantiation. The static semantic analyzer must implement the interface in listing 4.10. In the same listing are a tuple class for the error massages of the static-semantic analyzer and an enumeration of the kind of feedback the static-semantic analyzer can give. A tuple of feedback contains the type of feedback being an error, a warning or simply information. Then comes the error-message string followed by the error code. Lastly, the tuple contains the class in which the error was found.

Listing 4.10: Types used for the static-semantic analyzer

```
interface StaticSemanticAnalyzer{
  List<AnalysisMessage> getMessages();
  int getErrorCount();
}
public class AnalysisMessage{
  public Type type;
  public String message;
  public ErrorCodes ec;
  public Class errorClass;
}
public enum Type{
  Warning,
  Error,
  Info,
}
```

The analyzer is wrapped into a unit test. In listing 4.11 a unit test for doing the static-semantic analysis is implemented using the JUnit 4 framework. The test is annotated with Test, as is the convention for JUnit 4. The wrapper instantiates the static semantic analyzer with the user-classes which uses the framework. In this case, we analyze our catalog ComputerShop and all its contents. After the class has been instantiated, getMessages is called which retrieves the messages from the analyzer. They are written to the console. Finally the test fails if there are errors in the analysis.

Listing 4.11: A static-semantic analyzer implemented as a unit test

```
...
public class TestSemanticChecker {
  public void test(){
    Class [] cs = {
      no.uio.ifi.heim.martifag.ccm.test.catalogues.ComputerShopExample.
          class,
    };

    TestConstraints tc = new TestConstraints(cs);
    List<TestConstraints.AnalysisMessage> messages = tc.getMessages();

    for(TestConstraints.AnalysisMessage m : messages)
      System.out.println(m.type.toString() + ": " + m.message +": "+ m.
          state);

    if(tc.getErrorCount()>0)
      fail("There were errors in the framework usage");
  }
}
```

4.4 The modified Entity-Behavior Framework

The Entity-Behavior Framework is also modified to support static-semantic analysis. The goal for the Entity-Behavior Framework is also to be able to extract the DSL equivalent AST and then use the same static-semantic analyzer code on the framework as on DSL code.

The framework is described using the same example which was used for the Entity-Behavior Language and the original Entity-Behavior Framework, figure 3.6 and listings 3.15 and 3.30. For an abstract example showing more of the features of the modified version of the Entity-Behavior Framework, see Appendix D.

Listing 4.12 shows the implementation of the Order behavior. Substates are here defined as entries into the substates array. Each entry is another array containing two entries. The first entry is the substate name; the second entry is the substate class. The array is defined as both static and final since the static semantic analyzer accesses the array using reflection and the Order.class Class object. The static semantic analyzer also makes sure there are only two entries per entry into substates and that they are of the correct type.

Transitions are defined using a similar technique as for sub-states. Each entry into the transitions attribute defines a transition. These entries contain arrays with four entries. These are the transition name, the source state, the trigger signal class and the target state. The transitions attribute is checked in a similar manner as the substates attribute. The transition names are used to call a method with the same name, using reflection. Within this method is where the action code of the transition is written. These methods take the source state and the signal object, which triggered the transition, as parameters. It is recommended that the Command-pattern is used for these actions. The methods are checked for consistency against the transitions attribute by the static-semantic analyzer. The reason the transition information is separated from the transition action is to allow the transition action methods to be overridden using Java's subclassing mechanism, where methods override methods in the superclass with the same signatures.

Non-composite states, such as Shipped, are defined by subclassing the framework class StateImpl.

Listing 4.12: Order behavior for the modified EBF part 1

```
public class Order extends XorStateImpl{
  static final Object [][] substates = {
    {"Initial", InitialStateImpl.class},
    {"Ordering", OrderingState.class},
    {"In Processing", InProcessingState.class},
    {"Shipped", ShippedState.class},
    {"Confirmed At Customer", ConfirmedAtCustomerState.class},
  };
  static final Object [][] transitions = {
    {"initial", "Initial", EpsilonSignalImpl.class, "Ordering"},
    {"addCommodityToOrder", "Ordering", AddCommodityImpl.class, "Ordering"
        },
    {"removeCommodityFromOrder", "Ordering", RemoveCommodityImpl.class, "
        Ordering"},
    {"toInProcessing", "Ordering", EpsilonSignalImpl.class, "In Processing"
        },
    {"toShipped", "In Processing", ShippedImpl.class, "Shipped"},
    {"toConfirmedAtCustomer", "Shipped", ConfirmedAtCustomerImpl.class, "
        Confirmed At Customer"},
  };
  void initial(InitialState from, EpsilonSignal s){}
  void addCommodityToOrder(OrderingState from, AddCommodity s){}
  void removeCommodityFromOrder(OrderingState from, RemoveCommodity s){}
  void toInProcessing(OrderingState from, EpsilonSignal s){}
  void toShipped(InProcessingState from, Shipped s){}
  void toConfirmedAtCustomer(ShippedState from, ConfirmedAtCustomer s){}
}

class InProcessingState extends StateImpl{}
class ShippedState extends StateImpl{}
class ConfirmedAtCustomerState extends StateImpl{}
```

The implementation of the OrderingState basically uses the same techniques used for the Order state in listing 4.12. Therefore, there is no need looking at it in detail.

A concern might be raised for the duplication of information for the transition-attribute and the Java operation signature which implement the action. First of all, using an implementation language which supports default parameter values would render the transition array unnecessary. The reason is that two parameters could be added to the transition methods and given default values. But, since Java does not support default values, the duplication is necessary. The consistency of the duplication is checked by the framework's static-semantic analyzer.

4.4.1 Extracting the AST from Entity-Behavior Framework hot-spots

Since a state is declared as a substate of either XorStateImpl or AndStateImpl, it can thus be identified as one of them using the Class < ? extends U > asSubclass (Class < U > clazz) operation of java.lang.Class. For statechart inheritance, the Class < ? super T > getSuperclass() method gets the state which is inherited from. This is the same as the grammar from 3.17 for the two productions: State and Inherit.

Both the transition- and the substates-attribute are accessed similarly to the inCategories-attribute in Catalog-Composition Framework: By using the Field getField(String name) method to get the field and then get(null) on the Field object to get the static initialization. This covers the parts of the grammar in 3.17 for the productions of AndState, XorState, SubState and Transition. Substates are found in the substates variable, transitions in the transitions variable and abilities are just transitions with the same from- and to-states.

The actions and the inheritance mechanism are solved as follows.

Actions. The action methods are synchronized with the transition variable by getting all methods in the class using the Method[] getMethods operation. Then use Class<?>[] getParameterTypes on the Method class to get the parameter types. They can hence be checked against the transition array entries

Inheritance. Inheritance is implemented by accessing the substate and transition attributes of each class up the hierarchy of classes. They are collected them into one set of states and transitions. Overriding happens for the states and transitions having the same name.

4.4.2 Implementation

The XorStateImpl implementation for the modified version of the Entity-Behavior Framework is seen in listing 4.13. This is in contrast to the original implementation of the class in listing 3.34.

The implementation of the two operations acceptSignal and getPossibleSignals is not so different from the original implementation. For acceptSignal we can see that the transition action is returned as a Java Method object. The method is called using reflection. getPossibleSignals is basically the same as for the original implementation.

Listing 4.13: The modified XorStateImpl.java which supports static-semantic analysis

```
...
public abstract class XorStateImpl extends CompositeStateImpl implements
    XorState {
  protected State inState;
  ...
  protected XorStateImpl() {
    super();
    // Register history state observer
    for(State s : substateObjects.values())
      if(s instanceof HistoryState)
        attach((HistoryState)s);
    onEntry();
  }
  public void onEntry() {
    inState = getSubstate("Initial");
    inState.onEntry();
  }
  public void acceptSignal(Signal s) {
    TransitionInfo t = getTransitionFromAndSignal(inState, s.getClass());
    if(t == null){ // Consume signal or send it down
      inState.acceptSignal(s);
      return;
```

```
    }
    if(!InCriteriaOk(t)){
      return;
    }
    // Call transition
    Method m = t.m;
    m.setAccessible(true);
    try {
      m.invoke(this, inState, s);
    } catch (IllegalArgumentException e1) {
    } catch (IllegalAccessException e1) {
    } catch (InvocationTargetException e1) {
      if(e1.getCause() instanceof RuntimeException)
        throw (RuntimeException)e1.getCause();
    }
    // Set new in-state
    State oldState = inState;
    if(oldState instanceof ShallowHistoryState){
      inState = ((ShallowHistoryState)inState).getHistory();
    }
    if(!(oldState instanceof ShallowHistoryState) || (inState == null))
      inState = t.target;

    if(!(inState instanceof HistoryState)) notifyObservers();
    inState.onEntry();
  }
  public Set<Class<? extends Signal>> getPossibleSignals() {
    Set<Class<? extends Signal>> result = new HashSet<Class<? extends
        Signal>>();
    Map<String, TransitionInfo> ts = getTransitions();
    for(TransitionInfo t : ts.values()){
      if(t.fromState==inState && InCriteriaOk(t)){
        result.add(t.s);
      }
    }
    result.addAll(inState.getPossibleSignals());
    result.remove(EpsilonSignalImpl.class);
    return result;
  }
  public boolean isIn(State s) {
    if(inState.getClass() == s.getClass()) return true;
    if(inState instanceof CompositeState)
      return ((CompositeState)inState).isIn(s);
    return false;
  }
}
```

The biggest difference is that we know that the implementations for the modified versions will not throw certain run-time exceptions. The things that can throw run-time exceptions have been checked by the static semantic analyzer. Some parts of the implementation of the static semantic analyzer are seen in listing 4.14.

