DEVELOPMENT OF A NEW DOMAIN-SPECIFIC LANGUAGE FOR SOFTWARE ARCHITECTURE SPECIFICATION: DSL-SA

by

Sezer Akar

B.S., Computer Engineering, Ege University, 2007

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2012

ACKNOWLEDGEMENTS

Firstly, I thank my thesis supervisor Prof. M. Ufuk Çağlayan for continuous support and understanding during development of this work. His guidance and patience helped me to understand problems and encouraged me to work on them.

I would like to thank my family for their encouragement and belief in me. They helped me to stay on track.

Finally, I give my special thanks to my lovely friend, Berke Atabey. She always motivated me until I reach the end of this work. This work could not be finished without her endless support.

ABSTRACT

DEVELOPMENT OF A NEW DOMAIN-SPECIFIC LANGUAGE FOR SOFTWARE ARCHITECTURE SPECIFICATION: DSL-SA

The specification of software structures that conform to specific software architectures is the first step in software design during software development. Software structure may be the first design artifact that becomes inconsistent as software evolves. We believe this issue has two logical explanations. Firstly, specifying software structure that conforms to specific software architecture is not easy task. It requires considerable effort and this effort can be useless in future during software development. Secondly, once software structure that conforms to specific software architecture is defined, subsequent software components and modules may not conform to the software architecture that was selected. In this thesis work, a new specification language, called Domain-Specific Language for Software Architecture Specification (DSL-SA), is proposed. It is an attempt to specify software structure that conforms to a specific software architecture easily and to overcome problems mentioned above. In association with DSL-SA, a software tool, called DSL-SA Editor, has been developed. DSL-SA Editor will be used to specify software components that correspond to specific software architecture styles. By using DSL-SA Editor, one can specify software structures that conform to pipes-and-filters architectural style, layered architectural style and free architectural style. DSL-SA Editor supports the validation of software structures, to see whether the software structure conforms to the selected architectural style. DSL-SA Editor also supports high-level source code generation from software structure specification. We present three case studies to show how DSL-SA and DSL-SA Editor can be used to specify software components that correspond to a specific software architecture.

ÖZET

YAZILIM MİMARİSİ BELİRTİMİ İÇİN YENİ BİR ALANA ÖZGÜ DİL: DSL-SA

Belirli bir yazılım mimarisine uyan yazılım yapılarının belirtimi yazılım tasarımında atılan ilk adımdır. Belirli bir yazılım mimarisine uyan yazılım yapıları yazılım geliştikçe tutarsızlaşan ilk olgu olabilir. Bizce bu durumun iki mantıklı açıklaması vardır. İlk olarak, belirli bir yazılım mimarisine uyan yazılım yapılarının belirtimi basit bir iş değildir. Hatırı sayılır miktarda çaba gerektirir ve bu çaba yazılım geliştirme sürecinin ileriki aşamalarında önemsizleşebilir. İkinci olarak, belirli bir yazılım mimarisine uyan yazılım yapıları belirtildikten sonra tanımlanan yazılım bileşenleri ve birimleri, en başta seçilen yazılım mimarisine uymayabilir. Bu çalısmamızda, yeni bir alana özgü dil olan DSL-SA önerilmektedir. DSL-SA belirli bir yazılım mimarisine uyan yazılım yapılarının daha kolay belirtmek ve yukarıda bahsedilen sorunları çözmek için yapılan bir girişimdir. DSL-SA ile birlikte, DSL-SA Editor olarak adlandırlan bir yazılım aracı geliştirilmiştir. DSL-SA Editor belirli bir yazılım mimarisi biçemine karşılık gelen yazılım yapılarının belirtilmesinde kullanılabilir. DSL-SA Editor'u kullanılarak, "pipes-and-filters" yazılım mimarisi biçemi, katmanlı yazılım mimarisi biçemi ve özgür yazılım mimarisi biçemine uyan yazılım yapıları belirtilebilir. DSL-SA Editor yazılım yapılarının belirli bir yazılım mimarisi biçemine uyup uymadığını denetleyebilir. DSL-SA Editor aynı zamanda yazılım yapısı belirtiminden üst düzey kaynak kodları üretebilir. Son olarak, bu çalışmada, DSL ve DSL-SA Editor'un belirli bir yazılım mimarisine uyan yazılım yapılarını belirtmede nasıl kullanılabileceğini göstermek amacıyla üç tane durum çalışması gösterilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

AC-MDSD	Architecture-Centric Model Driven Software Development
ADL	Architecture Description Language
AST	Abstract Syntax Tree
CST	Concrete Syntax Tree
DSL	Domain-Specific Language
DSL-SA	Domain-Specific Language For Software Architecture
DSSA	Domain-Specific Software Architecture
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
GPL	General-Purpose Programming Language
GMF	Graphical Modeling Framework
GMP	Graphical Modeling Project
M2T	Model-to-Text Transformation
M2M	Model-to-Model Transformation
MDSD	Model Driven Software Development
MOF	Modeling Object Facility
EMOF	Essential Modeling Object Facility
Ecore	EMF Core
OMG	Object Management Group
MDA	Model Driven Architecture
PIM	Platform-Independent Model
PSM	Platform-Specific Model
SA	Software Architecture
UML	Unified Modeling Language
XML	Extensible Markup Language

1. INTRODUCTION

Software systems are almost everywhere in our daily life. We continuously build larger and more complex software systems, which need more effort to build them. Large and complex software systems require corrects software architectures to be built upon.

The specification of software structures that conform to specific software architectures is the first step in software design during software development. Software structure may be the first design artifact that becomes inconsistent as software evolves. We believe this issue has two logical explanations. Firstly, specifying software structure that conforms to specific software architecture is not easy task. It requires considerable effort and this effort can be useless in future during software development. Secondly, once software structure that conforms to specific software architecture is defined, subsequent software components and modules may not conform to the software architecture that was selected. Usually, software architecture is used as a reference and the software structure and subsequent source code are produced. Keeping source code and software architecture aligned becomes harder as software development progresses. It is hard to maintain software structure specification documents and the high level source code of the software structure to reflect the latest software modifications. We believe that domain-specific languages and model driven software development paradigm can play an important role to overcome these problems. Models can help to select a software architecture and define the corresponding software structure more easily and in a formal manner. The target software structure, high-level source code, configuration files etc. can be generated from this software structure model.

To specify software models, one needs to have a specification language first. UML is the first choice that comes to mind, but since UML is semi-formal, other specification methods are needed. Various architecture description languages such as AADL [5], xADL [6] are proposed to define software architectures. All of these approaches advocate using existing, generic languages for specifying software architectures [7] and require some training on software modeling and software tools. During software life-

cycle, software structure may become out-of-date and refactoring software structure based on a new software architecture needs considerable effort. Another problem is rapidly changing technology. Even a well defined software structure and/or selected software architecture may not live long and may need to be changed to reflect newer software technologies and/or software requirements. For example, adding more scalability and/or improving performance, say in terms of software system response time, may imply the selection of a new software architecture, therefore modification of the software structure that will conform to the new software architecture will be needed. Such modifications in software structure due to new software architecture can be painful especially for large and complex systems.

In this thesis work, a new specification language, called Domain-Specific Language for Software Architecture Specification (DSL-SA), is proposed. It is an attempt to specify software structure that conforms to a specific software architecture easily and to overcome problems mentioned above. Software structure that conforms to a specific software architecture specified by DSL-SA is not only a plain document, but also a work product which helps software architects, developers and all other stakeholders during software development.

In association with DSL-SA, a software tool, called DSL-SA Editor, has been developed. DSL-SA Editor will be used to specify software components that correspond to specific software architecture styles. Currently, DSL-SA Editor supports three architectural styles. By using DSL-SA Editor, one can specify software structures that conform to pipes-and-filters architectural style, layered architectural style and free architectural style. DSL-SA Editor supports the validation of software structures, to see whether the software structure conforms to the selected architectural style. DSL-SA Editor also supports high-level source code generation from software structure specification.

We present three case studies, namely, Software Architecture Specification of a Secure File Transfer System by DSL-SA, Software Architecture Specification of a Lowest Price Book Finder System by DSL-SA, and Software Architecture Specification

of an Airport Management System by DSL-SA, to show how DSL-SA and DSL-SA Editor can be used to specify software components that correspond to a specific software architecture.

The remaining part of the thesis is organized as follows:

In Chapter 2, an overview of software architectures and model driven software development are presented. Model-driven software development standards and practices are introduced. Development patterns of domain-specific languages are examined. Architecture centric model driven software development is described.

In Chapter 3, we describe our proposed language, DSL-SA. Overall structure of DSL-SA and DSL-SA development steps are explained in detail. Further, DSL-SA is categorized with respect to domain-specific language categorization introduced in previous chapter.

In Chapter 4, we introduce an editor, called DSL-SA Editor, for DSL-SA. Functions provided by DSL-SA Editor are explained. We demonstrate how DSL-SA Editor enforces software structure conformance to specific software architecture and how DSL-SA Editor generates high-level source code from software structure that is defined by it.

In Chapter 5, three case studies are presented to show how DSL-SA and DSL-SA Editor functions could be used to specify software structure that corresponds to a specific software architecture.

Chapter 6 concludes this thesis work by summarizing our work and listing potential future works.

2. BACKGROUND

2.1. Software Architecture

There is not any agreement on software architecture definition. In this work, when we say "software architecture" we try to differentiate it from detailed design. An example for formal specifications of detailed design can be found in [8]. Differentiating software architecture from detailed design is a hard task to achieve but there are some guidelines to help. For example, software architecture's primary concern is non-functional requirements, but detailed design's primary concern is functional requirements. A method, Intension/Locality Hypothesis [9], is proposed to decide what belongs to software architecture. According to Locality Criterion defined in this hypothesis, a statement belongs to architecture if and only if a program that satisfies it can be expanded to another program which does not. For example, a program which in client-server style can be expanded to another program which is not in client-server style by adding more peers. Hence client-server style is architectural. Locality criterion can also be interpreted in this way: If a rule can be broken in any component of the software, it is non-local, hence belongs to software architecture not software design. Pseudo code belongs to detailed design. UML class and state diagrams can appear in software architecture documents but UML sequence diagrams must appear in detailed design.

Problems of current state of software architecture discipline are [10]:

- Technology-driven: Software architectures are too much technology-driven. They are highly coupled to one specific technology platform implementation. These kinds of practices see software architecture in only underlying platform aspect. When somebody says that their system is using web-service architecture, he defines only one aspect of software architecture (communication) and binds software architecture to a specific implementation (web-service).
- Hype-factor: Nobody agrees on software architecture definition. For example,

when Service Oriented Architecture (SOA) is mentioned, it is often understood as web-services. There is not any clear definition for terms.

- Industry standards: Previously, a working solution becomes an industry standard. In current state of software architecture practice, every vendor defines standards from scratch without any prior experience, so standards become complicated.

Software architecture can be seen in two different stages, prescriptive and descriptive [1]. Prescriptive software architecture defines principles of designs prior to system built. It can be seen as architecture-as-intended. Descriptive software architecture defines what is actually implemented in system; hence it is architecture-as-realized. These do not match in most of real cases. This situation is called architectural degradation. Architectural degradation can happen in two ways: architectural drift or architectural erosion. If there are new principles introduced in descriptive architecture, and these rules do not conflict with prescriptive architecture, this is called architectural drift. Architectural erosion happens when developers introduce new rules to descriptive architecture which are conflicting with prescriptive architecture. As software architecture degrades, software architecture recovery takes place. Software architecture recovery is an attempt to extract software architecture from implementation artifacts, source code, .class files and so on. It is a hard and time-consuming job, so it is more logical to prevent architectural degradation instead of trying to recover software architecture after it degrades.

Software architecture definition ways can be categorized as Figure 2.1 [1]. Software architecture style is a specialization of element and relation types, together with a set of constraints on how they can be used [11]. Example software architecture styles are pipes and filters, blackboard, layered styles etc. Architectural pattern is a description of element and relation types together with a set of constraints on how they are used [11]. Example software architecture patterns are model-view-controller, three-tier layered etc. Finally, DSSA is software architecture for a particular domain and it is generalized for effective use in that domain. DSSA helps to build many similar applications in same domain.

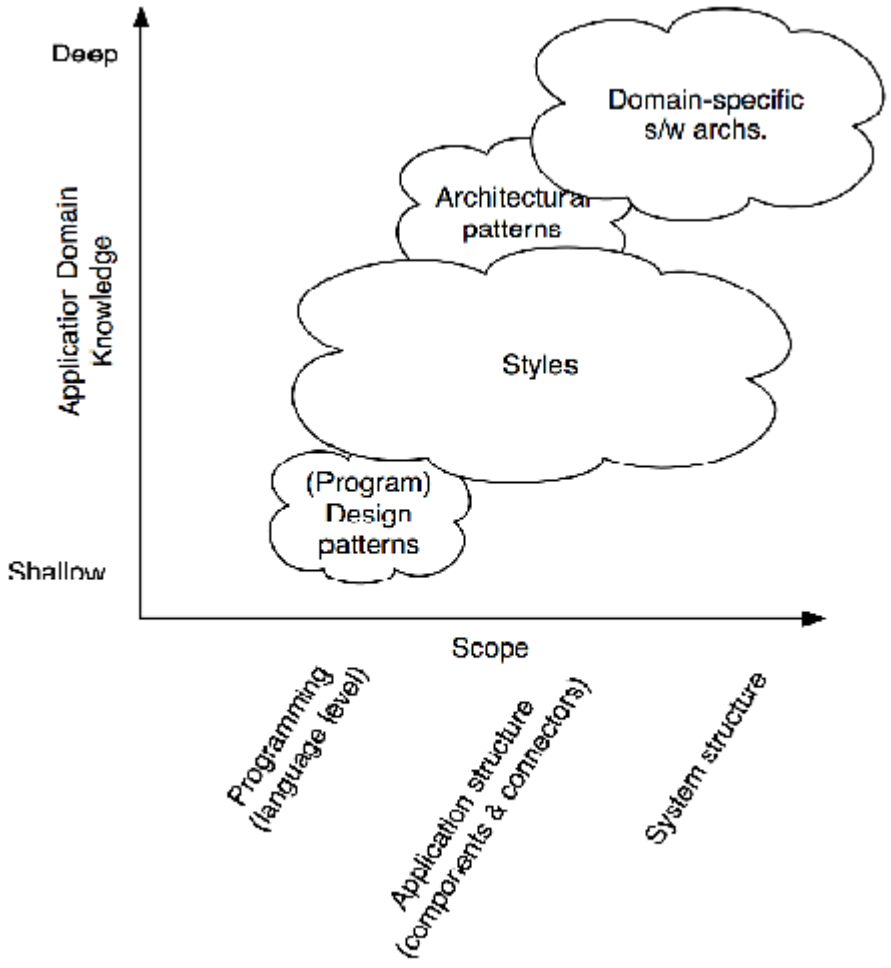

Figure 2.1. Architecture Styles, Patterns and Domain-Specific SA [1].

