Software Composition with Templates

PhD Thesis

Kostyantyn Yermashov

This thesis submitted in partial fulfilment of the
requirements for the degree of Doctor of
Philosophy, awarded by De Montfort University

June 2008

ALL MISSING PAGES ARE BLANK

IN

ORIGINAL

Abstract

Software composition systems are systems that concentrate on the composition of components. These systems represent a growing subfield of software engineering. Traditional software composition approaches define components as black-boxes. Black-boxes are characterised by their visible behaviour, but not their visible structure. They describe *what* can be done, rather than *how* it can be done. Basically, black-boxes are structurally monolithic units that can be composed together via provided interfaces. Growing complexity of software systems and dynamically changing requirements to these systems demand better parameterisation of components. State of the art approaches have tried to increase parameterisation of systems with so-called grey-box components (grey-boxes). These types of components introduced a structural configurability of components. Grey-boxes could improve composability, reusability, extensibility and adaptability of software systems. However, there is still there is a big gap between grey-box approaches and business.

We see two main reasons for this. Firstly, a structurally non-monolithic nature of grey-boxes results in a significantly increased number of components and relationships that may form a software system. This makes grey-box approaches more complex and their development more expensive. There is a lack of tools to decrease the complexity of grey-box approaches. Secondly, grey-box composition approaches are oriented to the experts with a technical background in programming languages and software architectures. Up to now, state-of-the-art approaches have not addressed the question of their efficient applicability by domain experts with no technical background in programming languages. We consider a structural visibility of grey-boxes gives a chance to provide better externalisation of business logic, so that even a non-expert in programming language could design a software system for his/her special domain.

In this thesis, we propose a holistic approach, called Neurath Composition Framework, to compose software systems according to well-defined requirements which have

ABSTRACT

been externalised, giving the ownership of the design to the end-user. We show how externalisation of business logic can be achieved using grey-box composition systems augmented with the domain-specific visual interfaces. We define our own grey-box composition system based on the Parametric Code Templates component model and Molecular Operations composition technique. With this composition system awareness of a design, comprehensive development and the reuse of program code templates can be achieved. Finally, we present a sample implementation that shows the applicability of the composition framework to solve real-life business tasks.

Declaration

I declare that the work described in this thesis is original work undertaken by me between October 2003 and February 2008 for the degree of Doctor of Philosophy, at the Software Technology Research Laboratory (STRL), Department of Computer Science and Engineering (CSE), De Montfort University, United Kingdom.

Acknowledgements

I would like to thank all the people who supported and inspired me during my research. I would like to especially express my sincerest and deepest gratitude to both of my supervisors, Professor Karl Hayo Siemsen and Professor Hussein Zedan, for their helpful advice and great care. I want to thank Karl Hayo Siemsen for giving me the idea to do the PhD and for providing excellent conditions to begin. Also, I want to express special thanks to Hussein Zedan for his constructive and crucial advice.

I wish to thank the whole Software Technology Research Laboratory team (De Montfort University, UK) for the extremely valuable discussions and pleasant working environment. Additional thanks go to the team of the Laboratory of Parallel Processes (University of Applied Sciences, Emden, Germany).

Finally, I would like to express my deepest emotional thanks to my parents, Nikolaj and Ludmila Yermashov, for always being there for me and for their immeasurable care.

Publications

(1) K. Yermashov, K. Wolke, K.H. Siemsen, Design of Domain-Specific Software Systems with Parametric Code Templates, The Third International Conference on Software Engineering, ICSE 2006, Venice, November 2006, ISSN 1305-5313.

(2) Kostyantyn Yermashov: "Architecture of the Neurath Basic Model View Controller". Int. Symposium XA2006 European Conference on Computer Science & Applications. "Tibiscus" University of TIMISOARA, RO, ISBN: 973-661-990-7, June 2006.

(3) K. Wolke, K. Yermashov, A. Austermann et al. "Das Java 2 JDK 5 Lehrbuch, Sprache und Bibliotheken"[engl. "The Java 2 JDK 5 Handbook, Language and Libraries"], C&L, Computer und Literatur Verlag GmbH, Böblingen, Germany, ISBN: 3-936546-25-8, January 2005.

(4) K. Wolke, K. Yermashov, K. H. Siemsen, R. A. Rasenack and C. Abt, "Abstrakte Syntaxbäume verwalten Quelltexte" [engl. "Abstract Syntax Trees for Source Code Management"], Toolbox magazine, Germany, September/October 2004.

(5) K. Yermashov, K. Wolke, K. H. Siemsen, C. Abt and R. A. Rasenack, "Vom Diagramm zum Quelltext"[engl. "From Diagram to Source Code"], Toolbox magazine, Germany, May/June 2004.

Contents

List of Figures

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

List of Tables

LIST OF TABLES

Listings

LISTINGS

List of Acronyms

ADLs ..... Architecture Description Languages, page 37

AHEAD ..... Algebraic Hierarchical Equations for Application Design, page 46

AOP ..... Aspect Oriented Programming, page 3

API ..... Application Programming Interface, page 100

ASLT ..... Abstract Syntax Language Tree, page 9

COM ..... Component Object Model, page 1

CORBA ..... Common Object Request Broker Architecture, page 1

COTS ..... Components-Off-The-Shelf, page 1

DS – PCT ..... Domain-Specific Parametric Code Template Language, page 80

DSC(*s*) ..... Domain-specific Component(s), page 88

DSL ..... Domain-specific Language, page 51

DSM ..... Domain-specific Modelling, page 51

DSO(*s*) ..... Domain-specific Operation(s), page 88

DSVCS ..... Domain-Specific Visual Composition System, page 69

DSVI ..... Domain-Specific Visual Interface, page 78

DSVL ..... Domain-specific Visual Language, page 51

EJB ..... Enterprise JavaBean, page 1

LIST OF ACRONYMS

FOP ..... Feature Oriented Programming, page 3

GoF ..... Gang of four, page 49

GOOD ..... Grammar-Oriented Object Design (GOOD), page 50

GUI ..... Graphical User Interface, page 24

HTML ..... HyperText Markup Language, page 51

ISC ..... Invasive Software Composition, page 46

Isotype ..... (International System of Typographic Picture Education, page 68

JavaCC ..... Java Compiler Compiler, page 217

MDSoC ..... Multi-Dimensional Separation of Concerns, page 41

MO(s) ..... Molecular Operation(s), page 134

NBT ..... Neurath Builder Tool, page 7

NCF ..... Neurath Composition Framework, page 5

NIP ..... Neurath Integration Platform, page 7

NMCs ..... Neurath Modelling Components, page 10

NML ..... Neurath Modelling Language, page 78

PCT ..... Parametric Code Template, page 7

PCT – L ..... Parametric Code Template Composition Language, page 7

SkwContext ..... Simple Knowledge Web Context, page 160

SOP ..... Subject-Oriented Programming, page 42

SQL ..... Structured Query Language, page 51

UIIE ..... User Interface Interaction Expressions, page 89

UML ..... Unified Modelling Language, page 33

XML ..... Extensible Markup Language, page 234

Chapter 1

Introduction

At the end of the 1960s, during the NATO Conference of Software Engineering, the scientific community recognised such terms as *software crisis* and *component technology*. These terms influenced the field of software construction for at least 30 years. The term *software crisis* stated that the size and complexity of software systems had grown so enormously that planning, implementation, and maintenance could no longer be managed. It was McIlroy who proposed a way to overcome these problems with the help of component technology. He proposed the application of building blocks (components) to build up software systems instead of the traditional at that time handcraft development by skilled individuals. Currently, many companies in the software industry try to build software from prefabricated components, components-off-the-shelf (COTS). COTS reduce development costs and result in a better time to market of software products. Examples of developed COTS systems include Enterprise JavaBeans (EJB) [74], COM [18] and CORBA [79].

The nature of components is not always the same. Components appear on different granularity levels, deal with different stakeholder requirements, or are simply design concepts. Components themselves have no practical value without the knowledge of answers to the following questions:

1. What should a component look like?
2. How should components be composed?
3. How is a system composed?

A *Composition System* incorporates this knowledge. Composition systems are systems that concentrate on the composition of components. They generalise many of the ap-

proaches to component-based engineering that we have seen in the last 40 years. Composition systems are defined through a *component model*, a *composition technique* and a *composition language*. The efficiency of a software process and products of this process depend on the qualities of the composition system used.

The ideas of McIlroy influenced and resulted in a large number of approaches for component technology presented by both research and industry. These approaches are still of great importance, as not all software composition problems are solved yet.

1.1 Motivation and Objectives of Research

Basic traditional software composition approaches are based on components such as modules, classes and services that were developed in the past five decades. Parnas et al. [71, 72] discussed the modularisation as a mechanism for improving the flexibility and comprehensibility of a system while enabling a reduction in its development time. Modules introduced encapsulation, combining data and behaviour in one package and hiding the implementation of the data from the user of the object. Modular systems can be flexibly recombined during design time.

At approximately the same time, object-oriented composition systems were introduced in order to simplify the development of software systems and to make it possible to change systems dynamically (during runtime). Objects were mainly introduced as elements of software composition during the development of Simula 67 [28] programming language, which was influential for the development of later object-oriented software composition models. Further, classic component-based (or service-based) composition systems were presented. In comparison to objects, components (services) represent stand-alone service providers. They represented the call-by-service paradigm.

Traditional software composition approaches *are characterised by applying the black-box component paradigm*. Black-boxes describe *what* can be done, rather than *how* it can be done. The black-box components represent a part of the system's implementation. They show the behaviour and hide the internal structure of components. Black-box components are structurally monolithic building blocks that could be connected and accessed via provided interfaces. Parameterisation of software systems based on black-boxes has been achieved with variables defined by the internal structure of components and by the interfaces. Assigning the values to variables, the black-box components could be cus-

tomised. With interfaces, the components could be connected in different ways. In industry, black-box approaches have been successfully applied for years.

The growing complexity of software systems and dynamically changing requirements to these systems required better parameterisation of components for achieving better reconfiguration and reuse. Starting from architecture-based composition approaches, *the elements of composition were treated more like grey-boxes*. As opposed to black-box components, the structure in grey-boxes is visible and therefore non-monolithic. Potential grey-boxes are opened for structural and thereby behavioural transformations. The architecture-based composition introduced connectors and ports. These components represented communication routines abstracted from the component implementations and could be exchanged by need.

State of the art composition approaches, such as Hyperspace Programming [88], Aspect Oriented Programming (AOP) [55], Feature Oriented Programming (FOP) [14] and Invasive Software Composition [8] define and apply grey-box components, such as *aspects*, *features* and *hypermodules*. Composition systems based on grey-box components significantly increase the structural parameterisation of software systems, potentially improving their extensibility and adaptability.

Grey-box based composition approaches are still in their infancy. The following problems are recognised:

P1: Grey-box based composition approaches are too complex to be efficiently applied in business;

P2: Component models for grey-box components are weak;

P3: The value of grey-box approaches for domain experts has not yet been fully recognised.

The *P1* problem means that compared to black-box based approaches, it is more difficult to maintain grey-box composition systems. Higher parameterisation of grey-box components results in a significantly increased number of entities that can take part in forming a software system. Typically, it costs more to create and use grey-box components, as specific expert knowledge is involved. Maintenance problems are the main obstacle in applying grey-box based composition approaches at a practical level. The *P2* problem means that the component models used to define grey-box components are not yet comprehensive enough. This can be optimised. The *P3* problem states that grey-box based composition can be valuable for domain experts. The non-monolithic nature of

grey-boxes allow for the separation and composition of domain-specific concerns. Up to now, grey-box approaches have not addressed this question.

The introduced problems result in the following objectives:

O1 : There is a need for an effective interface, or bridge, between domain experts and grey-box based composition systems. Such an interface shall abstract domain experts from the complexity of the composition system domain. With such an interface, the system can be configured at all levels of composition, such as domain-specific and language specific, simulateneously.

O2 : There is a need for a more comprehensive component model and composition technique to work with grey-box components.

1.2 Research Questions

During the literature review of the software composition systems, we recognised a tendency of applying grey-box (represented by program code *templates* in this thesis) components in the state of the art composition systems. Besides, we have seen a gap between the industry and these approaches. Taking these into consideration, we specify the main research question this thesis tries to answer:

"There is a need for holistic engineering process to system configuration, such that the end-user is provided with the ability to change the requirements at the highest level with automatic realisation of the changes at the lowest - implementation level. The end-user does not need to know the implementation level. How this can be achieved?"

Further research questions follow:

RQ1: What is the strategy to externalise business logic in template-based composition systems?

- What kind of a system is developed, who are the actors, where is a composition system going to be applied (non-functional requirements)?
- What are the main concepts and how do they relate (functional requirements)?
- What is the software life-cycle for such a composition system?

RQ2: What is the component model, the composition technique and the composition language of a template-based composition system?

- How can a template be encapsulated?
- What can be parameterised?
- What are interfaces to components?
- How are templates manipulated?
- How are the adaptability, extensibility and reusability of templates provided?
- What is a composition language for templates?

RQ3: How to externalise business logic in template-based composition systems up to the level of the domain expert?

- How are application domains described?
- What is the component model suitable to define the repository of components for any domain?
- What is the composition technique to define a repository of operations for any domain?
- What is the domain-specific composition language?
- How is the domain-specific composition language mapped onto the composition language?

RQ4: How is the perception and interaction with a template-based composition system increased?

- How are domain-specific visual interfaces (DSVIs) formed?
- What kind of interactions with DSVIs are possible and how they are defined?

1.3 Scope of the Research

The first research question, *RQ1*, is rather global. Answering this question results in a broad understanding of aims and results as well as concepts and their cooperation. Those are presented in the form of a conceptual framework called Neurath Composition Framework (NCF). The objective during the processing of the *RQ1* is to establish an overview

of facilities for the definition and application of template-based composition systems together with business logic externalisation mechanisms.

The research question *RQ2* is related to the definition and application of template composition systems. We define a *Template Level* of composition that collects all the concepts needed for the template-based composition.

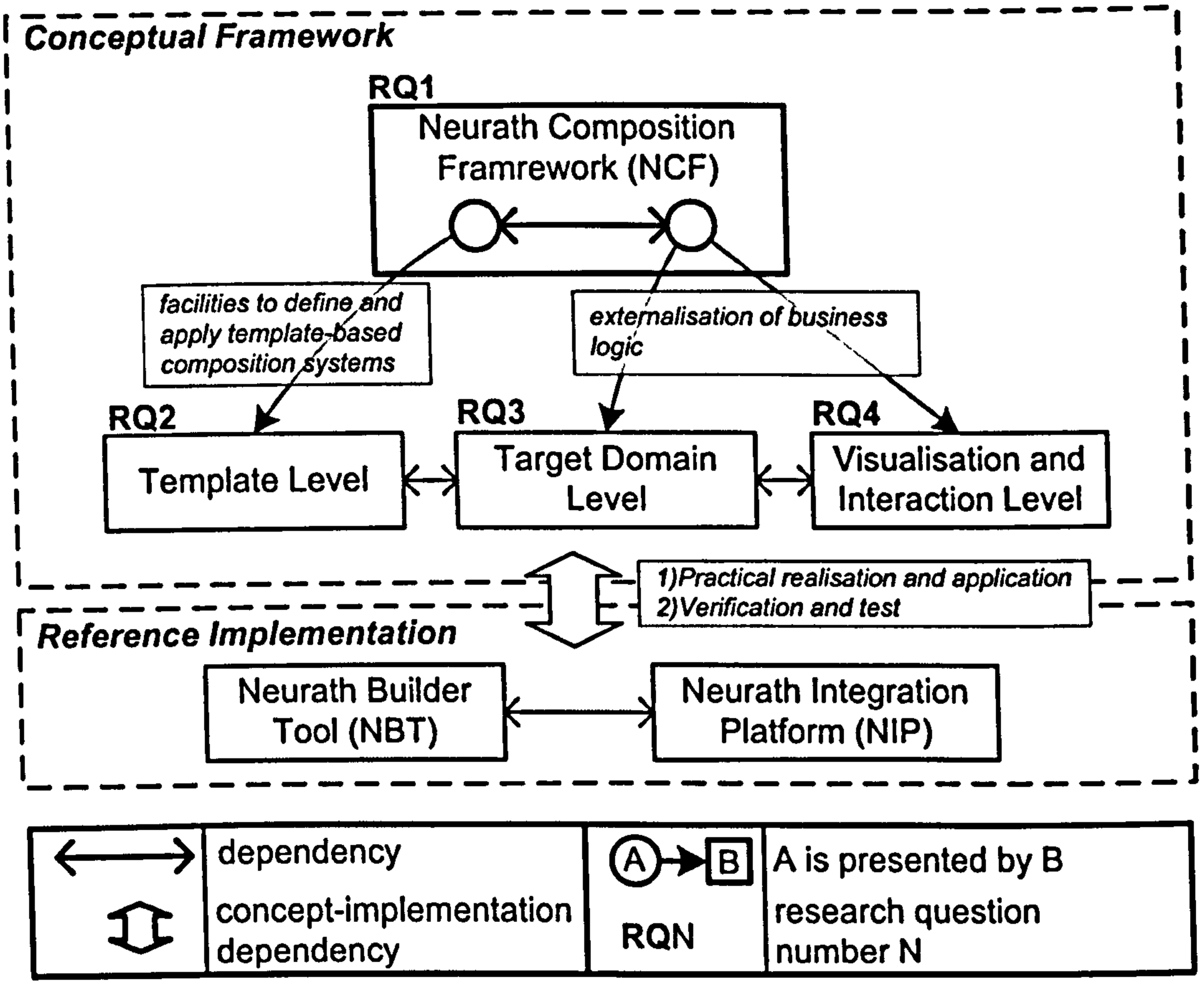


Figure 1.1: The basic model of the research

The research questions *RQ3* and *RQ4* are related to the externalisation of business logic in template-based composition systems up to the level of domain experts. At this stage the main objective is to provide a form of domain-specific representation of concepts defined at the Template Level of composition. To do this, we recognise two steps. The first step is a bi-directional translation between a terminology of a template composition system and a terminology of an application domain. Concepts, which are needed to perform the first step, are collected within the *Target Domain Level* of composition. The

second step is a presentation of results of the first step to domain experts as well as an interpretation of expert's actions that are applied to this representation. Concepts, which needed to perform the second step, are collected within the *Visualisation and Interaction Level* of composition.

In order to test and verify results of the investigation, the reference implementation of the NCF has been developed. It is important to admit that the reference implementation shows practical applicability of the NCF for the real-life problem, the development of a family of software systems for the house automation application domain. The object-oriented programming language Java has been chosen to program the reference implementation. It is required that a platform with a Java Runtime Environment (JRE) version 1.4 or higher be used to run the implementation. Basically, two tools were designed. Neurath Builder Tools (NBT) is a tool to define domain-specific visual composition systems. The production of the NBT is a subject of deployment into the Neurath Integratin Platform (NIP). The NIP is a tool where composition systems are used by domain experts to visually design software products. The reference implementation shall have a positive influence on the acceptance of the proposed framework. Figure 1.1 depicts the basic model of the research.

1.4 Original Contributions

The main contribution of this thesis is a NCF. It is a layered approach to compose software systems according to well-defined requirements which have been externalised, giving the ownership over the design to the end-user. The NCF defines a method to create template-based composition systems together with Domain-Specific Visual Interfaces on top of these systems. We have provided a reference implementation with help of which a practical applicability of the NCF was shown. More specifically, this thesis made the following original contributions:

A1: **A template-based Composition System:** We define a *component model* for components, called Parametric Code Templates (PCTs), that encapsulate program code templates. Additionally we present a *composition technique*, referred to as molecular operations, to compose those templates. Additionally, we introduce a simple *composition language*, called Parametric Code Template Composition Language (PCT-L), to define composition expressions. We collect all concepts to define and apply template-based composition systems under the *Template Level* of composi-

tion. Finally, we provide a reference implementation for the practical definition, application and testing of template-based composition systems.

A2: **A bridge between Composition System and Domain Experts:** We specify a strategy to map domain-specific languages onto template-based composition systems. The objective of such a mapping is to *give an ownership to domain experts over the template-based software composition process.* We collect all the concepts needed within the *Target Domain Level* of composition. Additionally, we provide a reference implementation to practically bridge the domain of template composition systems with different application domains.

A3: **An Interface for Visualisation and Interaction:** We present a strategy to improve interaction between domain experts and domain-specific languages that are products of the concepts defined at the Target Domain Level. We specify concepts to define domain-specific visual interfaces (DSVIs) on top of domain-specific languages. The objectives of DSVIs are the reduction of complexity by visualisation and an effective interaction with underlying DSLs through visual interfaces. The concepts to define DSVIs are collected within the *Visualisation and Interaction Level* of composition. Finally, we provide a reference implementation to define, apply and test DSVIs.

A4: **Concepts to define template-based composition languages and externalise them to be used by different domain experts:** we bound contribution items A1, A2 and A3 together in the Neurath Composition Framework. The objective of the NCF is *to give domain experts tools to perform complex program code manipulations via a domain-specific visual interface.* We give a reference implementation of NCF that consists of the NBT and the NIP.

Here, we would like to explicitly state that the contribution items A1, A2, A3 and A4 have been defined, developed and described by the author of this thesis. An exception is the ASLT [93], which is used for the definition of contribution A1 and is a collaborative work.

1.5 Organisation of the Thesis

The rest of this thesis is organised as follows:

Chapter 2 provides a broad overview on software composition systems. The notions of a component, a component model, a composition technique and a composition language are introduced. Further, the main requirements for software composition systems are specified. Existing approaches are analysed and divided into two groups. The group of black-box approaches is described in order to show what approaches are practically applied in industry. Afterwards, the grey-box composition approaches are presented to show the main current tendencies in software composition.

Chapter 3 gives an overview on the NCF. The architecture of the NCF is explained. We show basic organisation of the reference implementation for the NCF, as presented by the tools NBT and NIP.

Chapter 4 introduces a collaborative work, called Abstract Syntax Language Tree (ASLT). We explain a strategy to use the ASLT and introduce additional terms that extend the collaborative work, such as atoms of composition and atomic operations.

Chapter 5 presents concepts collected within the Template Level of composition. They describe how template-based composition systems are defined and used. Main functional blocks together with inputs and outputs are explained. The notion of templates is introduced. A component model for PCTs and a composition technique, represented by molecular operations, are presented. Additionally, a simple composition language PCT-L, to form composition expressions, is introduced.

Chapter 6 describes concepts collected within the Target Domain Level of composition. They describe the externalisation of business logic in template-based composition systems up to the level of different application domains. We explain the strategy to bridge a domain of template-based systems with different application domains with the help of domain ontologies and domain-specific languages. We present functional blocks that are defined at the Target Domain Level and show main inputs and outputs.

Chapter 7 presents a Visualisation and Interaction Level of composition. We explain how to extend domain-specific languages, defined by the underlying concepts from the Target Domain Level, with domain-specific visual interfaces (DSVIs). The way in which DSVIs are formed and mapped onto DSLs is explained. Additionally, we show what

kinds of interactions with DSVIs are possible and how these interactions are defined. Commonly, we present functional blocks of the Visualisation and Interaction Level and show main inputs and outputs. We explain a component model of visual components, called Neurath Modelling Components (NMCs), that constitute DSVIs.

In **Chapter 8**, we propose a reference implementation of the NCF. We show how all parts of NCF are implemented with the help of the object-oriented programming language Java and how they work together. Moreover, it covers the evaluation of the Neurath Composition Framework. With two use-cases, it shows how the framework is practically applied. Finally, the conclusions are made.

Chapter 9 summarises the work done in the thesis. It highlights the potential applicability of the contributions that have been made. Finally, the possible research and development directions that are based on the presented results are discussed.

Dependencies of chapters and some related annotations are illustrated in Figure 1.2.

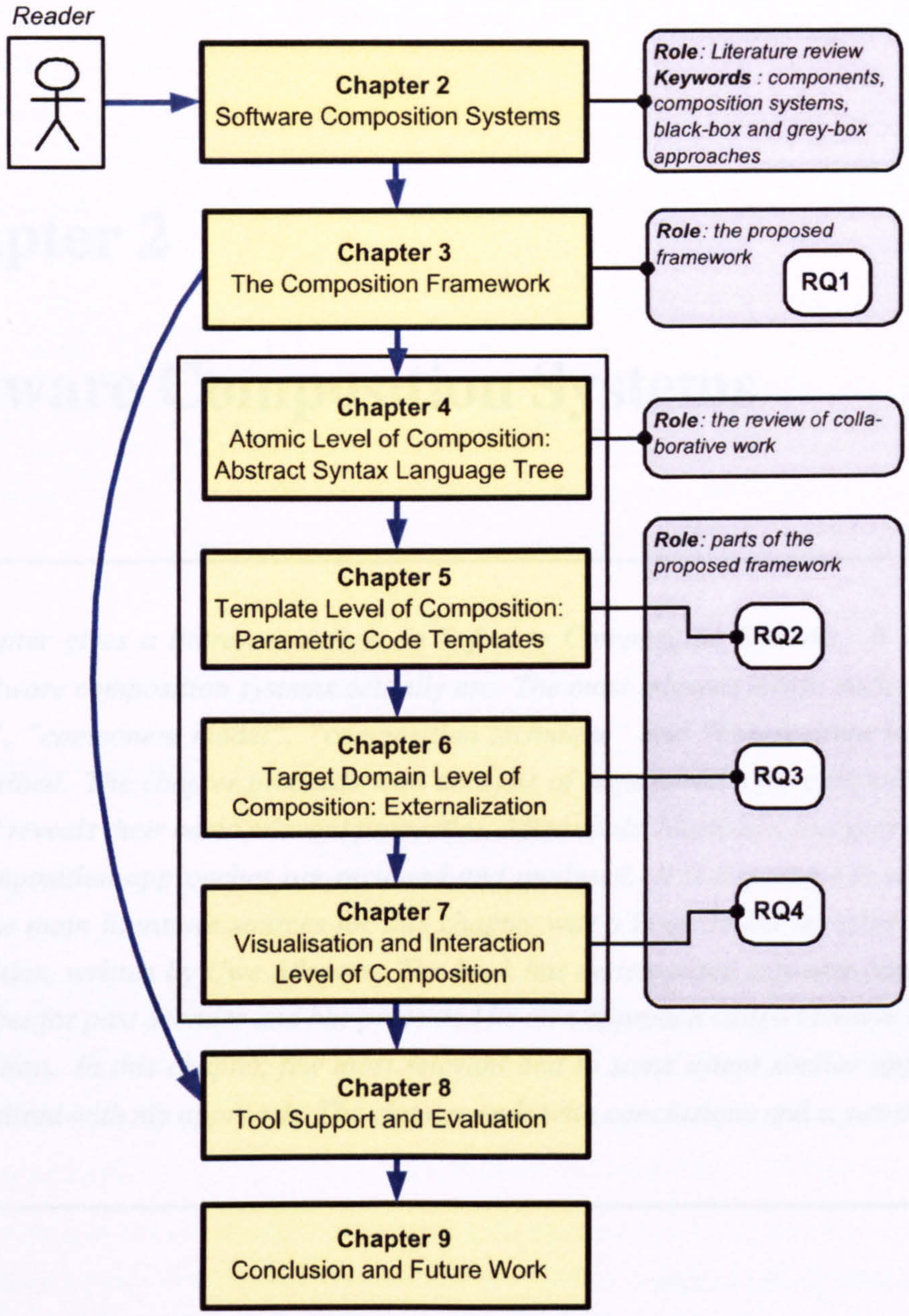


Figure 1.2: Reader's guide: chapter's dependencies

Chapter 2

Software Composition Systems

This chapter gives a literature review on Software Composition Systems. It discusses what software composition systems actually are. The most relevant terms, such as "components", "component model", "composition technique" and "composition language" are described. The chapter proceeds with analysis of requirements for composition systems and reveals their basic relevant properties. Afterwards, black-box and grey-box software composition approaches are reviewed and analysed. It is important to admit that one of the main literature sources for this chapter was a book called Invasive Software Composition, written by Uwe Aßmann. The book has systematised software composition approaches for past decades and has presented its own approach called Invasive Software Composition. In this chapter, few most relevant and to some extent similar approahces are compared with my approach. The chapter ends with conclusions and a summary.

2.1 Introduction

In the 1960s, software was not mass-produced. At that time, software construction was in the phase that assumed manufacturing of software by skilled individuals and by individual handcrafting. The need to overcome this phase of infancy was clearly stated by McIlroy [61]. The way to do this was through a mature technology that would provide parameterisation, assemble components in well-defined procedures and configure complete systems by pressing some buttons in a configuration tool [8]. McIlroy's ideas influenced and resulted in a large number of approaches for component technology presented by both research and industry. However, none of them have solved all of the problems.

Different fields of software engineering that concentrate on components underline the following questions that are still open:

- How can components be prepared for reusability [13]?
- How can they be adapted flexibly during composition?
- How can component-based software process be organised for large systems that consist of thousands of versions and variants?

In this thesis, we concentrate on growing sub-fields of software engineering such as *composition systems*. The term *composition* is used if a system is being built primarily from high-level components or building blocks [31]. Composition systems are systems that concentrate on the composition of components [8]. Further, in this chapter, we specify requirements to the composition systems, give a literature review on black-box and grey-box composition approaches and reveal the basic problems that have remained. The chapter ends with the conclusion and a summary.

2.2 Components

There are many existing definitions of components, some of which are given below. They focus on different aspects of software engineering. Components provide a service without regard to where the component is executing or its programming language [82] if:

- A component is an independent executable entity that can be made up of one or more executable objects;

- The component interface is published and all interactions are through the published interface.

Another definition is given in [26], where a component is defined as a reusable unit of deployment and composition that is accessed through an interface. D'Souza and Wills defined a component as a reusable part of software which is independently developed and can be combined with other components to build larger units [83].

Each component normally provides an interface to define and access the services within this component. From the other side, a component may specify required interfaces. Figure 2.1 shows a typical graphical representation of a component with provided and required interfaces.

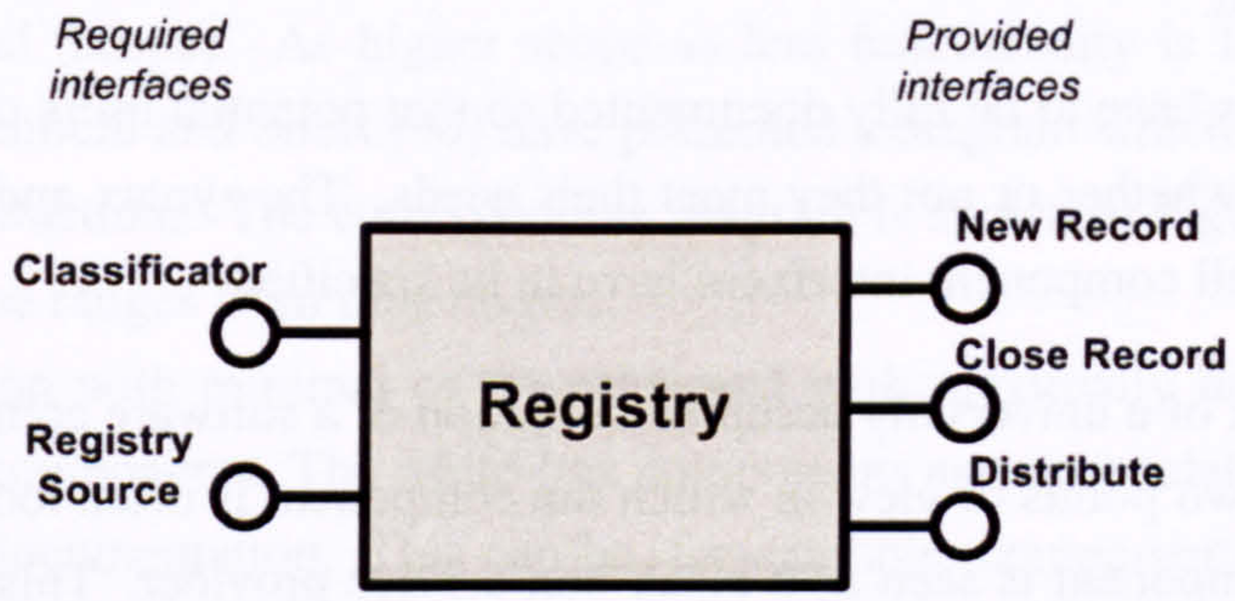


Figure 2.1: Example of a component

Although these definitions differ in some aspects, they all state that components are reusable units that provide functionalities via interfaces. Sommerville summarises the characteristics of components [82] as:

- Standardised.

  Component standardisation means that a component has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment.

- Independent.

  A component should be independent - it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.

- Composable.

  For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes.

- Deployable.

  To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed.

- Documented.

  Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.

There is a lack of a universally accepted definition of a software component [19]. We have recognised two points of view in which the component is often looked at. The first one is when a component is seen as a black-box service provider. This scope is related to the classical understanding of a component, first established with component-based development approaches, and described later in Section 2.6.3. The second point of view is broader, when a component is seen as a unit of composition. **In this thesis, we base it on a rather abstract definition of a component given by Aßmann in [8]:**

"A component is a software part that must be *composed* with other components in a *system composition* to form a final system."

Hence, a *software component* is simply a software item that is subject to software composition. It may appear on different levels of granularity, may be a design or implementation item, and may be in source or binary form.

There are defined many different standards for components. A component model is a definition of standards for component implementation, documentation and deployment. The component model specifies how interfaces should be defined and the elements that should be included in an interface definition. According to [58], a software component model should define:

- the syntax of components - how the components are constructed and represented;

- the semantics of components - what components are meant to be;
- the composition of components - how the components are composed or assembled.

Well-known component models include the EJB model [74], .COM model [18] and CORBA model [79, 30].