The constructor. In the constructor the state machine classes are taken apart and placed in lists. Then 13 operations are run on these classes, each of them containing several static-semantic checks.

void testSignalFromStateUniqueness(). This operation checks the uniqueness of the combination of from state and signal. This check is only relevant for xor-states. For each xor-state, the from-states are collected into a set. The from-states are taken from the transitions attribute through the getTransitionsWithOverride operation. Then for each of the from-states, it is checked that each signal is only occurring once for each from state. If not an error is added to the list of errors used by the client of the static-semantic analysis.

void testEpsilonTransitionsWhereAllowed(). This operation checks that epsilon signals are only in the statechart where allowed. This check is also only applicable for xor-states. Hence, for each xor-state-check, a transition is found which is triggered by an epsilon signal. It is checked that it is either from an initial state or from a history state. If not, it is checked that the transition is from a composite which contains a final state. If not, then an error is reported.

Listing 4.14: The static-semantic analysis implementation for the modified Entity-Behavior Framework

```
...
public class TestConstraints {
  private Set<Class> xorstateClasses = new HashSet<Class>();
  private class TransitionClassInfo{ ... }
  List<AnalysisMessage> messages = new ArrayList<AnalysisMessage>();
  List<String> errors = new ArrayList<String>();
  ...
  private void addMessage(Type type, String message, ErrorCodes ec, Class
      state){
    AnalysisMessage am = new AnalysisMessage();
    ...
    messages.add(am);
  }
  public List<AnalysisMessage> getMessages() { return messages; }
  public int getErrorCount() {
    return errors.size();
  }
  public TestConstraints(Class [] cs){
    testStateMachines.addAll(Arrays.asList(cs));
    for(Class c : stateClasses){ // For all states
      if(instanceOf(c, XorState.class)) // If state is xor-state
        xorstateClasses.add(c);
    }
    // Perform tests
    testSignalFromStateUniqueness();
    testEpsilonTransitionsWhereAllowed();
    ...
  }
  private void testSignalFromStateUniqueness(){
    for(Class<? extends XorState> c : xorstateClasses){
      Map<String, TransitionClassInfo> ts = getTransitionsWithOverride(c);
      Set<Class> froms = new HashSet<Class>();
      for(TransitionClassInfo t : ts.values())
        froms.add(t.fromState);
      for(Class from : froms){
        Set<Class<? extends Signal>> signals = new HashSet<Class<? extends
            Signal>>();
        for(TransitionClassInfo t : ts.values())
          if(t.fromState == from){
```

```
            if(signals.contains(t.s))
              addMessage(Type.Error, "Two transitions from the same state
                  with the same signal exists " + t.fromState + " and " + t.
                  s, ErrorCodes.DuplicateTransition, c);
            signals.add(t.s);
          }
        }
      }
    }
    private void testEpsilonTransitionsWhereAllowed(){
      for(Class<? extends XorState> c : xorstateClasses){
        Map<String, TransitionClassInfo> ts = getTransitionsWithOverride(c);
        for(TransitionClassInfo t : ts.values()){
          if(t.s == EpsilonSignalImpl.class){
            if(!(instanceOf(t.fromState, InitialState.class) || instanceOf(t.
                fromState, HistoryState.class))){
              if(!instanceOf(t.fromState, CompositeState.class)){
                addMessage(Type.Error, "Epsilon transition only allowed from
                    InitialState, HistoryState or CompositeState under certain
                    conditions, not true for transition " + t.m.getName(),
                    ErrorCodes.EpsilonFromRightStates, c);
                continue;
              }
              Class<? extends CompositeState> cs = (Class<? extends
                  CompositeState>) t.fromState;
              int count = 0;
              for(Class s : getSubstates(cs).values()){
                if(instanceOf(s, FinalState.class))
                  count++;
              }
              if(count != 1)
                addMessage(Type.Error, "Epsilon transition only allowed from
                    CompositeState if it contains a final state", ErrorCodes.
                    EpsilonFromXorWithoutFinal, c);
            }
          }
        }
      }
    }
}
```

4.5 Discussion

The discussions first look at static checks and where they enter the design of a system. Framework hot-spots are categorized and these categories are then analyzed with respect to static-semantic analysis. The discussion ends with a look at what is required to get static-semantic analysis for frameworks.

4.5.1 Three categories of checks

The static checks implemented in the experiment fall in three categories. The following categories were identified. See Appendix C for the complete list of checks implemented for each category and Appendix A for the source code.

1. Static checks secured by the DSL implicitly (e.g. with grammars), which must be checked for frameworks

2. Static checks secured by the framework implementation, which must be checked for DSLs

3. Static checks required for both frameworks and DSLs

Example. Checks in category 1 are, for example, checking that a required attribute exists in an instantiation of a framework class, that it is of the right type, contains the right amount of entries and contains entries of the correct type. For example, the instantiation X of XorStateImpl, which is shown in listing 4.15, require these category 1 checks.

Listing 4.15: Example where static semantics must be checked for the modified framework

```
...
public class X extends XorStateImpl{
  static final Object [][] substates = {
    {"Initial", InitialState.class},
    {"A", A.class},
    {"B", B.class},
  };
...
```

Checks in category 2 are, for example, that types have been defined before use and that there are no cycles in the subtyping graph. Continuing the example in listing 4.15, Java requires that classes A and B are defined somewhere. For example, in the code in listing 4.16 the classes are defined, which satisfies the Java static-semantic analyzer and hence secures the category 2 constraint.

Listing 4.16: Example showing the definition of classes A and B

```
...
class A extends StateImpl{}
class B extends StateImpl{}
...
```

Checks in category 3 are more complex, interesting and often domain specific checks. For example, the check that the categories in the Catalog-Composition Language and Framework, do not have cycles in their membership graph. □

It is interesting to see how much more work must be done with the framework compared to what one gets for free from the grammar of a DSL. The size of the categories 1 and 2 varies with respect to how similar the domain concepts are to the host language. The checks in category 1 are often trivial, since they are checked using context-free grammars or simpler in the DSL. The checks in category 2 are due to similarities between the domain concepts and the concepts in the implementation language, which then might be used to implement the domain model. The checks in category 3 are the really interesting and complicated checks, which contains the domain specific checks found in the domain analysis. The checks in category 3 are the ones in focus for the rest of the discussion.

4.5.2 Domain analysis and static-semantic analysis

Since we find the same static checks for both the DSL and the framework designed in this thesis, it looks like static constraints belong to the domain. In that case, domain

analysis uncovers the static semantics of the domain. The static constraints are then found before the choice of either a DSL or a framework. Hence, instead of focusing on which choice can implement the most static checks, subsection 4.5.3 looks at how the domain concept-instantiations are expressed in framework hot-spots differently from in DSL code.

4.5.3 Framework hot-spot types

In frameworks there are hot-spots where the framework user specifies the particulars for his or her application. If we take the notion of hot-spots and apply it to DSL-code, we find that the only type of hot-spot for a DSL is DSL code. The following looks into which hot-spot types there are for frameworks.

The study of 14 hot-spot types of five frameworks[2], in addition to the ones developed for this thesis, resulted in the following categorization into three categories. The categorizations are described as follows.

1. Give the framework information in the form of a value of a data type
2. Use the framework interface in your class, then interact with it in different ways
3. Use the framework instantiations (in the form of classes) with an application

In the first category we find, for example, XML-configuration files used by frameworks, in-line DSL code given in the form of strings, for example HQL in Hibernate, framework class instantiations in the form of a subclasses and behaviors implemented in the form overridden methods.

In the second category we find, for example, framework classes which provide functionality. When a framework has been instantiated, the instantiated classes can be used. For an OR-mapper, a class is used to query the database for objects. In the Catalog-Composition Framework, a catalog class is used to query the database for commodities.

In the third category we find the use of framework instantiations in other programs. For example, the tomcat web-server requires a class which implements a certain interface. The framework classes can, after instantiation, support such an interface and hence be usable by the tomcat webserver.

4.5.4 A canonical form as a middle step

Listing 4.17 shows an example from the experiment in this chapter. How is its static constraints checked?