2.2. Model Driven Software Development

Model Driven Software Development is a software development methodology which primary focus is creating models and generating software from models as much as possible. Microsoft Software Factory and Eclipse Modeling Framework (EMF) are examples of model driven software development (MDSD) approaches.

Basic entity in MDSD is model. A model is an abstraction of a problem domain by specifying problem elements and their relationships in a formal way, in terms of modeling language. A modeling language has constructs to build models, and it is a form of higher level abstraction. Modeling languages are also called as meta-models. Models conform to their modeling language. A modeling language has abstract syntax to define models. This abstract syntax can be seen as a result of even a higher model, which is called as meta meta-model. Meta meta-models provide generic abstractions and abstract syntax to build meta-models. Meta-models conform to meta meta-models to define themselves.

Model transformation plays an important role in MDSD. A model itself without transforming to another artifact can be just a good looking document. Models are transformed to source code directly in most cases. This transformation called as model-to-text (M2T) transformation. Text can also be XML, HTML, JSP or any other textual artifact which is useful. Another type of transformation is called as model-to-model (M2M) transformation. Models can be transformed to other models. This is often needed to decrease semantic gap between modeling and target artifact. For example, one model can be very high level to generate source code from it directly. In this case, this model can be transformed to some intermediary models, which cause a series of transformation and to source code at the end. These intermediary models can be also meaningful for user and can be taken as reference for other implementation issues. Basic ideas on MDSD can be seen in Figure 2.2 [2].

Goals of MDSD can be summarized as follows [2]:

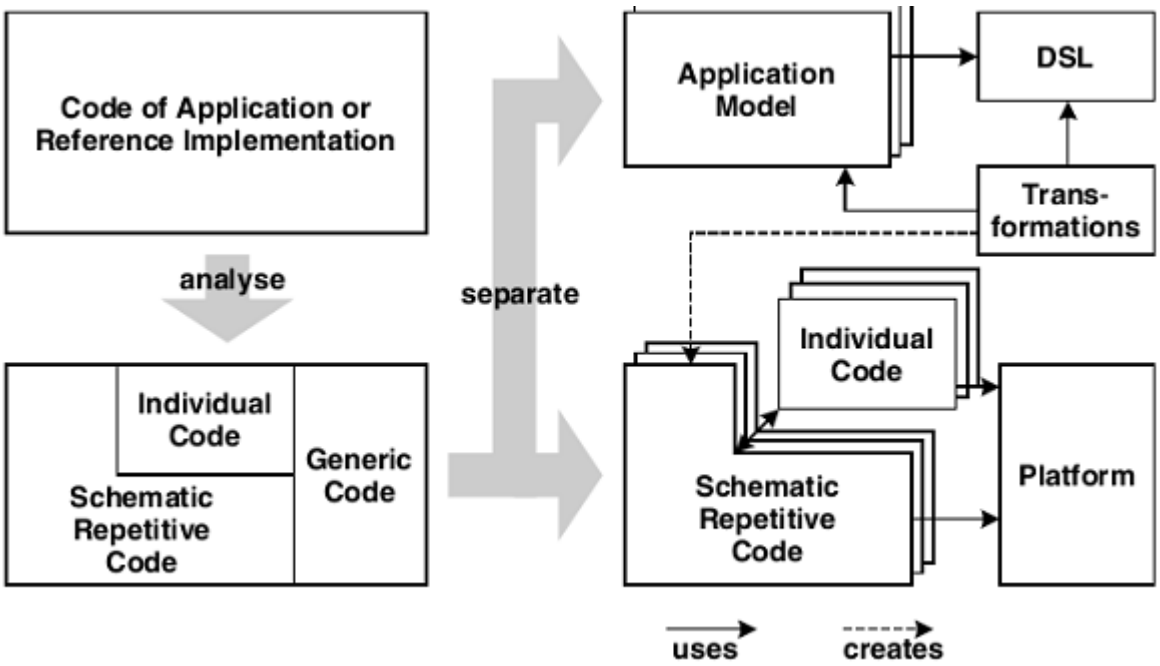

Figure 2.2. The basic ideas behind MDSD [2].

- Primary goal is to increase development speed through automation. Major part of source code can be generated from models.
- Formally defined modeling languages helps developers to increase software quality.
- Cross-cutting concerns can be developed one place i.e. transformation rules, which increase software maintainability.
- Modeling languages, models and created software architectures can be reused which may create a software product line.
- Modeling increases abstraction level from code level to model level. As a result, it helps to manage complexity.

MDSD helps on keeping software inner structure visible and consistent. There are also other approaches to do it, but they do not work well in practice, as MDSD does. One of them is code reviews. Reviewing all code with team to keep everything consistent is a very time-consuming job. Once project deadline reaches, these code review sessions are sacrificed first. Another is visualizing code through UML, and trying to keep UML and code consistent. Problem is unique mapping to code is impossible

because UML can not know application semantics.

MDSD is a forward engineering approach. Everything starts with modeling. After modeling activity takes place, model transformation, i.e. code generation can be done. It does not mean a waterfall approach have to be followed in MDSD. MDSD can fit in iterative and incremental software development methodology. Models can be developed in iterative and incremental manner. When we say MDSD is a forward engineering approach, it means reverse engineering does not play well with MDSD. Extracting models from source code does not end with good results mostly. Problem with extracted model is that, it has same abstraction level with source code. It is just a visualization of source code in UML like diagrams. Increasing abstraction level of extracted model involves manual mapping tasks and it is a time consuming effort.

We will look at Object Management Group (OMG) Model Driven Architecture (MDA) standard to understand MDSD domain better.

2.2.1. OMG MDA standard

OMG MDA standard can be seen in three layers, namely Platform Independent Model (PIM) layer, Platform Specific Model (PSM) layer and implementation layer. Technology independent, formal models are defined as PIMs. The idea of PIMs is that, concepts are less easy to be changed than technology. In most cases, PIMs are UML models. Difference of PIMs than other UML models is in their semantics. Not every UML model is formal but PIMs should be formal to allow transformation. Once PIM is defined, it should be mapped to PSM with M2M transformation. One example to PSMs is J2EE model. Usually, PSMs are defined with UML profiles. UML profiles are extension mechanism to standard UML. Once PSM is defined, M2T transformation takes place. Source code is generated for target environment based on PSM. Basic ideas of MDA can be seen in Figure 2.3 [2]:

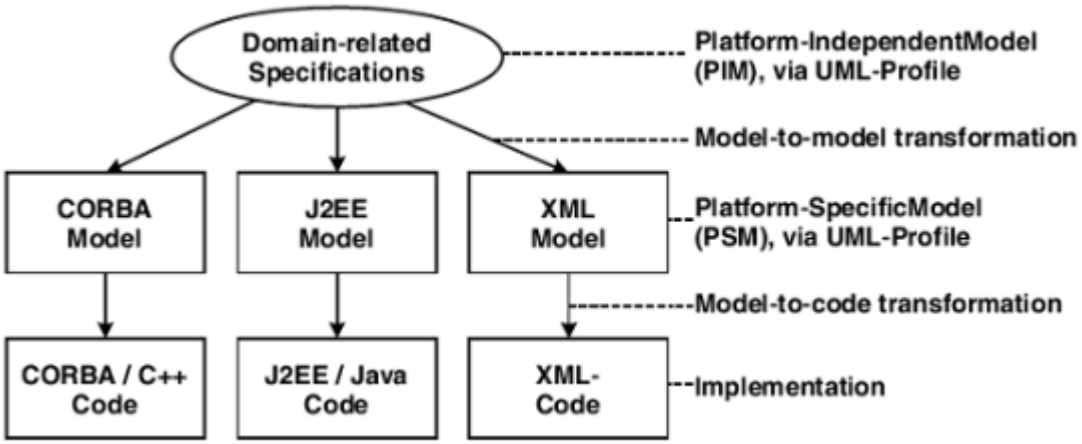

Figure 2.3. The basic ideas behind OMG MDA [2].

2.2.2. Eclipse Modeling Project

Eclipse Modeling Project (EMP) is a modeling and code generation project. Overview of EMP can be seen in Figure 2.4 [3].

EMP has many projects which are logically categorized as abstract syntax development, concrete syntax development and model-to-text or model-to-model transformation. It's meta-model called as EMF Core (Ecore) [12]. Ecore can be thought an effective JAVA implementation of a core subset of OMG Modeling Object Facility (MOF) API. It is very similar to Object Management Groups (OMG) Essential Modeling Object Framework (EMOF) meta-model. It provides abstract syntax tree development capabilities. EMF Query, Validation and Transformation capabilities complement EMF Core. Model-to-Text (JET [12] and XPAND [13]) and Model-to-Model (QVT and ATL) layer sub projects provides transformation from one form to another. In outer layer, concrete syntax development projects, GMF and TMF are used for graphical and textual representation of models.

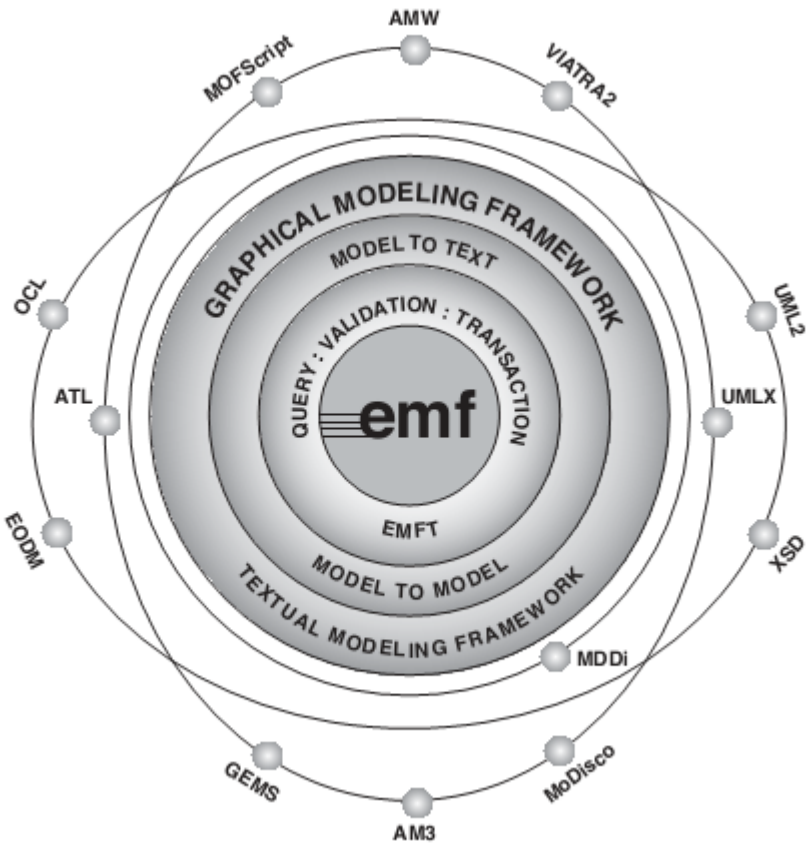

Figure 2.4. Overview of Eclipse Modeling Project [3].

2.2.3. Domain-Specific Languages

Although there is no agreement for definition of Domain-Specific Language (DSL), DSL can be considered as language that is constructed for a specific application domain. The concept is not something new, but become more popular due to domain-specific modeling and model driven design software methodologies. Example DSLs are VHDL for hardware design domain, Make for software building domain, and BNF for syntax specification domain. DSL chooses expressiveness in a specific domain over generality. This leads to higher level languages improving developer productivity and communication with business experts. DSL reduces the amount of domain and programming expertise needed, so they open up application domain to a larger group of software developers compared to General-Purpose Programming Language (GPL). There are also other reasons that people choose to develop DSL instead of using GPL [4]:

- Domain specific notations have more expression power than operator notation provided by GPL's.
- Domain specific abstractions and constructs may be too complex for mapping to functions or objects directly. So user may prefer to define them in a domain-specific language directly.
- DSL's offer possibilities for complex tasks, like verification, optimization and transformation. To achieve these tasks by using GPL's can be hard.
- DSL's need not be executable and they can be designed just for expression, analysis purpose. On the other hand, GPL's always have to be executable.

DSL's improve development productivity [14]. Use of DSL in software development, provide more clear way to show intent of system. It increases readability of code, so it is easier to find errors in code and to modify system via DSL. Also a DSL can be used as a façade to rest of system and user of DSL, here developers, might not be needed to understand whole system. It may be used to hide complicated, legacy systems from developers by letting them to develop system based on DSL.

Domain-specific languages help to improve communication between customers of a system [14]. It is a specialized language for a particular domain, and domain experts may easily read code written in DSL. Instead of communicating with customer in an ambiguous way by providing hundreds of requirement documents, a system expressed in a DSL might be more productive. Domain experts can read, even write to some extent, code with DSL, and communicate with software developers based on DSL.

Despite of its advantages, DSL has some problems too [14]. It takes considerable time and money to build a DSL and its surrounding environment. People will not be familiar this new language, so there will be a learning curve for project. Usually, a software project contains many DSLs together. Learning and using many languages might be harder than using one language.

According to [4], DSL development consists of five phases. Different patterns are mentioned in each phase.

(i) Decision: This is the phase which developer of DSL need to answer questions like "do we really need a DSL?", "what will it be used for?". The construction of DSL and tools is expensive and it should pay it cost. Example decisions to start a DSL development is automatize tasks, to start a software product-line, GUI construction etc. Examples for decision patterns can be seen in Table 2.1 [4].

(ii) Analysis: In analysis phase, domain knowledge is gathered by analyzing problem domain. Analysis patterns can be seen in Table 2.2 [4].
Analysis can be done in informal ways or by using formal methods. Also domain knowledge can be extracted from source code by source inspection or tools, a combination of both.