2.3 Black-box and Grey-box Components

Descriptions of components can have different levels of abstraction. The higher the level of abstraction of a component, the wider the range of the component's application. Elements of composition can be classified by the abstraction level versus the implemented functionality level (scope). As higher scope as less functionality is implemented by a component. Greenfield and Short [40] have presented a diagram which shows scope versus value for abstractions. The corresponding diagram is shown in Figure 2.2. The scale of scope and value ranges from none to one.

The abstraction with minimal or no value and with maximally broad scope represents white-box components. The white-box components are more related to the problem specification or documentation. This can be, for example, components presented by its specification in the form of source code. White-box components are completely opened for transformations by programmers or end-users.

The abstraction with maximal value and with maximally specific scope represents black-box components. The black-box components are directly related to the implementation or a solution. They are characterised by their functions and interfaces. Typically, black-box components are closed for structural transformations. An example is the binary form of JavaBean components [63].

Components that fall between those two are grey-box elements. As opposed to back-box components, grey-box components are opened for structural and therefore behavioural transformations. Compared to white-box components, grey-box components restrict allowed transformations. Examples of grey-boxes include design patterns and code fragments.

Sometimes it is spoken about glass-box components. These are white-box components that are closed for transformations. Figure 2.3 shows a schematic representation of black-box, white-box, grey-box and glass-box components.

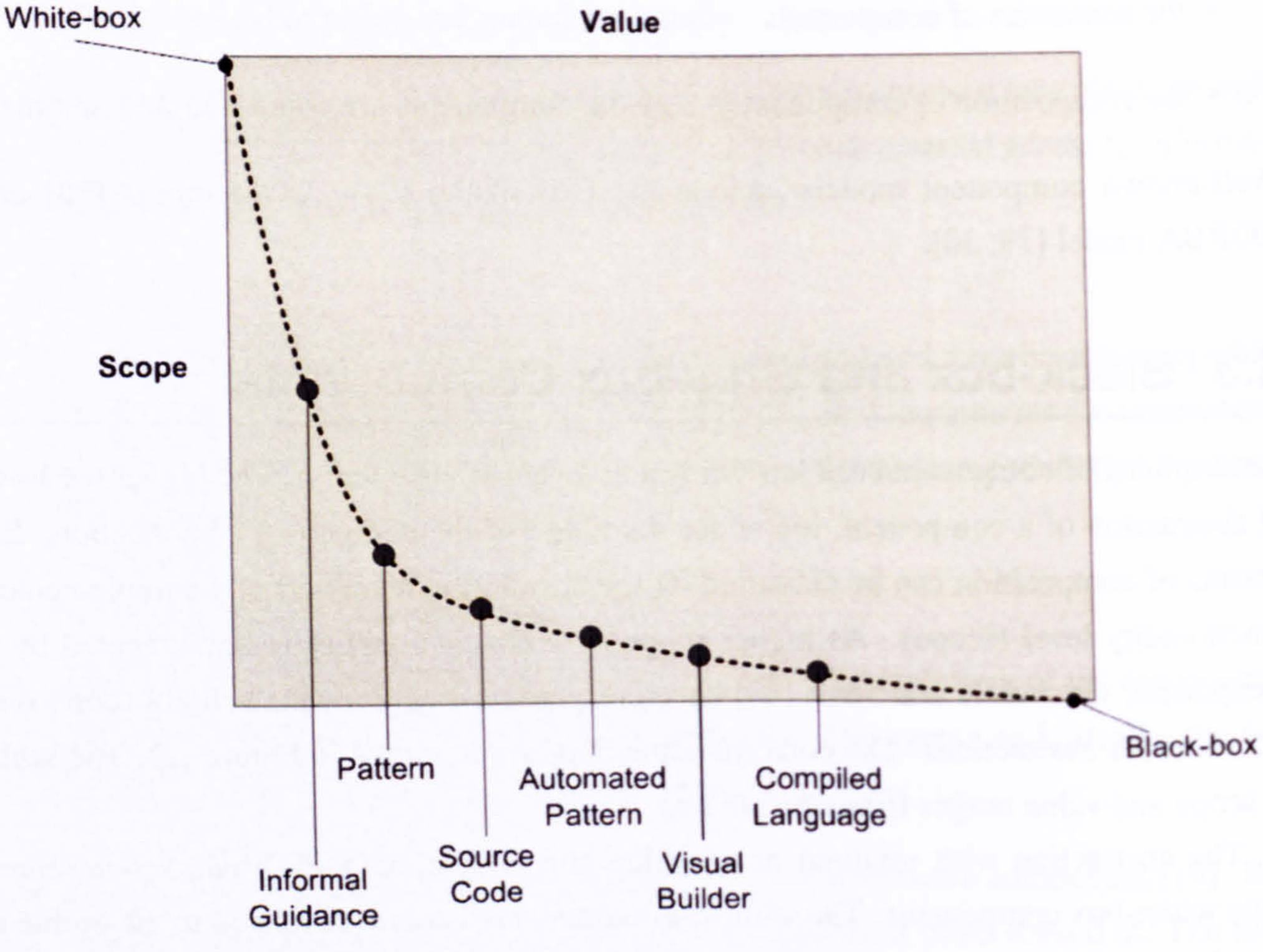


Figure 2.2: Scope versus value for abstractions [40]

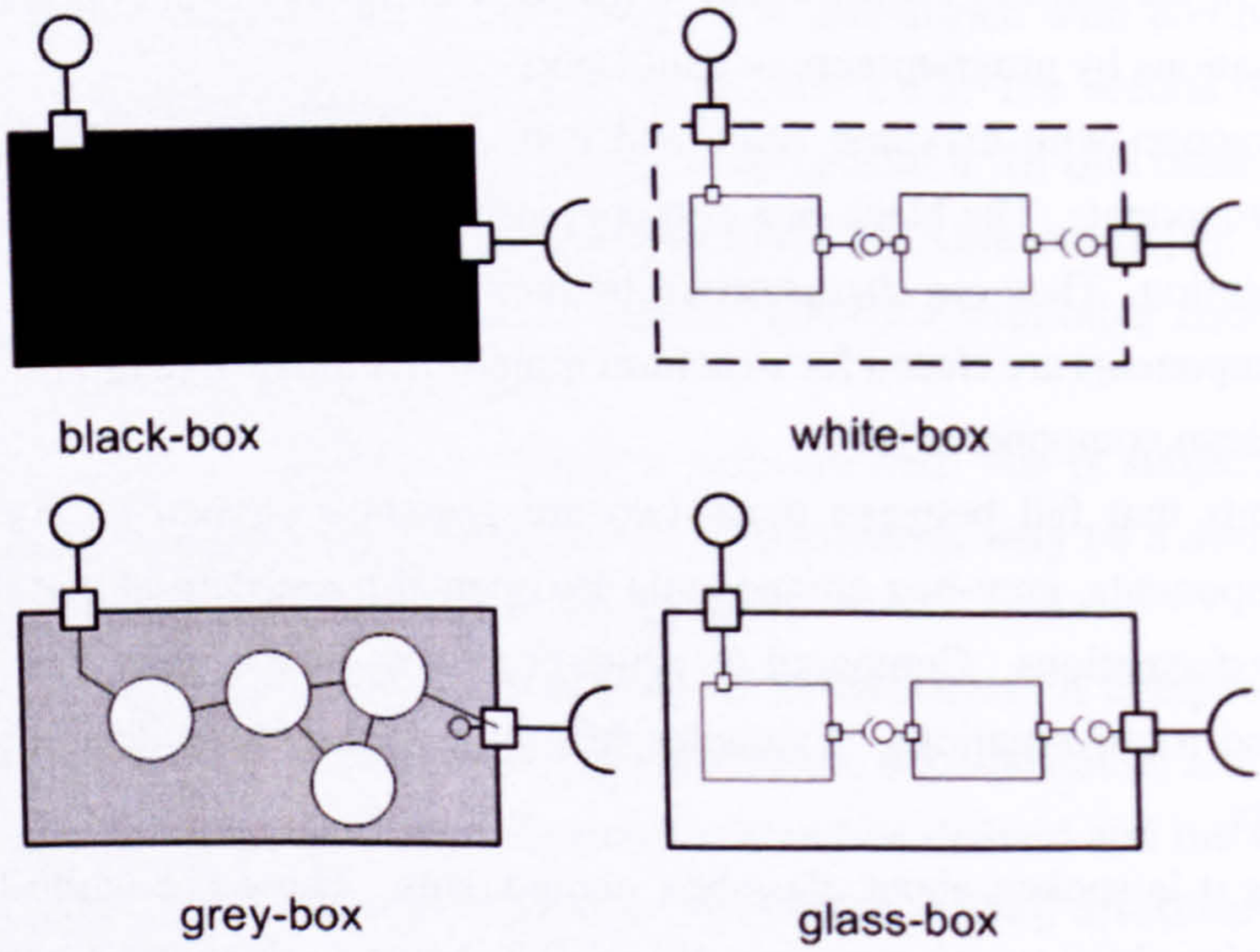


Figure 2.3: Different kinds of components

In this thesis, we classify components as a black-box type or grey-box type, as according to the visibility of their implementation. We refer to black-box components when the component is presented with its binary form and a program code of that component is not documented and therefore non-visible. The components that have program code described and are therefore visible are referred to as grey-boxes.

2.4 Composition Systems

Composition systems are systems that concentrate on the composition of components. They generalise many of the approaches to the component-based engineering we have seen in the last 40 years. Aßmann [8] presented the tower of component systems (Figure 2.4).

This figure shows different composition systems. In the middle of the figure there is a separation line that denotes two areas. The bottom area unites software composition approaches that are oriented on the *black-box* component paradigm. The top area unites software composition approaches that are oriented on the *grey-box* component paradigm.

Black-box components typically describe *what* can be done, rather than *how* it can be done. They represent the pieces of functionality that can be combined. Black-box components can not be directly modified by a programmer. Traditional composition approaches, presented within the bottom area of the figure, are characterised by applying black-box components. Compared to black-box components, grey-box components are opened for structural changes. Grey-box components introduce a higher level of parameterisation and therefore wider adaptability and reuse. Some traditional and many state of the art software composition approaches, presented within the top area of the figure, are characterised by applying a grey-box component paradigm.

According to Aßmann, all software composition systems can be compared in terms of three major aspects [8] which form a composition system (Figure 2.5) of a:

1. *Component model* that answers the question, "How do components appear and when can they be exchanged?".

2. *Composition technique* that answers the question, "How are components composed?". It offers a wide range of composition operators.

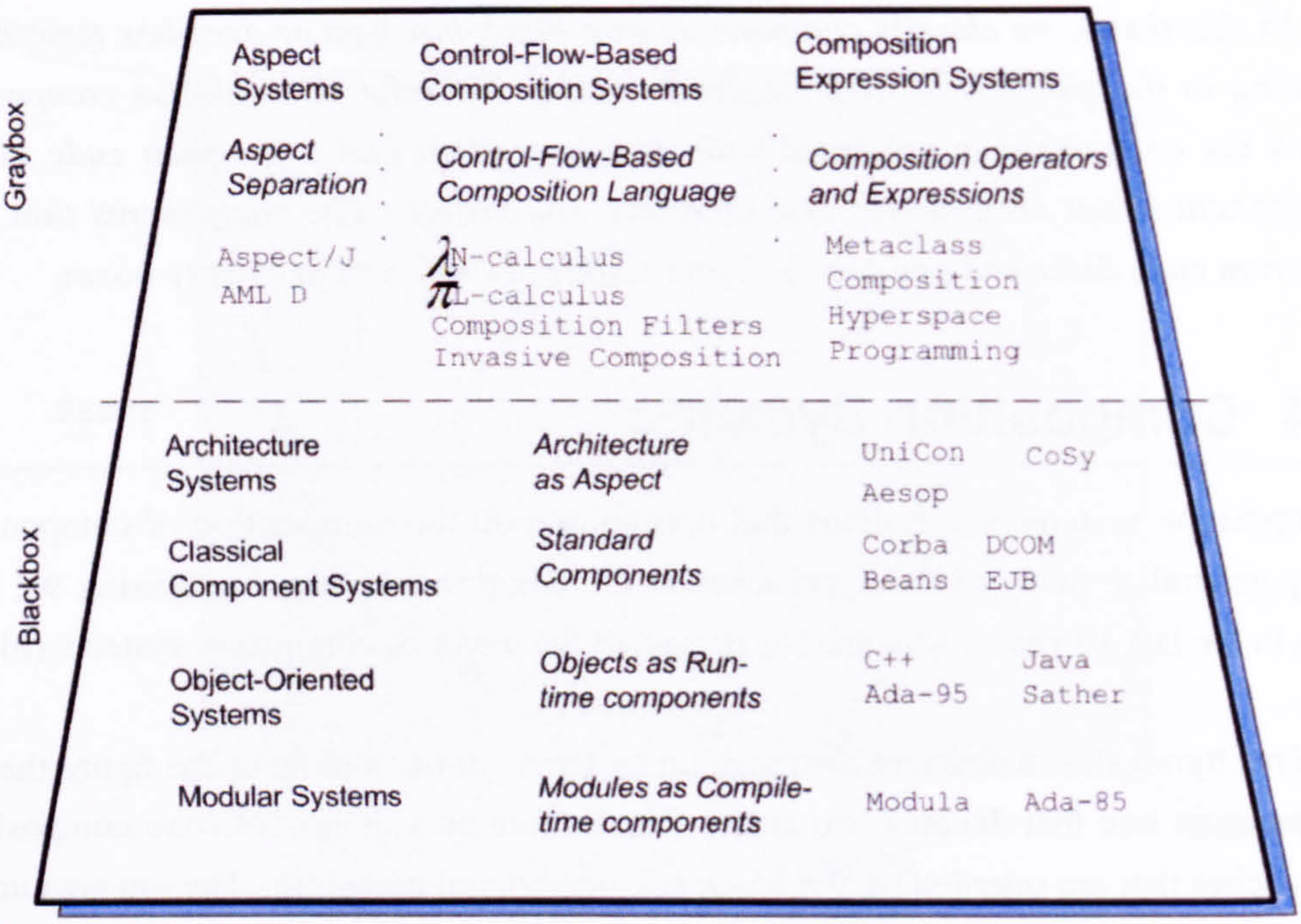


Figure 2.4: The historical and conceptual tower of component systems. Some of them are full-fledged composition systems. The central technical concept is denoted by an italic shape. Examples are denoted by typewriter font. [8]

3. *Composition language* that is needed to write composition recipes (composition specifications). These specifications describe how systems should be built from components and contain information about their architecture.

2.5 Requirements for Composition Systems

Composition systems typically consist of three parts: a component model, a composition technique and a composition language. Requirements for composition systems include a summary of requirements for each of these parts.

A component model is a definition of standards for component implementation, documentation and deployment. The main requirements for a component model are related to terms such as *modularity*, *parameterisation* and *conformance to standards*. Modularity is the property of components that measures the extent to which they have been composed out of separate parts (modules). To reduce costs and shorten time to market, it is required that components be reused. Figure 2.6 depicts reuse for two different software systems.

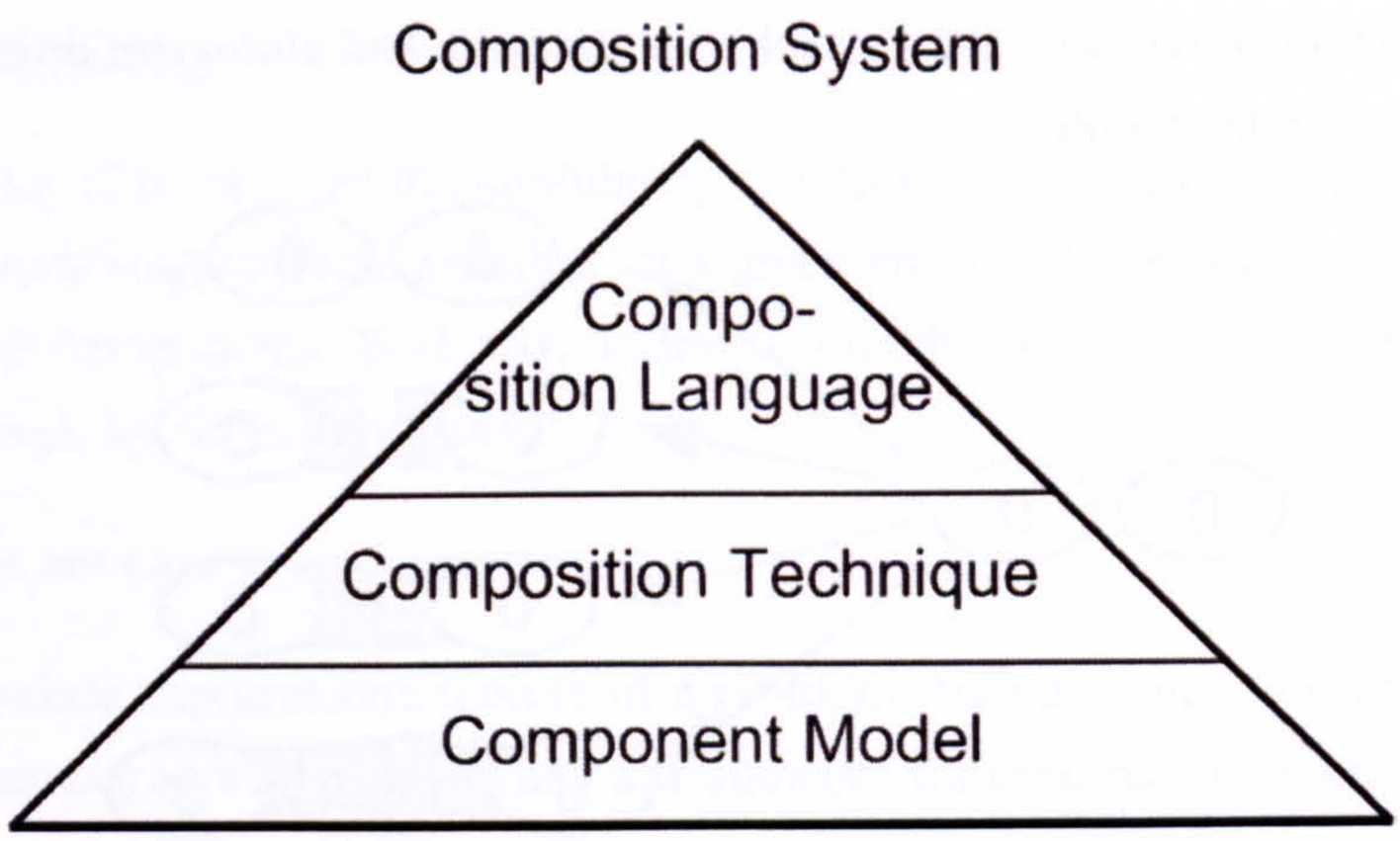


Figure 2.5: Parts of a composition system

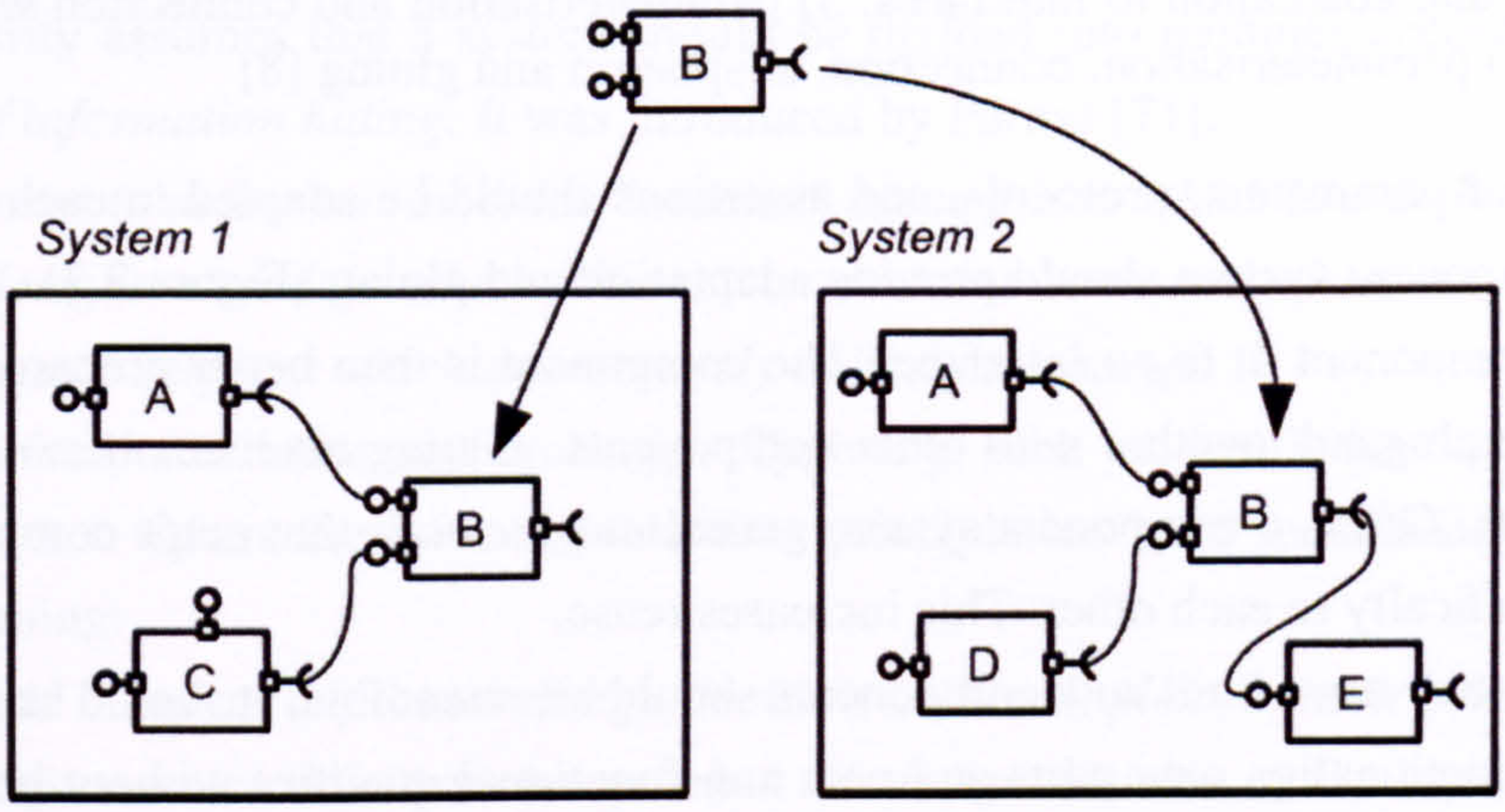


Figure 2.6: Reusing a component B in different software systems

An interface defines a communication boundary between two components. How a component will be reusable in different systems depends on the answer to the question, "What should the interfaces of a module look like?". Currently, parameterisation mechanisms are restricted to type parameters in languages with generic classes. The question, "Can we generalise these?" arises [8]. Components should conform to standards. They guarantee a better fit of components during composition and lower the learning curve for component-based development.

A composition technique covers the questions related to the manipulation with components and answers the question, "How are components composed?". Aßmann [8] relates to the main requirements of composition technique with terms such as *connection*, *exten-*

sibility and *aspect separation*. Terms such as *adaptation* and *gluing* are directly related to the component's connection.

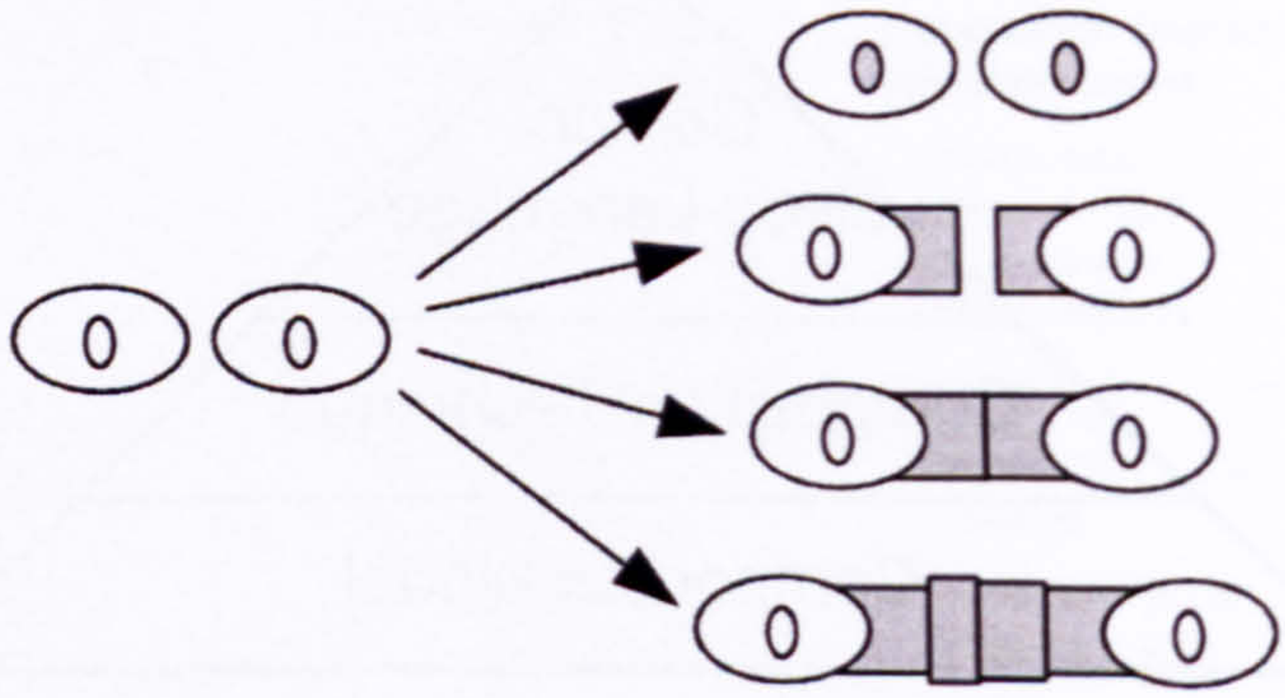


Figure 2.7: Tasks during composition from top to bottom: 1) parameterisation, 2) parameterisation and adaptation to interfaces, 3) parameterisation and connection with adaptation, and 4) parameterisation, connection, adaptation and gluing [8]

Types of parameters, protocols, and assertions should be adapted to each other. For this, a component system should provide adaptation and gluing (Figure 2.7). Adaptation makes a component fit to an interface. The component is then better prepared for reuse and can be plugged together with other components. Gluing mediates between specific components. Often, a component system generates *glue code* that maps component protocols specifically to each other. This increases reuse.

Software systems built with components should be extendible. It should be possible to add new functionality, new parts and new non-functional qualities without hand-editing old parts of the system. One way to reach higher parameterisation and extendibility is to not treat components as black-boxes. With the help of aspect separation, different functional and non-functional aspects should be distinguished.

A composition language is needed to provide explicit specifications or recipes on how a system or its part should be composed with components. Aßmann [8] relates to the main requirements on composition language with the following terms. *Product-consistency support* assumes that a composition language should help to ensure quality features of software systems. *Software-process support* assumes that the language should support the composition-based software construction process. Additionally, the language should be expressive enough to express variants and versions of product lines, and should be powerful enough to describe large systems. Additionally, the language should be easy to understand. Finally, the *meta-composition support* assumes that the composition recipes can grow with the system and the language itself should be based on composition.

2.5.1 Modularity

Budgen [20] has characterised the modularity as follows. The use of an appropriate form of modular structuring makes it possible for a given problem to be considered in terms of a set of smaller components. Basically, finding a suitable strategy to modularise a system provides, at least, the following benefits:

- modules are easy to replace;
- each module captures one feature of a problem, aiding comprehension (and hence maintenance) as well as providing a framework for description as a team;
- a well-structured module can easily be used for another problem.

Modularity assumes that a system should be divided into modules according to the *principle of information hiding*. It was introduced by Parnas [71]:
We can attempt to define our modules "around" assumptions which are likely to change. One then designs a module which "hides" or contains each one. Such modules have rather abstract interfaces which are relatively unlikely to change.
Modularity facilitates reuse and exchange. Typically, each module hides and maintains a module secret. Often, the information-hiding principle has been specialised as the following meaning:
Every module hides an important design decision and its implementation behind a well-defined interface which does not change when the design decision or the implementation changes.
The principle assumes that changes in implementation of deployed modules are not visible to the users of those modules. In this way, modules can be exchanged for variants with the same interface, so that product families can easily be built.

Aßmann proposed two basic requirements for the modularity of components:

1. *Information hiding*. A component has to maintain several secrets, such as implementation language, location, and uptime.

From Parnas' statement, two major problem areas for component models can be identified. The first question is *which* secrets a module hides. Since future software systems will be completely heterogeneous, dynamic, and distributed, information hiding should cover many other secrets that Parnas could not envision. For instance, the implementation language of a component must be secret so that components can

be programmed in the language that is most apt. Also, the location of a component should be hidden to deployers; access to remote components should be automatic. Lastly, the uptime of a component should be hidden. Some components should be active 24 hours per day, while others need not live permanently.

2. *Semantic Substitutability.* A component or composition system should offer a specification mechanism for component semantics so that it can be checked whether components can safely substitute another one. In most programming languages, module systems only guarantee *syntactic substitutability*, i.e. correct substitution concerning the typing of module interfaces.

Modularity is the most basic feature a component system has to support. The more secrets a component preserves and the more precisely its semantics are described, the better it can be exchanged.

2.5.2 Parameterisation

A parameterisation of components improves their adaptability and reuse. Aßmann distinguishes between static and dynamic parameterisation. In case of dynamic parameterisation, the principle of delegation is used. A component that delegates certain functionality to another object (delegatee) can be parameterised by exchanging the delegatee. Static parameterisation works at a compile-time for component classes and is represented by generic classes and types, e.g. generics in Java programming language [94, 47].

Beyond types, it would be convenient if components could also be parameterised by other program elements, such as declarations or statements. Components can be parameterised with code fragments. It is a known practice applied in industry. Such a form of a parameterisation can be implemented with the `#ifdef#` construct of the C preprocessor [54, 84]. This construct operates with compile-time variables that define the code fragments compiled.

Often it is required to configure components dynamically during runtime. For example, changing a graphical user interface (GUI). This requires certain configuration routines of GUI which can be performed through properties, i.e. their attribute values of GUI components.

Aßmann specified three requirements regarding the parameterisation of components:

1. *Generic Type Parameters.* Component systems should offer generic components. In the simplest case, these are *generic types* or *generic classes*. In more advanced cases, *nested generic parameters* should be supported.

2. *Parameterisation of Program Elements.* Beyond types, arbitrary program elements of a component can be generic. Such parameterisations should be checked according to the static semantics of the underlying programming language.

3. *Property Parameterisation.* At runtime, components should be parameterised by property values.

If a component model supports generic types, generic program elements, and property adaptation, both compile-time and runtime components can be adapted to reuse contexts flexibly. Hence, parameterisation is one of the major requirements for improving reuse and exchange.

2.5.3 Connection

Components are normally connected by means of their connection points that reveal how a component can be interconnected with its reuse context, which items are communicated to and from the environment, and which relations to the outer world must be established. Connection points are defined by the interfaces of components. Components can not always be connected directly. Often, certain adaptation mechanisms are required to build up a bridge between connection points. Often, it is necessary to select a sequence of adapters, which transforms component protocols step-by-step. This process is called *gluing* of components.

Aßmann specified two requirements regarding the connection of components:

1. *Automatic Component Adaptation.* Components must be adapted towards interfaces, in order to improve their reuse. *Adaptation* makes a component fit to an interface. This interface might be a standardised interface of the component system.

2. *Automatic Gluing.* A composition system should generate or interpret glue code, a sequence of adapters or mediators to map the protocols of components to each other.

2.5.4 Extensibility

Traditionally, software systems are built with black-box components. In such systems, certain adaptability is reached with the genericity feature of components. Due to information hiding, deployers of components are not able to investigate their internal structure. When end-users would like to add new functionality into the system, it is only possible by direct intrusion of the component manufacturers.

Problems with extension fall into two categories - the *fragile base-class problem* [87, 35, 42] and *the lack of use-based extensions* [8]. The fragile base-class problem is a fundamental architectural problem of object-oriented systems where super-classes (or base classes) are considered "fragile" because seemingly safe modifications to a base class - when inherited by the derived class - may cause the derived classes to malfunction. The use-based extensions assume a multiple definition of a component's use. For example, in C++ programming language, it is possible to extend a class in files other than the one in which the class had been defined. The lack in use-based extension results in difficulties in extending the system.

Views can be seen as a solution for both problems. Views permit the definition of several specifications of a component, e.g. to extend components with new functionality. View-based systems already exist [69, 37, 39]. View-based extensions help to avoid so-called object schizophrenia, when, with the help of internal delegation, a logical object is split into two physical ones. Those objects can shadow one another, resulting in failed calls [87].

Aßmann specified few requirements regarding the extensibility of components:

1. *Base Class Extension.* The base classes should be extended without recompiling their sub-classes.

   (a) *Generated Factories.* Components should be created by factories in such a manner so that they can be built in consistent component families. Additionally, these factories should be generated so that extensions are automatically incorporated.

   (b) *Generated Access Layers.* Properties of components should be accessed by layers that can be exchanged and regenerated when a system is extended.

2. *General View/Use-Based Extension.* A component should be extensible by specifying a new view on the component.

3. *Integrated Extensions.* To avoid delegation, it should be possible to integrate extensions into objects.

2.5.5 Aspect Separation

Part of the program that cross cuts its core concern is referred to as "aspect" [55]. Aspect specification unites specification items that define behaviours cluttered over the components (Figure 2.8). Normally, when non-united, those specification items have a negative impact on the software quality in terms of comprehensibility, adaptability and evolvability.