A commodity specification is a necessary part of an e-commerce system. The commodity specification must be entered into the system somehow. It can be entered either before compilation or at run-time. Either way, the static semantics of the commodity specification can be evaluated before the commodity is put into action in the system itself.

In a web-shop framework, a commodity-type is usually stored in a database together with a number representing the quantity of commodities available of that type. The full commodity specification can typically be extracted as a tuple. This tuple can be checked for consistency.

[2]Hibernate, Spring, EMF, Magento, JUnit

Take the domain concept of a general commodity from the experiment in this chapter. The general commodities are for example Mp3-players. An Mp3-player is a pack containing the mp3-player unit itself, but it must also contain headphones and a charger in order to be useful to the customer. A concrete subtype of an Mp3-player is an Apple iPod. An iPod is a proper Mp3-player as defined because it contains the iPod-unit, a couple of earplugs and a USB-charger.

Listing 4.17: A DSL example

```
GeneralCommodity Mp3Player
  mp3Unit : Mp3Unit
  headphones: HeadPhone
  charger: Charger

GeneralCommodity Mp3Unit
  interfaces: ChargerPlug, PhonoPlug

GeneralCommodity HeadPhone
  interfaces: PhonoPlug

GeneralCommodity Charger
  interfaces: ChargerPlug
```

[3]

The code in listing 4.17 might be inserted into a system in many different ways. It may be stored as source code such as in listing 4.17, as tuples in a database scattered over several tables (which is common for webshops), or as several java-classes as is the case for the modified Catalog-Composition Framework. If stored in either of these forms, the canonical form of the commodities as seen in listing 4.18 can be generated. If this representation can be created, the static semantics can be checked. The checks can be performed not only as a compile-time operation, but also at run-time by running a separate validator thread or process on the program or database contents. It is not a run-time check as long as the checks are reported to a developer and not used as a run-time exception to guide the execution of the program.

The code in listing 4.17 may be collected from any source and then transformed into the canonical form in listing 4.18 where the known and relevant concrete information is retained. A(x:X) means a commodity A having the part x of type X. B[X] means that the commodity B has the plug X. The two versions are identical. The first versions kept the identifier names to see how the transformation was performed.

Listing 4.18: Two versions of the same canonical form where one has kept the identifier names

```
Mp3Player(mp3Unit : Mp3Unit, headphones: HeadPhone, charger: Charger)
Mp3Unit[ChargerPlug, PhonoPlug]
HeadPhone[PhonoPlug]
Charger[ChargerPlug]

A(b : B, c: C, d: D)
B[E, F]
C[E]
D[F]
```

[3]The interfaces have been simplified to an interface-type which is the same on both sides. e.g. which does not come in a male and female version but where each plug is both male and female at the same time.

We can now state a couple of static constraint in terms of the parts of this canonical form. The parts of each element must form a connected graph by its interconnections. A is the only elements which contains parts. For A, C and D connect to B, which is a connected graph. It therefore passes the check.

This canonical form for static-semantic analysis for a language or a framework should be defined during the domain analysis. It does not seem to be bound to either a DSL or a framework implementation of a domain model, but seems to belong to it.

4.5.5 From framework instantiations to a canonical form

Static semantics is found to be identified in the domain analysis phase and it not specific to the DSL approach. The static semantics can be checked if a canonical form for static-semantic analysis can be created.

The following example looks into an example in order to consider why a canonical form may be hard to create from framework instantiation code.

Consider the Entity-Behavior part of the e-commerce system. For a statechart which is not intended to change at run-time there are several static constraints. One of these is the uniqueness of from-state and the trigger of every transition. As long as one is able to construct the canonical form, which in this case is a list of tuples of the following type: (transition name, from state, trigger), one can perform the static semantic analysis required for this check.

Listing 4.19 shows an example using the Entity-Behavior Framework. It constructs statecharts by means of method calls.

Listing 4.19: An example from the Entity-Behavior Framework

```
public class X extends XorStateImpl{
  public X(){
    addSubstate("A", new SimpleStateImpl());
    addSubstate("B", new B());
    setInitialState("A", new ActionToA());
    addTransition("A", "A", new ActionAtoA(), SigA.class);
    addTransition("A", "B", new ActionAtoB(), SigB.class);
    addTransition("B", "A", new ActionBtoA(), SigA.class);
  }
}
```

If this construction is the same for each execution, then the resulting statechart's internal structure can be analyzed to find the list of tuples. If the construction of the statechart depends on variables, other input data or otherwise complex general purpose code, then one cannot create the representation *since there is no statechart*. What one has is actually a more complex structure than a statechart.

Take for example a statechart where two transitions are conditional on construction of the statechart. The structure one must analyze is no longer a statechart. One of the transitions now actually has an [if x] constraint on it, the other has an [if y]. The constraint means in this context that the transition will either be present in the statechart or not, when constructed. The DSL corresponding to this extended statechart cannot be checked either. So this is not specific for frameworks. Consider the Entity-Behavior example shown in listing 4.20 which include some C-preprocessor-like commands. The corresponding framework code using the Entity-Behavior Framework is shown in listing 4.21.

Listing 4.20: Example using the Entity-Behavior Language where the canonical form cannot be constructed

```
XorState X
  a : A
#if x
  -> b / SigS
#endif
#if y
  -> c / SigS
#endif
  b : B
  c : C
```

Listing 4.21: Corresponding Entity-Behavior implementation of listing 4.20

```
public class X extends XorStateImpl{
  public X(boolean x, boolean y){
    addSubstate("a", new SimpleStateImpl());
    addSubstate("b", new SimpleStateImpl());
    addSubstate("c", new SimpleStateImpl());
    if(x)
      addTransition("a", "b", new ActionAtoA(), SigS.class);
    if(y)
      addTransition("a", "c", new ActionAtoB(), SigS.class);
  }
}
```

How can the static-semantic analyzer know whether "x and y" can ever be true? This depends on some other mechanism which must also have static semantic rules in order to be evaluated. If x and y is only known at run-time, then so is the "from state, trigger"-uniqueness constraint.

Take a framework based on the findings of a domain analysis. Say also that the domain contains static semantic constraints. One must make sure that one does not implement a more general or powerful concept than the one found in the domain analysis. If it is more general, then the static semantic constrains must be reconsidered for this more general domain. This is a problem for both DSLs and frameworks, since the static semantic constraints are a result of the domain analysis.

It might be hard to implement the domain concepts accurately in a framework. This might be because the user can do a lot in the code to mistreat the functioning of the framework. The framework should do as much as it can to not let the user hinder its proper functioning. In case it is hard to get the framework accurate to the domain concepts, a DSL is a better choice.

Is it hard or impractical to get the information out of a framework instantiation which does in fact accurately implement the domain concepts it is supposed to do? Or, the other way around, is it hard to program a framework in such a way that the domain concepts are truly implemented? This was at least not a problem with the modified E-Commerce Framework described in this report.

Even if the technology does not bar users from doing things which break the intended behavior, the framework developers could give the users certain instructions. The Entity-Behavior Framework manual could give directions as follows. Construct the statechart without using flow- control-structures of the host language. Simply build the statechart in a sequential fashion in one block. Also, build the statechart in an operation which might be called to retrieve the statechart without running the rest of the system, for example

just in a unit test. This statechart can then have its static semantics checked. Listing 4.22 shows an example.

Listing 4.22: Another way of having static-semantic analysis for frameworks

```
public class X extends XorStateImpl{
  public X(){
    addSubstate("A", new SimpleStateImpl());
    addSubstate("B", new SimpleStateImpl());
    addSubstate("C", new SimpleStateImpl());
    addTransition("A", "B", new ActionAtoA(), SigA.class);
    addTransition("A", "C", new ActionAtoB(), SigB.class);
    /* This operation locks the statechart
       to its current structure */
    lock();
  }
}

...
// Before run-time
XorState x = new X();
Results result = staticSemanticAnalyser.check(x);
System.out.println(results);
...
```

It is not a problem to have flow control within the statechart construction code if this is for other things. One could for example vary the action of a transition on each run without this being a problem. The criterion is that what has to be checked must be constructed according to the requirements of the concepts being implemented. Whether or not a transition is present is relevant. The presence or not of an action is not relevant to the example static check.

Also, if the framework was changed this way, the statechart construction operation no longer have to throw an 'illegal structure'-run time exceptions, since the structure is checked at compile-time. Hence the technique moves checks from run-time into compile-time. This makes the implementation code simpler, when the statechart-structure integrity concern is separated out to the static-semantic analysis.

The general mechanisms in the general purpose language seem to be the cause of the difficulty of implementing domain concepts accurately in a framework. Things in the host-language which must be considered with respect to domain-concept integrity is flow-control, annotations, generics and other features of the host language.