(iii) Design: Will DSL be used in a host language or will it be an independent i.e. a new language? The former named as Language Exploitation pattern, the latter named as Language Invention pattern. Language exploitation patterns are piggyback, specialization and extension. If existing language is partially used this pattern named as piggyback. In specialization pattern, existing language is restricted, so user of DSL can not use every feature in existing language. In

Table 2.1. Examples for Decision Patterns [4].

Pattern	DSL	Application Domain
Notation	MSC[SDL Forum 2000]	Telecom system specification
*Visual-to-textual	Hawk [Launchbury et al. 1999]	Microarchitecture design
	MSF [Gray and Karsai 2003]	Tool integration
	Verischemelog [Jennings and Beuscher 1999]	Hardware design
*API-to-DSL	SPL [Xiong et al. 2001]	Digital signal processing
	SWUL [Bravenboer and Visser 2004]	GUI construction
AVOPT	AL [Guyer and Lin 1999]	Software optimization
	ATMOL [van Engelen 2001]	Atmospheric modeling
	BDL [Bertrand and Augeraud 1999]	Coordination
	ESP [Kumar et al. 2001]	Programmable devices
	OWL-Light [Dean et al.2003]	Web ontology
	PCSL [Bruntink et al. 2005]	Parameter checking
	PLAN-P [Thibault et al.1998]	Network programming
	Teapot [Chandra et al. 1999]	Cache coherence protocols
Task automation	Facile [Schnarr et al. 2001]	Computer architecture
	JAMOOS [Gil and Tsoglin 2001]	Language processing
	lava [Sirer and Bershad 1999]	Software testing
	PSL-DA [Fertalj et al. 2002]	Database applications
	RoTL [Mauw et al. 2004]	Traffic control
	SHIFT [Antoniotti and Göllü 1997]	Hybrid system design
	SODL [Mernik et al. 2001]	Network applications
Product line	GAL [Thibault et al. 1999]	Video device drivers
Data structure representation	ACML [Gondow and Kawashima 2002]	CASE tools
	ASDL [Wang et al. 1997]	Language processing
	DiSTiL [Smaragdakis and Batory 1997]	Container data structures
	FIDO [Klarlund and Schwartzbach 1999]	Tree automata
Data structure traversal	ASTLOG [Crew 1997]	Language processing
	Hancock [Bonachea et al.1999]	Customer profiling
	S-XML [Clements et al.2004; Felleisen et al. 2004]	XML processing
	TVL [Gray and Karsai 2003]	Tool integration
System front-end	Nowra [Sloane 2002]	Software configuration
	SSC [Buffenbarger and Gruell 2001]	Software composition
Interaction	CHEM [Bentley 1986]	Drawing chemical structures
	FPIC [Kamin and Hyatt 1997]	Picture drawing
	Fran [Elliott 1999]	Computer animation
	Mawl [Atkins et al. 1999]	Web computing
	Service Combinators [Cardelli and Davies 1999]	Web computing
GUI construction	AUI [Schneider and Cordy 2002]	User interface construction
	HyCom [Risi et al. 2001]	Hypermedia applications

Table 2.2. Analysis Patterns [4].

Pattern	Description
Informal	The domain is analyzed in an informal way.
Formal	A domain analysis methodology is used.
Extract from code	Mining of domain knowledge from legacy GPL code by inspection or by using software tools, or a combination of both.

extension pattern, existing language is extended with new features that will be available to DSL users. Design patterns can be seen in Table 2.3 [4].

Table 2.3. Design Patterns [4].

Pattern	Description
Language exploitation	DSL uses (part of) existing GPL or DSL. Important subpatterns: •Piggyback: Existing language is partially used •Specialization: Existing language is restricted •Extension: Existing language is extended
Language invention	A DSL is designed from scratch with no commonality with existing languages
Informal	DSL is described informally
Formal	DSL is described formally using an existing semantics definition method such as attribute grammars, rewrite rules, or abstract state machines

(iv) Implementation: Implementation of DSL may involve interpreter or compiler/application generator patterns. Implementation patterns are shown in Table 2.4 [4].

(v) Deployment: After DSL developed, it will be deployed to target environment for usage.

Implementation of a DSL involves following steps [3] :

- Abstract Syntax Tree Definition: Metamodel of DSL is defined here. If we give

Table 2.4. Implementation Patterns [4].

Pattern	Description
Interpreter	DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. This approach is appropriate for languages having a dynamic character or if execution speed is not an issue. The advantages of interpretation over compilation are greater simplicity, greater control over the execution environment, and easier extension.
Compiler/application generator	DSL constructs are translated to base language constructs and library calls. A complete static analysis can be done on the DSL program/specification. DSL compilers are often called application generators.
Preprocessor	DSL constructs are translated to constructs in an existing language (thebase language). Static analysis is limited to that done by the base language processor. Important subpatterns: •Macro processing: Expansion of macro definitions. •Source-to-source transformation: DSL source code is transformed (translated) into base language source code. •Pipeline: Processors successively handling sublanguages of a DSL and translating them to the input language of the next stage. •Lexical processing: Only simple lexical scanning is required, without complicated tree-based syntax analysis.
Embedding	DSL constructs are embedded in an existing GPL (thehost language) by defining new abstract data types and operators. Application libraries are the basic form of embedding.
Extensible compiler/ interpreter	A GPL compiler/interpreter is extended with domain-specific optimization rules and/or domain-specific code generation. While interpreters are usually relatively easy to extend, extending compilers is hard unless they were designed with extension in mind.
Commercial Off-The-Shelf (COTS)	Existing tools and/or notations are applied to a specific domain.
Hybrid	A combination of the above approaches.

example from EMP, this is the step where we define DSL's Abstract Syntax Tree using Ecore metamodel.

- Concrete Syntax Definition: Users of the DSL should be able to create their instance of models using a concrete syntax. This can be achieved in two ways: Textual or Visual. If we give example from EMP, Xtext can be used to create textual DSLs and to define their concrete syntax, and GMF to create graphical DSLs.
- Model-to-Model (M2M) or Model-to-Text (M2T) Transformation: M2M tools transform one model to another. Examples from EMP are QVT and ATL. M2T tools transform models to source code, configuration files or any other textual representation. Examples from EMP are JET, XPAND.

2.2.4. Architecture Centric MDSD

One specific type of MDSD is Architecture Centric MDSD. The domain of AC-MDSD is software architectures. Its goal is to create software system families which allow you to create many architecturally similar products instead of building one unique product. This can really compensate cost of building DSLs. Main ideas of AC-MDSD can be seen in Figure 2.5 [2].

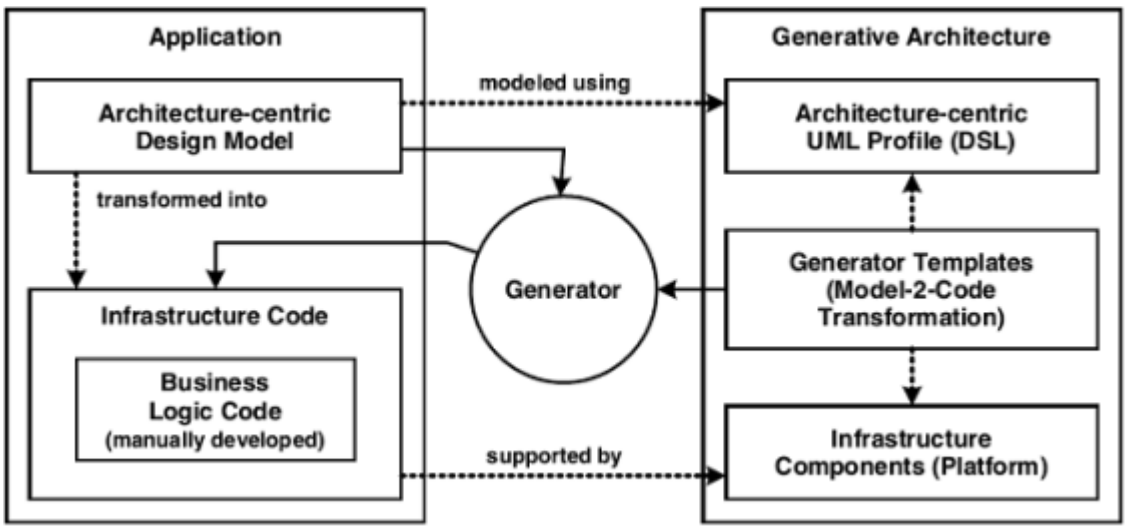

Figure 2.5. Architecture Centric MDSD [2].

In the current practice of software development, developers have to deal many complex tools to build a software system. Open source libraries, third party tools, database systems and so on. This infrastructure code may constitute 60%-70% of software. Goal of AC-MDSD is to generate architecture relevant infrastructure code automatically from a model as much as possible. Developers do not bother with infrastructure code and develop business domain specific code directly; as a result development efficiency increases.

AC-MDSD has also impacts on organization of development. Having a generative architecture and applications separates development team to two different development tracks. One team develops generative architecture and its creative tools, and other team creates unique products based on generative architecture. While software architects work on generative architecture, designers can work on unique product design and developers develop business logic code based on this design.

3. A NEW DOMAIN-SPECIFIC LANGUAGE FOR SOFTWARE ARCHITECTURE SPECIFICATION: DSL-SA

3.1. Motivation for DSL-SA

Almost all software systems have software architecture at design phase of software development. The software architecture that is selected may become out of date and/or the conformance of source code to selected software architecture may be broken as project evolves. Usually, software architecture is seen as only a plain document. Many alternatives are suggested to overcome such problems. Specifying software architecture with a domain-specific language (DSL) is one of them.

There are many approaches to specify software architecture. UML is the most common approach. UML can be seen a general specification language for modeling and it is not intended for only software architecture specification. Moreover, UML does not provide strong formalism needed for source code generation. In addition to UML, Architecture Description Language's (ADL's) are used for software architecture specification. ADL's have support for software architecture specification but they are not easy to extend and they are hard to learn. Most of ADL's may easily become out of date in a short time.

UML and ADL's require considerable modeling knowledge, so they have a major learning curve. They may contain many unnecessary elements to define software architectures in a specified domain. Furthermore, it is often not possible to find an ADL to fit a specific application domain and requirements. Instead of having an ADL to specify software architectures and trying to modify it, we choose to propose a DSL for software architecture domain. This DSL should be enough powerful to specify all architectural requirements for all products in same system family. In addition, DSL definition phase will help all stakeholders to understand related domain deeply. Our proposed work, a new domain-specific language for software architecture specification,

can be seen an example of these ideas.

This work is focused on the development of a new domain-specific language to specify software architecture. The goal is to increase traceability from requirements to source code and communication through stakeholders. A considerable amount of the source code should be directly derivable from software architecture that is specified by DSL-SA. Furthermore, software architecture should be validated for consistency, completeness with respect to the initial requirements. To achieve these goals, software architecture validation and meaningful code generation, proposed DSL should be formal. Specifying the overall structure of software architecture from the beginning is almost impossible for non-trivial domains. Therefore, DSL and models specified by it should be open to modification. Language should have rich editor support for practical purposes. Software architects and other stakeholders should be able to specify software structures that conform to selected software architecture by using DSL-SA and its language editor, DSL-SA Editor.

DSL-SA has been categorized by using the five phase guidelines given by [4]. According to [4], DSL construction contains five phases: Decision, Analysis, Design, Implementation and Deployment.

(i) Decision to develop DSL-SA: Software products have at least 60%-70% architecturally relevant infrastructure source code. Similar products in same company have similar architectural requirements, so infrastructure source code is often copied to other software products too. Instead of error prone copy-paste-modify process, software architecture can be defined formally and relevant source code can be generated automatically. Specifying software architecture by a formal DSL helps not only for one product but also for all architecturally similar products. However, DSL construction requires considerable effort, so investment in DSL construction should pay for itself. Using same DSL to produce many software products helps to achieve this goal. DSL-SA can be used to specify many software architectures in same system family and it can be used to start a software product line. Therefore, DSL-SA is an example of product line decision pattern

in Table 2.1.

(ii) Analysis of software architecture domain: Domain knowledge is source of input for DSL construction. Domain of DSL-SA, software architecture, is analyzed informally in our case. Therefore, DSL-SA is an example of informal analysis pattern in Table 2.2.

(iii) Design of DSL-SA: DSL-SA is a new language that is proposed for software architecture specification. It does not reuse any part of existing General-Purpose Programming Language or DSL. DSL-SA has some common ideas with UML, but it is designed from scratch. Therefore, DSL-SA is an example of language invention pattern in Table 2.3. Software architecture that is specified by DSL-SA is a formal model, so DSL-SA is an example of formal design pattern in Table 2.3.

(iv) Implementation of DSL-SA: Once a software architecture is defined by using DSL-SA, it is transformed to source code in base language, i.e. JAVA , by using XPAND templates. Software architecture specification is preprocessed by XPAND templates. Therefore, DSL-SA is an example of preprocessor implementation pattern in Table 2.4.

(v) Deployment of DSL-SA: Software architecture that is specified by DSL-SA can be used by software architects and developers to validate architecture properties and to generate high-level source code after it is deployed.

Summary of DSL-SA categorization by using the five phase guidelines given by [4] can be found in Table 3.1.

3.2. Overall Structure of DSL-SA

Software development phases can be seen in Figure 3.1. DSL-SA concerns with high-level design phase only. One can design software structures by DSL-SA after he selects an architectural style. High-level source code of software structures for a specific software architecture style can be generated by using DSL-SA Editor. Final source code will be obtained when the rest of source code corresponding to detailed

Table 3.1. Summary of DSL-SA categorization.

Phase	Pattern
Decision	To start a software product line
Analysis	Informal
Design	Language Invention and Formal design
Implementation	Preprocessor
Deployment	N/A

design is developed.

We will present overall structure of DSL-SA. DSL-SA specification contains three steps:

(i) DSL-SA Metamodel: We define DSL-SA metamodel by using Eclipse Ecore metamodel. This step is also called as Abstract Syntax Tree definition step.
(ii) DSL-SA Concrete Syntax: Language constructs and grammatical rules of DSL-SA are defined in this step.
(iii) Model-to-Text Transformation: Software architecture that is specified by DSL-SA needs to be transformed to source code. Model (software architecture instance) to text (source code) transformation is achieved in this step.

3.3. DSL-SA Metamodel

We define DSL-SA Metamodel (Abstract Syntax Tree) using Eclipse Ecore metamodel. DSL-SA metamodel for pipes-and-filters architectural style can be seen in Figure 3.2.