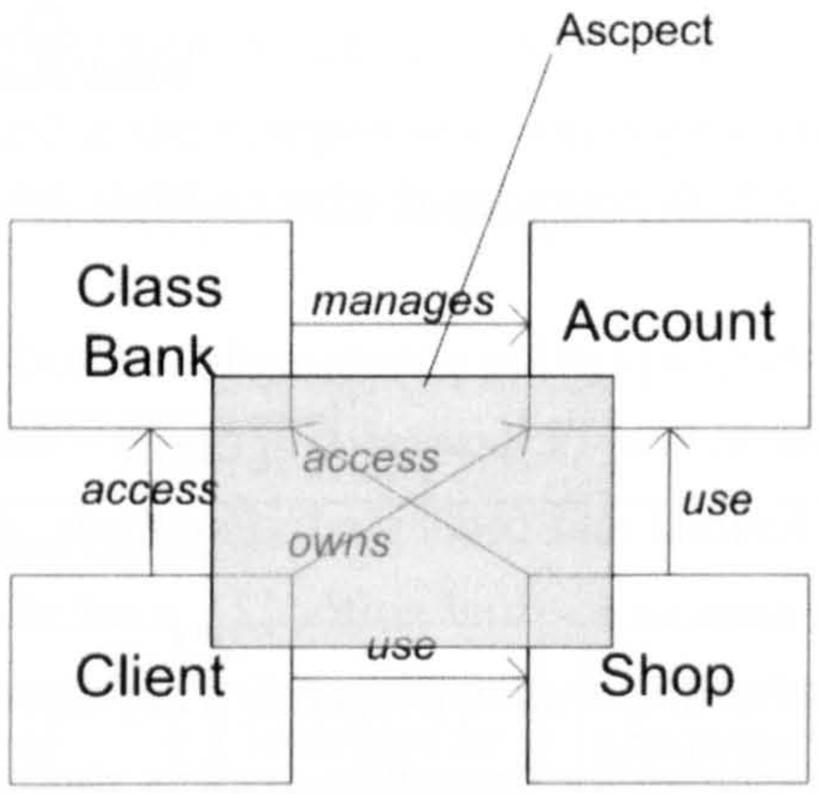


Figure 2.8: Behaviours cluttered over the components encapsulated by an aspect

A restricted form of aspect separation is realised when components can be accessed under different interfaces and each of the interfaces represents an aspect (Figure 2.9).

Aßmann specified two requirements regarding aspect separation for components:

1. *Aspect Weaving.* A composition system should support separation and composition of aspects.

2. *Multiple Interfaces.* A component may have several interfaces under which it can be accessed. Each interface covers one aspect.

2.5.6 Linguistic Support

Composition system requires a language. With this language the composition can be described. Actually, any language could be taken as an example, like a classical one such as a CSP - a simple language for describing parallel computations and their interactions.

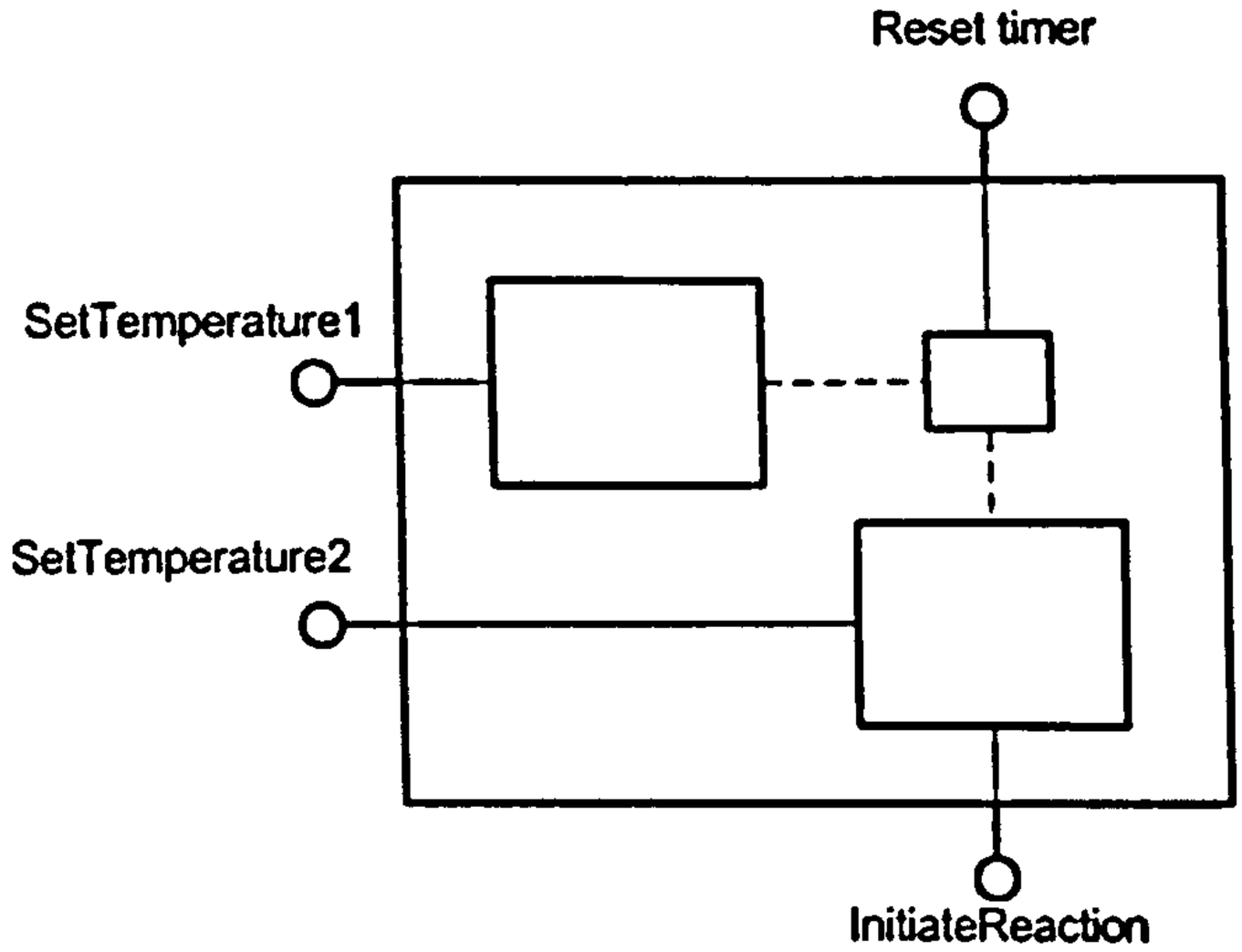


Figure 2.9: A component with multiple interfaces

A composition system should provide a systematic method for constructing large software systems. This can be in one or several languages. With those, the composition phrases or recipes can be formed and processed afterwards. Such recipes can be script [77], a set of editing sequences in a visual editor [2], a set of rules [34], a program in a functional language [24], or an imperative program. They should be able to automate the composition process, to be expressive enough to express variants and versions of product lines, and to be powerful enough to describe large systems. Additionally, the composition language should be easy to learn and easy to understand.

Aßmann specified a requirement for product consistency: "A composition recipe should be able to ensure consistency of the composed system. Hence, a composition language should adhere to formal features that can be checked."

Additionally, the formal features of the recipe can be used to ensure quality features of the software process. This requirement is different from the requirement for product-consistency, since it ensures the quality of the process, as opposed to the quality of the product.

Aßmann specified a requirement for software-process support: "A composition language should support the software-construction process naturally. Build management should be supported, both for eager and lazy builds. Additionally, formal features of the recipe should be used to ensure quality features of the composition-based software process.

As the composition recipes can grow with the systems, recipes themselves should be composed. Composition recipes should be available as components, and a composition language to compose these components should exist. Clearly, a composition-recipe component describes the architecture of a subsystem. If it can be composed to larger recipes with the help of composition, reuse of architectures is significantly enhanced."

Aßmann specified a requirement for meta-composition: "Composition recipes should themselves be constructed in a component- and composition-based way. This requires that a meta-composition language is available, and this may even be the same as the composition language."

Composition recipes and the composition language should be composition-oriented. Then, the concepts of the component model, composition technique, and composition language can also be introduced at the composition language level. This inherits all features of the composition level to the composition language. This is shown in Figure 2.10, taken from [8].

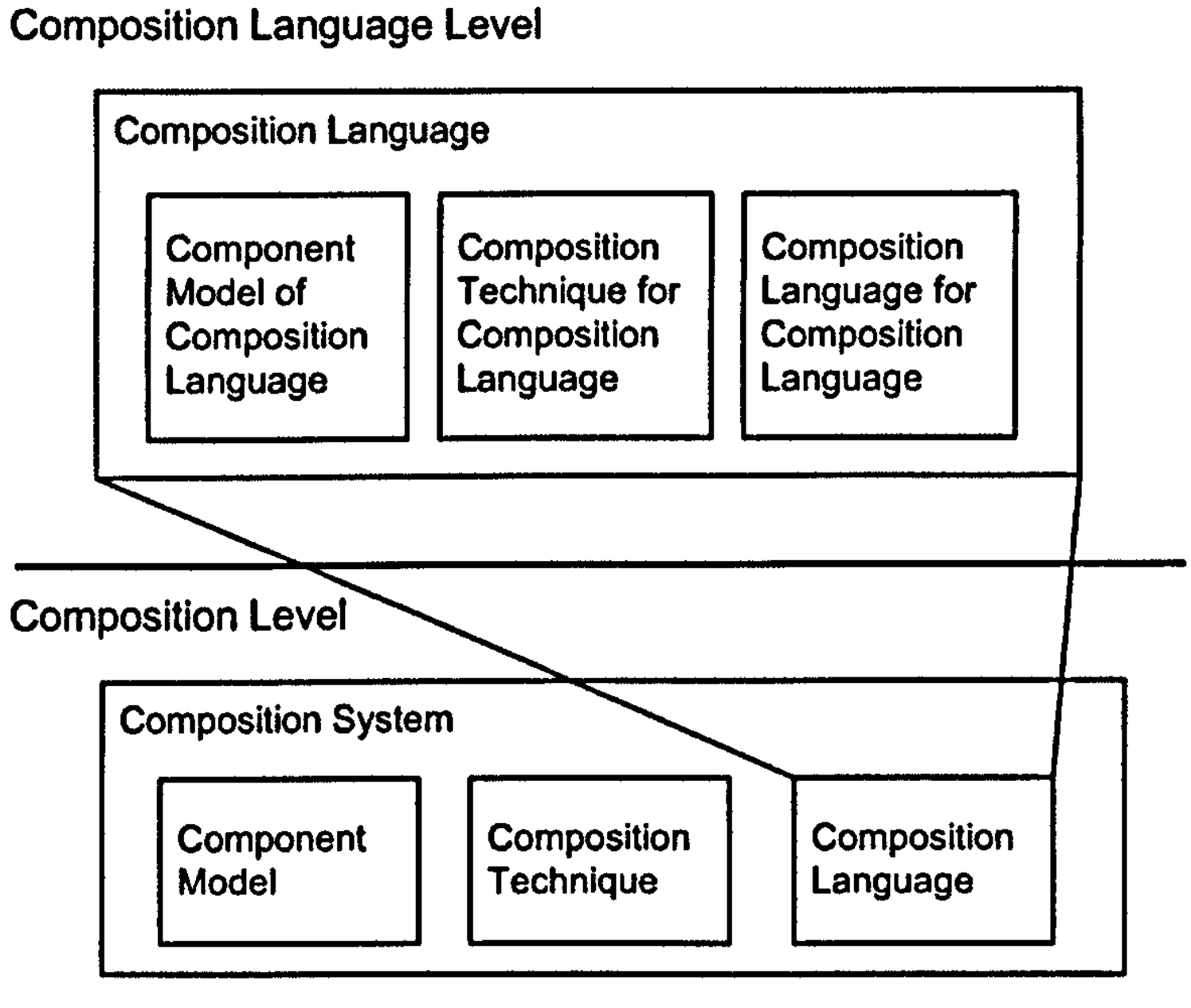


Figure 2.10: Composition-orientation of the Composition Language [8]

2.6 Black-box Approaches

Black-box composition approaches [95] have already been successfully applied in industry for many years. For the review, we have chosen the following approaches: modular composition, object-oriented composition, component-based composition and architecture composition. We start from with the oldest one - modular composition - and finish with the relatively young architecture-based approach. Each approach introduced new concepts that improved the quality of the process of composition and of the resulting products.

2.6.1 Modular Composition

Modules as black-box elements of software composition were mentioned in the 70s of the previous century. Parnas and Dennis were one of the initial contributors to the topic. Parnas et al. [71] discussed the modularisation as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. According to [72]:

A module is a work assignment for a programmer or programmer team. Each module consists of a group of closely related programs.

Modules introduced encapsulation - combining data and behaviour in one package and hiding the implementation of the data from the user of the object [47]. Coding techniques in the area of modular programming results in such major advancements as (1) low coupling and (2) replacement. Low coupling is expressed by the fact that a module can be written with little knowledge of the code in another module. Modularisation allowed for more flexible replacement and reassembling of modules without the reassembling the whole system.

Some examples of the modularisation can be found in [71]. In the 80s, software composition was mainly based on the modular programming. Such languages as Modula-2 [91], Pascal [52], or Ada-83 [49] have been developed.

Technically, a module is a unit of source code that can define constants, data types, variables, functions and procedures. A module can be compiled independently of other code. With an export declaration, a module can allow access to its internal entities by gaining access to another modules. In this case, we say the module exports some def-

initions. Respectively, modules that access those definitions import them. Imports are defined in the import declaration part within a module. The implementation part within the module defines constants, data types, variables and concrete functionality. Figure 2.11 depicts modules with their typical internal structure and an example of how they can be interconnected.

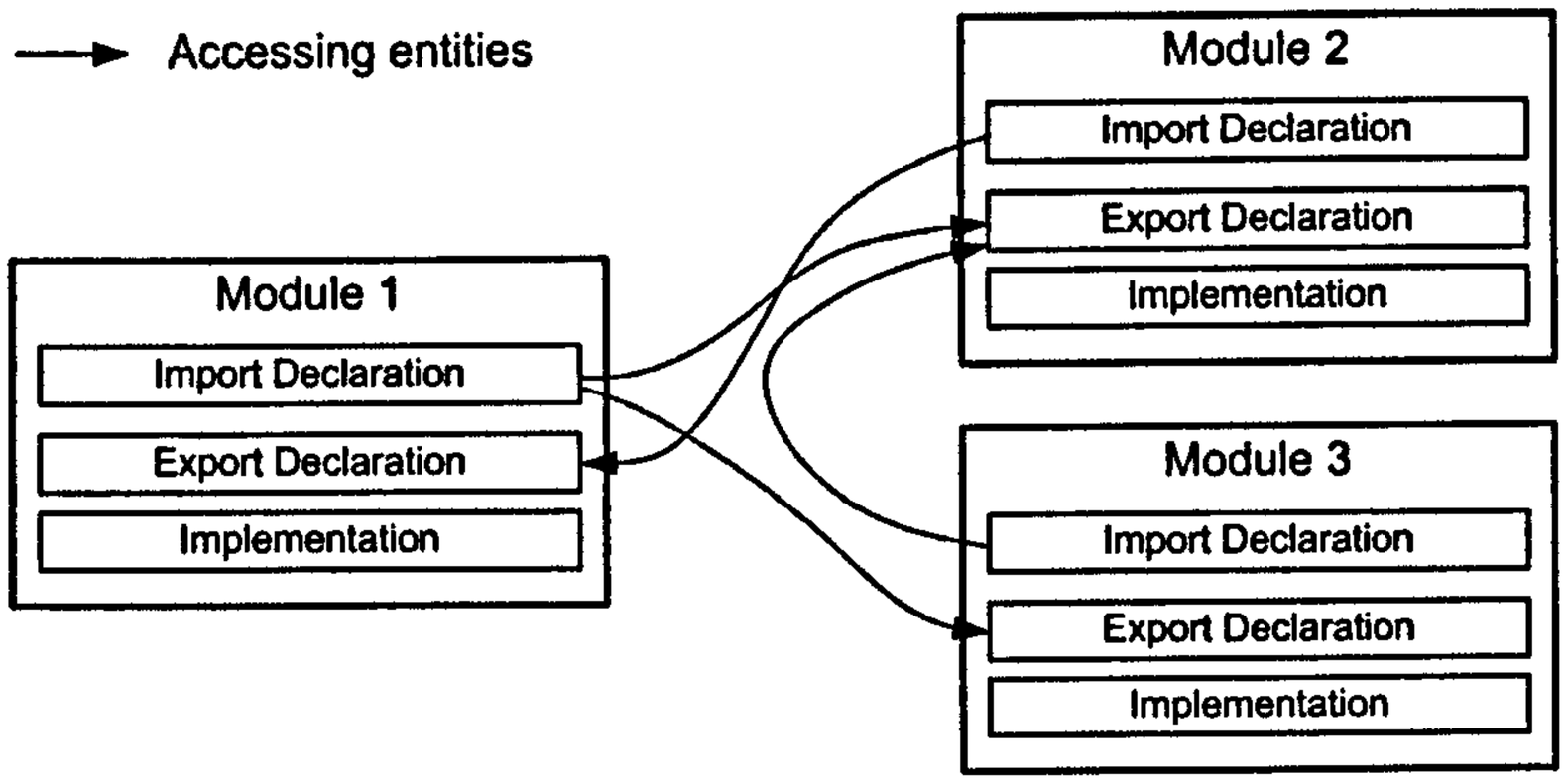


Figure 2.11: Syntactic structure of a module and an example of an interconnection

Modular systems can be flexibly recombined during design time. However, during run time, the modular system remains static, as it is not possible to create and link new modules dynamically. This limitation was removed on the next step in the software composition development. New object-oriented way of thinking and composing software was presented.

2.6.2 Object-Oriented Composition

Object-oriented software composition was introduced in order to simplify the development of software systems and to make it possible to change the system dynamically. Mainly, objects were introduced as elements of software composition during the development of Simula 67 [28] programming language, which was influential in the development of later object-oriented software composition models. The modular concept was enhanced and new elements of composition, so-called objects, were defined. This was based on the need to have better system descriptions and implementation techniques. In the object-oriented composition, important new terms were defined. These are *class*, *object*, *polymorphism* and *inheritance*.

In [17], Booch gives a base to define an object as a thing which has a behaviour, state and identity. Objects are more like real things. They are characterised by identity, states and behaviour. [85] gives a definition of these terms as follows.

The state of the object is all the data which is currently encapsulated. The behaviour is the way an object acts and reacts in terms of its state changes and message passing. It is exposed through methods within objects. Objects have an identity; they are not defined just by the current values of its data (attributes). An object has a continuing existence. The identity is a distinction criterion between different objects.

Consider the following example, taken from [86] and depicted in Figure 2.12. Figure 2.12 shows (a) a structure of software object and (b) a bicycle modelled as software object.

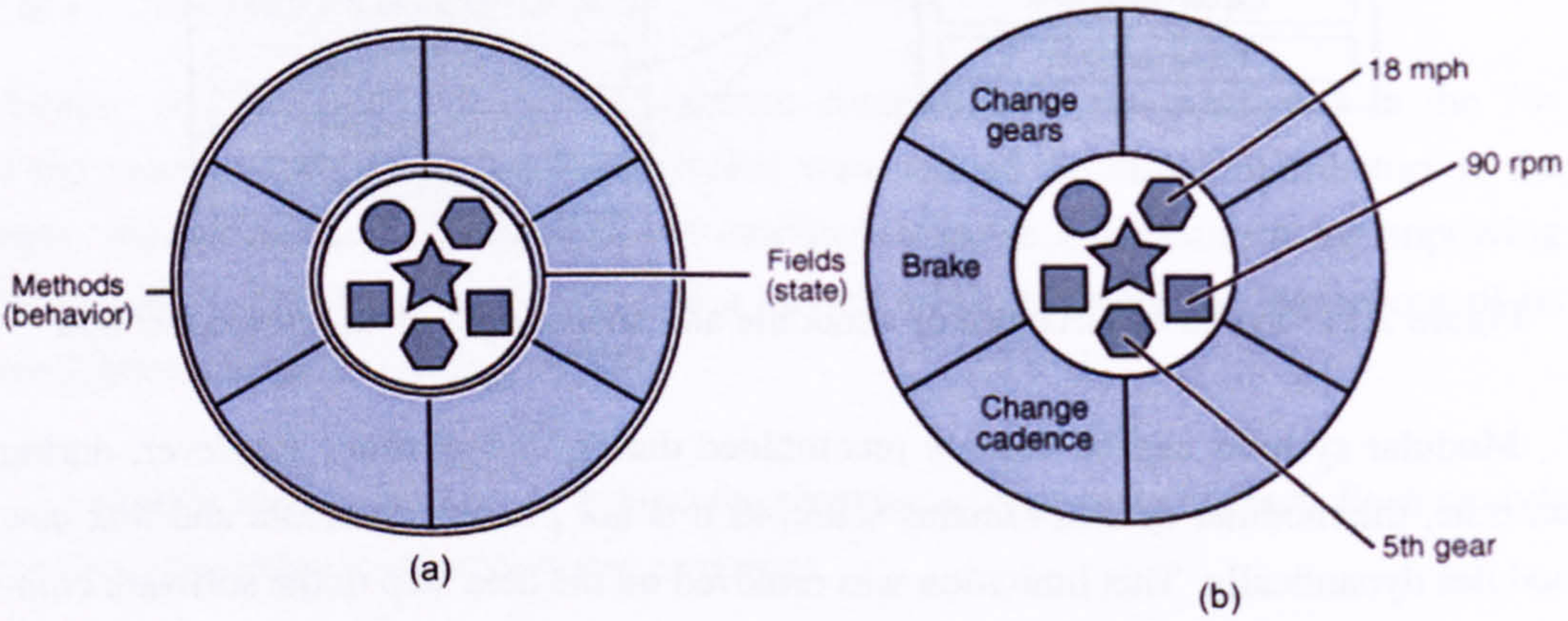


Figure 2.12: (a) - a structure of software object (b) - a bicycle modelled as software object [86]

In contrast to modules, it is easier to describe real world problems with objects. Aßmann [8] underlined the main difference of objects to modules:

"Object-oriented systems ... add support for runtime components, extension, and scalable composition. ...

In contrast to modular systems, in object-oriented systems, modules (i.e. classes) may be instantiated at runtime to objects.

...

The basic composition model for objects is the binding of calls to callee methods at runtime (polymorphism). Since objects can be exchanged dynamically with polymorphism, architectures that change dynamically can be built. This is the major improvement over modular systems."

Object-oriented software composition introduces classes, inheritance and polymorphism. Generally, classes are specifications of a set of objects with similar behaviour. A class can be written once and used later to instantiate an object. Figure 2.13 illustrates the relation between class and objects instantiated from the class. Objects are created according to the class specification and interact with some other objects that constitute a software system.

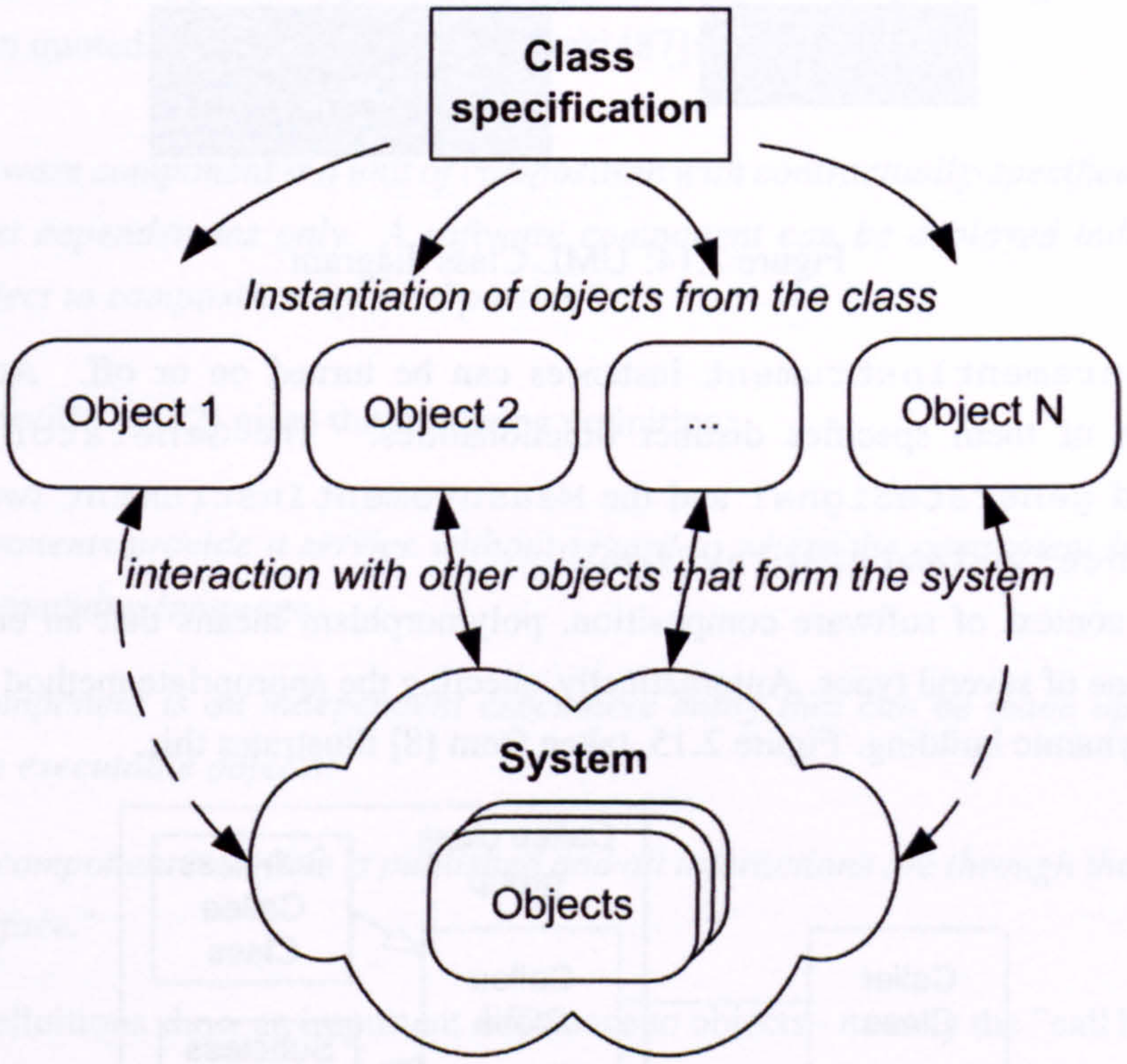


Figure 2.13: Classes and objects

Specifications encapsulated in classes can be further extended (specialised) with the help of inheritance. Consider the Unified Modelling Language (UML) [16, 76, 85] class diagram example shown in Figure 2.14.

The class Device describes all device objects. The behaviour for the Device is specified with two methods, `turnOn` and `turnOff`. The classes `Generator` and `MeasurementInstruments` specialise the device as two different forms. They inherit from the class `Device`. In this case, we can also say that they are sub-classes of a super-class `Device`. Sub-classes inherit behaviour from their super-classes. In the context of the example, this means that the `Generator` instances

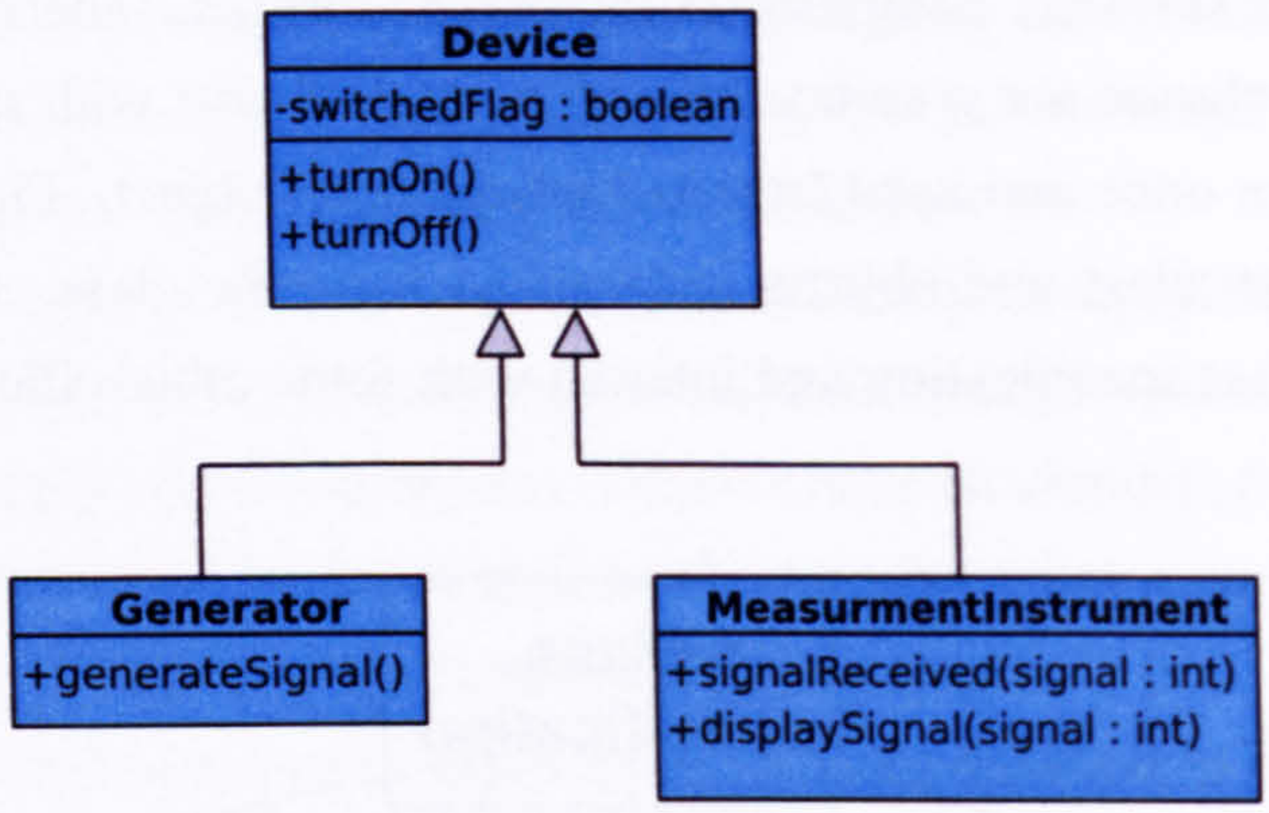


Figure 2.14: UML Class diagram

and `MeasurementInstrument` instances can be turned on or off. At the same time, each of them specifies distinct functionalities. The `Generator` class has the method `generateSignal` and the `MeasurementInstrument` two methods `signalReceived` and `displaySignal`.

In the context of software composition, polymorphism means that an entity could have any one of several types. Automatically selecting the appropriate method at runtime is called dynamic building. Figure 2.15, taken from [8] illustrates this.

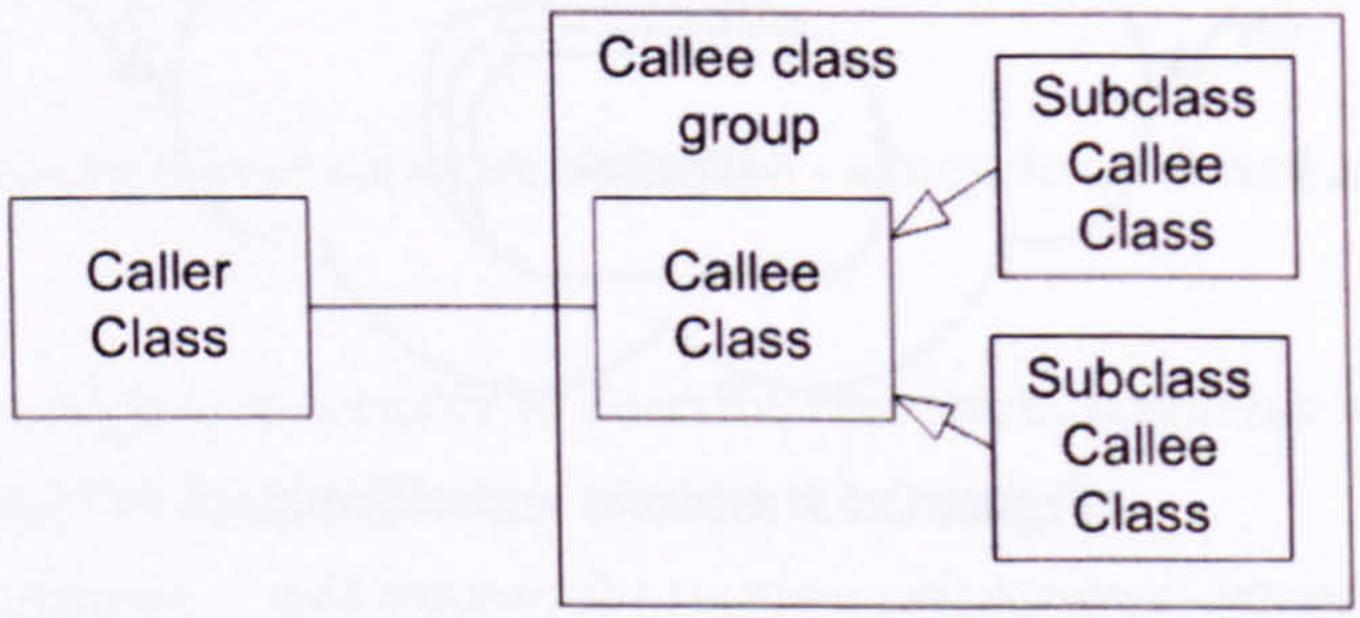


Figure 2.15: Polymorphism and dynamic binding. Objects of a super-class can be dynamically exchanged for objects of a sub-class at runtime

An object-oriented composition model provides major advantages over modular composition, such as dynamic extension of the system and a more flexible and comprehensive way of describing a target system. However, composition with an object is an identity-based approach, as the identity of an object is a criteria for required behaviour. a further step over the object-oriented approach was the composition of software systems with components (services).