4.6 Summary

Static check classification. Many static checks have been implemented for the modified E-Commerce Framework. These checks fell into three classifications. Class (1) is checks required for the modified E-Commerce Framework which are guaranteed by the E-Commerce Language grammar. Class (2) is checks which are required for the E-Commerce Language but are guaranteed by the modified E-Commerce Framework, since it reuses the static checks found in the Java language. Class (3) is checks which are required for both the E-Commerce Language and for the modified E-Commerce Framework. This class contains most of the domain-specific checks. This class of static checks is therefore the focus for the rest of the discussion.

Relation to the domain model. The same static checks were successfully implemented for both the E-Commerce Language and for the modified E-Commerce Framework. What the two systems have in common is the domain model. It was noticed that the static checks can be described in the domain model and found during the domain analysis. This result led the analysis to consider the question of how well the modified E-Commerce Framework implements the domain model in comparison to the E-Commerce Language.

Hot-spots. Instantiation of the modified E-Commerce Framework are code for its hot-spots. The hot-spots are the means with which the modified E-Commerce Framework implements the domain model. It is the nature of these hot-spots that determine whether the modified E-Commerce Framework can support static-semantic analysis.

A canonical form. Since static semantics are a part of the domain model, then there should be an implementation-independent form with the information required to perform static-semantic analysis. If a canonical form can be reached, then the static semantic analysis is also possible for the modified E-Commerce Framework. As long as the target is the canonical form, it does not matter whether the source is in terms of tuples in a database scattered over several tables, java-classes which contents are studied by reflection or an AST.

Enabling static-semantic analysis. The canonical form requires an accurate implementation of the e-commerce domain model. This is a weak point for the E-Commerce Framework and is also found to be the reason for why static-semantic analysis is harder for the E-Commerce Framework. If the implementation of a domain model is inaccurate, the canonical form created is different than what the domain model and the static checks require. If the E-Commerce Language was modified to produce the same canonical form, then the two implementations are equally accurate, and the static semantics are as hard to check.

4.7 Related work

It is taken for granted that static semantic analysis is not an issue for frameworks, while it is obvious for a DSL because it is a language. In [MHS05] it is said that DSLs have their power in being languages (being more expressive than frameworks, and being subject to static semantic analysis), but that most of them never come beyond being frameworks that are embedded in general purpose languages. The focus in [MHS05] is to give guidelines on when and how to develop DSLs, while this thesis notes that even if one decides not to develop a DSL, but rather stick to a framework, then one may still get the benefit of static semantic analysis that one would get with a DSL.

In [BV04] the idea is that class libraries and frameworks encapsulate domain knowledge, e.g. in terms of interfaces to these classes, and that this ought to be available in terms of concrete domain specific syntax, not just in terms of method calls in the form of the general purpose language in which the classes are made. Therefore the domain specific syntax is embedded into the host language. The benefit with embedding is that the features of both the host language and its tools are available, e.g. static semantics and type checkers.

[HORM08] takes the embedding a little further. The domain specific knowledge is still represented by interfaces, but one may have several implementations (and thereby different semantics) of the same interface.

Compared to these approaches, the approach in this thesis is equivalent to embed a DSL into a framework. That is, it works for the cases where you would rather not make a DSL, but stick to a framework, and still get the benefit of a DSL in terms of static semantic analysis.

[Tra08] has in the section called "8.3 Reporting DSL Defined Errors at Compile-Time", which is a part which uses reflection to perform static-semantic analysis. The DSL in question here is an internal DSL into the Converge language.

[Ekm06] tells about OpenJava.

> OpenJava [TCoKI00] adds a macro system to Java that uses a meta-object protocol (MOP), similar to Java's reflection API, to manipulate the program structure. [...] The MOP can be used to add additional analyzes on top of Java [...].

OpenJava uses a very similar method to perform static checking using reflection. Listing 4.23 shows an example program[4]. In this example, a new qualifier is added to Java. An Overriding qualifier is used on an operation which is meant to override a superclass method. The way this is checked in OpenJava is using the reflective methods getDeclaredMethods and getModifiers, to determine if a method has the required modifier.

[Ekm06] also talks about another system for extending Java.

> The most flexible solution for language extensibility is to provide support for extensions at both the syntactic and static-semantic analysis level. Polyglot is an extensible source-to-source compiler framework implemented in Java that relies on design patterns for extensibility, e.g., abstract factories, extensible visitors based on delegation, and proxies [NCM03].

Polyglot performs a source-to-source compilation from a language which is a modified version of Java. The modifications are the extensions to Java and the modifications are checked using a custom static semantic analyzer on the AST produced by this first compilation stage.

[DDLM07] is a paper about having sub-method reflection. This is interesting with respect to frameworks in that the contents of a method can be checked at compile-time using reflection. Sub-method reflection is also discussed as a possible solution in 3.4.5.

[4] http://www.csg.is.titech.ac.jp/openjava/OpenJava_1.1/tutorial/Override.html Confirmed available 21. April 2009

Listing 4.23: OpenJava example of static-semantic analysis using reflection

```
import openjava.mop.*;
import openjava.ptree.*;
public class OverrideCheckerClass instantiates Metaclass extends OJClass
{
  private static final String OVERRIDING = "overriding";

  public static boolean isRegisteredModifier( String keyword ) {
    if (keyword.equals( OVERRIDING )) return true;
    return OJClass.isRegisteredModifier( keyword );
  }

  public void translateDefinition() throws MOPException {
    OJMethod[] methods = getDeclaredMethods();
    for (int i = 0; i < methods.length; ++i) {
      if (! methods[i].getModifiers().has( OVERRIDING )) continue;
      String name = methods[i].getName();
      OJClass[] ptypes = methods[i].getParameterTypes();
      try {
        getSuperclass().getMethod( name, ptypes, this );
      } catch (NoSuchMemberException e) {
        System.err.println( "warning:␣␣" + methods[i] + "␣doesn't␣" +
                  "override␣any␣method␣in␣the␣superclasses." );
      }
    }
  }
}
```

Chapter 5

Conclusion

This thesis reports from an experiment in two parts. The first part was implementing the same general solution for an e-commerce domain as a DSL and as a framework. The second part was modifying a framework in such a way that it supports static-semantic analysis.

For the first part, both the DSL and the framework were implemented successfully. The benefits of the DSL version were determined to be as follows. The DSL was experienced to ensure working at the right abstraction level and thereby avoid the temptation for creating hack-solutions, over simplified solutions to problems which require a larger change. The DSL helps protecting the internal functioning of the system, since the user of a DSL does not have access to the constructs used to build the DSL. The DSL is a bit easier to extend because the DSL has a digital representation which allows for transforming the source code to a new extended version. Specialization of DSL are also a bit easier, the reason is that the DSL has the ability to implement the domain more accurately.

The framework was experienced to have another set of benefits. It was simpler to make, as the accuracy of the implementation is not as important as for DSLs. Classes which are parts of the framework and which the user makes, can be reused together with other frameworks in the object-oriented language. For a DSL, the constructs may be custom to the DSL, and cannot therefore directly be used together with other constructs from different DSLs. Using several framework to build an application is simpler than using many DSLs.

The question of whether static-semantic analysis is possible for frameworks leads to the second part of the experiment. For a framework one has to implement constraints normally implemented with grammars for DSLs as hand-coded static checks. The domain model was found to be the true place for static semantics. Since the domain model is common for both the DSL and the framework, the discussion lead to asking why it is more difficult for frameworks to support static checks. The reason was found to be the difficulty of implementing the domain model accurately. If a DSL is an as inaccurate implementation of the domain model as a framework, then the static checks are as hard for the DSL. Hence it is the actual domain model implemented which must be checked, and not the planned one.

The experienced benefits of the DSL and the framework are summarized in table 5.1. In this summary, the benefits are grouped into a common reason for why the benefits apply to either the DSL or to the framework. "One language" means the benefit is due to the implementation of a framework is in the same language as framework is used. "Accuracy" means that the DSL is a more accurate implementation of the domain model.

Table 5.1: Summary of comparison between the DSL and the framework

Feature	DSL	Framework	Reason
Domain specific concrete syntax	+	-	Other
Domain specific static semantic analysis	+	-	Accuracy
Static checks are a natural part	+	-	Other
Easier to make	-	+	Other
Constructs can be used in the host language	-	+	One language
Existing tools can be used	-	+	One language
Basing an app. on several systems	-	+	One language
Working at the right abstraction-level	+	-	Other
Integrity of the system's internal functioning	+	-	Accuracy
Extensibility	+	-	Other
Framework vs. DSL specialization	+	-	Accuracy

The primary thing lacking for the framework is being able to do an accurate implementation of the domain model. It would be interesting to know if a language such as Java could be extended in such a way to better allow for an accurate implementation of domain models in general. If it was possible, then the benefits of DSL would be applicable to the framework to a higher degree.