Meta-model of DSL-SA contains specialized elements like Source, Sink, Pipe and so on. DSL-SA meta-model has model elements to specify pipes-and-filters architectural style, layered architectural style and free architectural style. It can be extended

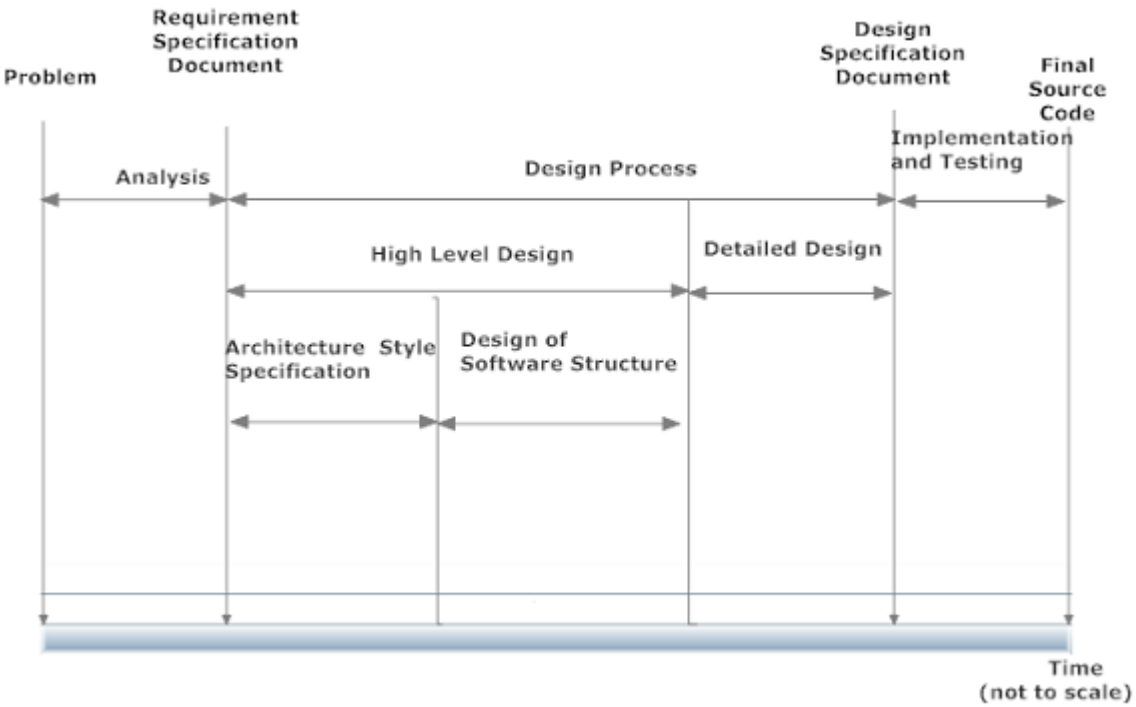

Figure 3.1. Software development phases.

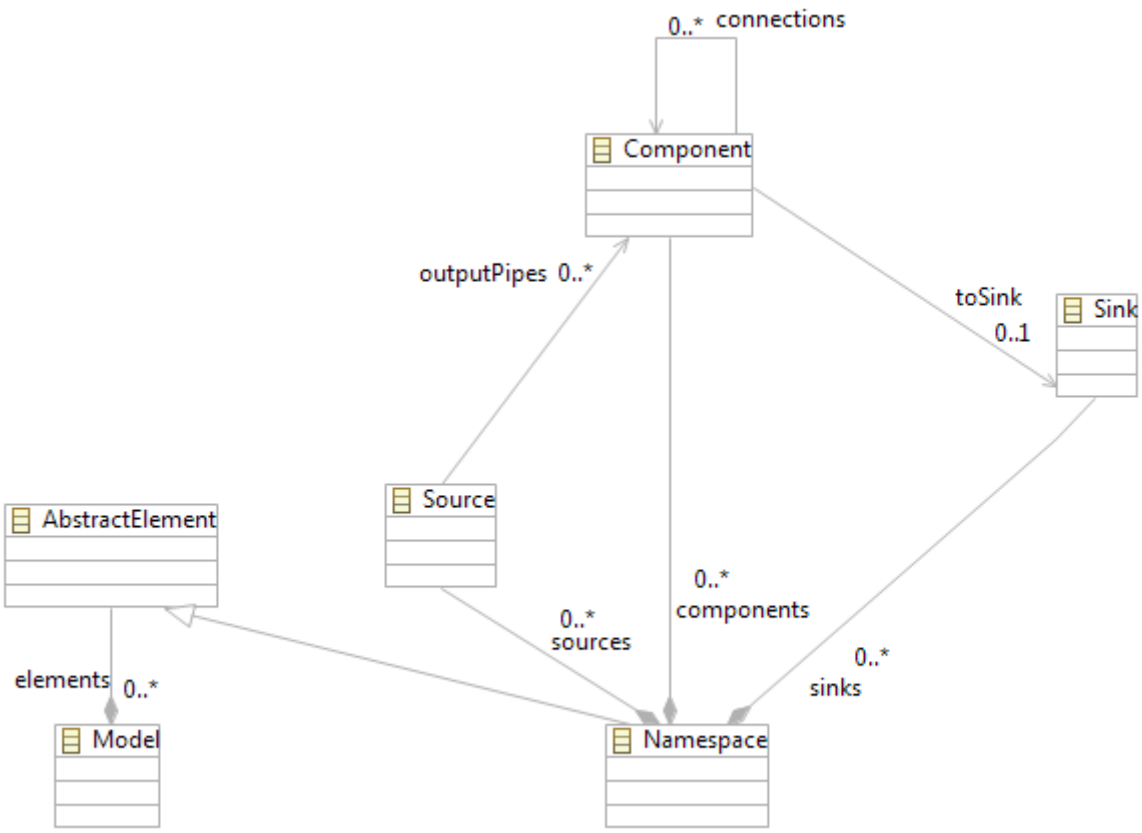

Figure 3.2. Abstract Syntax Tree of DSL-SA for pipes-and-filters architectural style.

to support other architectural styles. Every meta-model element should have a concrete syntax definition, so software architect can use it during software architecture specification step. In addition, every meta-model element should have a corresponding technology mapping in model-to-text step.

3.4. DSL-SA Concrete Syntax

Users of the DSL should be able to create many instances of models by using a concrete syntax. This can be achieved in two ways: Textual or Visual (graphical). If we give example from Eclipse Modeling Project, Xtext can be used to create textual DSLs and to define their concrete syntax. This is the step where we define concrete syntax of DSL-SA to derive DSL-SA editor that is based on Eclipse Graphical Modeling Framework. Steps of DSL-SA concrete syntax development can be seen in Figure 3.3.

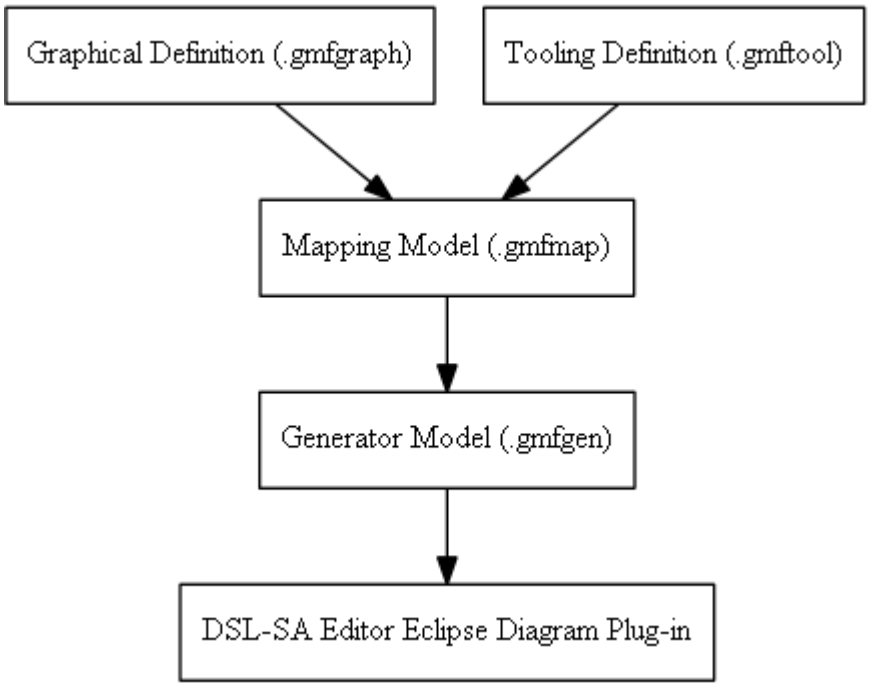

Figure 3.3. DSL-SA concrete syntax definition steps.

Meta-model elements are defined as language constructs in concrete syntax definition step. We define meta-model elements as graphical elements in DSL-SA. DSL-SA Editor utilizes concrete syntax definition to manage meta-model elements, so user can

actually use meta-model elements during specification of software structures that conform to selected software architecture. If we have a look at Figure 3.3, in "Graphical Definition" step, we define our meta-model elements' graphical definitions in a *.gmfgraph file. Element's shapes, rectangle, ellipse etc., figures, graphical relation between meta-model elements are defined in *.gmfgraph file. In "Tooling Definition" step, we define palette which is at rightmost in DSL-SA Editor. Palette contains meta-model elements as icons to drag and drop into editor. Palette is defined in *.gmftool file. DSL-SA meta-model elements, its tool and graphical definitions are mapped together in a *.gmfmap file in "Mapping Model" step. A generator file, *.gmfgen, is created from *.gmfmap file. Generator file generates source code of DSL-SA Editor Eclipse diagram plug-in. Files are created during DSL-SA concrete syntax definition step can be seen in Figure 3.4.

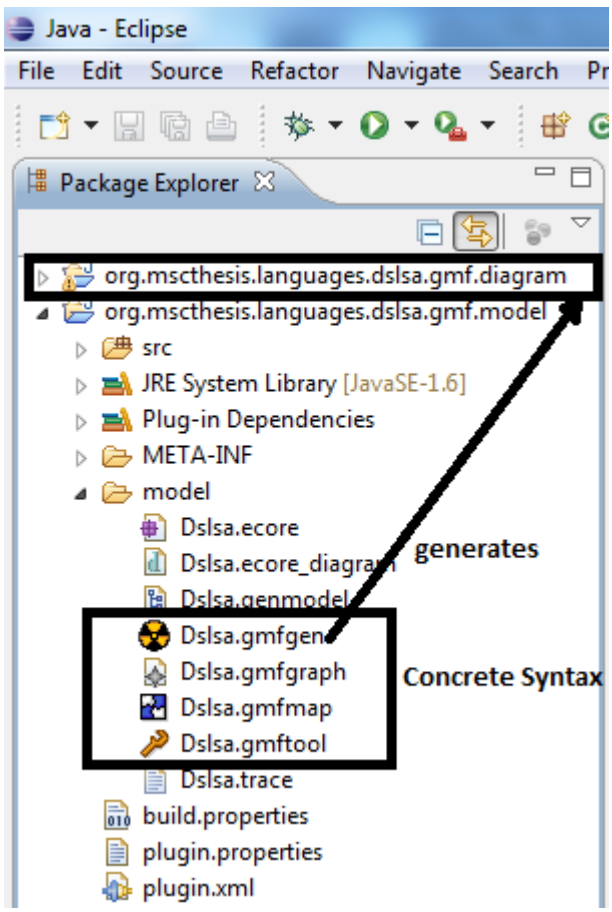

Figure 3.4. DSL-SA Concrete Syntax Definition in Eclipse Package Explorer View.

There are certain rules in software architecture styles. For example, sinks can not have any output pipes, sources can have only output pipes in pipes-and-filters architectural style. In addition, one layer can not contain other layers in layered architectural style. All of these rules are defined during DSL-SA concrete syntax definition. Therefore, once DSL-SA editor derived from this syntax, it is not possible to create flawed software structures for specified styles. It might be seen as trivial for small scale software architectures, but these kinds of language features can be really useful for large and complex software architectures.

3.5. Model-to-Text Transformation

After meta-model and language constructs are defined in meta-model definition step and in concrete syntax definition step, respectively, the last step is to define how user can transform models to some executable software artifact. This step is called model-to-text transformation or technology mapping. DSL-SA Editor use XPAND templates to transform software structures to high-level source code. An example for such XPAND template file is shown in Figure 3.5.

```
«DEFINE SourceClazz FOR Source»
«FILE "Source.java"-»
class Source extends Filter{
    @Override
    public StringBuilder execute(StringBuilder input) {
        «PROTECT CSTART "/*" CEND "*/" ID name»

        «ENDPROTECT»
    }

    @Override
    public void run() {
        for (Pipe outputs : outputPipes) {
            outputs.send(execute(null));
        }
    }
}
«ENDFILE-»
«ENDDEFINE»
```

Figure 3.5. XPAND Template for Source Element.

XPAND template file generates JAVA class file for meta-model element “Source”. Source code can be generated many times after technology mapping is done. DSL-SA uses JAVA programming language for its target language, but it is possible to change target language in future. It requires modifications only in XPAND template files to generate source code in a language other than JAVA. It does not require changing software architecture specification. This is a big advantage especially for large and complex software systems. Analysis and design phases require considerable effort in such systems. Renewing technology is often needed and it is a big problem. If software architecture is specified with DSL-SA, then renewing technology phase can be passed with less pain by changing XPAND templates. Fixing bugs in generated source code also requires modifications in only one place; mapping file.

4. DSL-SA EDITOR

Users of DSLs create DSL instances with the help of an editor. Editor is as important as its DSL. Ease of usage, rich tool support and integration to well-known IDEs are features of a good DSL editor. If editor does not meet such requirements, then DSL might be useless. We will present DSL-SA Editor in this chapter.

4.1. Overall Structure and Organization of DSL-SA Editor

DSL-SA Editor is developed based on Eclipse Graphical Modeling Framework [3] which is a part of Eclipse Modeling Project as it is shown in Figure 2.4. DSL-SA Editor has following functions:

- Software Structures Specification: A software architect can specify software structures that conform to a software architecture by using DSL-SA Editor. DSL-SA Editor currently has the support for pipes-and-filters and layered architectural styles. A software architect can also specify software structures without following any architectural style.
- Software Structures Validation: After software structures are specified, they can be validated for architectural style, backward compatibility and dependency requirements by DSL-SA Editor. Therefore, software structures validity can be checked before source code generation step.
- High-Level Source Code Generation: A valid software specification is input for high-level source code generation. DSL-SA editor generates source code for JAVA, and configuration files for Spring Framework from software structures specification.