2.6.3 Component-Based Composition

Component-based composition (development) stands for designing software systems from application elements that were constructed independently by different developers using different languages, tools, and computing platforms [1, 22]. In comparison to objects, components represent stand-alone service providers. Many definitions of components exist so far. They focus on different aspects of software engineering. The definition that is often quoted is one offered by Szyperski [87]:

"A software component is a unit of composition with contractually-specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Sommerville in [82] gives the following definition:

"Components provide a service without regard to where the component is executing or its programming language:

- *A component is an independent executable entity that can be made up of one or more executable objects;*
- *The component interface is published and all interactions are through the published interface."*

Both definitions show an important difference to objects - namely the "call by service" principle. From the user's point of view, a component represents certain services introduced by its interfaces. In classical component-based composition, components are rather black-boxes [7]. The user is interested in the delivered service and does not actually care about implementation and platform details. From the software designer's point of view, components represent a collection of design artefacts grouped together, documented and accessed via interfaces. Figure 2.16 shows a component with its interface specification.

Greenfield et al. [40] presented an extension and variation of the component specification metamodel provided by Cheesman and Daniels [21]. The key point in the definition of a component is that its implementation is separated from the specification. The component specification describes what a component does and how it behaves when its capabilities are used by others. Potential users of those capabilites do not concern as to how they

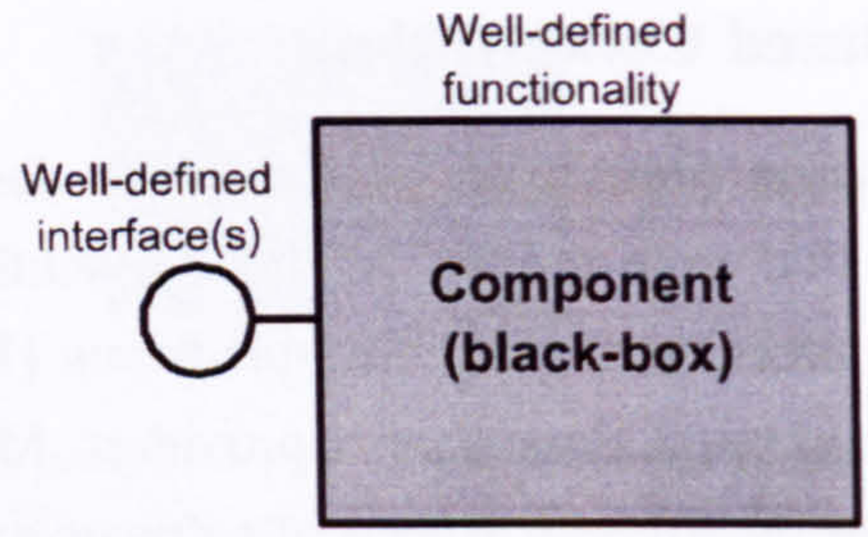


Figure 2.16: A black-box component

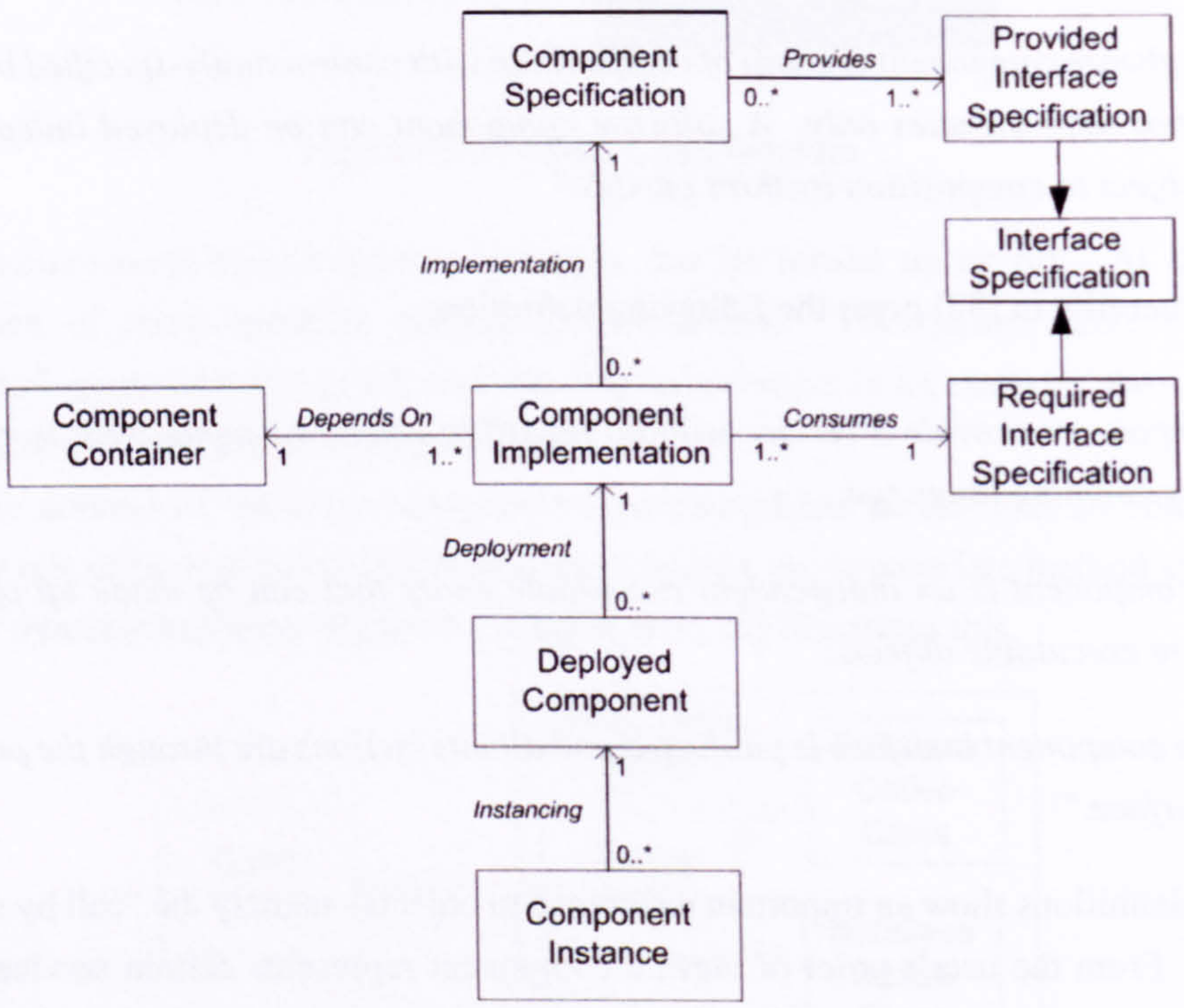


Figure 2.17: Component specification metamodel

are actually implemented. Each component normally provides an interface to define and access the services within this component. From the other side, a component may specify required interfaces, the implementation of which will be consumed by the existing implementation part of the component. There could be more than one implementation defined for the same component specification. These implementations can be created for different platforms and represent different performance or cost. Ready components are integrated

into the execution platform (deployed). During runtime components will be instantiated dynamically according to defined requirements.

There are many different standards defined for components. A component model is a definition of standards for component implementation, documentation and deployment. The component model specifies how interfaces should be defined and the elements that should be included in an interface definition. Well-known component models are Enterprise JavaBeans [75] and the Corba component model [65].

Components represented the call-by-service paradigm. From the scope of the developer annotation techniques are used in order to describe services. The potential user of a service can request a needed service and it will be bound dynamically. At further step in development in composition models is presented by architecture-based composition. This model is a step in the direction of using grey-box elements as elements of software composition.

2.6.4 Architecture-Based Composition

Architecture-based composition introduces a novel composition model with connectors and ports, abstracting communication routines from the component implementations. Connectors are modules for communication. Ports are connection points indicating when (but not how) data flows in and out of a component. In order to help the architectural designers with their tasks, the formal notations for representing and analyzing architectural design were proposed. These notations are referred to as "architecture description languages" (ADLs). Clements [23] specified ADLs as formal languages that can be used to represent the architecture of a software-intensive system.

Figure 2.18 depicts a system composed with components, ports and connectors. On the left side, there is a more detailed view is presented of the two components composed.

The component-connector composition helps avoid interface mismatch. Additionally, architecture can be reconfigured without affecting its components and wider applicability of components.

Several systems using the architecture-based composition have been developed. These are, for example, Acme [38], Darwin [59], Unicon [78], and Wright [5].

The communication mechanism in the architecture-based composition is encapsulated and represents a parameter through which a system can be adapted to a range of newly defined requirements. This results in a wider applicability of components. Taking this into consideration, it is possible to say that the architecture-based composition is a step

in the direction of using grey-box elements and the aspect-oriented composition models discussed below.

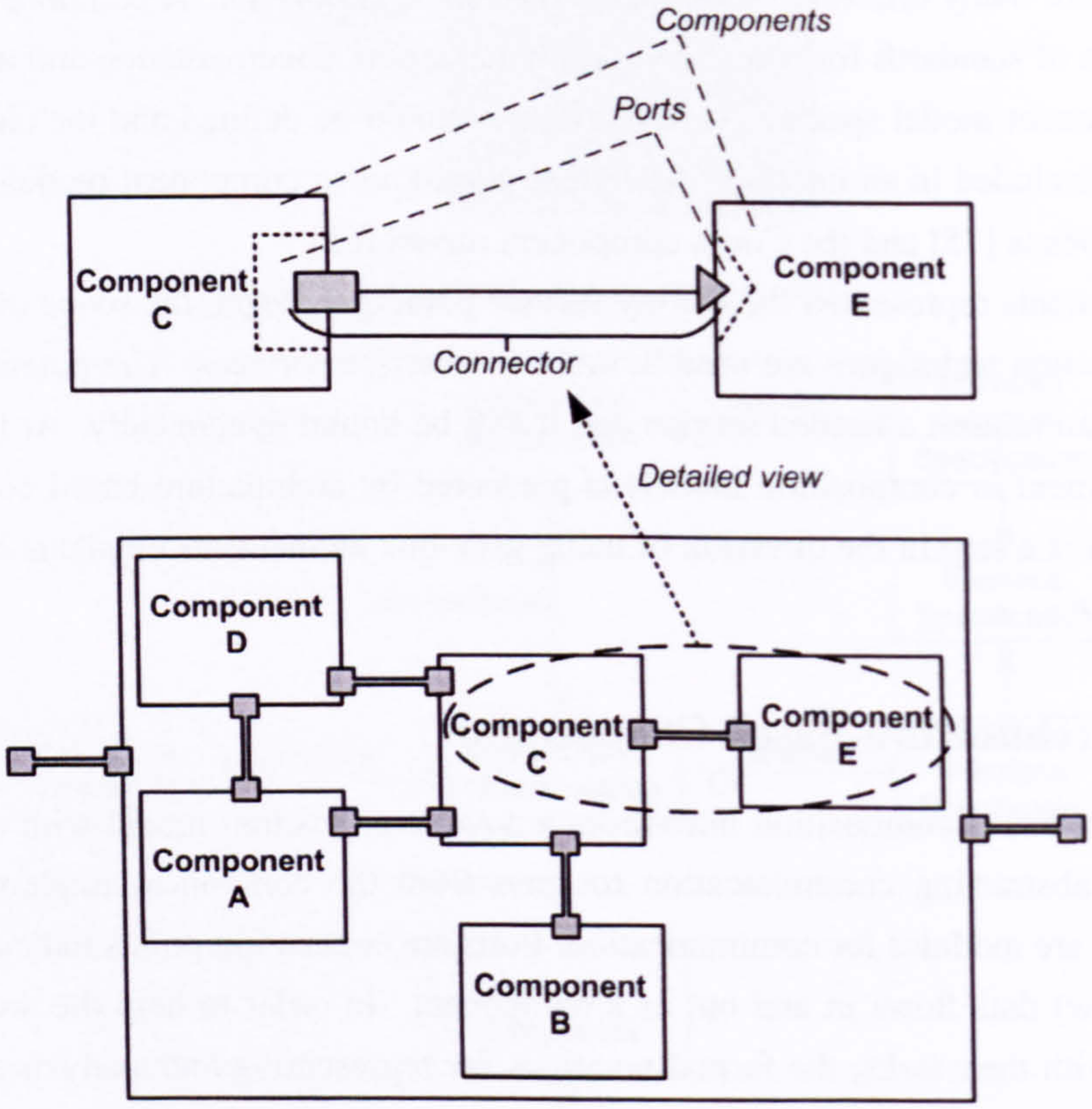


Figure 2.18: Two components are connected by means of connector and ports

2.7 Grey-box Approaches

Compared to black-box approaches, grey-box approaches apply grey-box components as units of composition. The typical example of a grey-box composition approach is the AOP. In this approach, a grey-box component is an *aspect*. Aspects maybe weaved by demand into other components (e.g. classes). The weaving mechanism makes classes (or other components) non-monolithic.

Grey-box approaches are in their infancy and have not been yet successfully applied in industry. This thesis reviews the following grey-box composition approaches: aspect-

oriented programming, composition approaches based on the multi-dimensional separation of concerns, feature-oriented composition and invasive software composition.

2.7.1 Aspect-Oriented Programming

Some problems appear during the process which concerns separation when the program is broken into distinct features. Ideally, the overlapping in functionality should be as minimal as possible. However, no matter how carefully a software system is decomposed into modular units there will always be concerns (typically non-functional ones) that cut across the chosen decomposition. The code of these crosscutting concerns will be spread necessarily over different modules, which has a negative impact on the software quality in terms of comprehensibility, adaptability and evolvability. Besides, some systems can not be built with traditional techniques as specified by Kitzales [55]. Figure 2.19 shows a concern which cuts across target modular decomposition.

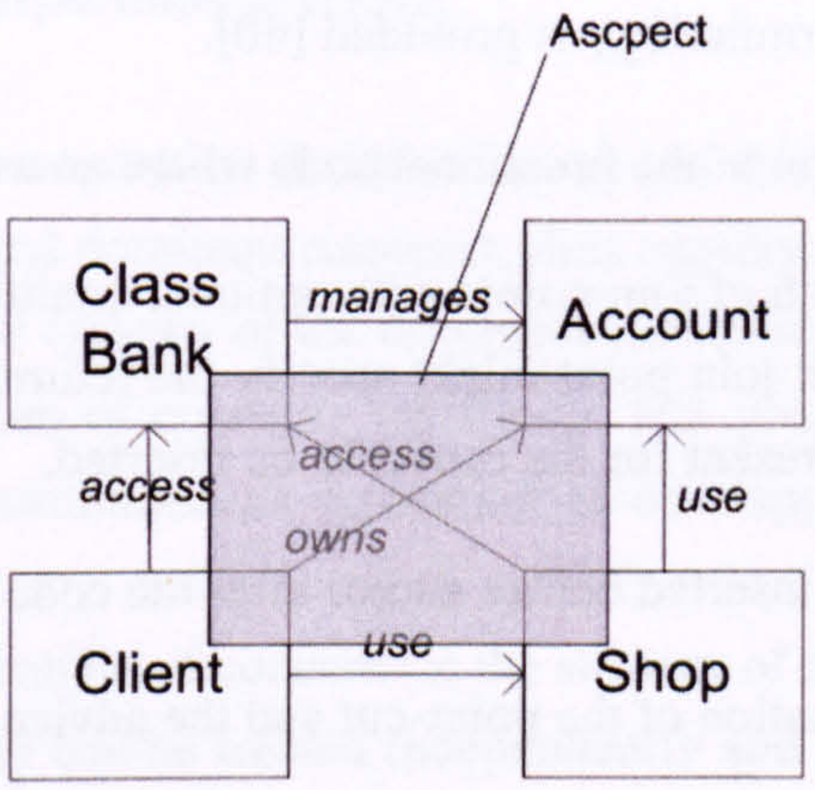


Figure 2.19: Modules and aspect

Modules in Figure 2.19 describe concepts such as Bank, Client, Account and Shop along with their signed relations. The grey area depicts that each element contains programming logic which is mixed in modules but performs some task which can be considered separately. This grey area specifies a concern which cuts across the given modules. For example, this can be security or logging mechanisms that are present in any of the depicted modules. The architecture-based composition only carries aspects such as application-specific functionality and communication.

Aspect-Oriented Programming [6, 66, 55] is a method developed at the Xerox Palo Alto Research Center (PARC) by Gregor Kiczales [55]. Part of the program that cross cuts

its core concern is referred to as "aspect". AOP allows programmers to first express each of a system's aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver™[55].

In comparison to composition models discussed in previous Sections 2.6.1, 2.6.2, 2.6.3 and 2.6.4, the AOP uses aspectual decomposition rather than functional decomposition. Kiczales et al. in [55] specified the core differences between aspects and functional units (modules, components, objects) as follows:

An important difference between aspects and functional units is that aspects fundamentally cross-cut both each other and the resulting executable code. As such a cohesive aspect of concern in the source program ends up being spread about and mixed in with other aspects in the output of the weaver.

The following basic terminology is provided [40].

1. Join Point: A position in the functional code where an aspect may be inserted.
2. Pointcut: A collection of join points with optional qualifiers. For example, a point-cut for a method call join point might specify the return value type and parameter types that must be present for the aspect to be inserted.
3. Advice: Code to be inserted before and/or after the code identified by a point cut.
4. Aspect: The combination of the point-cut and the advice.

In order to generate a target source, the processing of the input source code and aspects happens with help of Aspect Weaver. This is depicted in Figure 2.20.

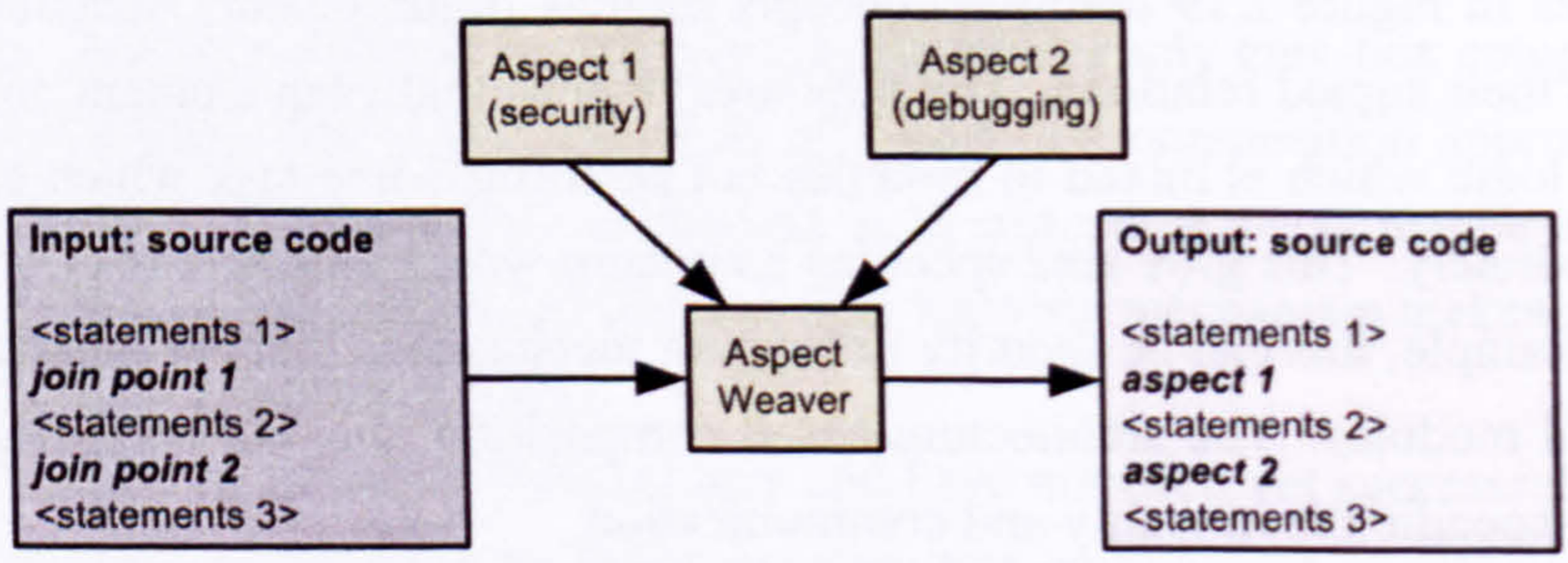


Figure 2.20: Weaving aspects with the help of Aspect Weaver

In the AOP, modules are treated as grey-box elements; thus, wider adaptability of target systems is seen. Technically, the adaptability is reached by the weaving of glue code around modules (classes, components, etc.) and by the exchanging of aspects. Next, we review the approach, called *N-Dimensional Separation of Concerns*, that took its roots from the AOP.

2.7.2 Compositions Based on N-Dimensional Separation of Concerns

Black-box composition approaches which are currently used in industry permit the separation and encapsulation of only one kind of concern at a time. Tarr et al. specified a related term, "tyranny of the dominant decomposition" [88]. More precisely:

"... existing formalisms at all lifecycle phases provide only a small, restricted set of decomposition and composition mechanisms, and these typically support only a single, "dominant" dimension of separation at a time."

Examples of tyrant decomposition include classes and modules. Non-dominant concerns cut across encapsulated dominant concerns, thus causing scattering and tangling.

To solve the problem of tyranny of the dominant decomposition, the strategy, called multi-dimensional separation of concerns (MDSoC), was proposed by Tarr et al. It is based on the support for simultaneous separation of overlapping concerns in multiple dimensions.

Multi-dimensional separation of concerns is the strategy of dividing a complex system into several dimensions that can be treated independently and are simpler to understand [8]. Ossher and Tarr [67] referred to a dimension as a kind of concern, like class or feature. More precisely, MDSoC was defined by Ossher and Tarr [67], such as the separation of concerns involving:

- Multiple, arbitrary dimensions of concern.
- Simultaneous separation along these dimensions.
- The ability to handle new concerns, and new dimensions of concerns dynamically, as they arise throughout the software lifecycle.
- Overlapping and interacting concerns; it is appealing to think of many concerns as independent or "orthogonal", but they are rarely in practice.

Additionally, feature such as on-demand remodularisation was introduced. This feature will let the developer choose the best modularisation at any time, based on any or all of the concerns.

The MDSoC is a new, fresh strategy and it opens the door for further research questions to be investigated. Basically, it is not yet integrated into the real working industry solutions. Next, we will shortly review subject-oriented programming as an initial step in the direction of MDSoC, followed by hyperspaces, as one of the currently proposed approaches for MDSoC.

Subject-Oriented Composition One of the initial steps in developing the MDSoC was the subject-oriented programming (SOP). The SOP was introduced by Ossher et al. [43]. The SOP supports the packaging of object-oriented systems into subjects. The SOP provides a means for composition designers to write composition expressions.

A general meaning of the term "subject" is given [43]:

"We use the term "subject" as a collection of state and behaviour specifications reflecting a particular gestalt, a perception of the world at large, as seen by a particular application tool.

...

Subjects generally describe some of the state and behaviour of objects in many classes."

More precisely, each subject is an object-oriented program or its fragment that models its domain in its own, subjective way [43]. Subjects are written in an object-oriented source language and compiled using a subject compiler for that language. A combination of subjects is a cooperating group called a composition. A composition defines the composition rule that specifies combination details. The SOP provides, for example, rules such as the merging two subjects of a class (merge) and the updating of a class feature list with the features of another class (replace).

The SOP provides the following advantages [43]:

- Unanticipated extension and composition, without changing or recompiling existing source code

- Decentralised class development. Authors of different applications that share objects can write definitions of their views of shared classes separately; these definitions can later be composed.

- Requirement-based (feature-based) development. The code that implements a requirement or feature can be built as a coherent subject, rather than being interleaved amongst other code in a way that makes it hard to identify and maintain.

Further, the development of SOP is introduced by the Hyperspace programming - a generalised SOP.

Hyperspaces The hyperspace approach is a particular approach to MDSoC. Hyperspaces permit the explicit identification of any concerns of importance, encapsulation of those concerns, identification and management of relationships among those concerns, and integration of concerns [67]. The main concepts of hyperspace programming are *units*, *hyperspaces*, *hyperslices*, and *hypermodules*. A unit is a code fragment of the underlying language, e.g. a class or a member definition. All fragments of a system are assembled in a hyperspace. All code fragments belonging to a single concern are grouped together to a hyperslice with a user-specified concern mapping. Hyperslices are building blocks of composition and composed by composition rules [68] taken from SOP. Hyperslices are a set of units that are declaratively complete in order to eliminate coupling between hyperslices. A compound set of hyperslices is called a hypermodule. They can be composed to larger hypermodules until the final system results. As such, hypermodules are hierarchical component models whose basic components are hyperslices.

The major advance over SOP and AOP is the ability to simultaneously support the clean separation of multiple different kinds of potentially overlapping concerns, with on-demand remodularisation.

Hyperspaces can be applied to any design or implementation language. For Java programming language, the hyperspace programming is exemplified in the tool Hyper/J [48]. Hyper/J non-invasively facilitates several common development and evolution activities, including: adaptation and customisation, mix-and-match of features, reconciliation and integration of multiple domain models, reuse, product line management, extraction and replacement of existing parts of software and on-demand remodularisation.

Let's consider the following example, taken from [8]. There is a package `passiveDevices` collecting the classes `WorkPiece` and `ConveyorBelt`, and a package `activeDevices` collecting classes which describe presses and robots. There is one more package `tracing` that contains tracing code. Lising 2.1 shows the hyperspace `ProductionCell` as a collection of all classes in all packages.

```
1 hyperspace ProductionCell
```

```
composable class passiveDevices.*
composable class activeDevices.*
composable class tracing.*
```

Listing 2.1: Hyperspace ProductionCell

Initially, there are groups of hyperslices - dimensions - defined. The `Feature` dimension groups core functional features of classes into different hyperslices. The `Logging` dimension groups fragments that deal with logging functionality. The next step is to define concerns and concern mappings. First, the concern `Feature.WorkPieces` is defined. By default, it includes every member in the package. Afterwards, it is specified for certain members that they belong to a second concern `Feature.Transfer`. Listing 2.2 shows this.

```
package passiveDevices: Feature.WorkPieces
  operation lifeCycle: Feature.Transfer
  field ConveyorBelt.pieces: Feature.Transfer
  operation setPieves: Feature.Transfer
  operation setPiecesNumber: Feature.Transfer
  operation getPiecesNumber: Feature.Transfer
```

Listing 2.2: Concern mappings for the package passiveDevices

In a similar way, as Listing 2.3 shows, the concern mappings are defined for the package `activeDevices`.

```
package activeDevices: Feature.ActiveDeviceBehaviour
  operation Press.takeUp: Feature.Transfer
  operation Robot.takeUp: Feature.Transfer
  operation lifeCycle: Feature.Action
```

Listing 2.3: Concern mappings for the package activeDevices

The next listing, 2.4, shows another concern `Logging`. `Tracing` groups all methods from the same class `TracingAttribute`.

```
package tracing: Feature.Tracing
  operation TracingAttribute.enterAttribute : Logging.Tracing
  operation TracingAttribute.leaveAttribute : Logging.Tracing
```

Listing 2.4: Concern mappings for the package tracing

Now, the hypermodule can be defined, which describes the transfer of work pieces in the production cell. This is done by grouping the hyperslices `Transfer`, `WorkPieces`, and `Tracing` together. This is shown in Listing 2.5.

```
1 hypermodule TracedProductionCellTransfer
2  hyperslices: Feature.Transfer, Feature.WorkPieces,Logging.
     Tracing
3  relationships: mergeByName
4  bracket "*","*"
5    before Logging.Tracing.TracingAttribute.enterAttribute()
6    after Logging.Tracing.TracingAttribute.leaveAttribute()
```

Listing 2.5: Hypermodule: the transfer of work pieces in the production cell

The hypermodule `TracedProductionCellTransfer` merges the three hyperslices by name and brackets all operations of all classes with tracing code. No code concerned with actions is included.

The hyperspaces composition model is advantageous in comparison to the architecture-based composition and aspect-oriented programming. It provides an explicit grey-box component model and supports explicit composition expressions.

The work of the MDSoC and hyperspaces, respectively, is at an early stage and largely unproven in practice to this point. Some of the remaining questions were specified in [67]. They involve topics of discussion such as the understanding and specification of concerns, their scalability, tool support, improvement of the software process and adaptation to the real industry needs.

2.7.3 Feature-Oriented Composition

A feature in software composition represents a certain piece of functionality. Batory et al. [14] defines a feature as a characteristic that programs of a product-line can share; distinct programs in a product-line are described by distinct combinations of features. In [73], Prehofer proposed a new model for the flexible composition of objects from a set of features. This model was called Feature-Oriented Programming (FOP). Unlike object-based composition, the feature-based composition made it possible to create objects with individual services by selecting the desired features. Like the Aspect-Oriented composition, the FOP was developed in order to overcome problems that appeared when using classical decomposition techniques like OOP. These problems include the presence of

crosscutting concerns (Section 2.7.1). The FOP modularises crosscutting concerns and is based on collaboration design and refinements and can be classified as a part of the MD-sOC concept. A typical approach to FOP is AHEAD (Algebraic Hierarchical Equations for Application Design).

AHEAD Batory et al. [14] defined AHEAD as an approach to FOP based on step-wise refinement. Step-wise refinement asserts that a complex program can be synthesised from a simple program by progressively adding features. An AHEAD model uses the GenVoca [15] design methodology for creating application families and architecturally-extensible software. The GenVoca model shows how the code representation of an individual program is expressed by an equation. Consider the following constants that represent base programs with different features:

```
1 f //program with feature f
2 g //program with feature g
```

Listing 2.6: Programs with features

A refinement is a function that takes a program as input and produces a refined or feature-augmented program as output:

```
1 i(x) // adds feature i to program x
2 j(x) // adds feature j to program x
```

Listing 2.7: Refinements

A multi-features application is an equation that is a named expression. Different equations define a family of applications, such as:

```
1 app1 = i(f) //app1 has features i and f
2 app2 = j(g) //app2 has features j and g
3 app3 = i(j(f)) //app3 has features i,j and f
```

Listing 2.8: Family of applications defined though the equations

Thus, the features of an application can be determined by inspecting the equation.

2.7.4 Invasive Software Composition

Aßmann [8] proposed a composition technique called Invasive Software Composition (ISC) that enables the composing of software components by program transformation.

In ISC, "component" is defined as a set of program elements or program fragments. This definition of component goes beyond the treatment of components as the back-box elements. The following definitions were proposed:

A fragment box is a set of program elements. A fragment box has a composition interface that consists of a set of hooks.
A hook is a point of variability of a fragment box, a set of fragments, or positions that are subject to change.
A composer (or composition operator) is a program transformer that transforms one or more hooks for a reuse context.

Composers instantiate, adapt, extend, and connect fragment boxes by transforming their hooks. Fragment boxes are a collection of code templates.

Fragment boxes are parameterised with hooks. There can be of two kinds of hooks - a code hook and a position hook. Hooks containing only program elements are called code hooks. Position hooks are those which consist of positions in the component's code, which may or must be filled during composition. Additionally, hooks can be implicit or declared. Implicit hooks are those contained in every component by definition of its programming language. Declared hooks are those declared to be subject to change. Moreover, hooks are classified according to their structure. There could be atomic and nested hooks. A nested hook consists of a collection of subhooks that must be bound or extended together. Figure 2.21 shows an example of code fragment with hooks.

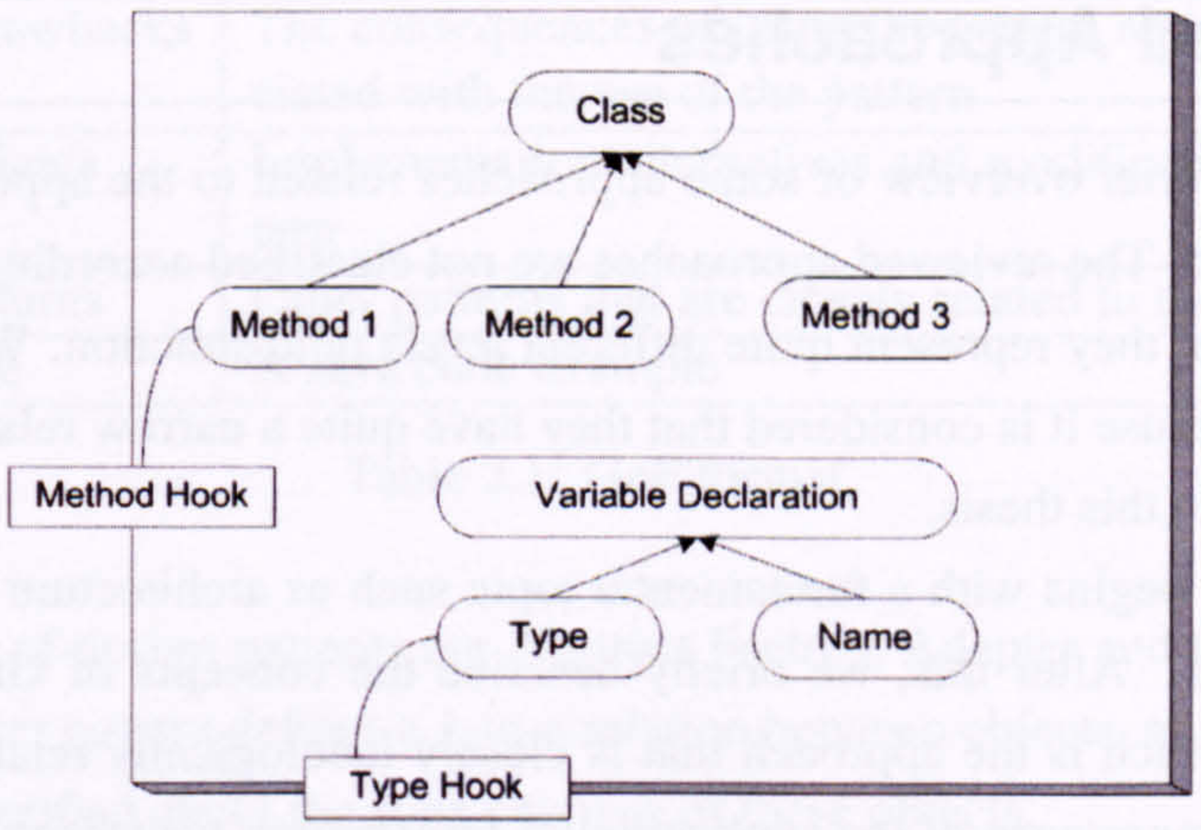


Figure 2.21: An example of fragment with hooks [8]

The figure shows a class box with program elements, a method hook and a type hook; these are subject to change during class composition.