If a general purpose language is well suited to describe domain models accurately, then the language could be a candidate for a standard language. What are the missing features of a language such as Java which would make this possible is an interesting question to answer. Such a language could serve as the common ground for domain models, breaking what [Fow05] calls the *symbolic barrier* between DSLs and general purpose languages.

Bibliography

[AP93] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1:27–38, 1993.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, and Peter Sommerlad. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[Bos98] Jan Bosch. Design patterns & frameworks: On the issue of language support. In *Object-Oriented Technologys*, pages 133–136. Springer Berlin / Heidelberg, 1998.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, May 2005.

[BV04] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM.

[Chr09] Steve Christey. 2009 cwe/sans top 25 most dangerous programming errors. http://cwe.mitre.org/top25/, January 2009.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):275–295, 2007.

[Deu97] A. van Deursen. Domain-specific languages versus object-oriented frameworks: A financial engineering case study. In *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39. Ilmenau Technical University, 1997.

[Ecl08] Eclipse Foundation. The eclipse modeling framework (emf) overview, July 2008. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html.

[Ekm06] Torbjörn Ekman. *Extensible Compiler Construction*. Doctoral dissertation, Department of Computer Science, Lund University, Box 118, SE-221 00 Lund, Sweden, 2006. http://progtools.comlab.ox.ac.uk/members/torbjorn/documents/EkmanThesis.pdf.

[Fow05] Martin Fowler. Language workbenches: The killer-app for domain specific languages? No note, 2005.

[FS97] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[Gli95] Martin Glinz. An integrated formal model of scenarios based on statecharts. In *Proceedings of the 5th European Software Engineering Conference*, pages 254–271, London, UK, 1995. Springer-Verlag.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HJE95] Hermann Hueni, Ralph E. Johnson, and Robert Engel. A framework for network protocol software. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, 1995.

[HORM08] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of dsls. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 137–148, New York, NY, USA, 2008. ACM.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[Lau01] Anthony Philip James Lauder. *A PRODUCTIVE RESPONSE TO LEGACYSYSTEM PETRIFICATION*. PhD thesis, THE UNIVERSITY OF KENT AT CANTERBURY, 2001.

[LW01] Barbara H. Liskov and Jeannette M. Wing. Behavioural subtyping using invariants and constraints. In *Formal methods for distributed processing: a survey of object-oriented approaches*, pages 254–280, New York, NY, USA, 2001. Cambridge University Press.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[ND04] Oscar Nierstrasz and Marcus Denker. Supporting software change in the programming language. *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[Pre94] Wolfgang Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, pages 150–162, London, UK, 1994. Springer-Verlag.

[SB04] Ehikioya SA and Ola B. A comparison of formalisms for electronic commerce systems. *Computational Cybernetics, Second IEEE International Conference on*, pages 253–258, 2004.

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99, February 2001.

[SV06] Thomas Stahl and Markus Völter. *Model Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[Swe08] Tim Sweeney. Unrealscript language reference, May 2008. http://udn.epicgames.com/Three/UnrealScriptReference.html.

[TCoKI00] Michiaki Tatsubori, Shigeru Chiba, Marc olivier Killijian, and Kozo Itano. Openjava: A class-based macro system for java. In *in Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.

[WK05] Jos Warmer and Anneke Kleppe. Why combine uml and ocl?, December 2005. http://www.klasse.nl/ocl/ocl-reasons.html.

Appendix A

Implementation and source code

The source code is available for download at the following URL: http://heim.ifi.uio.no/martifag/masterthesis/experimentsrc.zip

The source consists of several Maven projects. Each Maven project has a standard Maven directory layout as seen in figure 3.15.

- ECL - The Catalog-Composition Language and the Entity-Behavior Language compilers
- CCF - The Catalog-Composition Framework
- EBF - The Entity-Behavior Framework
- CCFSSA - The Catalog-Composition Framework with static-semantic analysis
- EBFSSA - The Entity-Behavior Framework with static-semantic analysis using reflection
- ECF - Webshop framework component integrating the Catalog-Composition Framework and the Entity-Behavior Framework
- ECFShopManager - The webshop salesclerk, providing a layer on top of the ECF project
- ECFWebView - The view component for accessing the webshop with a browser

A.1 Building and viewing the source code and models

The source code can be accessed by just unzipping the source archive and using maven. This section guides you through the process of setting up the development environment for viewing the source code and the models.

The development-environment requires the following software.

- Sun Microsystems - Java Standard Edition 1.6
- Apache - Maven 2.0.9
- Eclipse - Eclipse Ganymede 3.4.1

- Eclipse - Java Development Tools 3.4.2
- Sonatype, Inc. - Maven Integration for Eclipse 0.9.7
- Eclipse - EMF 2.4.0
- CEA LIST - Papyrus UML2 Editor 1.11.0

In order to run the web system in a browser, the following client side software is needed. Also, setting java-permissions listed in listing A.1 is required.

- Sun Microsystems - Java Plug-in 1.6.0_13
- Mozilla - Firefox 3.0.8

A.1.1 Importing into Eclipse

After having installed the required software, open eclipse and select "File", "Import ...". Expand "General", choose "Maven projects" and click next. Select the unzipped project archive. Choose between importing CCF and EBF or ECL, CCFSSA, EBFSSA, ECF, ECFShopManager and ECFWebView. The reason is that there are namespace conflicts between the -SSA-versions of two of the projects. These two sets of projects must hence be imported into separate eclipse workspaces. The -SSA-versions are modifications of the original projects with support for static semantic analysis. Due to this modification of the same projects, the namespace conflict occurs. Once having chosen the projects, the eclipse-maven-plug-in will configure Eclipse according to the project POMs, download the required dependencies and build the workspace.

The ECFWebView project requires a bit of work in order to get running. This is discussed in a subsection A.2.

A.1.2 What the source contains

The seven projects are each structured as standard maven projects.

In src/main/model the class models of each project are found. The .ecore file is the model itself. The .ecorediag is the diagram-file containing several diagrams for each ecore model. The .genmodel file contains information for the generation of the model code.

In src/test/java the test classes are found. These can be run in Eclipse by right-clicking on the project and selecting "Run As" and then "JUnit Test".

In src/main/java the implementation is found. The interfaces are in the base package, while their implementation is in the impl package. If the class model is changed, the genmodel file can be used to synchronize the interfaces and the implementation code in this folder. Open the genmodel file, right click on the top-level element in the tree view, and select "Generate Model Code". All the project code is synchronized with the class model. Sometimes the "Generate Model Code" feature causes some errors which simply can be ignored.

A.1.3 The E-Commerce Language Project

The project called ECL is a bit different from the other projects. The reason is that the project contains the implementation of two parsers written using grammars for the ANTLR system. The src/main/antlr directory contains the two parser implementations, Ccl.g and Ebl.g.

One manual operation is required to get this project working. Right-click and select "Maven", "update project configuration".

The src/test/ directory contain two directories with DSL-examples for the two DSLs. These are compiled during the test-phase of project build. The results are placed in the target/generated-code/java directory. The code generated works with the CCFSSA and EBFSSA frameworks.

A.2 The webview

A.2.1 Starting the webview

The Java policies must be set in order to allow the client side applet to run. Open java.policy in the Java installation folder and add the following lines to the "default permissions granted to all domains". The policy changes are needed because the webshop-system is only an experiment and not a completed product. Remember to remove these policies when done testing the web system.

Listing A.1: Permissions required in java.policy

```
permission
  java.lang.reflect.ReflectPermission
  "suppressAccessChecks","";
permission
  java.lang.RuntimePermission
  "accessClassInPackage.sun.plugin.dom.html","";
permission
  java.lang.RuntimePermission
  "accessDeclaredMembers","";
```

Prebuilt version

The webview is built and available at the following URL: http://heim.ifi.uio.no/martifag/masterthesis/ECFWebView/index.xhtml.

Build from source

In order to build the webview, install all the seven projects using mvn install, then do an mvn assembly:directory on the ECFWebView project. The command line commands are shown in listing A.2.

Listing A.2: How to build the webshop example system

```
cd CCFSSA
mvn install
cd ../EBFSSA
mvn install
cd ../ECF
mvn install
cd ../ECFShopManager
mvn install
cd ../ECFWebView
mvn assembly:directory
```

As a result of executing this code, the completely built webshop system is placed in this directory ECFWebView/target/ECFWebView-1.0-SNAPSHOT-distribution.dir. Open index.xhtml in Firefox in order to run the system. A web server is not required to run the webshop application, as it is not prepared to run on a web server in this version.

A.2.2 Screenshots of the webview running

Figure A.1 shows the front page of the webshop. This webshop is based on a very small catalog containing a few commodities. On the left-hand side, we can see the view of the categories and their subcategories. This is a standard tree-view found in most webshops. In the center we see the commodities in the currently selected category, which is FrontPage. Each commodity can be added to the current order, shown on the right.