DSL-SA Editor is composed of two Eclipse plug-ins:

- DSL-SA Editor Diagram plug-in: This plug-in supplies diagram editor and tool palette. Diagram plug-in is generated from concrete syntax definition directly.

Some source code is developed inside generated diagram plug-in for highlighting errors in software structures specification. Software structures specification rules are enforced by diagram plug-in.

- DSL-SA User Interactions plug-in: This plug-in contains actions, views which are developed not generated. Validation and high-level source generate functions are supplied by this plug-in.

DSL-SA Editor has been developed in JAVA programming language. DSL-SA Editor overall structure and its relation to DSL-SA can be seen in Figure 4.1.

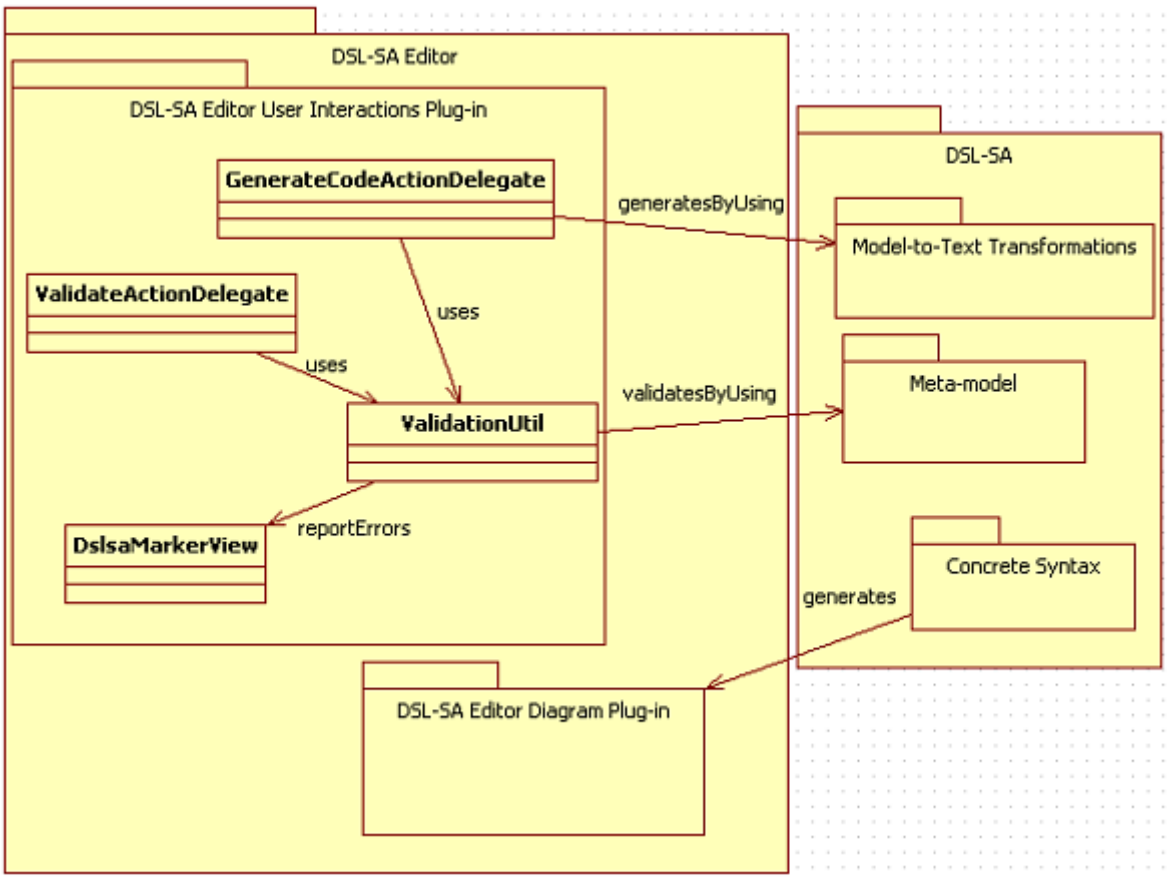

Figure 4.1. DSL-SA Editor Overall Structure.

GenerateCodeActionDelegate and ValidateActionDelegate classes are DSL-SA Editor actions. GenerateCodeActionDelegate uses DSL-SA model-to-text transformations to transform software structures to high-level source code. Both ValidateActionDelegate and GenerateCodeActionDelegate classes use ValidationUtil class to

validate software structures. ValidationUtil class validates software structures according to DSL-SA meta-model. Validation errors are reported on view which is created by DSLSAMarkerview class. DSL-SA Editor Diagram plug-in is generated by DSL-SA concrete syntax definition as illustrated in Figure 3.4, so details of DSL-SA Editor Diagram plug-in are not mentioned here.

4.2. Software Structures Specification by DSL-SA Editor

DSL-SA has a built-in editor to support software architects to specify software structures that conform to a software architecture easily. It has support to define pipes and filters, layered architectural styles, and free architectural style. Software architect can select predefined architecture styles before he specifies software structures as it is illustrated in Figure 4.2. DSL-SA editor supports project creation for free (means no architecture style), pipes-and-filters and layered architecture styles. After an option is selected, editor and its elements, i.e. palette, views, are loaded according to selection.

DSL-SA Editor has drag and drop feature for model elements like Source, Filter etc. Every model element in palette can be dropped to appropriate place in editor. An example in DSL-SA Editor for pipes and filters architectural style can be seen in Figure 4.3. Model elements, Source, Filter, Pipe and Sink which are defined in palette, can be dropped into editor for pipes and filters architectural style.

One layer element can contain other components, but it can not contain other layers in layered architectural style. So a component from palette can be dropped into a layer, but a layer can not be dropped into another layer. DSL-SA Editor limits this. An example for this limitation can be seen in Figure 4.4.

Some model elements can not exist by themselves. They need to be defined inside of another model element or attached to one. For example, Provides and Requires ports in free architectural style need to be attached to one Component. Defining a Port without a Component is meaningless and DSL-SA Editor does not allow it. Once you define a Component, you can click on it to define Provides or Requires ports for

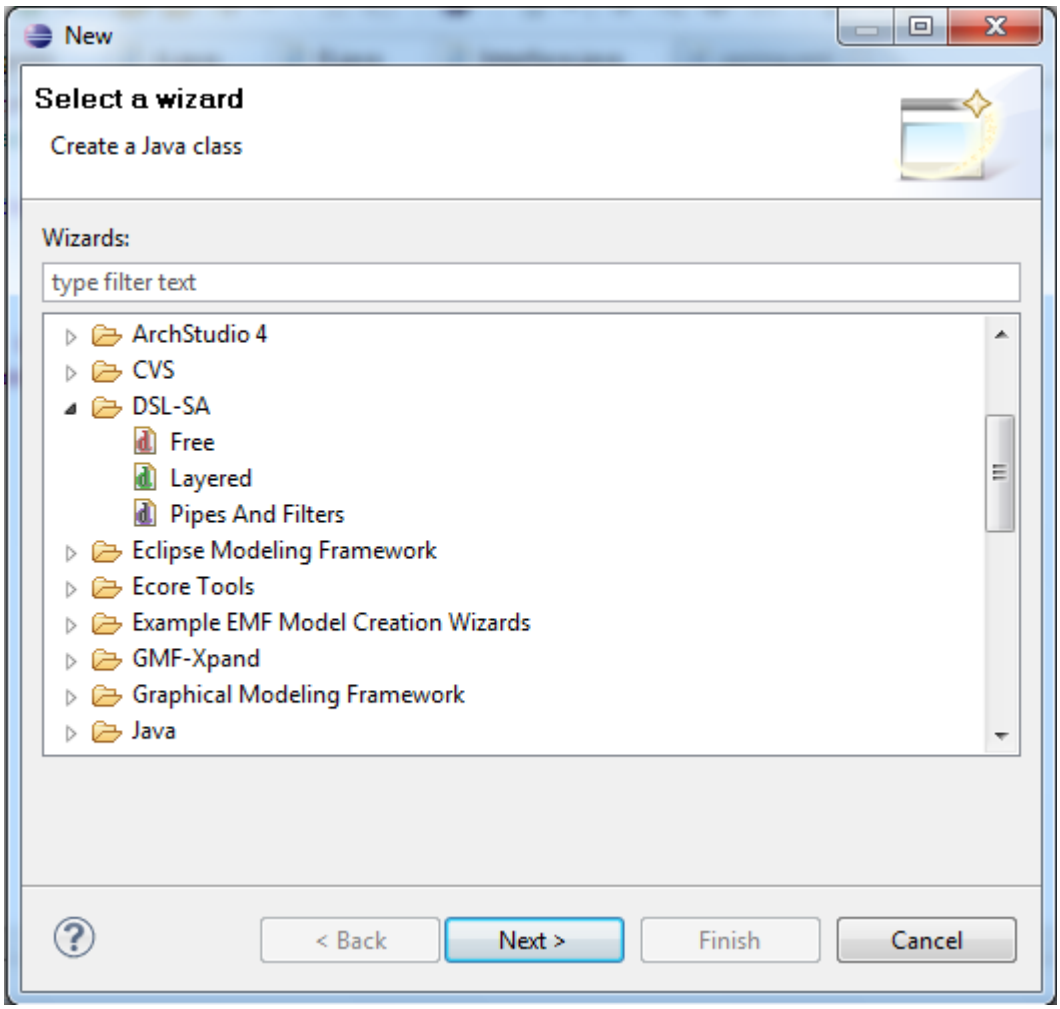

Figure 4.2. DSL-SA New Project Wizard In Eclipse.

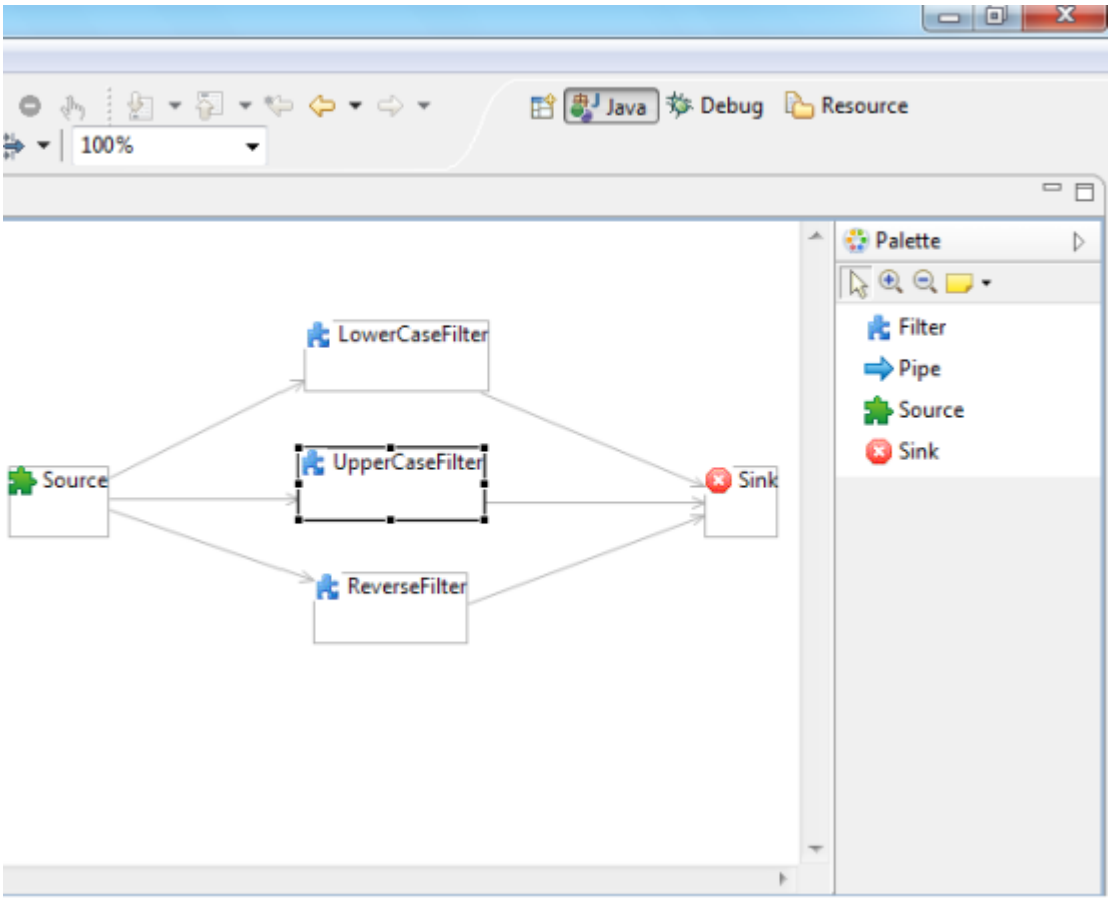

Figure 4.3. Software structures that conform to Pipes-and-Filters Architectural Style in DSL-SA Editor.

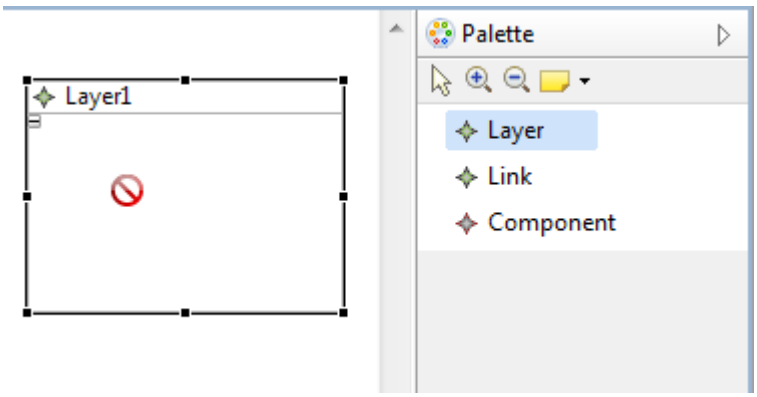

Figure 4.4. Software Structures Specification Rule: Layer can not be dropped into another layer.

that Component. On the other hand, Component, Link and Interface elements can be defined standalone, so they can be dropped into editor directly. Therefore, they are shown in editor palette which is at rightmost in Figure 4.5. Note that, Provides and Requires port elements are shown in tool-tip of Component element. They are not defined in editor palette.