Composers are procedure or metaprograms which manipulate code fragments according to specified hooks. Let's consider the example given (Listing 2.9).

```
1 MethodBox requestDevice;
2 Hook methodExit = requestDevice.methodExit(''methodExit'');
3 MethodBox debugRequestDevice = methodExit.bind(''Debug.info(\''
    leaving method requestDevice\''););
```

Listing 2.9: Metaprogram demonstrating a composer

The example demonstrates a composer - a metaprogram that manipulates target code. This manipulation includes looking for the hook denoting exit point within the method `requestDevice` and inserting the code to print out the debug information. A software composition system for ISC was introduced [9]. Compost [45] is a Java class library for invasive software composition with static metaprogramming.

Invasive software composition not only provides a full-fledged composition language, but also offers a grey-box component model for the construction of tightly connected systems. It enables generic programming, connectors in a standard language, inheritance calculi, view-based programming, and aspect-based development. Invasive software composition also has some restrictions. It is a code-based component technology, i.e. it works at compile-time and requires classes.

2.8 Related Approaches

In this section, a brief overview of some approaches related to the appoach presented in this thesis is given. The reviewed approaches are not classified according to some special criteria. Moreover, they represent quite different levels of abstraction. We collected them in this section because it is considered that they have quite a narrow relationship with the topic researched in this thesis.

The overview begins with a fundamental topic such as architecture reuse in the face of design patterns. After this, we briefly describe the concepts of Grammar-Oriented Object Design, which is the approach that is closely ideologically related to this thesis. Afterwards, the importance of the application of information visualisation techniques on software models is shown with the Geon Diagrams.

2.8.1 Design Patterns

Gamma et al. [36] made claims which can be summarised in the following hypothesis [31]: "Reusing designs through patterns yields faster and better maintenance." According to Alexander, a pattern is a relationship between a context, forces often encountered in that context, and a strategy for solving the forces [3, 4].

A design pattern is a schematic description of a possible solution to a design problem [31]. Design patterns are typically informal descriptions that answer the following questions: "How is the design pattern identified?", "What does it do for the developer?", "How does it solve the problem?" and "What are the benefits of this design pattern?".

There are two forms in which the patterns are usually represented. These are the "Canonical" and the "Gang of four" (GoF) forms. For example, the GoF description includes the specifications as shown in Table 2.1.

Topic	Explanation
Name	A descriptive name of the design pattern.
Also known as	Alternate names, if any
Pattern Properties	The pattern's classification. Includes type and level
Purpose	A short explanation of what the pattern involves
Introduction	A brief description of a problem which might be faced where the pattern may be useful. An example is provided
Applicability	When and why the design pattern might be used
Description	Detailed description of the pattern, what it does and how it behaves
Implementation	What must be done to implement the pattern
Benefits and Drawbacks	The consequences of using the pattern and tradeoffs associated with the use of the pattern
Pattern Variants	Implementation alternatives and modifications of the pattern
Related Patterns	Other patterns that are closely related to the pattern
Example	A Java code example

Table 2.1: GoF format

The examples of design patterns are Abstract Factory, Adapter and Observer. For example, the Observer pattern defines a 1-to-n relation between objects, so that all interested abstractions are notified about the state changes of these objects.

Design pattern theories have been applied to real-world problems [62, 33, 25]. Still, design patterns are difficult to track, modularise and reuse, as their elements tend to van-

ish in the code. Denier [29] proposed experiments with aspect-oriented programming as a modular technology to give new insights on the expression of design patterns. He used mechanisms offered by aspect languages to express elements of design patterns implementation. It was concluded that aspects can benefit from insights of pattern-driven solutions on the way to building robust implementations. Moreover, it was underlined that aspect-oriented programming enables the study of variability and configuration, as design patterns in the code are often generic infrastructures mixed with specialisation for the target concern.

2.8.2 Grammar-Oriented Object Design

The Grammar-Oriented Object Design (GOOD) [12, 11] facilitates the creation of software enterprise component architectures that realise the following characteristics:

1. Dynamic [re-] configuration based on business domain languages,
2. Component manners to manage collaboration and
3. Self-description that defines the context and abilities of a component that can be queried without violating encapsulation.

These characteristics are needed to fulfil the demands of component-based software engineering today and in the future. According to [12], these demands are stated as follows. The configuration and collaboration of software components in software architecture are designed to adaptively conform to a set of business requirements that often need to be updated to reflect changing business needs and models. Altering collaboration sequences, business rules and processes within applications often creates unacceptable maintenance overhead in a tight delivery window, whereby the constraints of high quality software need to be balanced with (and is sometimes compromised for) rapid time-to-market.

The work done around GOOD is often related to the term *externalisation*. According to [11]: "Externalisation involves taking the encapsulation or separated design decision about the dynamic aspect of application flow and making it re-configurable by presenting it as meta-data within the grammar of a domain-specific language."

2.8.3 Domain-specific Languages

Domain-Specific Languages (DSL), or little languages, are those that are tailored to a particular problem domain. Through the appropriate use of notations and abstractions, they provide the expressive power to better describe specific solutions to problems in that domain [89].

Advantages of DSL include expression at an appropriate level of abstraction, employment of the concepts familiar to practitioners and better validation and optimisation at the domain level. DSL examples include Graphviz, HTML (HyperText Markup Language) and SQL (Structured Query Language). DSL can be seen as the composition of DSL components designed by a domain expert. According to [70], DSL components describe properties of a language, e.g. parts of the lexical or syntactical structure, scope rules, typing, or the mapping to a target language. Visual notations and abstractions for DSLs are more appropriate to model systems.

When using visual notations instead of textual ones for DSLs, we speak about Domain-Specific Visual Languages (DSVL) [32]. The quality of visual representations and the level of how they are accessible to the human intuition depend on the information visualisation technique used. Information visualisation is the visual presentation of abstract information spaces and structures to facilitate their rapid assimilation and understanding [10]. The complexity of information can be reduced by means of information visualisation methodologies and concepts [50, 46, 64]. Diagrams are essential in documenting large information systems. They capture, communicate and leverage knowledge that is indispensable for solving problems and are conceived to act as "cognitive externalisations" [99]. A diagram provides a mapping from the problem domain to the visual representation by supporting cognitive processes that involve perceptual pattern finding and cognitive symbolic operations [57].

2.8.4 Domain-Specific Modelling

According to [51], Domain-Specific Modelling (DSM) is an approach that uses generative programming [27] to improve the quality of the software development or system development process. The DSM approach uses so-called "domain-concepts" directly as program language concepts.

DSM aims to eliminate the double work of transforming the system model into the ready implementation. Normally, this translation is done automatically. DSM allows au-

tomatically generating low-level programming code from the high level domain-specific model.

The process of DSM can generally be seen as a definition of domain-specific language when the domain expert encapsulates his expert knowledge about the domain concepts, their properties and rules that apply into the modelling language. Afterwards, this language is applied to specify the application in a relatively simple and quite expressive manner. The example shown on Figure 2.22 is taken from [51].

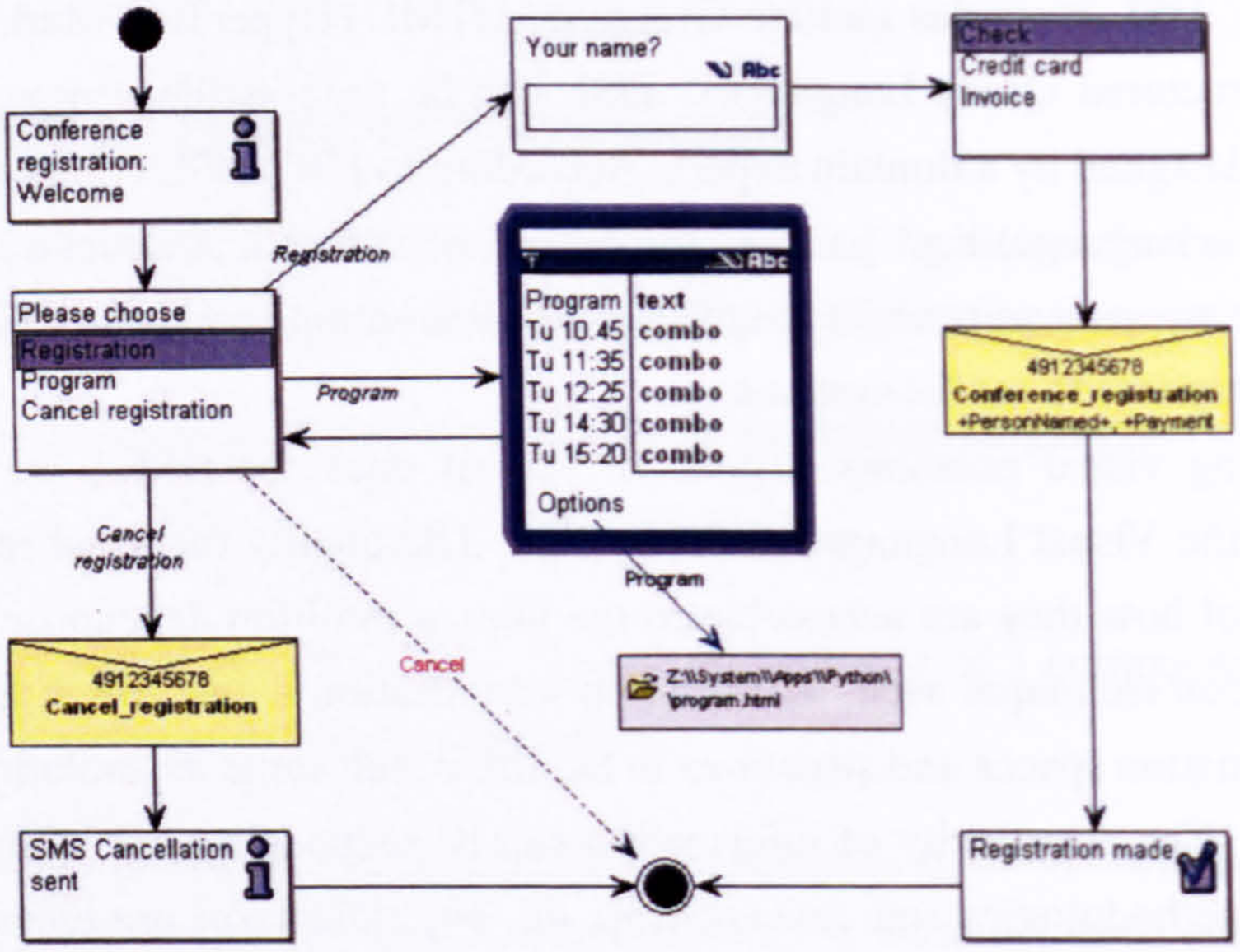


Figure 2.22: DSM of mobile phone application [51]

The figure shows an example of the defining of a mobile phone application with a domain-specific modelling language. The model only uses concepts from the mobile phone domain, i.e. all coding concepts are hidden. Still, the model is complete enough to generate the full application code automatically.

2.8.5 Enhancing Semantic Content: Geon Diagrams

Irani et al. [50] discuss aspects of structured object recognition theory and show how this can be used to make 3D diagrams that are more easily analyzed (visually) and remembered. Irani defined so-called Geon Diagrams based on Biederman's geons (geometrical ions). They performed a number of tests where the most suitable visual representation for several modelling concepts was selected. As criteria, a number of votes from all par-

ticipants was taken. As modelling concepts, the following were taken: generalisation, dependency, strength of relationship, multiplicity of relationship and aggregation. For each modelling concept, there were several visualisations presented. For example, Figure 2.23 shows dependency's possible visual representations to be selected from.

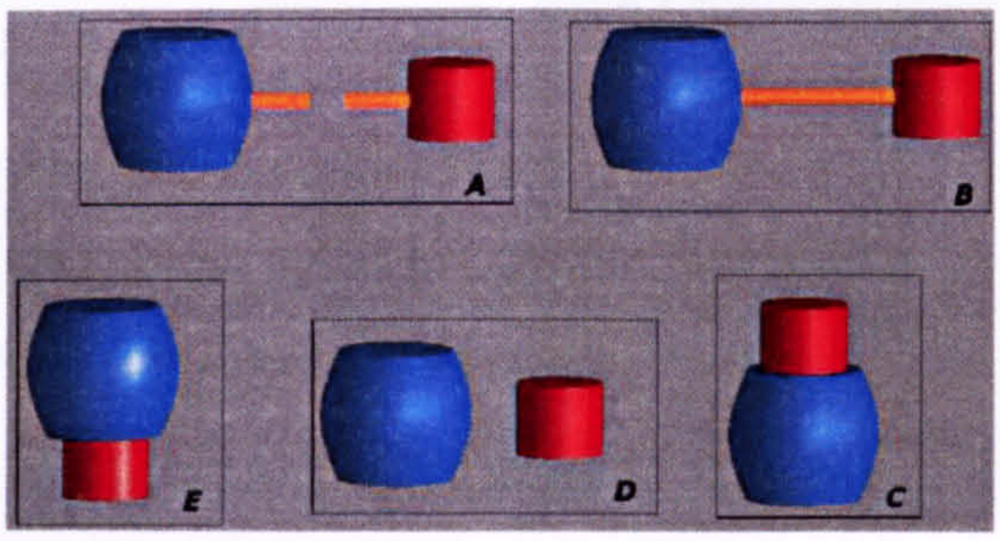


Figure 2.23: Visual representations of dependency concept [50]

From 1-5 (best to worst) test participants were asked to rank the representation denoting that one object depends on another. After organizing and proving statistics, it was clear that the most suitable visual representation for the dependency was "C". Figure 2.24 shows perceptual notations for all modelling concepts which were tested.

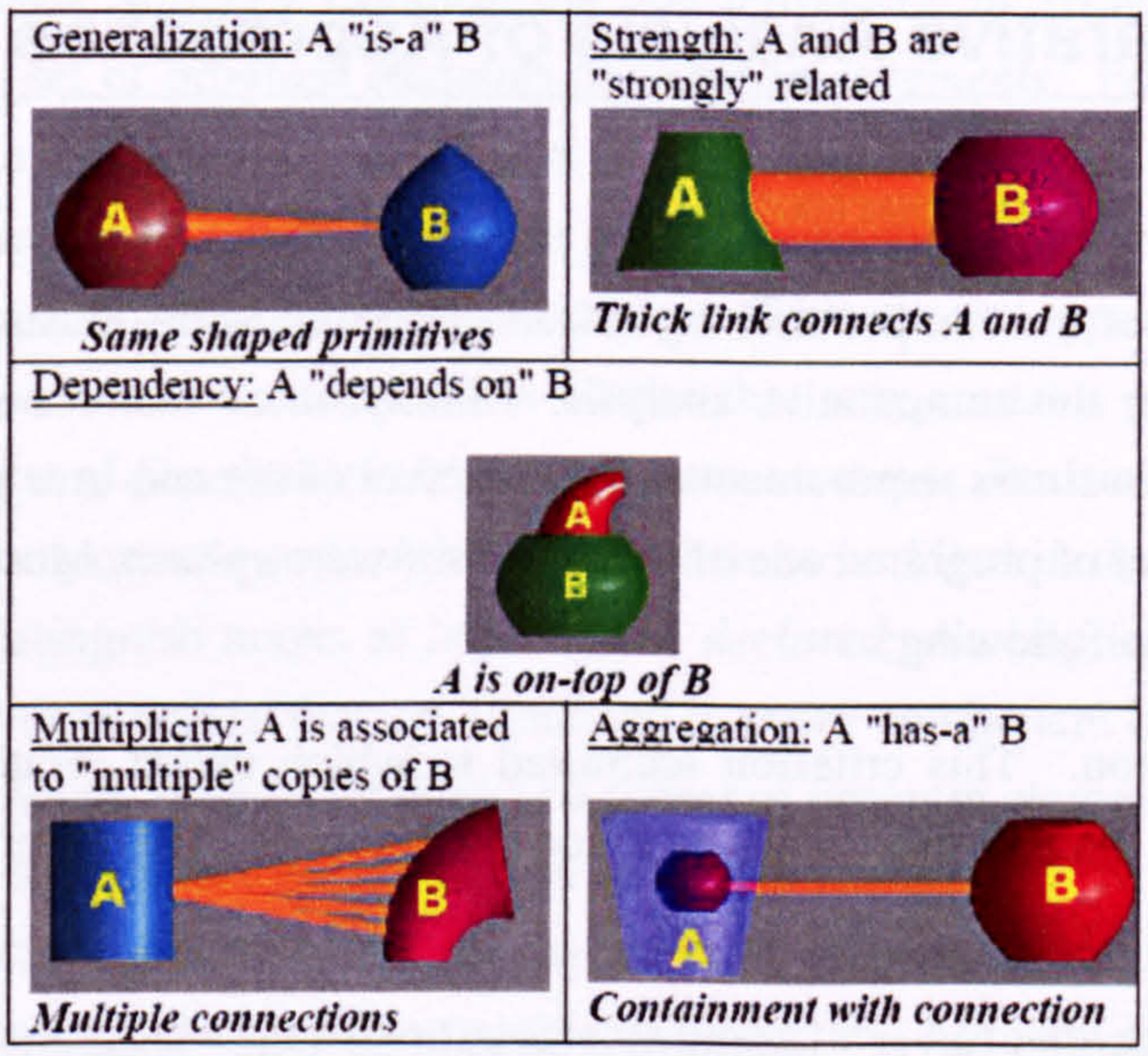


Figure 2.24: Perceptual notation for target modelling concepts [50]

Irani provided several tests in order to compare error rates in identifying relationships in the UML Class diagram and the corresponding geon diagram; see Figure 2.25.

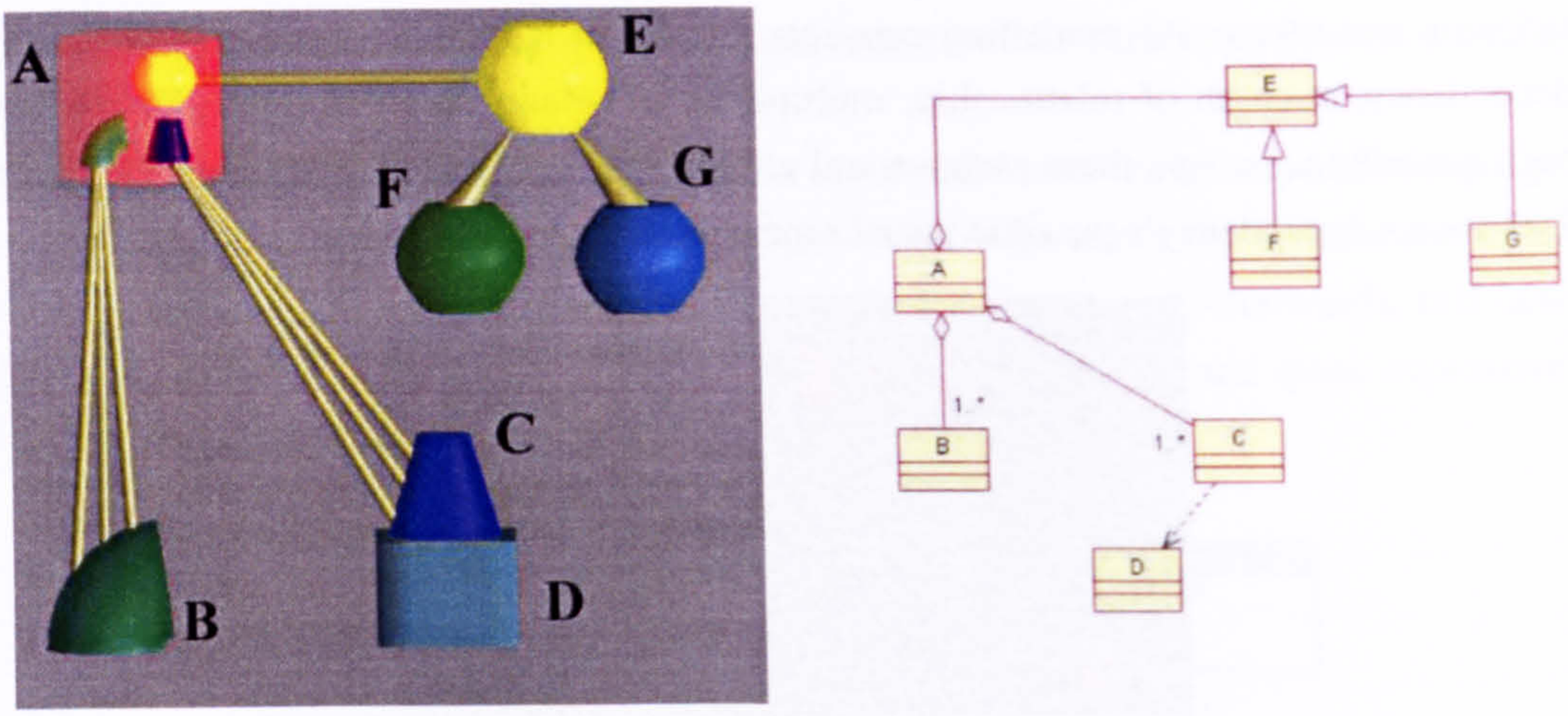


Figure 2.25: Left - a geon diagram representation; right - a UML representation equivalent to the geon diagram on the left side [50]

There were almost five times as many errors deciphering relationships between entities using UML notation than using the perceptual syntax.

2.9 Comparative Analysis of Approaches

2.9.1 Means of Comparison

The core features of the composition framework proposed in the thesis are taken as distinction creitria for the comparative analysis. Mainly, these features are related to the externalisation of business requirements up to the level of the end-user and to the configurability at the level of program code of the target software system. More specifically, the features include the following:

1. Externalisation. This criterion identified to which extent requirements may be changed by the end-user which is a non-programmer. The criterion includes two sub-criteria:

   (a) Visualisation Layer - identifies a domain-specific visual environment to simplify configuring underlying program code of a software system.

   (b) Target Domain Layer - denotes a set of mechanisms to describe the program code of a software system with the help of domain-specific terms.

2. Component Model (Language Level). This criterion identifies a component model to encapsulate templates at a program code level. The component model defines what can be configured. The criterion includes three sub-criteria:

   (a) Code templates - identify existence of templates as parts of a program code composition.

   (b) Context awareness - identifies feature of being context-aware for templates.

   (c) Configurability at the program code level - identifies the feature of configurability of a software system at a program code level.

3. Composition Technique (Language Level). This criterion identifies a composition technique to configure code templates defined by the component model.

4. Object-oriented technology (at the meta level). This criterion denotes application of the object-oriented approach to describe, implement and configure code templates.

2.9.2 Invasive Software Composition

The topic of Invasive Software Composition was introduced in Section 2.7.4. In ISC, components are a set of program elements or *program fragments*. Fragment boxes are collections of code templates, i.e. containers for code fragments. A system, built up with ISC, is parameterised with *hooks*. These are points of variability of a fragment box, a set of fragments, or positions that are subject to change. There are two types of hooks - a *code hook* and a *position hook*. A code hook is a hook that only contains program elements. This set may consist of placeholders for other syntax elements. Position hooks consist of positions in the component's code, which may or must be filled in during composition. It is possible to distinguish hooks as *implicit* and *declared*. An implicit hook is a set of program elements or positions that are contained in every component by definition of its programming language. A declared hook is a subset of program elements or positions on a fragment box that have been declared to be subject to change.

The ISC provides the connection and extension of software systems with the help of Composers. A composer transforms components invasively. Aßmann gives the following definition of a composer: *"A composer is a program transformer that transforms one or more hooks for a reuse context."* Composers may be defined using object-oriented technology which makes the development of composers coherent.

2.9.2.1 Similarities and differences

S1: The ISC applies code templates which seem to be quite similar to the parametric code templates of my approach.

S2: The ISC also applies object-oriented technology to make development coherent. Code fragments and composers provide configurability at the program code (implementation) level.

D1: The ISC approach concentrates more on the composition of fragments oriented to the expert in programming languages and architectures. There is no evidence regarding the externalisation of requirements up to the level of domain expert found in the literature. Unlike my approach, the ISC do not provide concrete mechanisms of configurability at the high domain-specific level.

D2: Additionally, the ISC does not provide context awareness of code templates.

2.9.3 Jenerator

Voelter and Gaertner in [60] introduced a Jenerator approach for extensible code generation for Java. By using its extension mechanisms, complete high-level, product-line specific generators can be built, enabling an automated creation of systems on a source-code basis.

Jenerator provides classes to create classes, methods, members, interfaces, etc. By applying object-oriented techniques such as inheritance and delegation to Jenerator classes, higher level, domain-specific generators can be implemented and used.

The Jenerator is defined by the following extension techniques:

1. Subclassing: By subclassing generator classes, higher level, domain-specific generators can be created.

2. Parametrisation: By parameterizing the generator classes, the behaviour of a generator can be controlled.

3. Delegation: Generator classes can use each other, creating more complex results.

Additionally, the Jenerator approach defines techniques such as Macros and Classgroups. Macros is a generator which can be applied to a class, doing a specific modification. It can also act on parts of the class, such as methods. Classgroups allow for the grouping of related classes and applies macros to all of them.

2.9.3.1 Similarities and differences

S1: As with my approach, the component model of Jenerator is defined with the same aim - to manipulate program code structures.

D1: However, the Jenerator approach does apply to a relatively simple component model. The possibility to create simple code structure is mentioned, but it does not define how these could be related together in the form of a code template. As such the component model in Jenerator is not well suited for the encapsulation of program code templates within the fashion of how it is done in my approach.

S2: Moreover, the Jenerator approach applies object-oriented technology to form components, thus winning features such as categorisation and easier development.

S3: In introducing subclassing, Jenerator enables the developing of a domain-specific composition system.

D2: No concrete requirements for externalisation mechanisms are provided from the other side.

S4: It is mentioned [60] that generator classes can be parameterised. We assume this results in the feature of configurability at the implementation level. We found the evidence provided by authors was not strong enough and rather abstract.

D3: The Jenerator approach does not define an explicit component model to encapsulate program code templates as well as composition techniques for their composition.

2.9.4 Visual Tool for Generative Programming

Grigorenko et al. [41] introduced the concept of a visual tool for generative programming. The tool demonstrates a way of combining object-oriented and structural paradigms of software composition. Metaclasses are introduced. These are components with specifications called metainterfaces. According to [41], metaclass is a source of classes that can be synthesised from its metainterface and its methods. Metainterfaces are wrappers that provide flexibility to classes and contain information about their usability.

Automatic code generation is used that is based on the structural synthesis of programs. This guarantees that problems concerning the handling of data dependencies, order of the application of components, usage of higher-order control structures etc. are handled automatically.

The approach is oriented towards two user groups. A Language Designer is responsible for writing the necessary Java classes and metainterfaces, merging them into metaclasses and then adding visual extensions. A Language User, who does not have to be a software expert, works with the defined visual language, arranging and connecting objects to create a scheme. Manipulating the scheme - a visual representation of a problem - is the central part of user's activities. After a scheme has been built by the user, the following steps remain: parsing, planning and compiling are automatically taken by the computer.

2.9.4.1 Similarities and differences

S1: The component model of the approach as described by Grigorenko et al is based on metaclasses and metainterfaces. The component model and the composition technique is described rather abstractly, with no explicitly defined specifications of how classes are actually manipulated and synthesised. Therefore, we assume that configurability at the implementation level is provided by the approach.

D1: However, the component model is not suited for the definition of code templates, context awareness mechanisms and operations to compose code templates.

S2: The Visual Tool for Generative Programming provides a basic externalisation mechanism. It is expressed with visual classes that are formed by adding visualisation information to the meta-classes.

D2: However, the approach does not porvide any details regarding the kind and/or model of visualisation information. With no ontology support, the externalization approach is rather simple.

2.9.5 Jacob: Configuring Component-Based Specifications for DSLs

Pfahler and Kastens [70] introduced the "Jacob" system for configuring component-based specifications for domain-specific languages. The "Jacob" system supports a language design process on a very high level of abstraction, enabling experts from application domains to design their own domain-specific languages. A language design is specified by composing and configuring language components. These components are provided by an expert in the field of computer languages and their implementation. Such components consist of two parts: the implementation part and the interface part.

The Jacob approach introduces the principle of bringing more ownership over the design process to the end-user. More specifically, the end-user should own the language implementation to be able to make his own modifications independently from the manufacturer of the language implementation. A domain-specific language can be described as a composition of DSL components. Being made configurable, DSL components may form a whole family of domain-specific languages.

2.9.5.1 Similarities and differences

S1: The Jacob approach explores the topic of externalizing business logic so that using appropriate notations, domain experts can design software by themselves. Jacob encapsulates implementation constructs in DSL components which, when being configured, can form further DSLs.

S2: Language specifications behind DSL components are produced according to the Eli language implementation system [90]. Patterns or templates in Eli are defined with the help of regular expressions. It is possible to say that with Eli, the basic configurability at the program code level is provided.

D1: In comparison to my approach, this is quite a reduced form of templates as, for example, no dependencies between templates can be captured.

2.9.6 Design Components

Keller and Schauer in [53] tried to overcome problems with design patterns during their implementation and maintenance by suggesting a more systematic approach to define, implement, and trace design patterns within a component-based development cycle. They introduced the core idea of the creation of software architectures which are based on well-defined and proven design patterns packaged into tangible, customisable, and composable *design components*. More specifically the design component was defined as follows:

A design component is a reified design pattern that allows for design composition through instantiation, specialisation, alteration, adaptation, provision, and generation. A structure of design components is defined with *constituents*, *notation*, *services* and *instantiation modes*.

Constituents are mainly determined by the constituents of the underlying design pattern. The list of constituents is adopted from Gamma [36] and extended with constituents for source code, executable code, access privileges, and design history.

The informal constituents, such as intent, motivation, or consequences, are described with plain text and graphics. Keller and Schauer have suggested HTML as the document standard. The formal constituents, such as structure, collaborations, and implementations are desrcibed with UML which is extended a little bit.

Design components should provide the following services:

1. Persistence of the constituents is managed in a repository.

2. Inspection exposes the constituents of the design component to the outside world. This service provides a graphic front-end for design components.

3. Event handling allows design components to interact with each other.

4. Security manages access privileges to design components and their constituents throughout the composition process.

5. Code generation provided plug-in interfaces for both a source code generator, and a compiler, which generates executable code.

6. Integration is the mechanism that allows external tools to use design components.

Design components can be instantiated and customised to meet given requirements. Two modes of instantiation are defined: remote and local.

The composition of design components is based on *reification*, *instantiation*, *specialisations*, *adaptation*, *assembly*, *alteration*, *provision* and *generation*. Design composition starts with a set of design patterns which are reified as design components and maintained in a repository. Instantiation provides a new instance of a design component for specialisation, alteration, adaptation, or assembly with other instances. Through specialisation, hierarchies of abstract and concrete design components can be created. Adaptation tailors generic design components to meet the needs of the specific demand on hand. Assembly increases the scope of a design solution by joining basic design components. Alteration creates a new revision of a design component, leading to hierarchies of original and current design components. Generation transforms the design component instance into source code frames or an executable class hierarchy stored with the component. Provision allows for evolving models to be shelved in the repository. There is no restriction about the level of concreteness, specificity, scope, and revision of design components.

2.9.6.1 Similarities and differences

S1: The approach tries to manage design patterns with design components. The coherent development of design patterns through adaptation and specialisation is introduced.

S2: A requirement such as event handling is mentioned. We consider, principally, that this can lead to the context awareness feature.

S3: As with my approach, the Design Components approach introduces a component model that may capture with design patterns.

D1: However, the description technique is based on UML, which does not provide many features defined by my approach (i.e. dependencies between templates, structure/-value configurability and nested templates).

S4: The design components can be connected with GUI components which play a front-end role. We consider that this can be seen as a very simple case of externalisation via visual language. My approach has a more powerful externalisation mechanism that is based on ontologies and a defined component model for GUI components from different

visualisation libraries.

D2: The Design Components approach does not explicitly specify a composition technique to compose design patterns to form a software system at the implementation level.

2.9.7 Results of Comparison Analysis

Results of the comparison analysis are collected in the form of a table (see Figure 2.26). The first column lists approaches which have been compared. The first row specifies fields which denote criteria (features) for comparison. Fields coloured in yellow are of particular interest, as they represent the core difference between my approach and the other approaches. A green "check" sign denotes a feature which characterises an approach specified in the first column. A black field shows that a specified feature is not supported by an approach specified in the first column.