After having added a few commodities to an order, the order details are shown in figure A.2. This page requires some explanation. On the top we see the commodities in the order. Under them we see a list of signal names. These signals are the ones which will trigger a transition for the order behavior. The signal called AddressDetailsImpl has HTML code associated with it. This HTML code lets the user construct a signal object of AddressDetailsImpl. Such HTML code has not been associated with the other signals in this version of the webshop. The link called Next initializes the signal and sends it to the order. After having completed the checkout process, the view looks like figure A.3. In this view, the order behavior has fewer signals which will cause a transition. This protects the order from invalid signals, such as when the customer decides to use the back-button to try to do other things. The order is now waiting on the ShippedImpl signal.

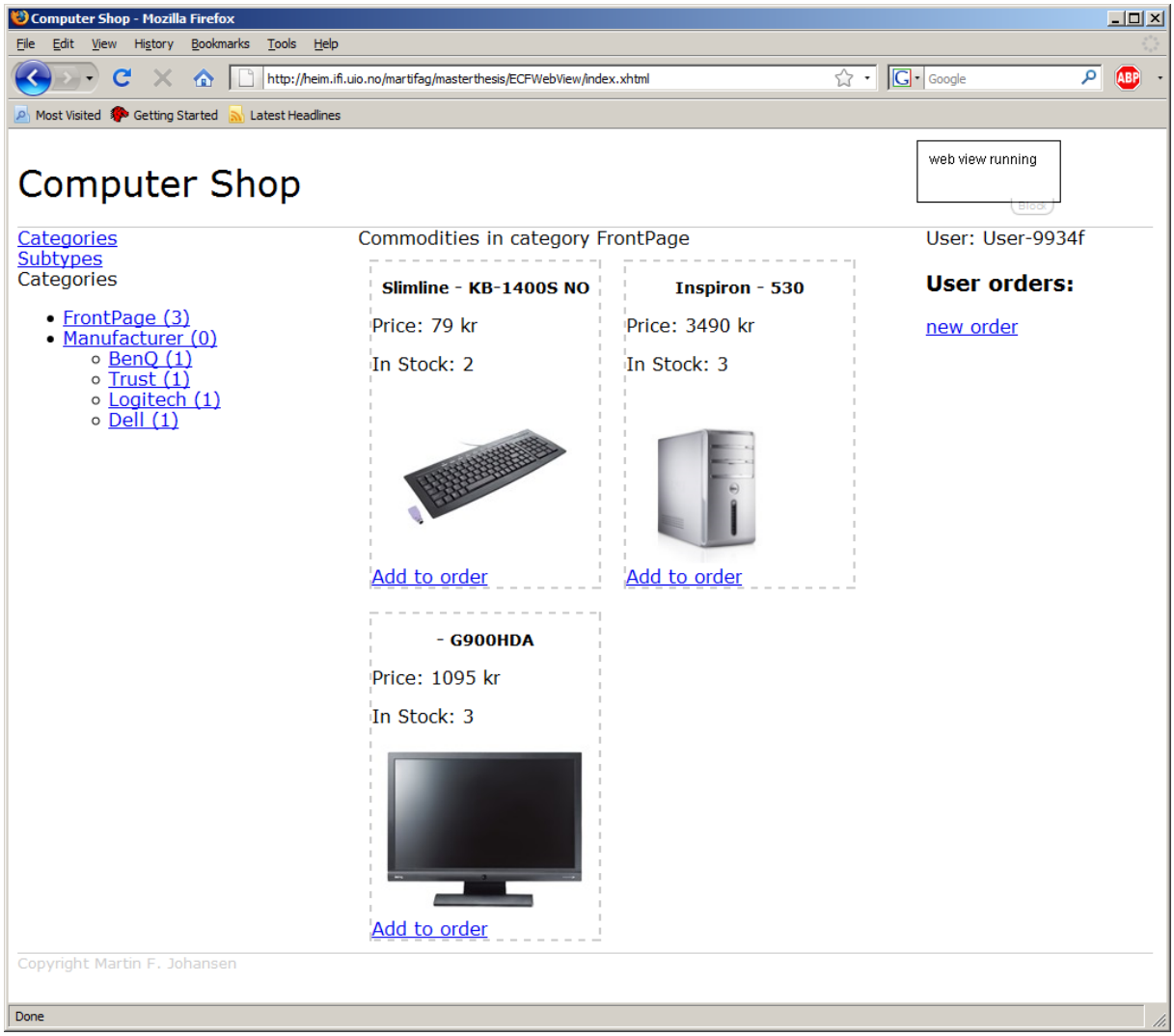

Figure A.1: Front page of the webshop

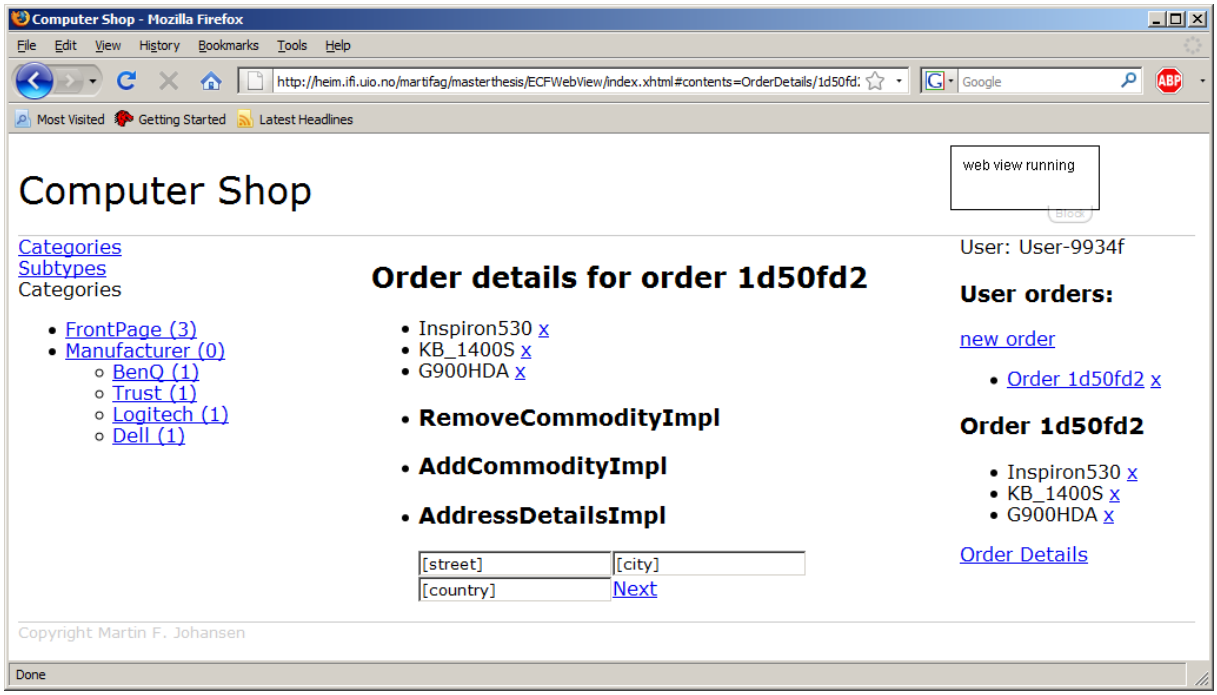

Figure A.2: Viewing an order in the webview

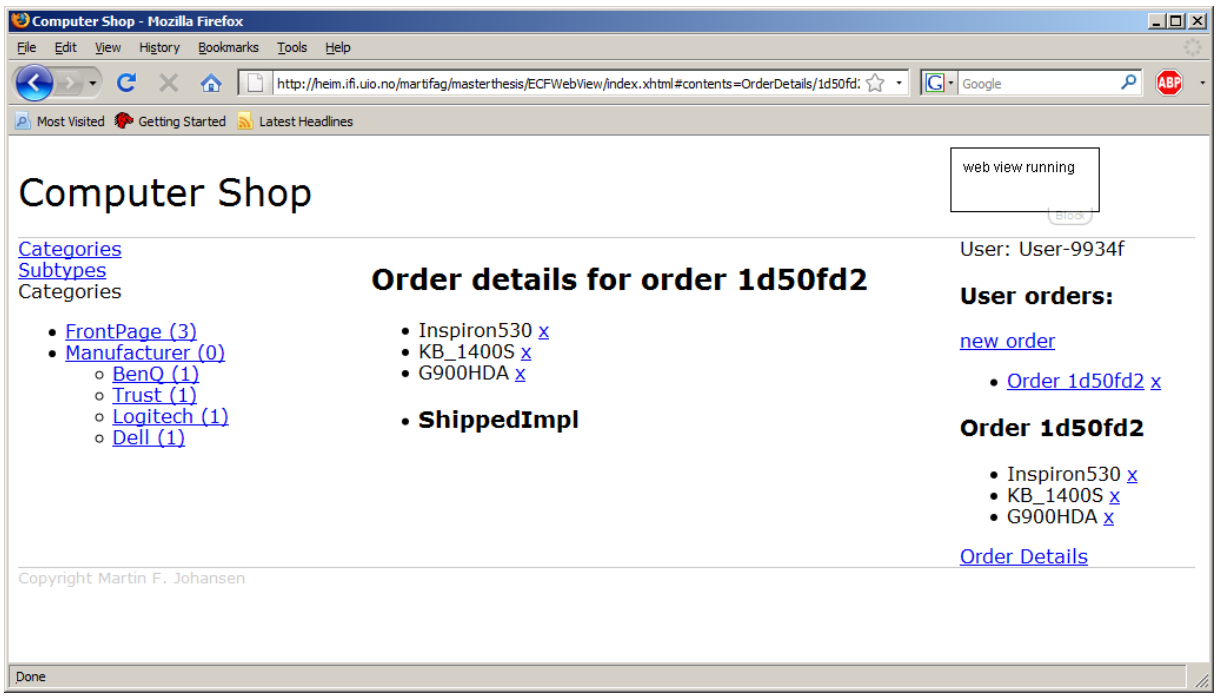

Figure A.3: Having completed an order in the webview

Appendix B

Requirements used to create the specification

The following nine webshops were used to produce the requirements listed in the sections of this chapter. The sections that follow contain the lists of requirements for the e-commerce system. This is the list that was used during the development stage. It is quite crude and is included here for reference only.

- Moobi webshop (used for dangaard.no, shop.netcom.no and shop.tele2.no)
- Magento
- osCommerce
- Zen Cart
- VirtueMart
- Komplett.no (source code not studied)
- cdon.com (source code not studied)
- iTunes (source code not studied)
- Interchange (source code not studied)

B.1 Selections

- One should be able to choose a selection of products to see.
- One should be able to have nested selections.

B.2 Verification

- All input into the system must be verified and the user should be informed about the error.

B.3 Security

- Security must be built in from the start.

B.4 Basket

- Gathering products, adjusting quantities and performing a checkout.
- Support one-click shopping.
- Orders must have protection against invalid values.
- Orders must have specified maximums and protection against invalid values.
- It should be possible to enter orders (as text for example) not using the WebUI for efficiency purposes.
- The products in the basket should be able to affect the checkout process.
- When products in the webshop system are changed, there may be users which have the product in their baskets, and there might be users who have ordered the product.
- All events should be explicitly defined.
- Support a scripted checkout. i.e. define an order by a comma-separated list of products.
- Support automatically added products when beneficial. For example, buy 3 for the price of 2-type features.
- Products should have variable timeouts or shared if so desired.
- An interesting product is one which is a subscription which expires or needs renewal. This is the ability so to say, to access certain material for a while, and then lose the ability.

B.5 User-system

- A permanent online account is a good thing, as the customers don't have to reenter all personal information all the time.
- There should be user-groups. The groups should define what the user gets access to.
- There is always a user. The first time someone enters the shop, a user is created.
- It must be possible to have an age-check on the products in the basket, then shop more non-check items and then purchase all without another check.
- If a product is checked, then the user should be able to remove the product from the basket, and then re-add it later, and it should still be checked for this user.

- It should be possible for an admin to act on behalf and as a user in the system, so that they can do something on their account which the user then sees later.

- It should be possible to deny a range of IPs/hostnames and single IPs access to the site.

- It should be possible to configure the access to the site of different users.

B.6 Payment systems

- There are many forms of payment: Credit card being the most common. But the shop system should support many alternatives, also external services such as Google checkout and PayPal.

- A user should be able to pay parts of the price with different payment methods.

- It should be possible to get refunds into the shop as credit.

B.7 Delivery system

- Product delivery is important. Different products have different delivery schemes. Music and movies can be downloaded or streamed. Physical objects must be shipped. Products might also be shipped from the closest location. The customer should be able to view this clearly. Products might also be subscriptions.

B.8 Order system

- Reports on different aspects of the system.

- Support reserved products.

- Support historic products, in old orders and in the existing orders. They must continue to work even when they are changed by the administrators.

- Support shipping parts of an order, to different places.

- Must support buying a product, but choosing a vendor or reseller from which to get the concrete product with a different shipping scheme.

- It must be possible to offer free products.

- Support physical and non-physical products. Also non-physical products which need something sent physically e.g. a password or key.