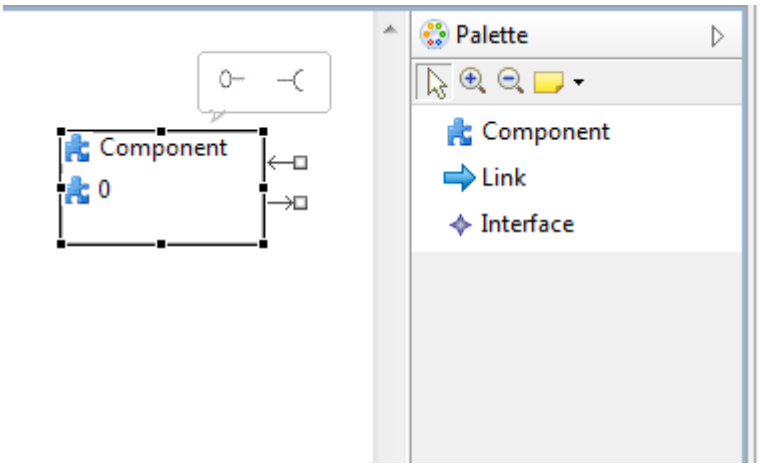

Figure 4.5. Software Structures Specification Rule: Ports can not be defined without a component.

4.3. Software Structures Validation by DSL-SA Editor

After software structures are specified, they need to be validated for architectural style, backward compatibility and dependency requirements. DSL-SA Editor support various type of software structures validation.

4.3.1. SA Style Checks by DSL-SA Editor

DSL-SA Editor has built-in validation support for software architecture that is specified in a known architectural style. For example, there are some rules in pipes and filters architectural style, i.e. Source element can not have any input pipe, Sink element can not have any output pipe. These rules are enforced by editor, so software architect can not create software structures that do not conform to selected software architecture. This is called as Style check feature of DSL-SA Editor. It is shown that

in 4.6, DSL-SA Editor does not allow creating an input pipe from Filter1 to Source element.

Figure 4.6. Pipes-and-Filters Architectural Style Check Error: Source can not have any input pipe.

DSL-SA Editor does not allow Sink to have output pipes, as it is shown in Figure 4.7. An output pipe from Sink element to Filter2 element can not be defined.

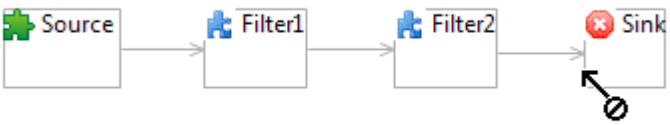

Figure 4.7. Pipes-and-Filters Architectural Style Check Error: Sink can not have any output pipe.

DSL-SA supports Layered architectural style too. In Layered style, every layer can call methods one layer below, nothing lower. Therefore, as it is illustrated in Figure 4.8, User Interface can not invoke any method in Data Layer directly, but it can invoke methods in Business Layer only. After DSL-SA Editor validates software structures specification, it reports such prohibited invocations.

4.3.2. Dependency Checks by DSL-SA Editor

Besides pipes-and-filters and layered architecture styles, DSL-SA let software architect to specify software structures without obeying any architectural style. This is

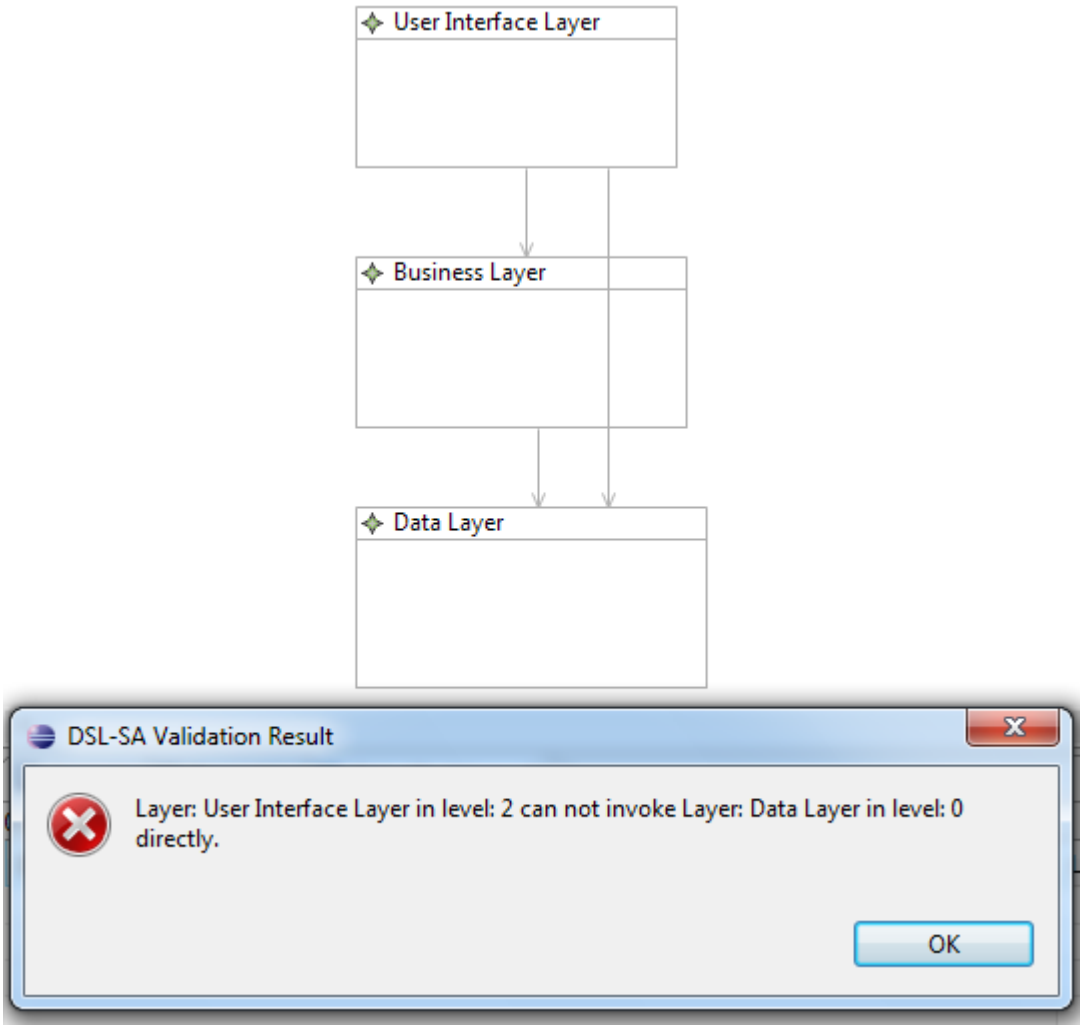

Figure 4.8. Layered Architectural Style Check Error: User Interface Layer can not invoke Data Layer directly.

called as “Free” style in DSL-SA. An example can be seen in Figure 4.9. In this version, every Component has Provides and Requires ports and these ports are bound to some Interfaces. Provides and Requires ports are shown as circle and rectangle figures, respectively. One component may have multiple Provides and Requires ports defined. Relationship between components based on their ports, and ports are bound to interfaces. For example, if component A publishes some interfaces, i.e. implements some Provides port, another component B, which has Requires port in same Interface type, can use functionality provided by component A. Components do not know internal state of each other’s because they communicate based on defined interfaces. We can say component B depends on component A. DSL-SA Editor checks software structure whether these kinds of dependencies are correctly managed or not. This is called as dependency check feature of DSL-SA Editor.

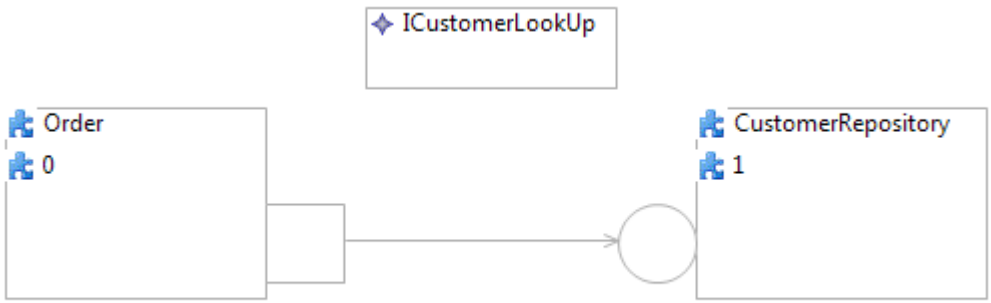

Figure 4.9. An example software structure for free architectural style in DSL-SA Editor.

Component provides and requires some ports. If a component requires a port in Interface type I, but there is not any component to provide this, then software structures specification is invalid because dependency can not be provided. This can be seen in Figure 4.10. Also if a component A which has Provides port in Interface type I, but no components in software structures specification requires this, there is a redundant Interface defined. This can also be a problem and needs to be avoided. Such error can be seen in Figure 4.11. DSL-SA can identify these kinds of dangling ports when software structures validation is done.

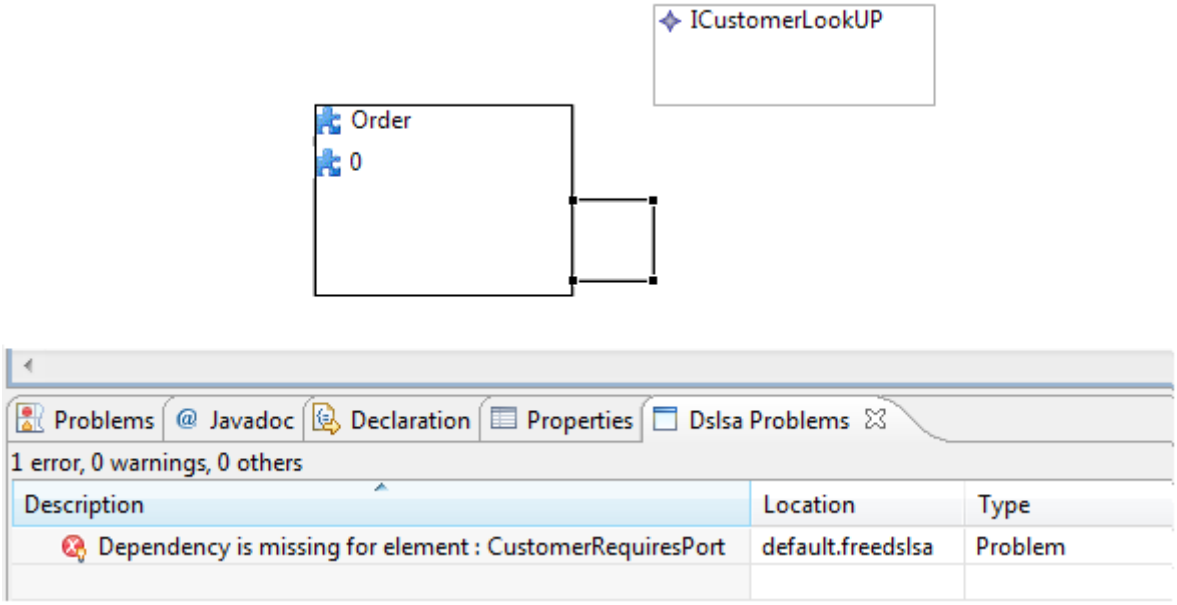

Figure 4.10. Dependency Check Error: Required dependency is not provided.

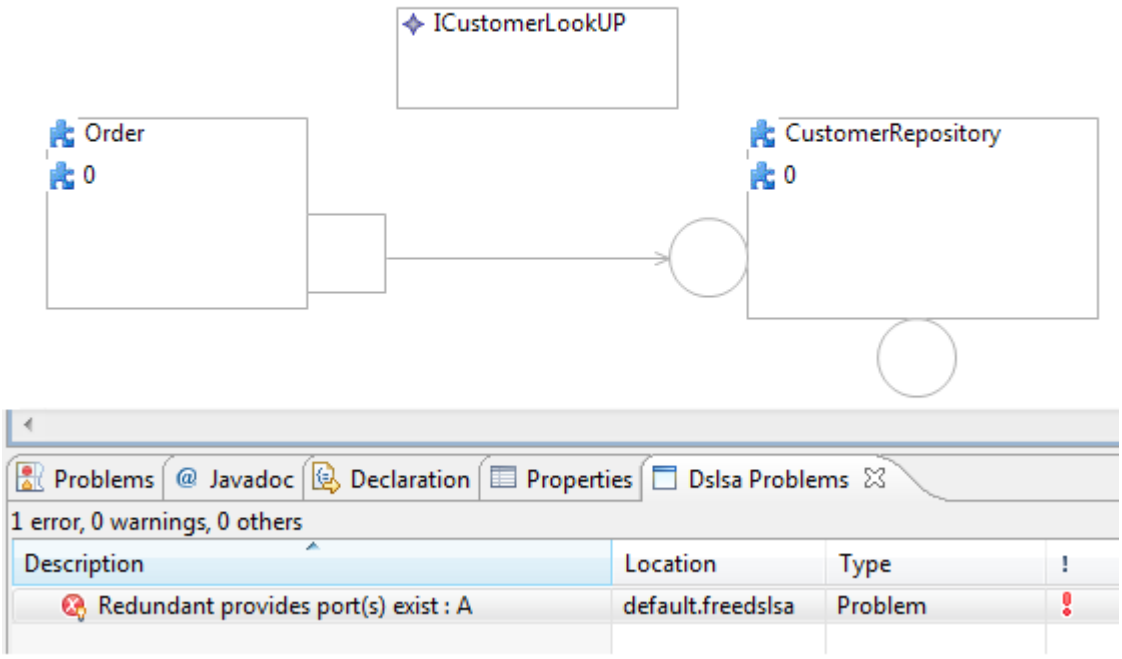

Figure 4.11. Dependency Check Error: Redundant provides port defined.