Approach	Externalization		Component Model (Language Level)			Composition Technique (Language Level)	OO Technology (on the meta level)
	Visualisation Layer	*Target Domain Layer*	*Code Templates*	*Context awareness*	*Configura-bility(impl.)*		
My Approach	✓	✓	✓	✓	✓	✓ Molecular Operations	✓
Configuring CB specifications for DSLs with Jacob (Pfahler, Kastens)		✓⊖	✓⊖		✓⊖		
Invasive Software Composition (Aßmann)			✓		✓	✓ Composers	✓
Design components (Keller)	✓⊖			✓⊖	✓⊖		✓
Visual tool for generative programming (Grigorenko et al)	✓⊖	✓⊖			✓⊖		
Jenerator (Voelter and Gaertner)		✓⊖			✓⊖		✓

Figure 2.26: Comparison of related approaches

2.10 Conclusion

We have reviewed such growing sub-fields of software engineering as composition systems - systems that concentrate on the composition of components. Composition systems extend the classical understanding of a component established with the early component-based development approaches, when a component is seen as a black-box service provider. Instead, composition systems see a component as a software part that must be composed with other components in a system composition to form a final system. Hence, a software component is simply a software item that is subject to software composition. It may appear on different levels of granularity, may be a design or implementation item, and may be in source or binary form.

The review has presented black-box and grey-box composition approaches. Black-box approaches apply black-box units of composition. Basically, these units are characterised by the described (visible) behaviour, but not by the invisible source structure. Black-box composition approaches, i.e. classic EJB components, have already been successfully applied in industry for the past decade. Compared to black-box approaches, grey-box approaches increase parameterisation of software systems by applying grey-box components. In addition to black-boxes, grey-boxes are characterised by the visible source structure which is seen as a subject of manipulation. Grey-box approaches, i.e. Feature Oriented Programming, were mainly applied in academia in recent years. Higher parameterisation of grey-boxes results in higher reusability, extensibility and adaptability of components (and therefore a designed system). At the same time, new problems appear. We recognise and concentrate on the following main problem:

The gap between new approaches and the business domain.

Due to the nature of grey-box components, grey-box software composition approaches, introduce additional types of building units. For instance, for the AOP, such additional units are aspects, advices, pointcuts and join points defined on top of black-box components. During the design process for each kind of a unit there could be a number of exemplars created which are quite high. Hence, compared to black-box approaches, the complexity of a new software composition system is significantly increased. Therefore, it costs more to create and use grey-box components as quite specific expert knowledge is

involved. Problems with maintenance are the main obstacle in applying grey-box based composition approaches at a practical level by end-users in the business domain.

In this thesis we are stating, that to solve the gap-problem, there is a need to define mechanisms in order to bridge the gap between composition systems and end-users (representing the business domain). Those mechanisms should reduce the complexity of composition systems. Some kind of an interface to different domain experts should be provided. From one side, this interface should reflect the meaning of a composition by understandable means for different domain experts. From the other side, this interface should interpret actions performed by domain experts into terms that are understandable for the composition system. The interface should externalise well-defined requirements according to a software system which can be designed by the end-user who is a non-expert in programming languages. Based on what was written above, we specify a new additional requirement of externalisation to composition systems:

Composition systems should externalise business logic.

Additionally, as we consider it to be most relevant, we have introduced few related concepts. We consider that design patterns, presented by Gamma et al., introduced the way in which to think in patterns. This approach has influenced many of the grey-box approaches. Design patterns' approach has additionally stressed the feature of architectural descriptiveness and reuse. Design patterns have typically represented relatively abstract informal solutions that could be used by developers. Self descriptiveness was presented by the GOOD approach as one of the requirements to fulfil the demands of component-based software engineering today and in the future. The notion of externalization is crucial in GOOD; it is also crucial in our proposed approach. We have presented further concepts, such as DSLs, DSM and Geon Diagrams in order to show the general tools the domain experts (end-users) could work with. Domain-specific notations as well as information visualisation (or software visualisation) could indeed potentially take part in forming the interface to externalise business logic.

2.11 Summary

In this chapter, the field of software composition systems has been reviewed. This growing sub-field of software engineering concentrates on the composition of components. We

introduced a notion of a component. It was stated that components may appear on different levels of granularity, may be a design or implementation item and may be in source or binary form. We classified components according to their level of description and we recognised black-box, white-box, grey-box and glass-box components.

Furthermore, basic requirements to composition systems were formulated. Those requirements are the summary of requirements to three parts that constitute each composition system. These parts include the component model, composition technique and composition language. Meaningful terms - such as `modularity`, `parameterisation`, `extensibility` - that influence the quality of composition systems were presented.

The chapter also presented black-box and grey-box composition approaches. The review on composition systems helped to recognise some tendencies. It was discovered that traditional composition approaches are rather characterised by applying the black-box component paradigm. State of the art approaches tried to increase parameterisation of software systems by applying the grey-box component paradigm. We have recognised a gap between grey-box approaches and business and have proposed an additional requirement of externalisation to composition systems. We believe such a requirement points towards a decreased complexity of grey-box approaches, making them more applicable by end-users from the business domain. The following chapters introduce a proposed composition framework to define and apply grey-box composition systems with respect to the externalisation requirement.

Chapter 3

The Composition Framework

This chapter gives an overview of the composition framework which is the main contribution to this thesis. We called it the Neurath Composition Framework. The framework specifies mechanisms for the definition of template-based software composition languages and for the externalisation of business requirements up to the level of domain experts. The architecture of the composition framework and the life-cycle of the development process it defines are explained. The main concepts and their collaborations are generally described. Moreover, we give an overview on the organisation of the reference implementation for the composition framework.

3.1 Introduction

In this section, we present the Neurath Composition Framework. This framework is the main contribution in this thesis. The NCF defines a methodology to specify template-based software composition systems and define domain-specific visual languages on top of these systems. With this technology domain experts could domain-specifically and visually design and maintain the program code of a designed software system.

The word "Neurath" in the framework's name comes from researcher's name Otto Neurath. One of his research interests that we have been inspired by was the narrative expressiveness of visual material. In 1936, he introduced the word *Isotype*. He defined the Isotype (International System of Typographic Picture Education) as a set of pictographic characters used "to create narrative visual material, avoiding details which do not improve the narrative character" [64]. Examples of pictographic symbols based on Isotype include pictograms for a cafeteria, a luggage check and a maintenance station; they are often seen in public places in our everyday life (see Fig. 3.1).

Figure 3.1: Pictograms

Isotype is a method with a special visual dictionary and a special visual grammar; that is, a new visual world which is comparable to our book and word world. In this symbol language charts, pictures, models, movies, games and illustrations can, with a little related text, show the main facts and explain the important problems in any field of knowledge [80]. Otto Neurath wrote, "The first step in Isotype is the development of easily understood and easily remembered symbols. The next step is to combine these symbolic elements". For example, there is a symbol for "shoe" and another for "factory". By joining these two symbols to make a new one, we can talk about a factory in which shoes are made. With another combination of symbols, we can discuss shoes made by machinery and shoes made by hand. Similarly, we can add the symbol for "coal" to the symbol for "worker"; we can make also make an Isotype for mechanised mining and for pick mining. We can place symbols on a map to show geographical distribution, or range them in rows to express statistical relationships. The aim of the visual method introduced by Neurath is to humanise and democratise the world of knowledge and intellectual activity.

The NCF is to define and apply template composition software systems with respect to the requirement of the externalisation of business logic. Further, in Section 3.2, we specify more narrow and precise constraints of externalisation defined within the context of this thesis. Afterwards, Section 3.3 introduces concepts of templates and design components which represent a base of a template composition system. These two sections form the main requirements for the process of composition with respect to the requirement of the externalisation of business logic. After this, the Domain-Specific Visual Composition System (DSVCS) which should implement these requirements is presented in Section 3.6. In further sections, the conceptual framework to implement and apply DSVCSs, called Neurath Composition Framework, is explained. Section 3.8 introduces tools for practical realisation of the NCF. Finally, Section 3.9 gives a summary of this chapter.

3.2 Externalisation of Business Logic

In the conclusion of Chapter 2, we introduced an additional requirement to the composition system:

Composition systems should externalise business logic.

In order to define an architecture of a composition system which meets this requirement, it is necessary to define more precisely what the composition process with externalised business logic is.

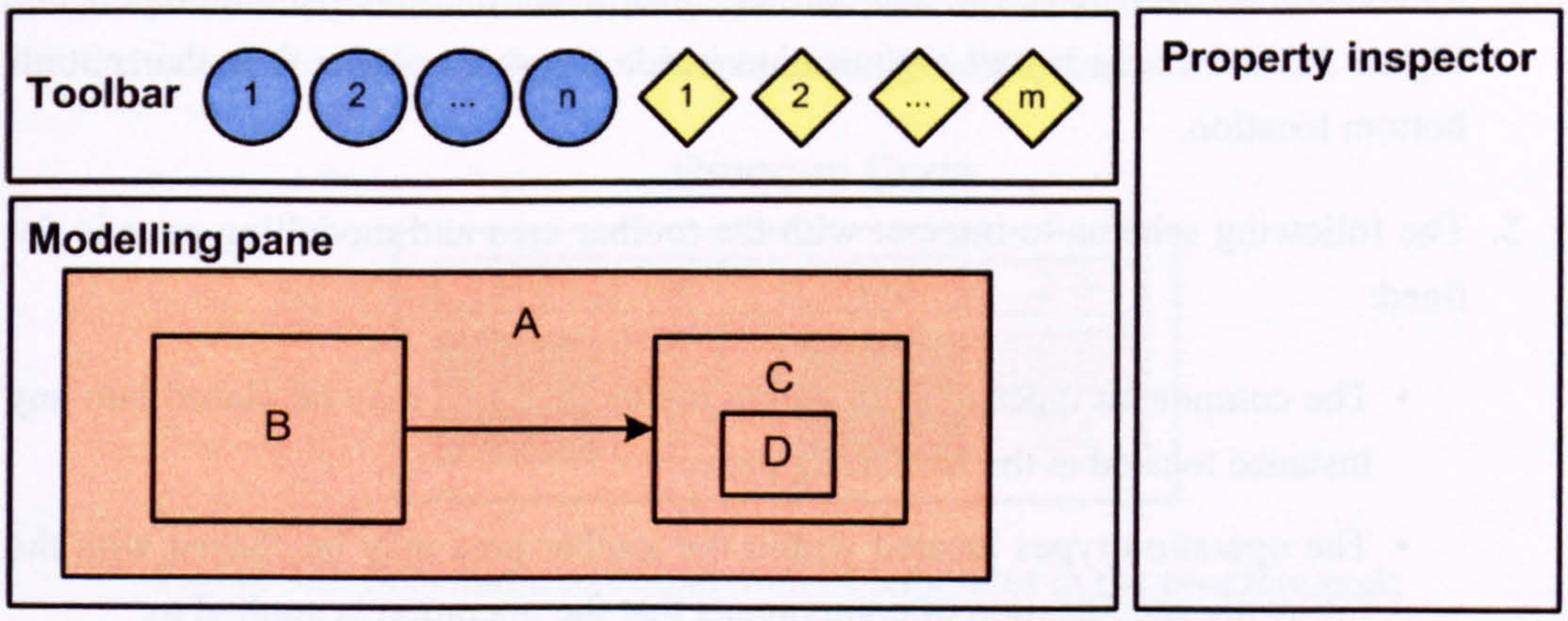


Figure 3.2: Design Environment oriented to Domain Experts

We define the following requirements for the externalisation for the case of a domain-specific composition system:

1. The visual design environment consists of elements such as a *toolbar area*, *modelling pane area* and *property inspector* (see Figure 3.2). These elements are just basic ones which are relevant here.

2. The toolbar area contains a repository of visual components (icons) that represent components and operations to design a software system within the defined domain. Figure 3.2 shows components as circles and operations as diamonds. The components represent domain-specific terms that the designed system may consist of. The operations represent rules to manipulate with components. A toolbar area contains type definitions of components and operations.

3. The modelling pane area domain-specifically reflects a state of a designed system. It contains visual components that represent the terms which the system consists of. The modelling pane contains instances of components. Figure 3.2 shows components A, B, C and D.

4. Visual components placed in the modelling pane are characterised by *linkage* with other components, *containment hierarchy* and *layout*. The linkage is a graphical relationship between two visual components. Figure 3.2 shows graphical linkage between B and C. The containment hierarchy is formed by visual components that are nested inside other visual components. Figure 3.2 shows the visual components B and C nested in A, and D nested in C. The layout of a visual component is defined by constraints to manage the location and size of the internal visual components. Figure 3.2 shows the layout organisation inside C, which places D at the central-bottom location.

5. The following schema to interact with the toolbar area and modelling pane is defined:

   - The component types located within the toolbar area may be placed into any instance located at the modelling pane.
   - The operation types located within the toolbar area may be chosen with the following specification of components that the operation is applied to.
   - Information about the chosen instances in the modelling pane is shown in the property inspector.

6. The interaction steps done by the Domain Expert cause transformations of the underlying program code. The resulted in program code should have the structure and the behaviour as according to the expectations of the Domain Expert. The state of the program code should be visually reflected in the modelling pane.

According to the specified requirements, if the Domain Expert has a repository of components and operations, it can visually (and domain-specifically) transform the program code of the software system being built. A structure and behaviour of the designed software system is a subject of manipulation. There is a need to use grey-box components in order to introduce a parameterisation of the structure of a composition unit. We defined templates as grey-box units of composition together. Additionally, we introduce components which we call *design-components*. These incorporate knowledge about how templates can be composed. The next section gives an overview about templates and design-components.

3.3 Templates and Design-Components

A template is a group of program code fragments which are combined (merged) together in order to implement a specified feature or behaviour of a software system. Figure 3.3 shows the schematic representation of templates in the program code. The program code consists of 12 composition units. Each composition unit is signed with two numbers. The first one refers to the type of template. The second number refers to an instance. The figure shows three different types of templates and seven instances.

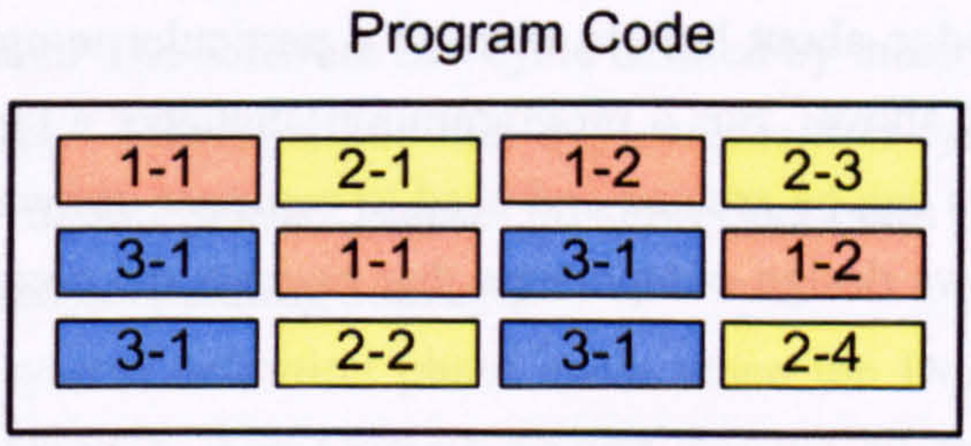


Figure 3.3: Schematic representation of templates in the program code

We define the following requirements for templates:

- *Modularity*. A template is a module or unit of composition that has a well-defined interface by which it can be configured and analysed.

- *Compositionality*. Templates can be composed (combined) together in order to form new templates.

- *Parameterisation*. The structure of the template, as well as the values carried by the structure, are subjects of parameterisation. With the help of the parameters provided, a template can be configured (configurability).

- *Reactivity*. Changes that occur within a template or outside of it may initiate a reaction of that template in order to adapt itself to these changes.

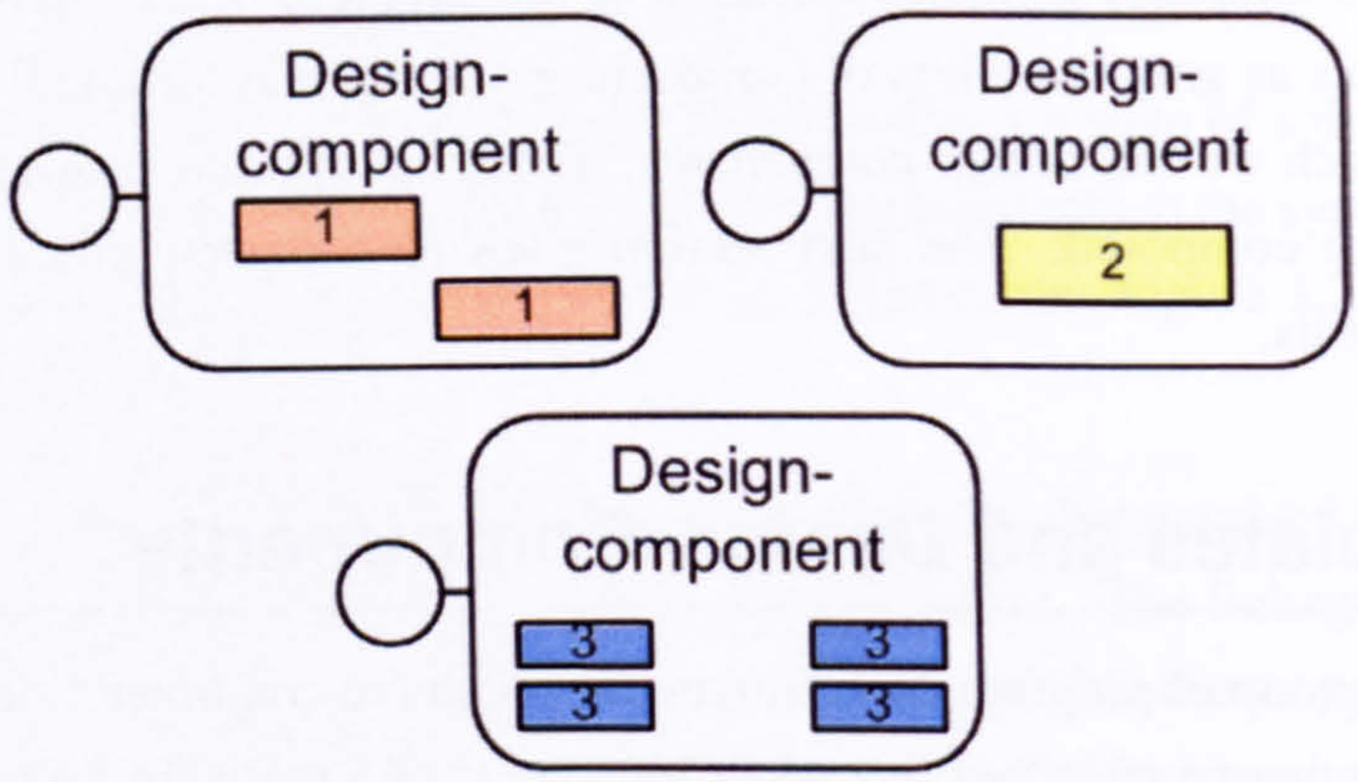


Figure 3.4: Schematic representation of design-components that manage particular template types

We defined the term *design-component*, a run-time unit working during the design time when the program code is composed with templates. A design-component incorporates all the knowledge about how to manage a particular template according to the requirements specified above. For a programming language, a repository of templates and design components can be created and used to compose software systems with templates. Figure 3.4 shows design-components that manage the composition of templates shown in Figure 3.3.

3.4 Domain-Specific Visual Composition System

We have specified the main requirements for the process of composition with respect to the requirement of the externalisation of business logic. Once implemented, these requirements form a *Domain-Specific Visual Composition System*. This system is based

on two main functional blocks: (1) *Externalisation System (ES)*, which is defined on top of the (2) *Template Composition System (TSC)*. Figure 3.5 shows this.

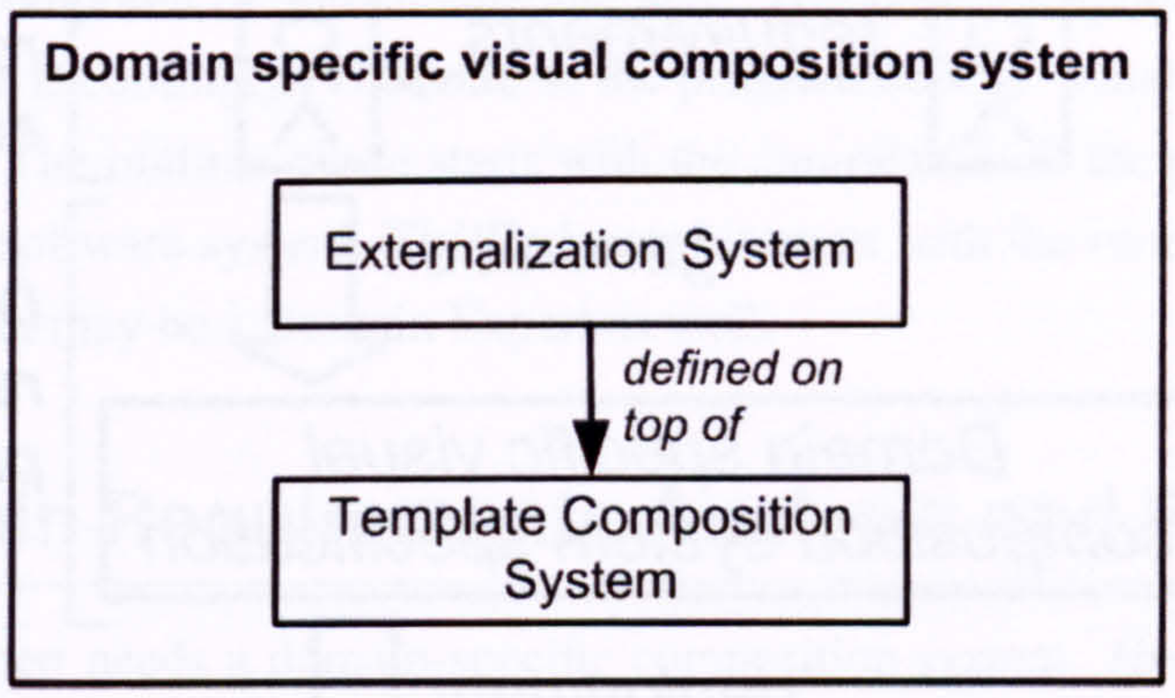


Figure 3.5: Domain-Specific Visual Composition System

The ES incorporates the requirements specified in Section 3.2. The ES is needed to provide a visual reflection of the structural and behavioural properties of the underlying designed system. Moreover, the ES is a facility that the Domain Expert interacts with to design a software system. The TCS provides a component model, a composition technique and a composition language to compose the program code of the software system with templates.

3.5 Software Life-Cycle

With the DSVCS, a domain expert can visually design a software system within the defined application domain. The software life-cycle defined by the composition framework consists of three phases. These include the: (1) *composition system definition phase*; (2) *design phase*; (3) *runtime* phase. Figure 3.6 shows the main phases of the software life-cycle and the related cooperating concepts.

The composition system definition phase starts when the Domain Expert specifies requirements for the DSVCS. The Domain Expert may be an expert in the particular application domain the software system has to be designed for. Basically, the requirement specification may include descriptions of domain-specific terms and relations that may form the designed system.

The Software Architect, an expert in software architectures, programming languages and templates, receives the requirement specification and performs the *domain require-*

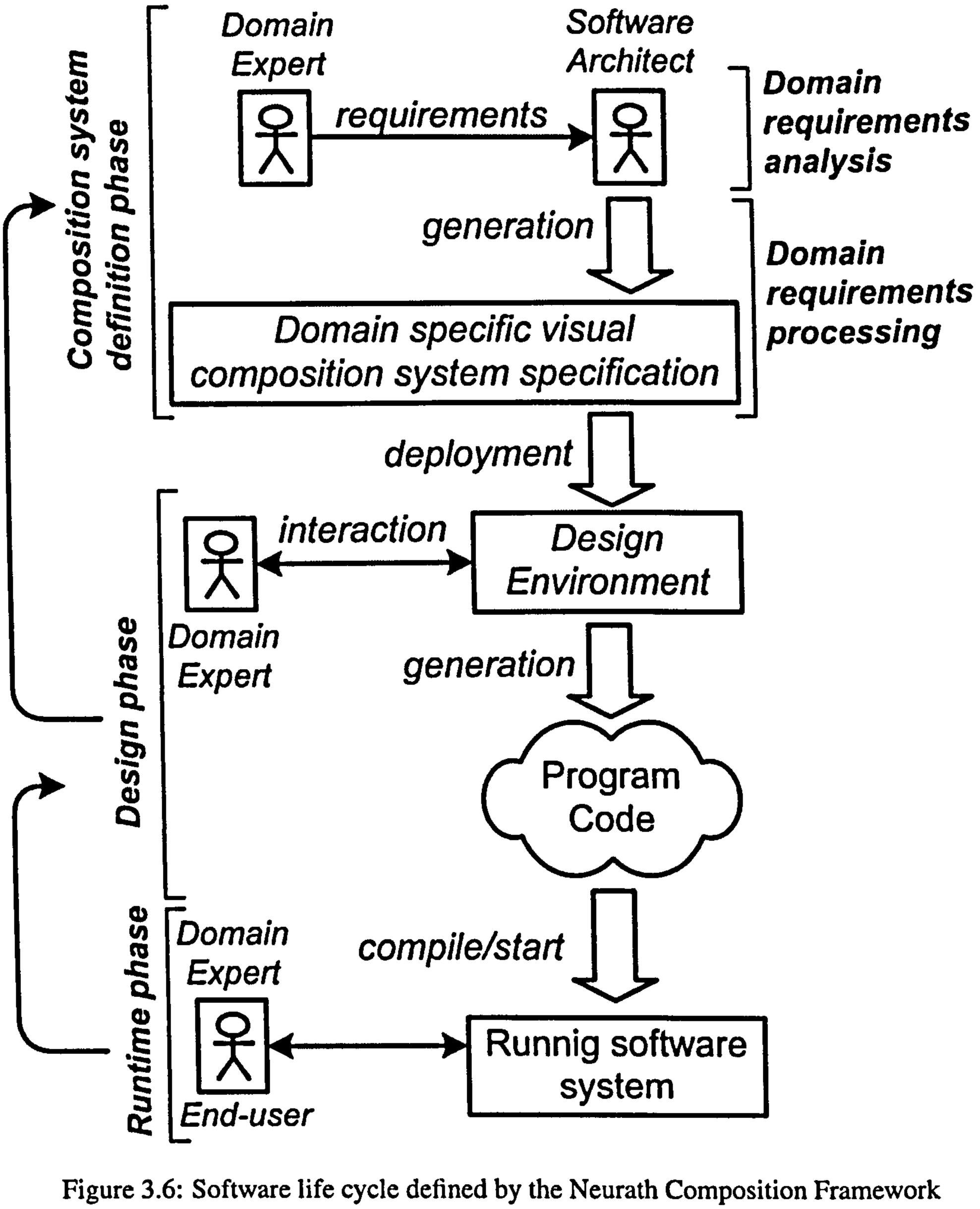


Figure 3.6: Software life cycle defined by the Neurath Composition Framework

ments analysis and *domain requirements processing* routines. He/She produces the specification of the DSVCS.

The produced specification is deployed into the design environment. With the deployment, the design phase starts. At this phase, the Domain Expert designs a software system

with the help of the deployed DSVCS. The process of design can be marked with the word "interaction". This assumes that the Domain Expert "tells" the system how the program code should be composed in domain-specific way. The system responds and transforms the program code accordingly. The state of the program code is visually reflected in the modelling pane. The runtime phase starts with the compilation of the program code and execution of the software system. The End-user interacts with the running software system. The End-user may be a Domain Expert as well.

3.6 Domain Requirements Analysis and Processing

The Domain Expert needs a domain-specific composition system. He/She requests that the Software Architect establishes such a system. Together, they form the requirements that describe how the system should be designed, which includes the following aspects as defined in terms of an application domain:

1. Structure statics and dynamics
2. Behaviour statics and dynamics
3. Appearance statics and dynamics

More specifically, the requirements specification may contain the following:

1. What domain-specific terms may form a software system?
2. What are the possible relationships between these terms?
3. When and how are these relationships established?
4. What is the visual appearance for the domain-specific terms?
5. How is the visual appearance changed with the state of a designed system?

Usually, the requirements specification is expressed in details, albeit quite informally, i.e. in the form of English text. However, to ease the development, a more structured form of description is awaited. Here, we propose the following strategy to describe the requirements to design within an application domain:

1. **Requirements as English text:** the requirements specification in the English language.

2. **Domain Ontology:** statics of the application domain are described with the Domain Ontology. Ontology is a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for a particular topic [44]. The domain ontology provides a vocabulary for referring to the terms in a subject area as well as a taxonomy that is a hierarchical categorisation of entities within a domain. Domain ontology is similar to the UML class diagram and can be described in terms of this diagram. We use notation, as shown in Figure 3.7.

3. **Term-English table:** this is a table showing relationships between the domain-specific terms defined by the ontology and the requirements specification parts written in English. The Software Architect may add additional requirements which must not affect the requirements provided by the Domain Expert. The fields of the table are the *Term* and the *Description.*

4. **Relation-English table:** this is a table showing relationships between the domain-specific relations defined by the ontology and the requirements specification parts written in English. The fields of the table are the *Relationship* and the *Description.*

5. **Actions-State table:** this is a table showing dependency between the sequences of actions and changes in the system's state. By default, such sequences of actions are defined as: (1) *click component type* and then *click instance*; (2) *click operation type* and then *click component instance*. The fields of the table are the *Actions*, *State 1* and *State 2.*

6. **Types/instances-appearance table:** this is a table showing dependency between domain-specific elements (types and instances taken from the Action-State table) and visual appearance at the modelling pane. The following fields should be provided: *Term*; *Appearance* (graphic); *Kind* (instance or type); and by demand, the *Description* field.

7. **ORel-GRel table:** this is a table showing the dependency between relationships defined by the domain ontology and relationships describing graphical appearance. The fields of the table are *Ontology Relation* and *Graphic Relation.*

8. **Layout Strategy table:** this is a table that specifies the layout constraints for the containers that are defined by terms from the Term-Appearance table in English. The fields of the table are *Term* and *Description.*

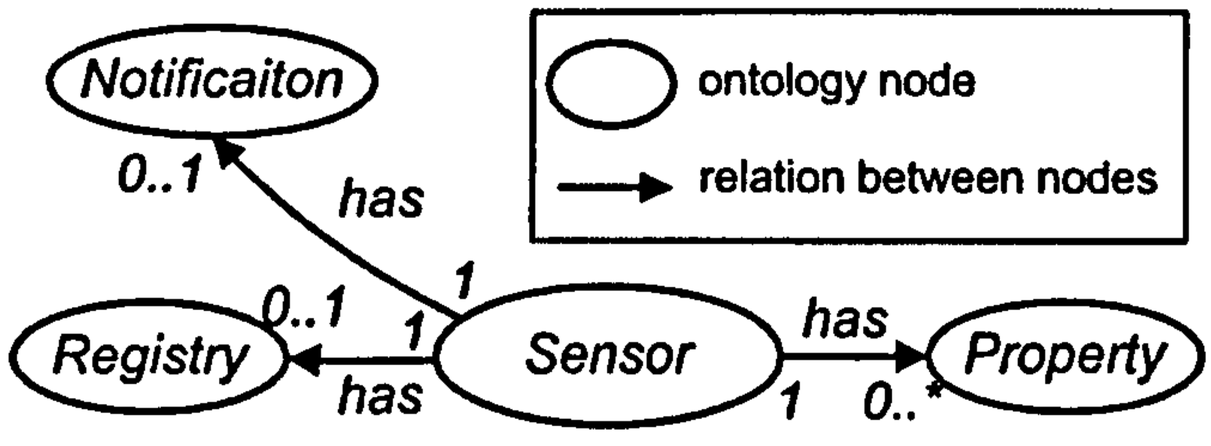


Figure 3.7: Example of a Domain Ontology specification

When all requirements are specified, the Software Architect starts processing the requirements and developing the domain-specific visual composition system. The NCF that specifies how such a system is defined and applied. Further, we explain the architecture of the NCF.

3.7 Architecture

In this section, we explain the architecture of the NCF. The NCF specifies how the DSVCS is defined and applied. We distinguish between four groups of concepts that form NCF. Each group is defined at one of the following levels: *Atomic Level*, *Template Level*, *Target Domain Level* and *Visualisation and Interaction Level*. Before we describe these levels, we are going to explain the criteria for the provided grouping.

Each group specifies how the composition system is defined at a different level of *granularity* and *domain orientation*. These are two criteria according to which the groups of concepts that form NCF are defined.

The granularity of a composition system is a measure of the size of composition units that make up a system; it can vary from fine-grained to coarse-grained. The domain orientation denotes what application domain the units of composition represent. We distinguish between two values of the domain orientation: (1) language domain; (2) application domain (or target domain). The language domain represents the concepts of software architectures, programming languages and templates. The Software Architect who is an expert in the language domain. The application domain represents the concepts of the particular domain that the DSVCS is designed for. The Domain Expert is an expert in the application domain.

The following levels of composition are defined:

1. Atomic Level. It collects concepts to encapsulate program code as related atoms of the programming language. These atoms are terminals and non-terminals of the BNF grammar specification.

2. Template Level. It is built on top of the Atomic Level and collects concepts describing the specification and application of template-based composition systems. At the template level, a program code is seen as a composition of code templates.

3. Target Domain Level. It collects concepts that describe bi-directional translation between a terminology of the template composition system and a terminology of different application domains. At the template level, a program code is seen as a domain-specific expression.

4. Visualisation and Interaction Level. It collects concepts for describing the visualisation of terminologies of different application domains. Moreover, it collects concepts for interpreting designer's actions over the visualisation and forwards them back to the Target Domain Level.

Figure 3.8 shows levels of composition and specifies the main activities defined at each level.

Further, we explain cooperation between the levels of composition for each phase of the life-cycle defined by the NCF. Afterwards, we explain each level of composition separately.