B.9 Product catalog

- A webshop is really an online catalogue of products. The shopping cart software might be added in order to make the catalogue a webshop.
- "Enterprise resource planning" is the term used for a product which handles the resources of a company. It might for instance handle is stock of products.
- The webshop system will probably need a local representation and copy of the external product, order, etc - systems. They will need to be synchronized with each other. What about observers?
- A bread-cumber bar is considered good.
- "Customers who bought this also bought" - feature
- "Related things"-feature.
- Must be able to select products and buy the selected products.
- Support exploding and collapsing trees and tree roots.
- There is a difference between saying "the box contains these products" and "we have a product which consists of these products together".
- The product-catalogue should support products with choices the user can specify about what it contains.
- It should be possible to get data about products from another place, e.g. a webservice.
- Support several groupings such as Products->Screens->TVs->Flat TVs->Samsung Flat TV->Model X10->A concrete Model X10
- Manufacturers must be represented in the system somehow.
- It should be possible to sell one type of commodity at different prices for each concrete or a group of concretes.
- It should be possible to work on additional products which are not yet on the shelf.
- Products should support variable parts, where the user can choose one of a set of alternatives.
- The user should be able to choose any valid member of a generalization or group. By resolving with many products or generalizations, the user may choose among those alternatives, just a few products or between many generalizations.
- Support bonus products if one buys according to some criteria.
- Support "offer of the week"-type of offers.
- It should be possible to comment or review of a product.

- Certain products may be combined with other products, in that the socket fits, for example.
- The system should support selling non-physical things such as subscriptions.
- Create-Read-Update-Delete must be supported for every aspect of the system.

B.10 Product catalog view

- It must be possible to change the order and structure of the many and the products in the view.
- It must be possible to hide things, and specify that they are visible for some time period.

Appendix C

Static constraints

C.1 Catalog-Composition Framework

The following static constraints were implemented for CCF.

C.1.1 Category 1: Only needed for the framework

- Statechart not a subclass of State
- Xor-state must have constructor
- Xor-state must have field substates
- Xor-state must have field transitions
- substates must be final
- substates must be static
- substates must be of type Object[][]
- transitions must be final
- transitions must be static
- transitions must be of type Object[][] not
- First entry in state specification is not String
- First entry in state specification is not Class
- Composite states must have at least two states (and-states 2, xor-states 2 including initial)
- From state must be a substate
- And-states can only have composite substates
- A final state cannot have outgoing transitions (implicitly defined final state in DSL)
- To state cannot be initial state
- State used twice or more in one composite state in state

C.1.2 Category 2: Only needed for the DSL

- That types are defined, reuses the host language mechanisms for doing the same.

C.1.3 Category 3: Needed for both the DSL and the framework

- The transition source and target must be existing substates within the same state
- Only one transition per signal per state
- No cycle in state hierarchy
- An epsilon transition can only be from either an initial state, a history state or from a composite state with a final state.
- Abilities has corresponding transitions in commodity behavior

C.2 Entity-Behavior Framework

The following static constraints were implemented for EBF.

C.2.1 Category 1: Only needed for the framework

- Not a subclass of Catalogue
- A bundle must be final
- Bundles cannot have interfaces
- Checks for cycles in the category memberships.
- Bundle must have at least one part
- The types specified in the in category- and subtype-relations must be of the correct type

C.2.2 Category 2: Only needed for the DSL

- Cycles in the subtype hierarchy is not allowed,
- A type used must be defined somewhere.

C.2.3 Category 3: Needed for both the DSL and the framework

- Choice-set must have only unique choices
- Is checking that the overriding parts of a subtype have types which are corresponding subtypes of the parts they override.
- A sub type must be as inter-connectible with its interfaces as its super type,
- All parts of a general commodity must be connectible.

Appendix D

Additional code examples

D.1 EBL: User behavior example

The example is shown in figure 3.7. It consists of three statecharts, one for an e-commerce customer and one for each of the two commodities. The statechart called Customer defined in listing D.1 has one sub-state InShop, meaning that the customer is in the shop. XorState is a keyword used to define a statechart, called Customer in this case. After this a substate called InShop is defined. The first substate in an xor-state is also its initial state. The Customer statechart contains no transitions.