Another good feature of DSL-SA Editor is identifying circular dependencies. An example to circular dependency is as follows: Component A has some Provides ports which are used by component B, and component B has some Provides port which is used by component A. Both A and B know each other. This is called as "Circular dependency". Circular dependency is not a good practice in Object Oriented principles because it is against to "Low Coupling/High Cohesion" principle. After software architecture is specified, DSL-SA identifies circular dependencies and it warns software architect. An example can be seen in Figure 4.12

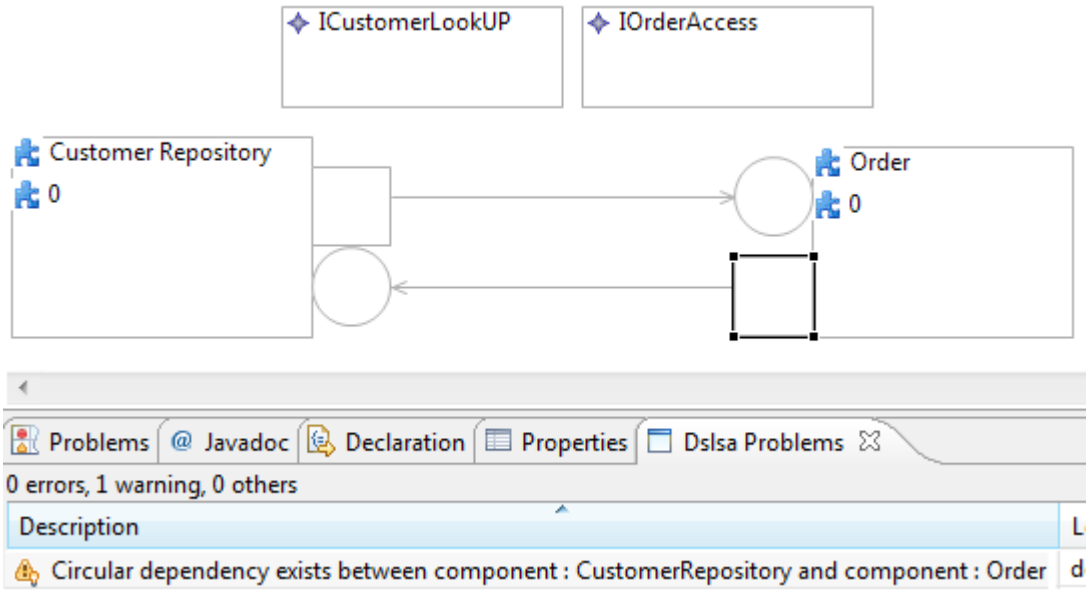

Figure 4.12. Dependency Check Warning: Circular dependency.

4.3.3. Version Checks by DSL-SA Editor

Software requirements may change in time. At some point, newer versions of same component may need to be implemented. This introduces a new problem: backward compatibility. Newer version of component should not break backward compatibility with older versions. It means newer version should not define a new Requires port or should not remove a Provides port. DSL-SA editor helps software architect to check whether these kinds of components exist or not. DSL-SA Editor checks all components and compare with previous versions to see if a Requires port is introduced or a Provides

port is removed. This is the version check feature of DSL-SA Editor. If version check fails, DSL-SA Editor reports an error so software architect can eliminate these errors and generate high-level source code from software structures specification. Examples can be seen in Figure 4.13 and Figure 4.14.

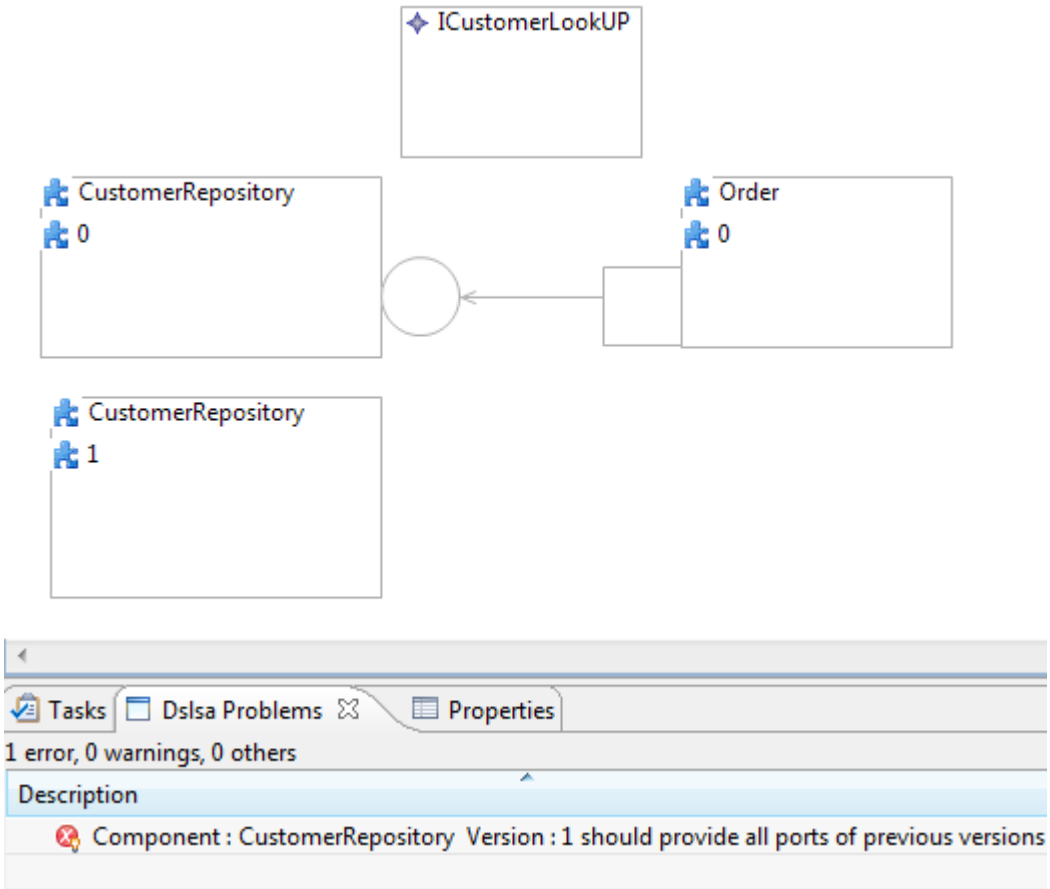

Figure 4.13. Version Check Error: Newer version should provide all ports of previous versions.

4.4. High-Level Source Code Generation by DSL-SA Editor

High-level source code is generated based on software structures that are specified by DSL-SA. After valid software structures are specified, DSL-SA Editor can generate high-level source code from software structures specification. Developers can use generated high-level source code as skeleton classes. For example, generated high-level source code enables messaging infrastructure for pipes-and-filters architectural style.

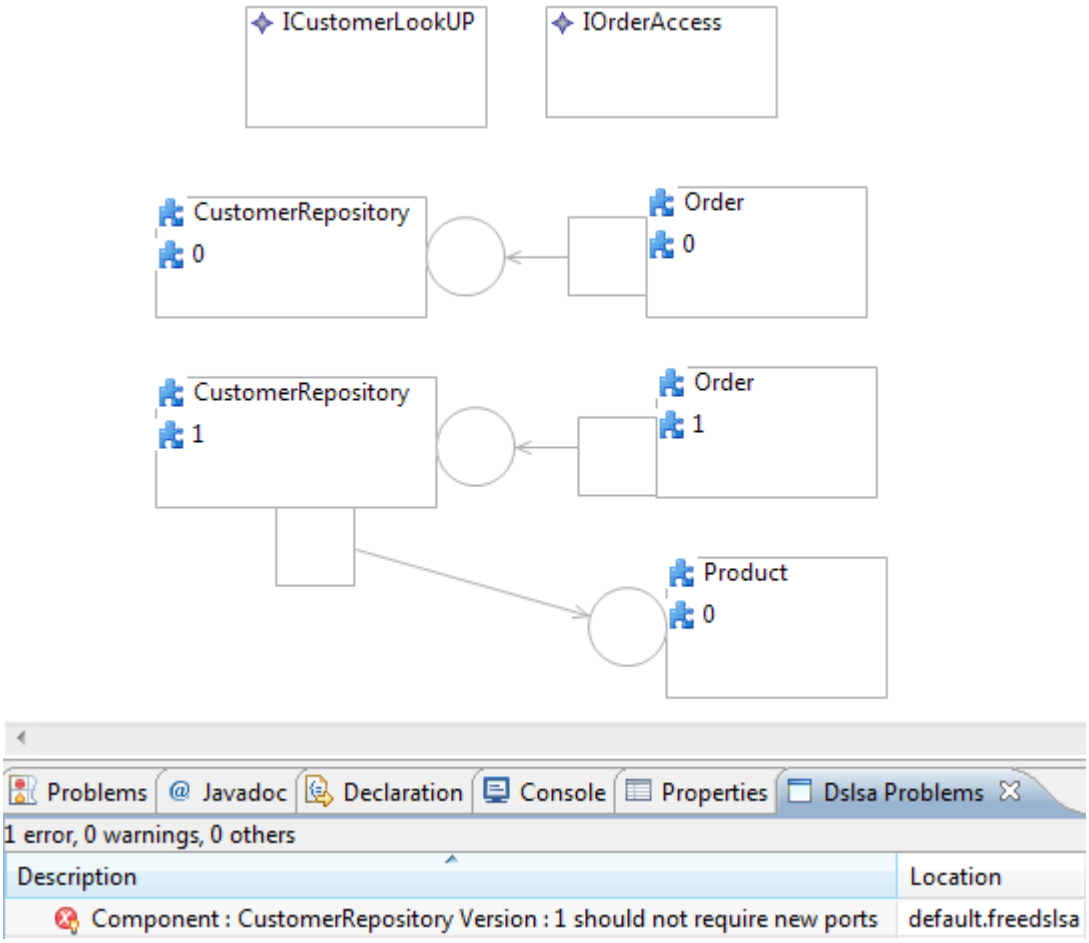

Figure 4.14. Version Check Error: Newer version should not require new ports.

Developers just need to process received messages according to business logic and/or construct new messages to be sent to messaging system. Developers do not need to do any coding to set-up messaging infrastructure. Another example, DSL-SA Editor generates JAVA classes and Spring Framework [15] XML files to manage dependencies in free architectural style. Business logic is implemented inside generated JAVA classes. Because developers can not introduce new dependencies in high-level source code, high-level source code always conforms to software structures specification.

To enable high-level source code generation, DSL-SA metamodel elements need to be mapped to elements in target language. This is called technology mapping. An example for technology mapping can be seen in Table 4.1.

Table 4.1. Technology mapping for JAVA.

DSL-SA Metamodel element	Transformed element in JAVA
Component	Class
Interface	Interface
Message	Method
Port	Variable

As it is shown in Table 4.1, Component element is mapped to JAVA class. Interface element is mapped to JAVA Interface declaration, Message element is mapped to JAVA method declaration and Port definition is mapped to instance variable declaration in JAVA class. One can use XPAND template programming language for technology mapping. DSL-SA Editor utilizes XPAND template file and generates high-level source code from software architecture specification. An example for XPAND template file is illustrated in Figure 3.5.

DSL-SA Editor also generates Spring Framework dependency XML files from software architecture specification. Technology mapping for Spring Framework in DSL-SA Editor is shown in Table 4.2.

Table 4.2. Technology Mapping for Spring Framework.

DSL-SA Metamodel element	Transformed element in Spring Framework
Requires Port	Setter Dependency Injection
Provides Port	Interface Declaration

Requires and Provides Ports are mapped to XML elements for "Setter Dependency Injection" method [15] and XML elements for interface declarations in Spring Framework, respectively. Developer does not need care about dependency management, because XML dependency files for Spring Framework are generated by DSL-SA Editor.

5. DSL-SA CASE STUDIES

We will present three case studies to show how DSL-SA can be used to specify software architecture more easily.

5.1. Software Architecture Specification of a Secure File Transfer System by DSL-SA

We will develop a software system which will transfer files between two computers in a secure way. The whole system can be seen as outgoing port for a company. A file which needs to be sent outside company will be transferred to secure file sender system first. Once file is received, it will be hashed based on a hash algorithm. After file is hashed and message digest is created, it will be encrypted with company private key. After encryption, digital signature will be attached to file, and file will be transferred to outside of company.

Figure 5.1. Secure File Transfer System Architecture Specification.

If we consider outside of the company, which will receive secured file as mail attachment, its SA can be constructed in pipes and filters architectural style too. Once a secured file received to company, source component reads mail attachment and delivers to Decryption filter. File has decrypted in this filter with sender company public key, so sender is authenticated. After file is decrypted, a message digest is constructed in hashing filter. If calculated message digest and received one is same, file is not changed on the way. This phase ensures files integrity. After hashing filter,

file is relayed to output file sink which will store file on the file system.

Figure 5.2. Secure File Receiver System Architecture Specification.

Filters are not necessarily to be deployed on same machine, mostly they are not. Sender filter can be deployed in one machine, hashing and encryption filters on other machines. Hence secure file transfer system should have a network distributed architecture. A node matches to one physical machine in network and it may contain one or more filters. Messaging between different nodes will be based on Java Messaging Services (JMS). So filters will be mapped to nodes, and pipes will be mapped to JMS queues. For secure file transfer system, we will use point-to-point message delivery.

As a result of having software architecture defined at the beginning, incorrect messaging is not allowed. For example Node C, on which Encryption Filter deployed, can not send any message to Node B, on which Hashing Filter deployed. Such messaging code will be lost in next code generation, so developer will not develop any code outside of code skeletons. He will restrain from breaking software architecture. If we do not define software architecture and program messaging infrastructure code directly, this kind of illegal messaging can not be detected easily, especially if system is quite large. If one messaging service rather than JMS needs to be used, only corresponding technology mapping file needs to be changed. This is another benefit defining software architecture at the beginning. Secure File Transfer System (SFTS) JMS queues and message counts are displayed in Active MQ admin console in Figure 5.3.