3.7.1 Composition System Definition Phase

During the composition system definition phase at all levels of composition, a composition language, called Neurath Modelling Language (NML), is formed (see Figure 3.9). This language can be seen as a visual domain-specific language based on the template composition system. During a further design phase, the NML is used by domain experts to visually compose the program code of a software system. The result of processing the sentences of the visual language is a Domain-specific Visual Interface (DSVI). This interface is needed for the reflection of the state of the system's design and for the interpretation of further designer's actions.

Figure 3.9 explains the nature of the Neurath Modelling Language that resulted at the composition system definition phase.

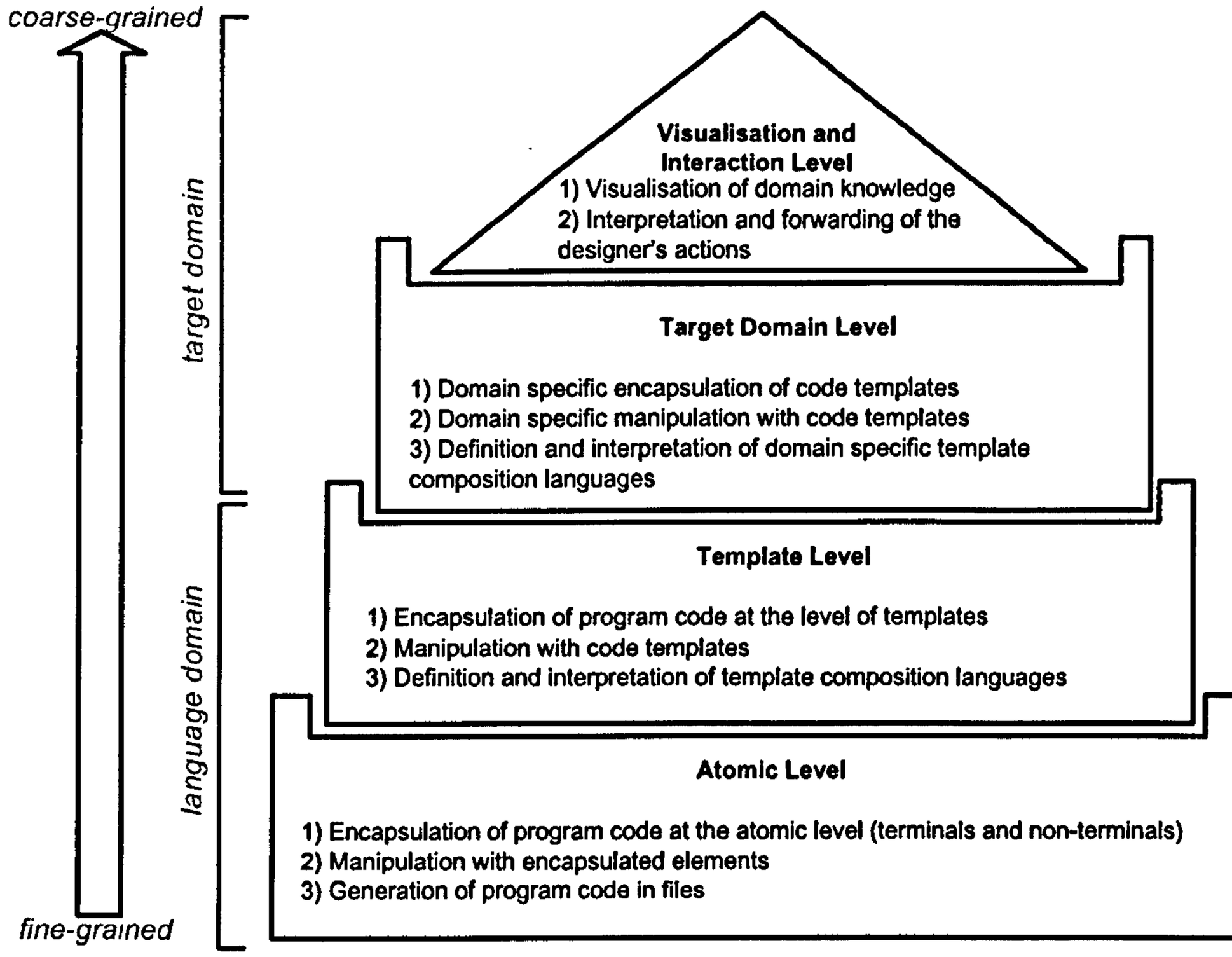


Figure 3.8: Levels of composition within the Neurath Composition Framework

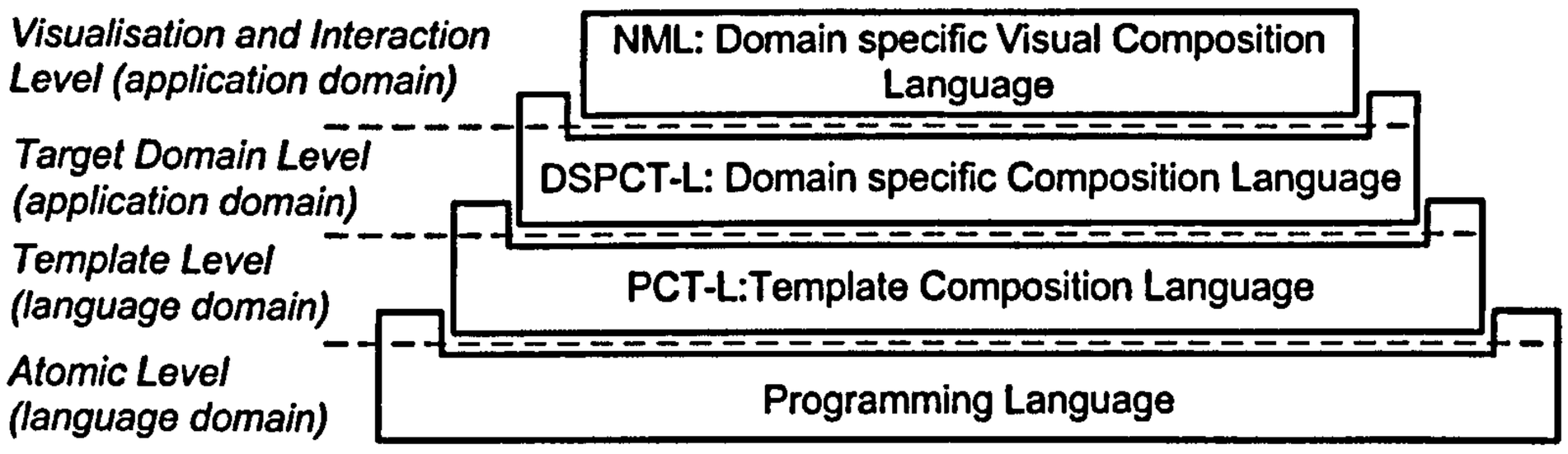


Figure 3.9: The nature of the Neurath Modelling Language

The figure shows four languages that belong to four levels of composition. A Programming Language is a base for the PCT-L template composition language. The Programming Language belongs to the Atomic Level of composition, where components are distinct language constructs, or - more specifically - terminals and non-terminals of the language grammar. The Template Composition Language belongs to the Template Level

of composition, where a component is a template-like structure formed with one or more groups of related atomic components.

A Template Composition Language is a base for the domain-specific composition language, called Domain-Specific PCT-L (DS-PCTL). The DS-PCTL belongs to the Target Domain Level of composition, where components represent domain-specific terms mapped onto the templates of the PCT-L. The DS-PCTL is a base for the domain-specific visual language, called Neurath Modelling Language. The NML belongs to the Visualisation and Interaction Level, where a component is a piece of domain-specific visual interface that has a visual representation mapped onto components of DS-PCTL; it can interpret actions done by designers.

Moreover, Figure 3.9 presents two main kinds of domain at which composition languages are defined and applied. The first one is the language domain. This domain defines a terminology for Software Architects who are experts in programming languages, software architectures and program code patterns. The second kind is an application domain. This can be any possible application domain for which business logic should be externalised. This domain defines a terminology for an expert in this domain, e.g. Security Manager, Supply Chain expert and Chemist. More concrete descriptions of languages defined at each level of composition and the components they operate with are given in Chapters 4,5, 6 and 7.

Further, more details of cooperation between different levels of composition during the composition system definition phase will be shown. Figure 3.10 shows interrelationships between different levels of composition.

Initially, at the Atomic Level of composition, atoms and atomic operations have to be provided. These atomic elements typically represent constructs of underlying programming language, dependencies between these constructs and rules to manipulate these constructs. Atoms and atomic operations are building blocks at the Template Level. At this level, repositories of templates and operations, called molecular operations, are provided to manipulate with these templates. Templates and molecular operations are formed with atomic elements.

The figure depicts Atomic and Template Levels of composition as levels that belong to the Language Domain. A Software Architect works within the Language Domain at the Atomic and Template Levels. The work at the Template Level results in template composition languages that can be used to form a program code, written in the underlying programming language.

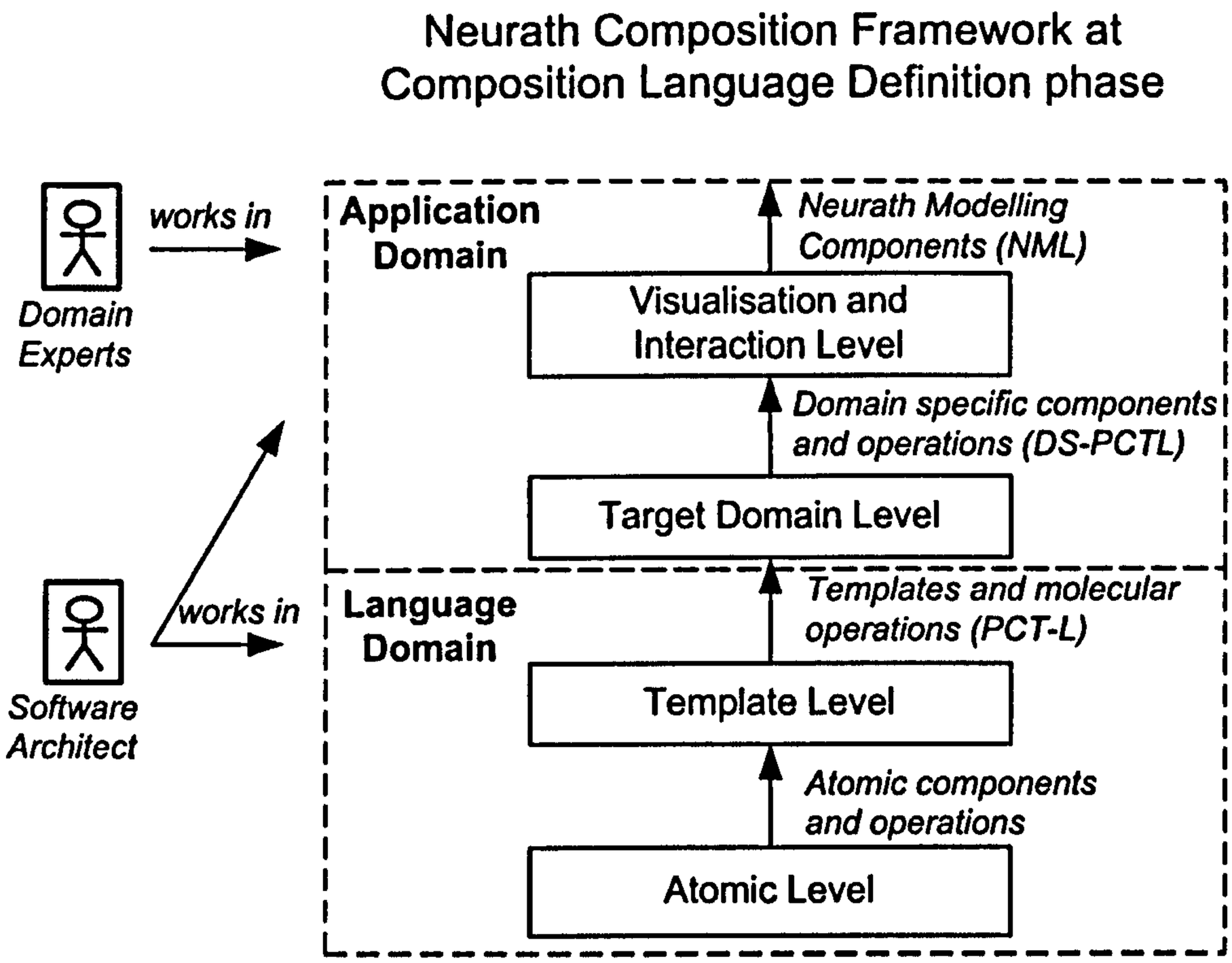


Figure 3.10: The NCF at the composition system definition phase

Templates and molecular operations can be used to develop a wide range of software systems. At further levels of a composition template, composition languages are externalised up to the level of domain experts. At the Target Domain Level, templates and molecular operations are extended into domain-specific components and operations that form domain-specific language. This language can be used by domain experts to build program code with templates in a domain-specific way. Software Architects and Domain Experts cooperate in order to create the right domain-specific components and operations.

The quality of externalisation is improved at the Visualisation and Interaction Level of composition. At this level, domain-specific components and operations are extended with domain-specific *visualisation* and *interaction* mechanisms. These mechanisms are incorporated in components called Neurath Modelling Components; they are the main production at the Visualisation and Interaction Level during the composition system definition phase. With NMCs, domain experts can visually design software systems for required application domains during the design phase. Visualisation strategies are necessary to build a visual model of the underlying model of the designed system during design time. The

visual model increases the perception and understanding of the designed system for different domain experts. Interaction mechanisms are necessary to interpret design-actions of domain experts in order to initiate composition processes at the underlying levels of composition.

3.7.2 Design Phase

During the design phase, the NML composition language is used by the Domain Experts to design a software system within the required application domain. They work at the Visualisation and Interaction Level of composition. Domain experts receive information about the designed system from the domain-specific visual interface that reflects the states of the system. Moreover, domain experts provide actions, describing - in a visual domain-specific way - what and how domain-specific parts should be composed together. The whole process of composition is supported by concepts defined at four levels of composition. Figure 3.11 depicts cooperation between the levels of composition during the design phase.

At the Visualisation and Interaction Level, actions coming from domain experts are interpreted into sentences of domain-specific language. This language was defined at the previous composition system definition phase. At the Target Domain Level, these sentences are translated into sentences of a template composition language which is also defined at the previuos phase. Further sentences of template composition language are translated into sequences of atoms and atomic manipulations that are processed at the Atomic Level of composition. This results in modifications in the underlying program code that describes a software system being built. Additionally, at the Target Domain Level of composition, the changed state of the system, expressed in domain-specific terms, is forwarded back to the Visualisation and Interaction Level during the translation. At this level, the state is translated into the domain-specific visual interface that reflects the state in a visual domain-specific visual. Moreover, the domain-specific visual interface introduces newly generated visual components (NMCs) that may be subjects of further designer's actions.

3.7.3 Runtime Phase

The Runtime Phase starts with the starting of the Software System that is generated from the design phase's resulting program code. A Software System can be of any type, i.e.

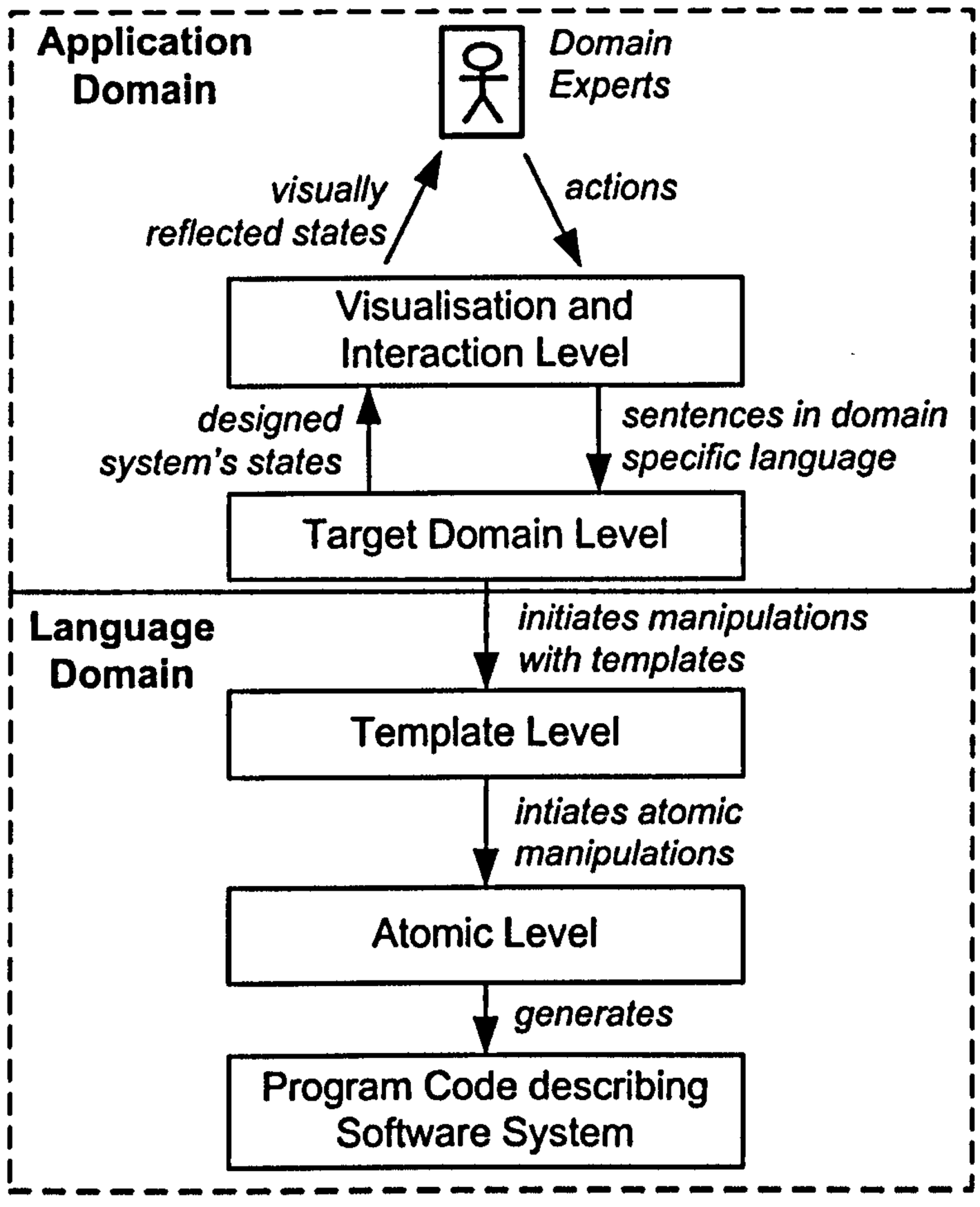


Figure 3.11: The NCF during the design phase

executable application, a component, a textual and visual applications. During the Runtime phase, the concepts defined at the Visualisation and Interaction Level may be useful for inspecting a state of the running system. Figure 3.12 shows a cooperation between concepts during the Runtime phase.

During the design phase, a state of the designed system, defined by structure and values, is reflected. During runtime, the system's state is a subject of change. The concepts defined at the Visualisation and Interaction Level during the design phase can be used to visually reflect states of the running system. Designers can use this feature to test and verify the system before supplying it to the end-user.

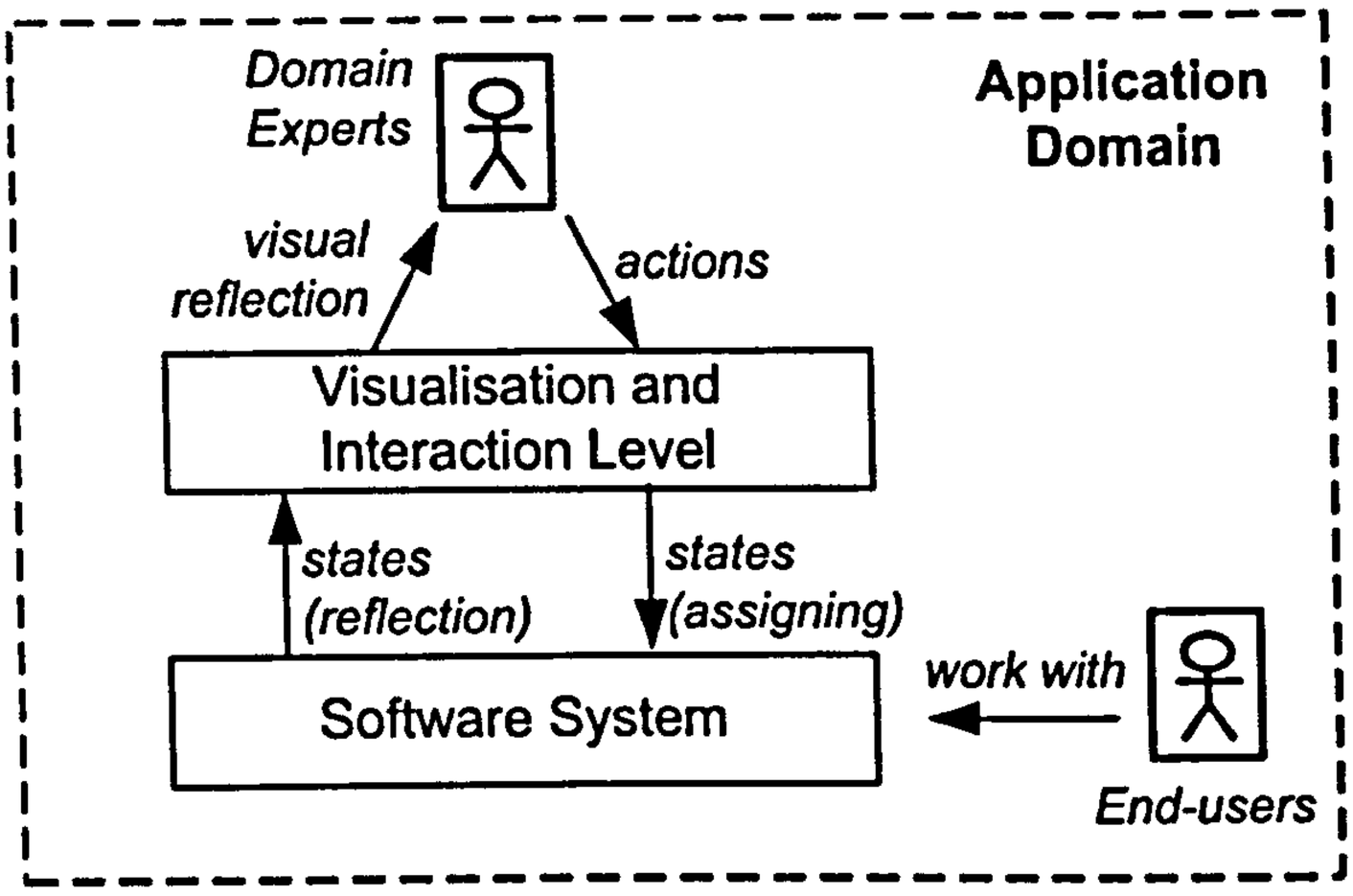


Figure 3.12: The NCF during the Runtime phase

In this thesis, concepts defined at the Runtime phase are out of scope.

3.7.4 Language Domain

At the Language Domain, two levels of composition are defined - the Atomic Level and the Template Level. Table 3.1 explains the activities at each level during the composition system definition phase and the design phase of the NCF software life-cycle.

Figures 3.13 and 3.14 show concepts defined at the Atomic Level and the Template Level at two different phases. Figure 3.13 depicts concepts that are relevant during the composition system definition phase.

At the Atomic Level - depending on the grammar of the programming language - a software system is going to be written in, with definitions of *Atoms* and *Atomic Operations* provided. Atoms represent terminals and non-terminals defined by the grammar of the programming language. Atoms can be manipulated with atomic operations. Examples of manipulations include creation, replacement, and the search and deletion of specified atoms. The main production at the Atomic Level is the definitions of atoms and atomic operations. These are material in order to form template definitions at the Template Level. Types of templates are defined according to the Parametric Code Template component model (PCT component model).

Phase	Level	Activity
Composition system definition phase	Atomic Level	Providing of atomic components and operations that are used to form compound components at the Template Level
	Template Level	Definition of new templates and operations to manipulate these templates or choose existing ones
Design phase	Atomic Level	Transformation of an underlying program code by the processing of sequences of atomic manipulations with atoms
	Template Level	1) Processing sentences of the template composition language 2) Forwarding requests of atomic manipulations to the Atomic Level 3) Holding the state of the designed system by means of templates

Table 3.1: Activities of the Atomic Level and the Template Level at each phase of the NCF software life-cycle

A Parametric Code Template is a name for a kind of component we define to encapsulate code templates. Moreover, at the Template Level, manipulation rules - called *molecular operations* - to manipulate templates are defined. The production at the Template Level is repositories of templates and molecular operations. These can be used by Software Architects for the template-based composition of program code. Section 3.7.5 explains how templates and molecular operations are used at the further levels of composition.

Figure 3.14 depicts concepts defined at the Atomic Level and the Template Level which are relevant during the design phase.

At the Templates Level, sentences of the template composition language are processed and translated into sequences of atoms and atomic operations. The elements of template composition language include templates and molecular operations. Sentences formed by these elements are expressions. When executed, they transform a designed system which is described as a superior template that contains other templates and interrelationships between them. At the design phase, the production of the Template Level involves sequences of atoms and atomic operations to perform concrete modifications of program

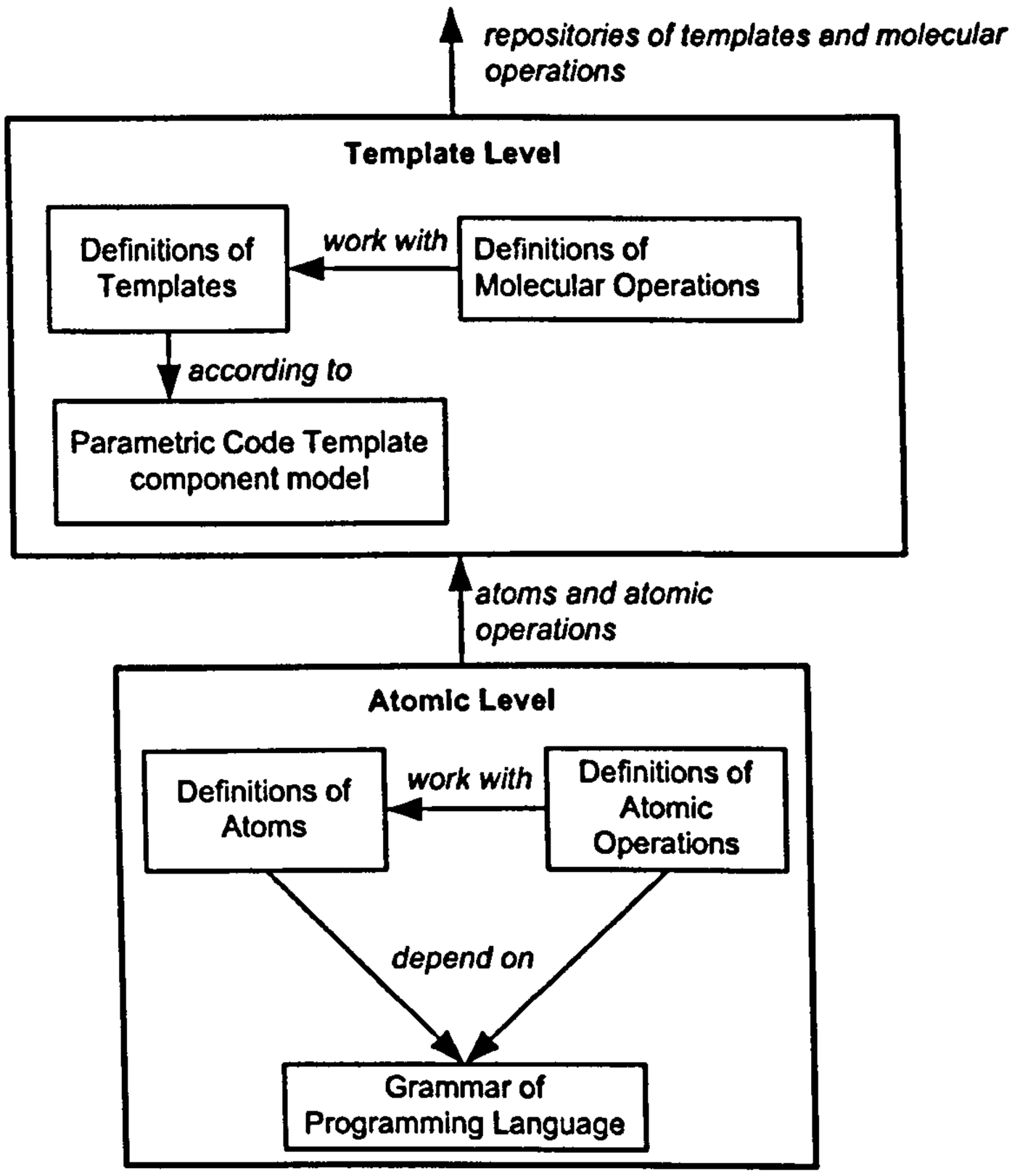


Figure 3.13: Concepts defined at the Atomic Level and the Template Level during the composition system definition phase

code at the Atomic Level. There, program code is described with the ASLT. Its nodes are atoms that describe constructs in the program code. The ASLT is a collaborative work, defined by Wolke [93, 92]. Currently, it is implemented for the Java programming language. With atomic operations, the nodes in the ASLT can be manipulated, thus transforming a program code at the atomic level.

3.7.5 Application Domain

At the Application Domain two levels of composition are defined - the Target Domain Level and the Visualisation and Interaction Level. These levels define concepts to externalise the business logic of software systems up to the level of domain experts. Tables 3.2,

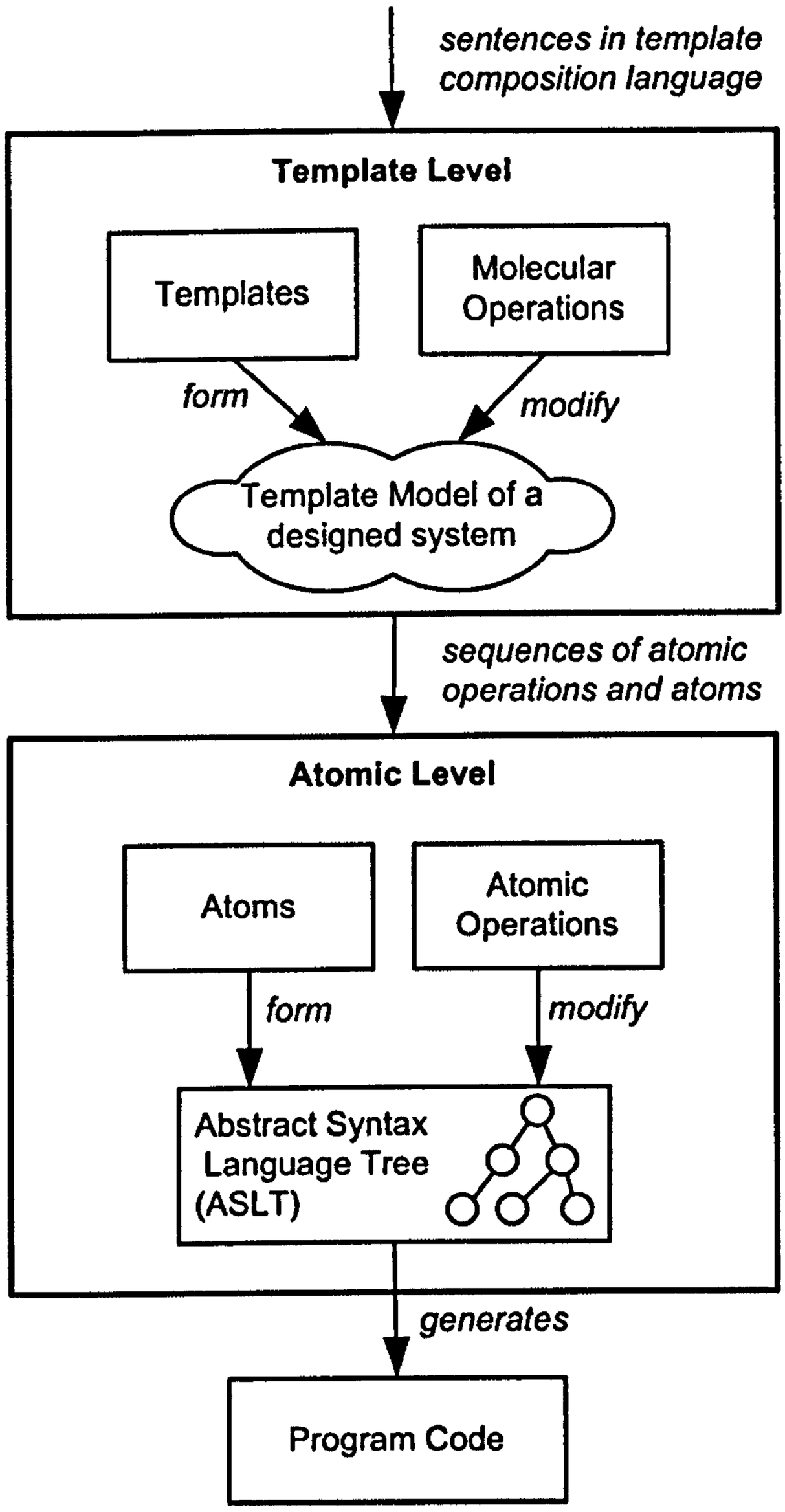


Figure 3.14: Concepts defined at the Atomic Level and the Template Level during the design phase

3.3 and 3.4 explain the activities at each level during the composition system definition phase, the Design and the Runtime phase of the NCF software life-cycle.