Listing D.1: EBL example of a statechart definition

```
XorState Customer
  inShop : InShop
```

The next example is more complex. Listing D.2 shows the definition of InShop. AndState is the keyword used to define an and-state, in this case one called InShop. This state is concurrently in both sub-states. Two xor-states are defined as sub-states of the and-state. The first xor-state contains one sub-state called NormalAccess. This is meant to be the access a normal user has to the system. The asterisk symbol specifies a signal which the customer might pass on into the system, to for example a commodity. This is a domain concept called 'the ability' of the user. Concurrently to entering the NormalAccess state, the customer also enters the AnonymousBuyer state. In this state the customer might perform one transition. A transition is defined with a -> symbol, which is an arrow, meaning that a transition from Anonymous buyer into Known Buyer might be performed by sending a Register signal. Similarly, the user might unregister to go the other way. Several transitions and abilities may be listed after a sub-state declaration. When a user registers, he or she enters the Known Buyer -state. It gives the customer more abilities as described by its definition.

Listing D.2: EBL version of the InShop sub-state

```
AndState InShop
  a : A
  b : B

XorState A
  normalAccess : NormalAccess
  * PurchaseNormalCommodity

XorState B
  anonymousBuyer : AnonymousBuyer
  -> knownBuyer / Register
  knownBuyer : KnownBuyer
  -> anonymousBuyer / Unregister
```

The code in listing D.3 is similar to the statechart in listing D.2. The customer may try to authorize to gain the abilities of buying commodities which requires him to be of age of majority and commodities which require him to be a member of the organization X.

Listing D.3: EBL code example of the KnownBuyer statechart

```
AndState KnownBuyer
  c : C
  d : D

XorState C
  notAgeOfMajority : NotAgeOfMajority
  -> ageOfMajority / AgeVerification
  ageOfMajority : AgeOfMajority
  * PurchaseAgeOfMajorityRequiredGood

XorState D
  notAMemberOfOrganizationX : NotAMemberOfOrganizationX
  -> memberOfOrganizationX / ProofOfMembership
  memberOfOrganizationX : MemberOfOrganizationX
  -> notAMemberOfOrganizationX / MembershipExpired
  * PurchaseMembershipInXRequiredGood
```

The commodity behaviors are also defined by statecharts. In listing D.4, the abilities of the customer have corresponding transitions to the customer behavior in listing D.2 and D.3.

Listing D.4: EBL code examples of commodity behaviors

```
XorState Good
  inStock : InStock
  -> sold / PurchaseNormalCommodity
  sold : Sold

XorState RestrictedGood
  inStock : InStock
  -> sold / PurchaseAgeOfMajorityRequiredGood
  -> sold / PurchaseMembershipInXRequiredGood
  sold : Sold
```

Since the DSL only specifies the behavior in terms of states and transitions between these, some general purpose language must be used to express the required actions of the

transitions. In this case Java is chosen. Signals have corresponding classes with the same names as the signals. Transitions have corresponding Command objects[1] with a name consisting of the from-state and the trigger-name, since this is unique for every transition.

D.2 EBF: An abstract example

In figure 2.2 is a statechart X containing one composite state B. Under it we see a new statechart Z which inherits from X. Z redefines the state B by adding a history state to it. If a signal SigA is received by the statechart X when B is in D, cause B to reset. The history state makes sure the state is remembered when leaving B. This is implemented using the statechart framework as seen in listing D.5. Subclass the class XorStateImpl in order to make a new xor-state. It is set up in the constructor of X. First add two states, A and B. A is a simple state, while B is a custom state. Set the initial state to be A, and give the transition subclass to be executed on entering the xor-state. For the transitions, specify from- and to-state, the transition action and the signal class. For statecharts which inherits the behavior, transitions can be overridden. A new to-state and action can be specified, the from state and signal must remain the same.

Listing D.5: Example implementation of a statechart using the Entity-Behavior Framework

```
public class X extends XorStateImpl{
  public X(){
    addSubstate("A", new SimpleStateImpl());
    addSubstate("B", new B());
    setInitialState("A", new ActionToA());
    addTransition("A", "A", new ActionAtoA(), SigA.class);
    addTransition("A", "B", new ActionAtoB(), SigB.class);
    addTransition("B", "A", new ActionBtoA(), SigA.class);
  }
}
```

Listing D.6 shows a transition class. This way, the action is encapsulated. This is an implementation of the command-pattern from [GHJV95].

Listing D.6: Example transition command class

```
class ActionAtoA extends TransitionImpl{
  public void execute(){
    // Action code here
  }
}
```

The state B is set up in the same manner as state X. The code is seen in listing D.7.

Listing D.7: Example implementation of an xor-state

```
class B extends XorStateImpl{
  public B(){
    addSubstate("C", new SimpleStateImpl());
    addSubstate("D", new SimpleStateImpl());
    setInitialState("C", new ActionToC());
    addTransition("C", "D", new ActionCtoD(), SigD.class);
  }
}
```

[1] An application of the command pattern of [GHJV95]

In order to inherit the behavior of X the code in listing D.8 simply extends the statechart class X. super must be called in the constructor of Z to properly set up the superclass X. The state B is overridden by using the same name, but a different class when using addSubstate.

Listing D.8: Example use of inheritance for a statechart

```
public class Z extends X{
  public Z(){
    super();
    addSubstate("B", new NewB());
  }
}
```

Listing D.9 redefines B by adding a history state. The new state is added and the initial state is overridden. An epsilon signal is the trigger going out of the history state, since it happens automatically. EpsilonSignal is a framework class used to specify epsilon signals that is a signal which automatically is triggered.

Listing D.9: Another example of state inheritance. Now of state B

```
class NewB extends B{
  public NewB(){
    super();
    addSubstate("ShallowHistoryState", new ShallowHistoryStateImpl());
    addTransition("ShallowHistoryState", "C", new ActionHtoC(),
        EpsilonSignal.class);
    setInitialState("ShallowHistoryState", new ActiontoH());
  }
}
```

D.3 Modified EBF: An abstract example

An implementation of figure 2.2 is described for the modified Entity-Behavior Framework.

X, defined in listing D.10, contains three sub-states and four transitions. Instead of adding states to the statecharts, they are now listed in an array. The attribute substates is required by the XorStateImpl superstate and contains pairs. The first entry is the state name; the second entry is the state class object. InitialStateImpl is a framework class. The states A and B are defined later.

Transitions are defined in the same way as sub-states. The four-tuple contains transition name, from-state, trigger signal and to-state. In order to specify an action, create a method with the same name as the transition. This method is called when the transition is performed. Since a transition is only dependent on the from-state and the trigger signal, these are the transition parameters.

Listing D.10: Example implementation of a statechart using the modified Entity-Behavior Framework

```
public class X extends XorStateImpl{
  static final Object [][] substates = {
    {"Initial", InitialStateImpl.class},
    {"A", A.class},
    {"B", B.class}
  };

  static final Object [][] transitions = {
    {"initial", "Initial", EpsilonSignalImpl.class, "A"},
    {"aToA", "A", SigA.class, "A"},
    {"aToB", "A", SigB.class, "B"},
    {"bToA", "B", SigA.class, "A"},
  };

  void initial(InitialState from, EpsilonSignal s){}
  void aToA(A from, SigA s){}
  void aToB(A from, SigB s){}
  void bToA(B from, SigA s){}
}
```

Listing D.11 shows the definition of state A, defined to be a simple state. State B is a composite state and is defined in a similar way as state X.

Listing D.11: Implementation of states A and B

```
class A extends StateImpl{}
class B extends XorStateImpl{
  static final Object [][] substates = {
    {"Initial", InitialState.class},
    {"C", C.class},
    {"D", D.class}
  };

  static final Object [][] transitions = {
    {"initial", "Initial", EpsilonSignalImpl.class, "C"},
    {"cToD", "C", SigD.class, "D"},
  };

  void initial(InitialState from, EpsilonSignal s){}
  void cToD(C from, SigD s){}
}
class C extends StateImpl{}
class D extends StateImpl{}
```

In listing D.12 is the definition of state Z shown. State Z inherits the functionality of X. This is also done in this version using the sub-classing mechanisms of Java. In order to redefine B, specify the same name in the first entry to the sub-states attribute and give another class.

Listing D.12: Definition of state Z for the modified framework

```
public class Z extends X{
  static final Object [][] substates = {
    {"B", NewB.class}
  };
}
```

Listing D.13 shows the implementation of the state NewB. In it the transition called initial is overridden and given a new target state. Java allows methods to override each other given the same signature. Hence, the transition action is overridden by using the same transition name and signatures on the action method. In the transition attribute, only the to-state might be changed.

Listing D.13: Definition of NewB for the modified framework

```
class NewB extends B{
  static final Object [][] substates = {
    {"ShallowHistoryState", ShallowHistoryStateImpl.class}
  };

  static final Object [][] transitions = {
    {"initial", "Initial", EpsilonSignalImpl.class, "ShallowHistoryState"},
    {"hToC", "ShallowHistoryState", EpsilonSignalImpl.class, "C"},
  };

  void initial(InitialState from, EpsilonSignal s){}
  void hToC(ShallowHistoryStateImpl from, EpsilonSignal s){}
}
```