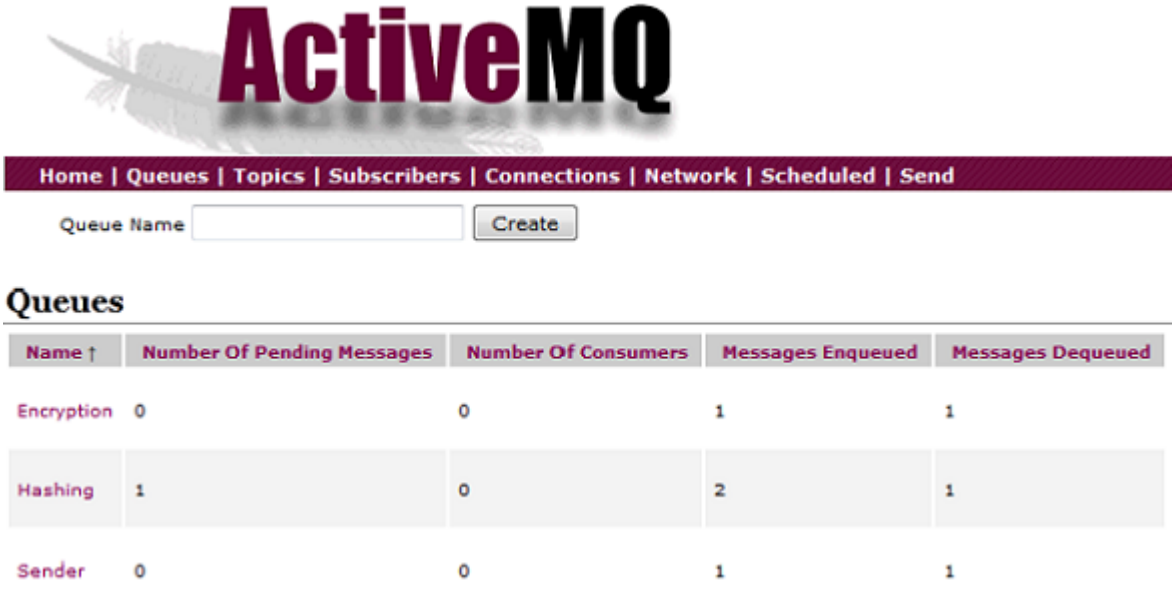

Figure 5.3. SFTS JMS Queues and Message Counts in ActiveMQ Admin Console.

5.2. Software Architecture Specification of a Lowest Price Book Finder System by DSL-SA

This case study is a modification of the example presented in [16]. We have a customer who wants to buy a certain book with the lowest price. We are constructing a system which can return lowest price bookstore address for a particular book. The problem is every bookstore has its own messaging format, so we need to transform one message format to another message format.

Software architecture is constructed in pipes-and-filters style. Every request come to system is logged by Logger filter and then passed to Multicasting Router filter. This filter send request message to every bookstores. As stated before, some of bookstore has its own messaging format. A transformer filter added between multicasting router and message gateway to do this message transform. Message gateway filters are filters that relay message to outside system, i.e. Amazon.com, and waits for message response. Once gateway receives message it sends it to Aggregator filter. Again a transformer filter can take place between gateway and aggregator filters. Aggregator filter waits for

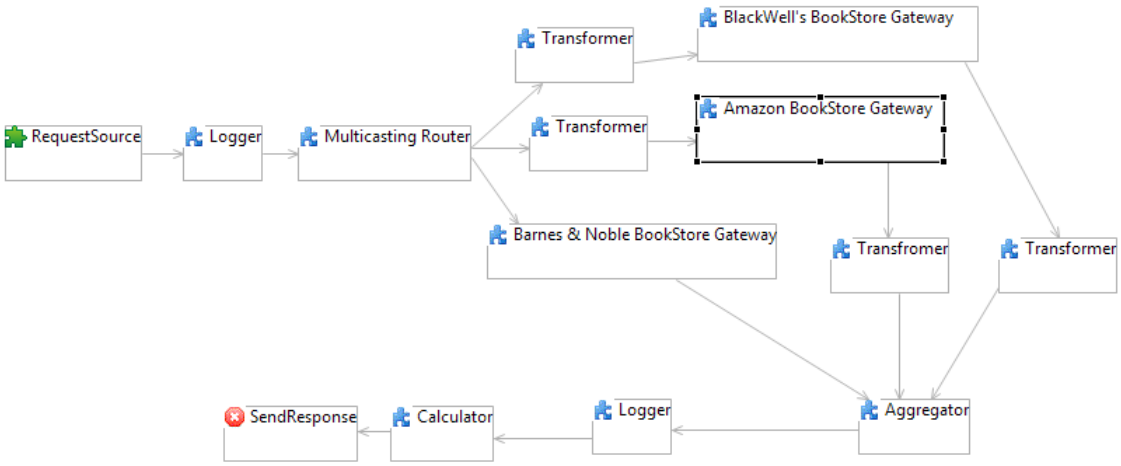

Figure 5.4. Lowest Price Book Finder Architecture Specification.

all messages to come, so it works in synchronized mode. It collects all messages and forwards to Logger for logging. After logging part, Calculator filter calculate lowest price bookstore, filter out others, and return bookstore address.

5.3. Software Architecture Specification of an Airport Management System by DSL-SA

Airport Management System Architecture Specification case study is changed from one of inspiring related works for our study. It is an example given in [7]. Author develops textual DSL in his work to define SA. We will show how same SA can be developed with DSL-SA. DSL-SA is a graphical DSL, and we believe a graphical DSL requires less effort than a textual DSL does for this case study.

An airport management system has several functionalities like reporting take-off and landing times for flights, registering and unregistering flights and so on. Airport management system should contain many components to implement these functionalities ; as a result it is a large and complex system. There should be a software architecture specified at design phase of such systems. Otherwise, it is impossible to manage complexity in this scale. We will use DSL-SA to define SA of airport man-

agement system and we will derive high-level source code from defined SA. A part of airport management system SA defined by DSL-SA can be seen in Figure 5.5.

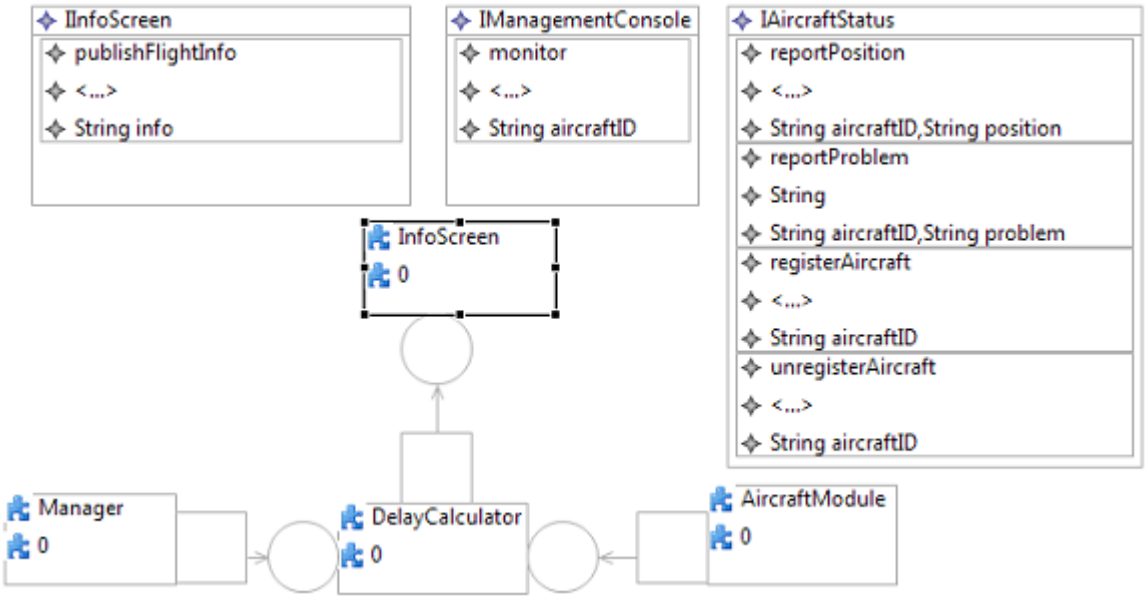

Figure 5.5. Airport Management System Architecture Specification.

Airport management system SA is an example of free DSL-SA. It does not have any architectural style like pipes and filters or layered styles. As it is shown in Figure 5.5, it has three interfaces, IInfoScreen, IManagementConsole and IAircraftStatus and four components, InfoScreen, Manager, DelayCalculator, AircraftModule respectively. Dependencies between components are clearly defined with requires and provides ports and these ports are bound to specific interfaces. Interfaces define messages. For example, IInfoScreen defines one message, publishFlightInfo. This message takes one parameter , named as info which is in String type.

According to validation rules defined at "Software Structures Validation" section, there is not any validation error in this software architecture and DSL-SA Editor validates it successfully. Once software architecture is validated, DSL-SA Editor generates JAVA classes and Spring Framework [15] dependency XML files. AircraftModule and DelayCalculator component's JAVA classes are given as examples in Figure 5.6 and 5.7 respectively.

```
class AircraftModule {

    IAircraftStatus calculator;

    public void setCalculator(IAircraftStatus calculator) {
        this.calculator = calculator;
    }
    public IAircraftStatus getCalculator() {
        return calculator;
    }

}
```

Figure 5.6. AircraftModule Component JAVA Class.

AircraftModule has a Requires port, which is in IAircraftStatus type. This required dependency is provided by DelayCalculator with its aircraft port. DelayCalculator component implements all messages in IAircraftStatus interface. These messages are mapped to method declarations. A component which has IAircraftStatus as a required port, is translated to a JAVA class which has an instance variable in Interface type. As it is shown Figure 5.6, AircraftModule class has an instance of IAircraftStatus type, so it can call all methods defined in that interface. DelayCalculator, which provides this interface, implements methods in that interface type. So once AircratModule instance invokes methods in IAircraftStatus, like reportPosition, method in DelayCalculator class is called. Here developer only needs to write business logic codes inside methods. For example, he needs to do required operations to report a problem. He should not define new methods (hence new messages) in component level directly, because it will break source code conformance to SA definition. It will be also lost in subsequent high-level source code generation from SA.

```
class DelayCalculator implements IAircraftStatus, IManagementConsole {

    IInfoScreen screen;

    public void setScreen(IInfoScreen screen) {
        this.screen = screen;
    }
    public IInfoScreen getScreen() {
        return screen;
    }
    public void reportPosition(String aircraftID, String position) {
        // TODO Auto-generated method stub

    }
    public String reportProblem(String aircraftID, String problem) {
        // TODO Auto-generated method stub
        return null;

    }
    public void registerAircraft(String aircraftID) {
        // TODO Auto-generated method stub

    }
    public void unregisterAircraft(String aircraftID) {
        // TODO Auto-generated method stub

    }
    public void monitor(String aircraftID) {
        // TODO Auto-generated method stub

    }
}
```

Figure 5.7. DelayCalculator Component JAVA Class.

6. CONCLUSION

In this thesis, a new specification language, called Domain-Specific Language for Software Architecture Specification (DSL-SA), is implemented. Software structure that conforms to specific software architecture specified by DSL-SA is not only a plain document, but also a work product which helps software architects, developers and all other stakeholders during software development.

In association with DSL-SA, a software tool, called DSL-SA Editor, has been developed. DSL-SA Editor is used to specify software components that correspond to specific software architecture styles. Support for three architectural styles, pipes-and-filters architectural style, layered architectural style and free architectural style has been developed for DSL-SA Editor. DSL-SA Editor supports the validation of software structures, to see whether the software structure conforms to the selected architectural style. DSL-SA Editor also supports high-level source code generation from software structure specification.

We presented three case studies, namely, Software Architecture Specification of a Secure File Transfer System by DSL-SA, Software Architecture Specification of a Lowest Price Book Finder System by DSL-SA, and Software Architecture Specification of an Airport Management System by DSL-SA, to show how DSL-SA and DSL-SA Editor can be used to specify software components that correspond to a specific software architecture.

Our work is in intersection of software architecture, model driven software development and domain-specific languages fields. It can be classified as architecture-centric MDSD which is one flavor of model driven software development. We focused on to define architecturally relevant elements with DSL-SA and to generate high-level source code skeletons based on software architecture specification. We leave details of design and final source code production to software developers.

As future work, we plan to improve DSL-SA editor to support abstract state machines. Adding security properties to DSL-SA and applying model checking to software structure is another future direction for our work. Our aim is to utilize DSL-SA to have a software product line to create architecturally relevant products to maximize its benefits.

REFERENCES

1. Taylor, R. N., N. Medvidovic and E. M. Dashofy, *Software Architecture: Foundations, Theory and Practice*, Addison-Wesley, 2007.

2. Stahl, T. and M. Völter, *Model-Driven Software Development: Technology, Engineering, Management*, Wiley, Chichester, UK, 2006.

3. Gronback, R. C., *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, Addison-Wesley Professional, 2009.

4. Mernik, M., J. Heering and A. M. Sloane, "When and How to Develop Domain-Specific Languages", *ACM Computing Surveys (CSUR)*, Vol. 37, No. 4, pp. 316–344, 2005.

5. Feiler, P. H., D. P. Gluch and J. J. Hudak, *The Architecture Analysis & Design Language (AADL): An Introduction*, Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, 2006.

6. Khare, R., M. Guntersdorfer, P. Oreizy, N. Medvidovic and R. N. Taylor, "xADL: Enabling Architecture-Centric Tool Integration with XML.", *Hawaii International Conference On System Sciences*, 2001.

7. Völter, M., "Architecture as Language", *IEEE Software*, Vol. 27, No. 2, pp. 56–64, 2010.

8. Pelliccione, P., P. Inverardi and H. Muccini, "CHARMY: A Framework for Designing and Verifying Architectural Specifications.", *IEEE Transactions on Software Engineering*, Vol. 35, No. 3, pp. 325–346, 2009.

9. Eden, A. H. and R. Kazman, "Architecture, Design, Implementation", *Software Engineering, International Conference on*, 2003.

10. Völter, M., "Software Architecture - A pattern language for building sustainable software architectures", *European Conference on Pattern Languages of Programs*, 2006.

11. Institute, S. E., *Software Architecture Glossary*, 2010 `http://www.sei.cmu.edu/architecture/start/glossary/index.cfm`, accessed at July 2012.

12. Steinberg, D., F. Budinsky, M. Paternostro and E. Merks, *EMF: Eclipse Modeling Framework*, 2. edn., Addison-Wesley, Boston, MA, 2009.

13. Efftinge, S. and M. Völter, "oAW xText: A framework for textual DSLs", *Eclipsecon Summit Europe 2006*, 2006.

14. Fowler, M., *Domain-Specific Languages*, Addison-Wesley Professional, 2010.

15. Johnson, R., J. Hoeller, A. Arendsen, T. Risberg and C. Sampaleanu, *Professional Java Development with the Spring Framework*, Wiley, 2005, `http://www.springframework.org/node/128`.

16. Rademakers, T. and J. Dirksen, *Open-Source ESBs in Action*, Manning Publications Co., Greenwich, CT, USA, 2008.