Phase	Level	Activity
Composition system definition phase	Target Domain Level	1) Specification of domain-specific language based on template composition language 2) Domain analysis and description of an application domain
	Visualisation and Interaction Level	1) Specification of DSVI that is mapped onto template composition during design phase 2) Specification of Neurath Modelling Language to transform DSVI

Table 3.2: Activities of the Target Domain Level and the Visualisation and Interaction Level at the composition system definition phase

Phase	Level	Activity
Design phase	Visualisation and Interaction Level	1) Interprets actions performed by domain experts into sentences in domain-specific language (the connection with the Target Domain Level) 2) Visually reflects the state of the designed software system (the connection with domain experts)
	Target Domain Level	1) Interprets sentences written in domain-specific language into sentences written in template composition language (connection with the Template Level) 2) Generates a domain-specific desciption of the state of a designed software system (connection with the Visualisation and Interaction Level)

Table 3.3: Activities of the Target Domain Level and the Visualisation and Interaction Level at the design phase

Figures 3.15 and 3.16 show concepts defined at the Target Domain Level and the Visualisation and Interaction Level at two different phases. Figure 3.15 depicts concepts that are relevant during the composition system definition phase.

The figure shows that repositories of templates and molecular operations defined at the Template Level (see Fig. 3.13) are extended with definitions of *Domain-specific Components (DSCs)* and *Domain-specific Operations*. These are formed according to the rules

Phase	Level	Activity
Runtime phase	Target Domain Level	Describes the state of the running system with the domain-specific terminology
	Visualisation and Interaction Level	1) Domain-specifically visualises the state of the running software system 2) Interprets actions of domain experts in order to perform 3) and 4) 3) Manipulates with visual representation 4) Assigns specified states to the specified components of the running software system

Table 3.4: Activities of the Target Domain Level and the Visualisation and Interaction Level at the Runtime phase

defined by the DSC model and depend on *Domain Ontology* to describe the application domain. DSCs represent terms defined by the domain ontology. Domain-specific operations work with DSCs and represent changes that could be done in the domain-specific system.

As depicted in Figure 3.15, the main production at the Target Domain Level inlcudes DSCs, domain-specific operations and the domain ontology. These are extended at the Visualisation and Interaction Level with the DSVI specifications. DSVIs are basically defined by a set of NMCs and *Views*. NMCs are parts of DSVI that visually represent part of the state of the designed system during the design phase. Moreover, during this phase, the NMCs are manipulated with the help of *Views*. At the composition system definition phase, Views are described by *View Specifications*. NMCs are defined according to rules provided by the *NMC model*.

Figure 3.16 depicts concepts relevant during the design phase. At this phase domain experts design a software system within the required application domain via DSVI.

Basically, the DSVI visually reflects the state of the designed system as well as interprets actions done by designers in order to perform template based transformations of the designed system. DSVIs are composed with NMCs. The figure shows that designer's actions applied to DSVIs are interpreted into *User Interface Interaction Expressions (UIIE)*. Then, with the help of *UIIE Parser*, these expressions are interpreted into sentences written in domain-specific language. These sentences are expressions that consist of Domain-specific Components and Operations. At the Target Domain Level these expressions are

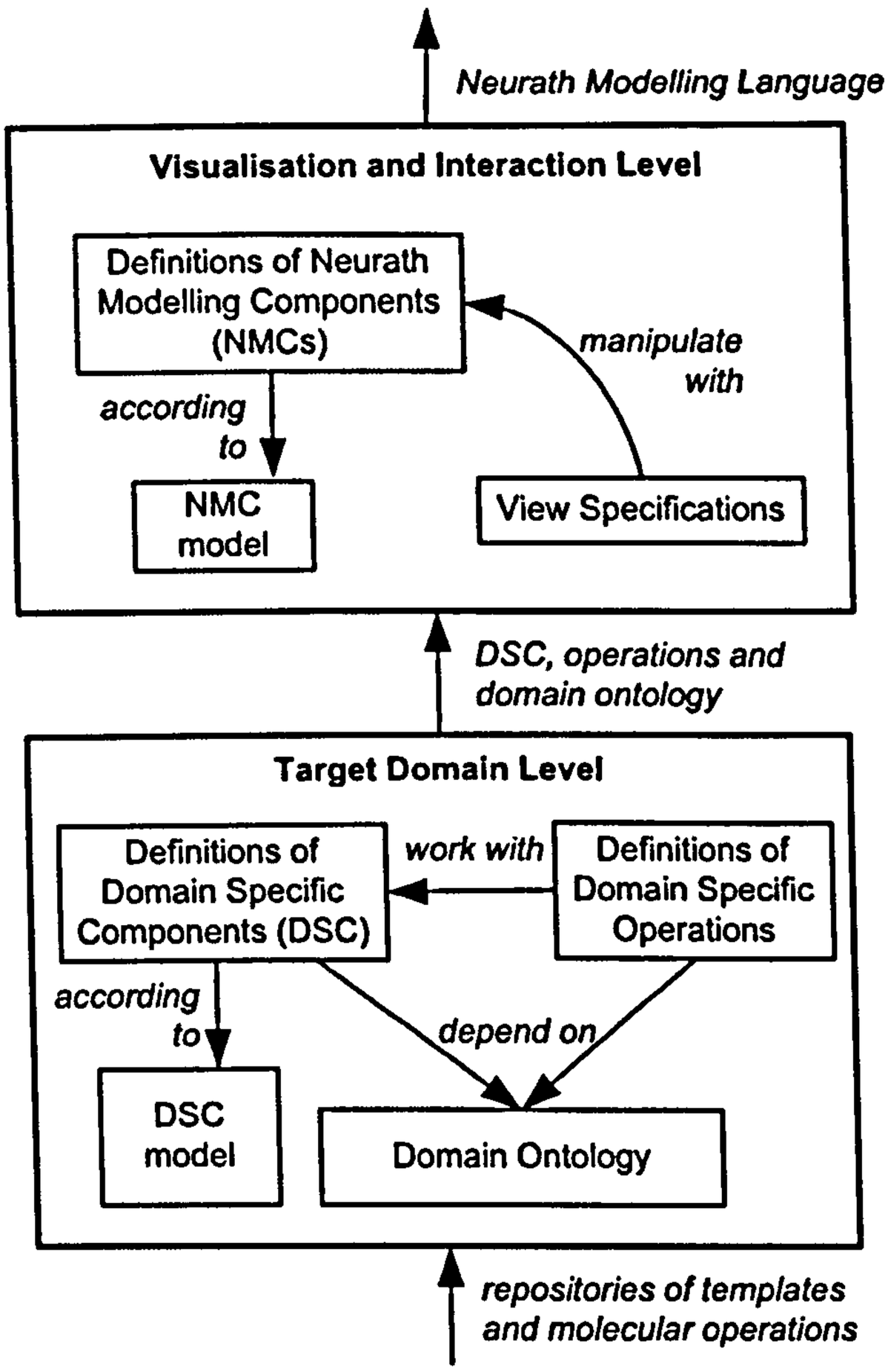


Figure 3.15: Concepts defined at the Target Domain Level and the Visualisation and Interaction Level during the composition system definition phase

processed. From one side, this results in sentences written in template composition language which are forwarded to the Template Level for further processing (see Fig. 3.14). From the other side, interpretation of the expressions results in a modification of the domain ontology reflecting the state of a designed system. The state is forwarded to *views* that are defined at the Visualisation and Interaction Level. Views interpret the state and update the DSVI.

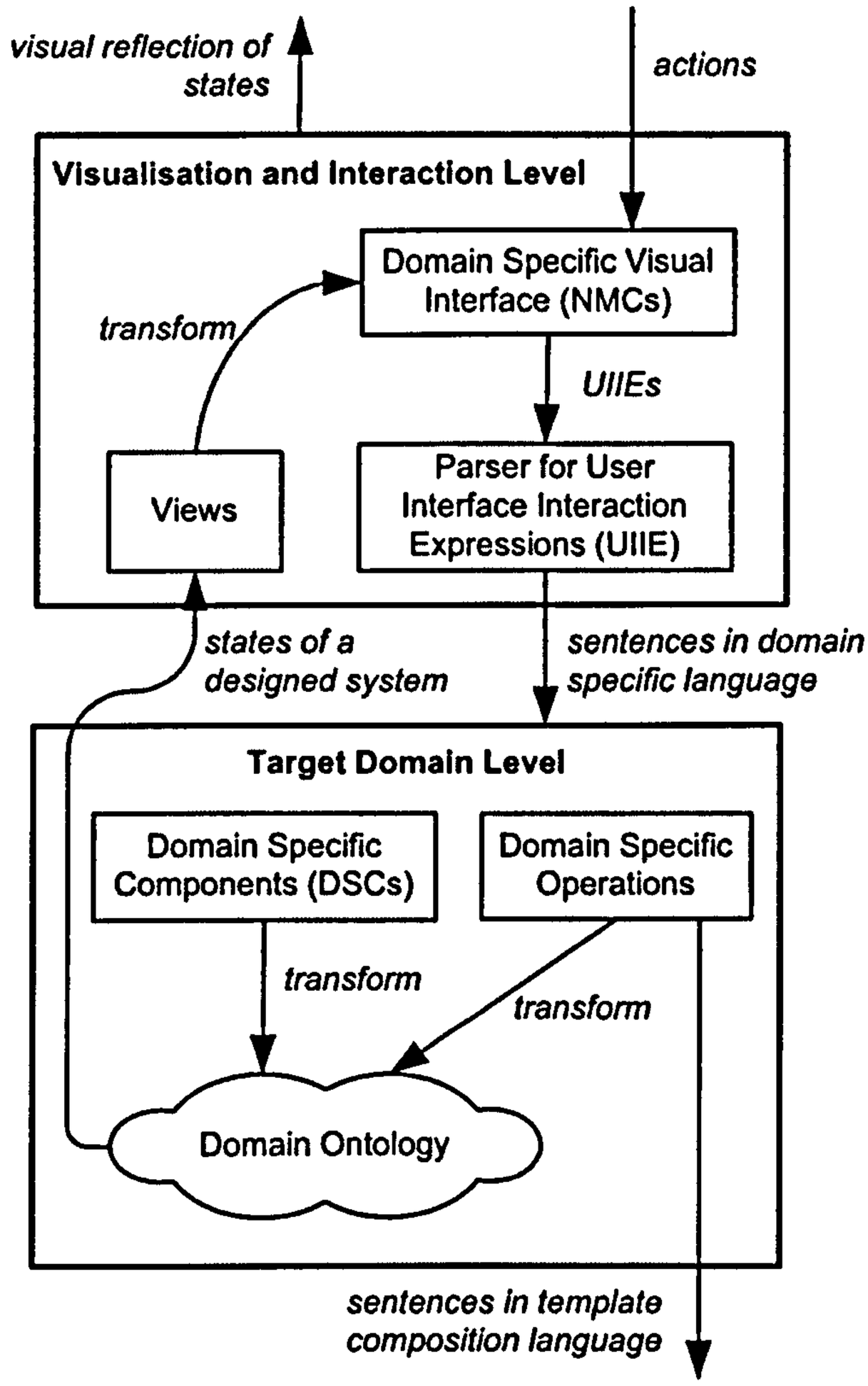


Figure 3.16: Concepts defined at the Target Domain Level and the Visualisation and Interaction Level during the design phase

3.8 Practical Realisation

Figure 3.17 depicts the basic idea of how NCF is practically realised. The NCF realisation consists of two main parts (tools). The first tool is called NBT. It is a tool for working at the composition system definition phase. With this tool, Software Architects - in cooperation with Domain Experts - create and test NMLs. The NBT has the following features:

1. Specification and test of a template composition system. This includes work with repositories of templates and operations to manipulate these templates.

2. Specification and test of DSLs. This includes work with domain-specific components and operations defined on top of template composition systems.

3. Specification and test of domain-specific visual interfaces defined on top of DSLs.

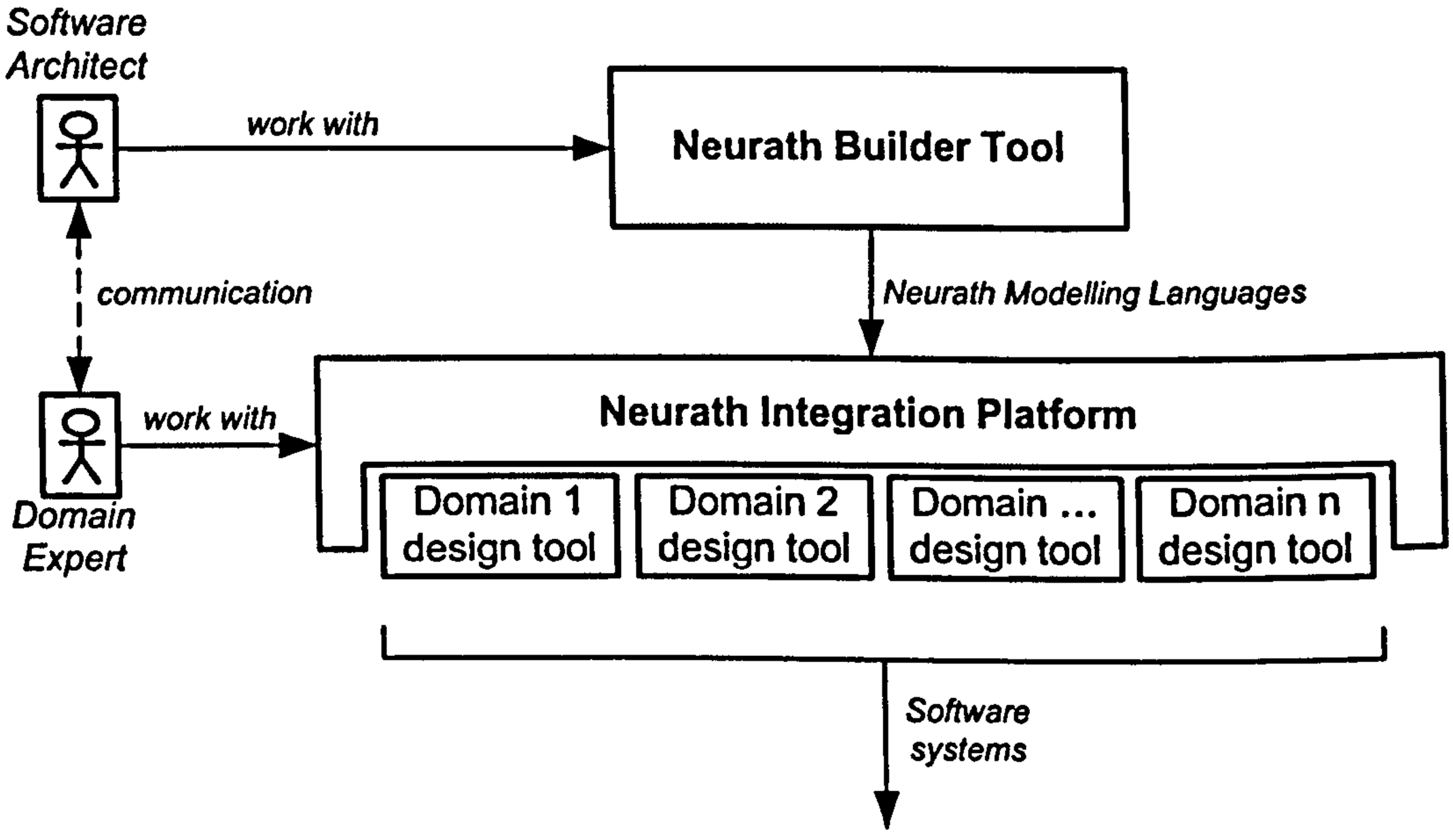


Figure 3.17: Tools for Practical Realisation of NCF

The second tool is called NIP. It is a tool to deploy (integrate) and use NMLs. The NIP is applied during the design phase and the Runtime phase. The NIP has the following features:

1. Can deploy NMLs.

2. Provides a design environment for NMLs. This includes a pane to visually design the software system, an inspector of properties of objects, toolbar, message pane and some other facilities.

3. Can generate a program code from the domain-specific visual model and execute it if it is executable.

Typically, the NIP is based for further different domain-specific tools, such as the UML tool, House Automation Software Design tool, Supply Chain Software Design Tool etc. Currently, NBT and NIP are implemented to define and use domain-specific visual languages that are defined on top of the template composition system for Java programming language.

3.9 Summary

In this chapter we have introduced basic requirements to the NCF. We have described an architecture of NCF and the life cycle of the development process that the NCF defines. Finally, we gave an overview on tools to practically implement the NCF. Further chapters describe the concepts in detail.

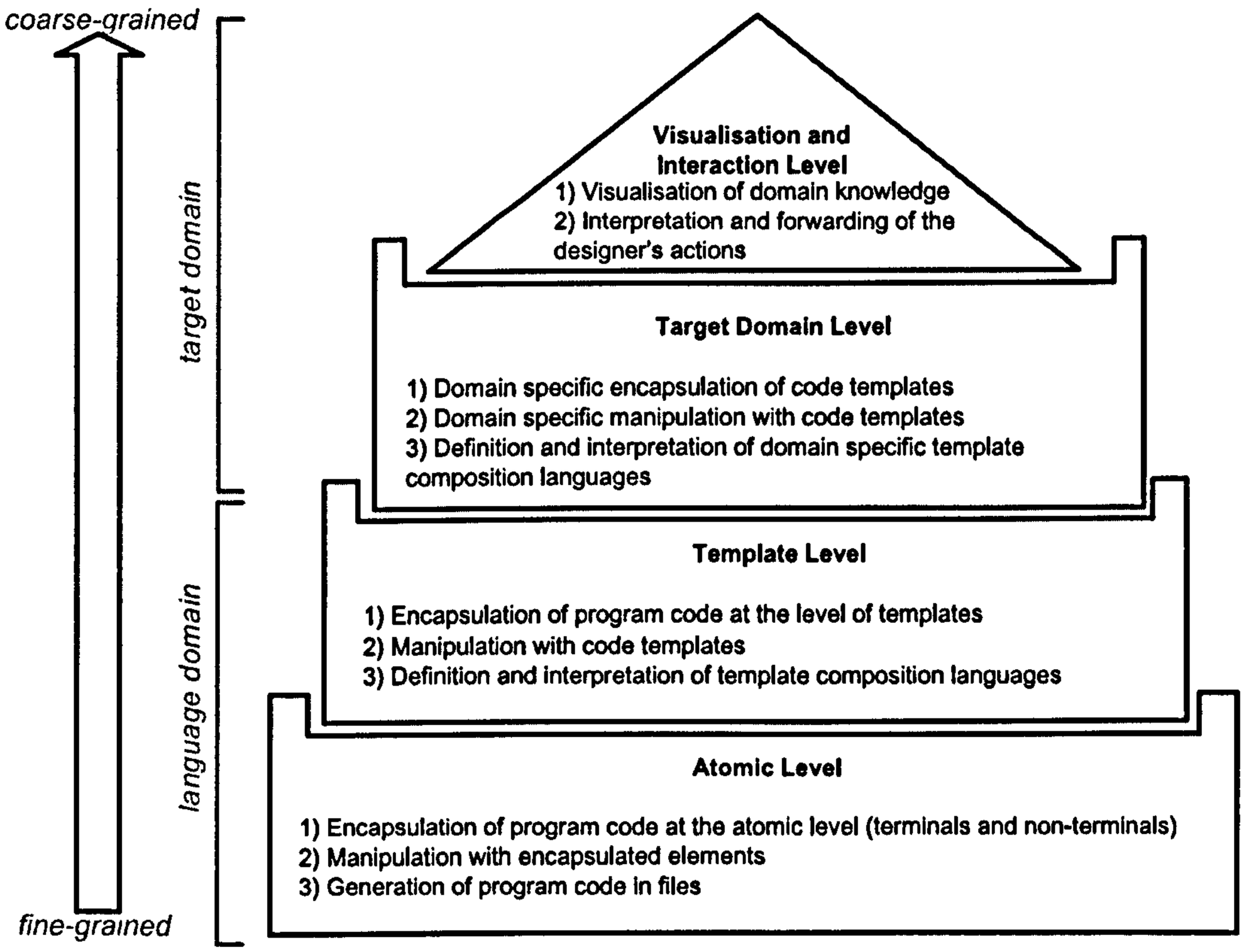


Figure 3.18: Levels of composition within the Neurath Composition Framework

Further, we explain concepts defined at each level of composition (see Fig. 3.18) for different phases of the NCF life-cycle. Chapter 4 presents the Atomic Level of composition with atoms, atomic operations and the Abstract Syntax Language Tree. Afterwards, Chapter 5 explains Template Level and introduces templates, molecular operations and template composition language. Chapter 6 presents the Target Domain Level of composition, describing the basic externalisation of a template composition system up to the level of domain experts. It presents concepts such as domain-specific components and operations together with domain-specific languages defined on top of template composition systems. Thereafter, Chapter 7 introduces the Visualisation and Interaction Level of composition. The concepts for better externalisation are explained. This includes Neurath Modelling Language and Views. Finally, Chapter 8 shows how the NCF is practically implemented with the Neurath Builder Tool and the Neurath Integration Platform.

Chapter 4

Atomic Level

This chapter introduces the atomic level of the composition. Basically, at this level a program code is seen as a set of related constructs that are defined by the grammar of a given programming language. We introduce a concepts to manipulate with a program code at the Atomic Level. These are atoms, atomic operations and Abstract Syntax Language Tree which is a collaborative work [93]. Additionally we introduce terms and few concepts as an extension to the ASLT.

4.1 Introduction

For software designers a program code is often seen as a piece of text written in some programming language. The programmers directly interact with files that are holding programs. Sometimes tools are provided that automatically generate parts of the code. The textual form of a program code is a classical one and still very useful in practice. Usually, the existing tools that perform certain code generation routines require a special data structure that often holds a program code in a tree-like form. It holds information about the program code, like its structure, items that the code contains and their basic interrelationships. That data structure that holds a program code in a tree form is very comfortable to perform analysis routines over that code.

The Neurath Composition Framework that is described in this thesis defines several levels of composition. In this section the lowest level of composition is described. It is called *Atomic Level*. Concepts defined at this level are used on a further higher *Molecular Level* of composition. We use the ASLT which is a collborative approach to perform program code composition routines at the Atomic Level.

4.2 Architecture

At the Atomic Level of composition we work with a program code with help of the ASLT. It defines a parse tree for a program code, parsing mechanisms and an interface to manipulate with that tree. Nodes of the parse tree are atoms the program code is composed with. The ASLT is a collaborative work defined by Wolke in [93]. In this thesis the ASLT (applied for the Java programming language) is used to perform atomic composition of a program code. An atomic composition of a program code means that the code can be specified with a sequence of atoms and atomic operations. Figure 4.1 shows main concepts defined at the Atomic Level of composition.

The program code written in some programming language is parsed and represented in the form of ASLT. The ASLT has an interface to access its nodes. In this thesis we define the term *atomic operations*. Atomic operations are rules to work with nodes of the ASLT. This includes such routines as the search of nodes, their creation, deletion, replacment, modification etc.

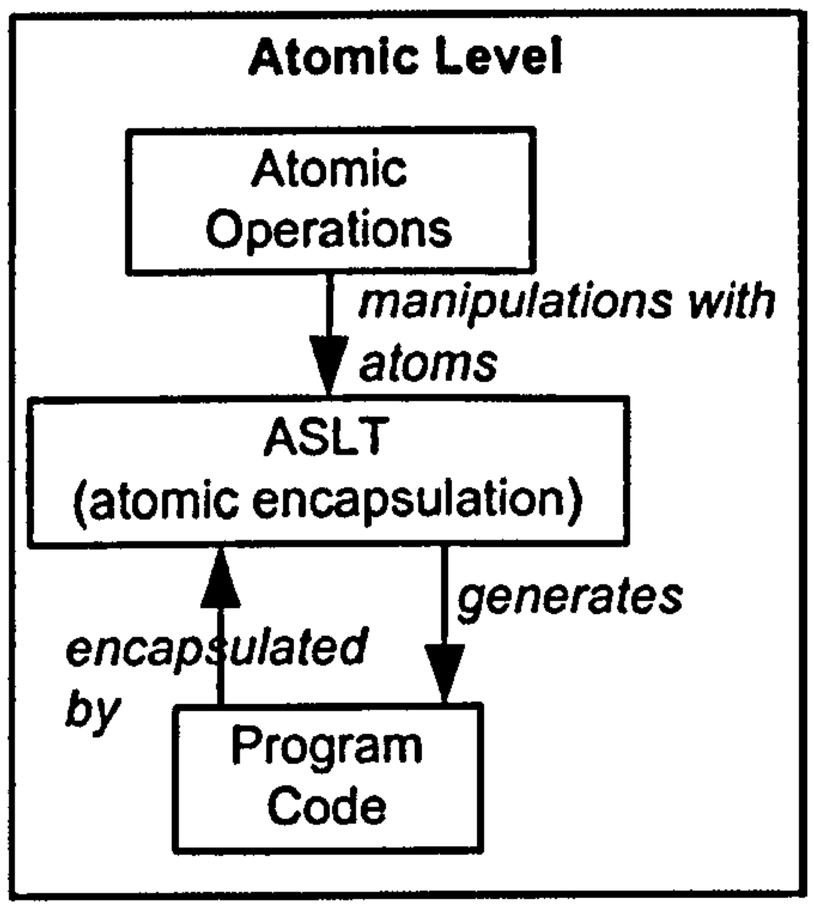


Figure 4.1: Concepts defined at the Atomic Level of composition

4.3 Atoms

At the Atomic Level of compostion a program code is composed with elemtents, that are programming language constructs. These are typically constructs which are represented by terminals and non-terminals of the BNF [1] grammar [81] definition of the programming language. We define a term *atoms* to denote these elements. For example, for the Java programming language, atoms are constructs such as a method signature, an indentifier of a class, a class body and a statement. Figure 4.2 depics the program code defined according to the grammar of some programming language. At the left side the same code is schematically shown in the ASLT form. The structure specification of the ASLT depends on the grammar of the programming language a program code is written in.

A program code consists of sentences written in some programming language. The structure of the sentences is defined by the syntax of the programming language. Listing 4.1 shows the part of the syntax specification of the Java programming language:

```
1 ...
2 <class declaration> ::= <class modifiers> class <identifier> <
     super> <interfaces> <class body>
3 <class modifiers> ::= <class modifier> | ...
4 <class modifier> ::= public|abstract|final
```

[1] Backus-Naur notation (more commonly known as BNF or Backus-Naur Form) is a formal mathematical way to describe a language, which was developed by John Backus and Peter Naur to describe the syntax of the Algol 60 programming language

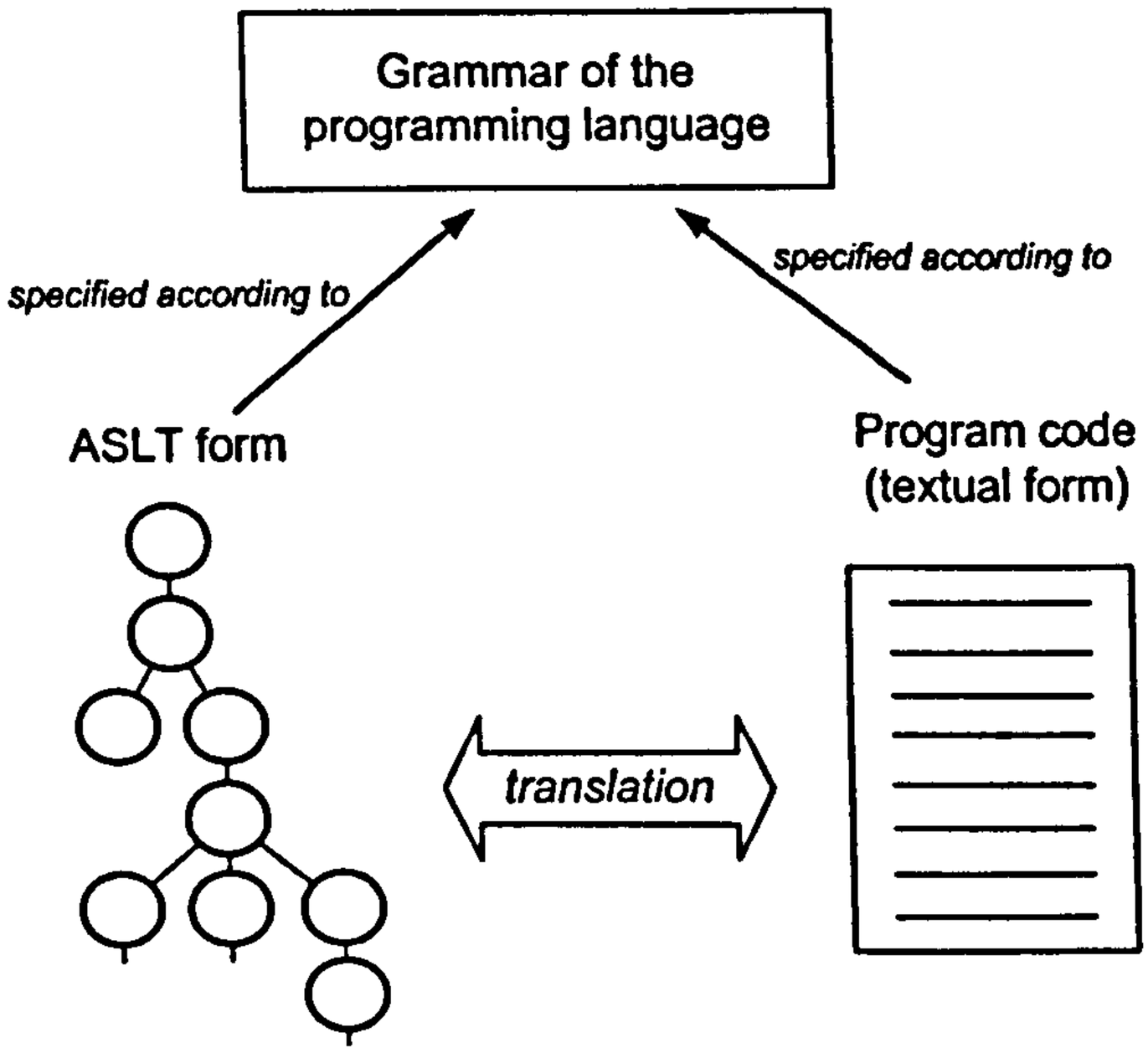


Figure 4.2: Atoms

```
<super>          ::= extends <class type>
...
<class body>     ::= { <class body decls> }
<class body decls> ::= <class body decl> | ...
<class body decl> ::= <class member decl> | ...
<class member decl> ::= <field decl> | <method decl>
...
```

Listing 4.1: Part of the syntax specification of the Java programming language (BNF notation)

Encapsulation of code at the atomic level means parsing a program code into a form of a parse tree. Figure 4.3 partially depicts the ASLT for the syntax specification of Listing 4.1. Each node represents both terminal and non-terminal symbols from the BNF of the programming language. Figure 4.3 shows what kinds of nodes may exist and what children they may have. Each node has attributes to access the concrete elements of a concrete program code.

Figure 4.4 depicts the code fragment written in Java programming language and the ASLT encapsulating this fragment. It is shown for some nodes what part of the code fragment they encapsulate.

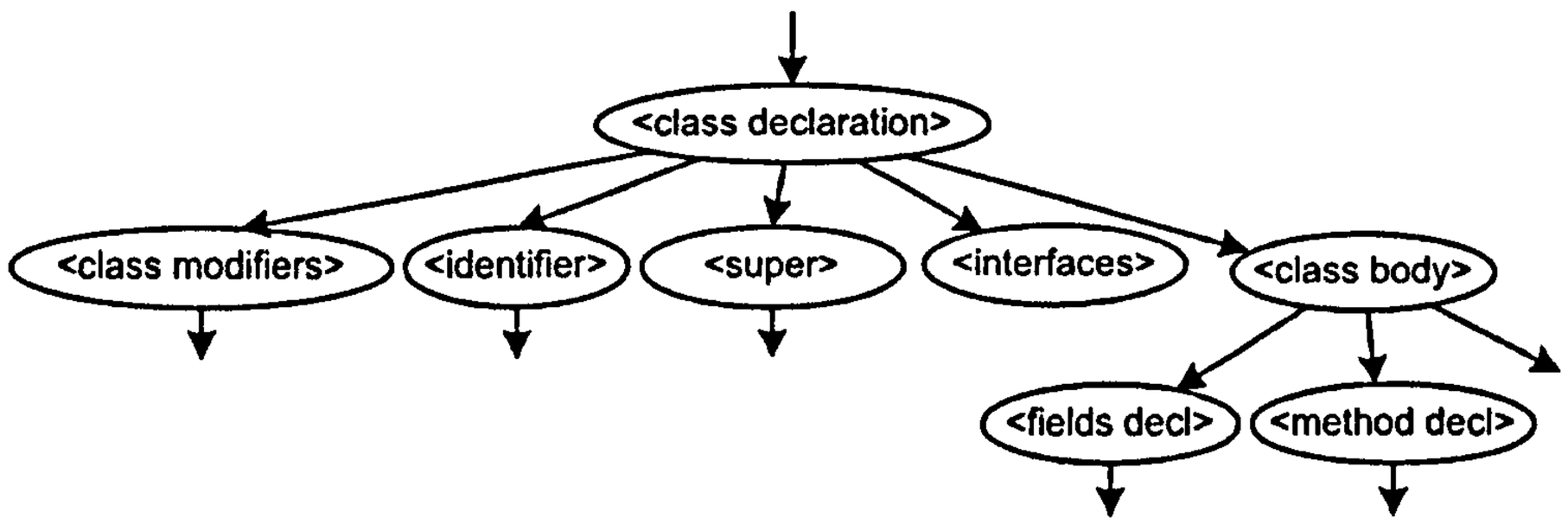


Figure 4.3: Structure of the ASLT

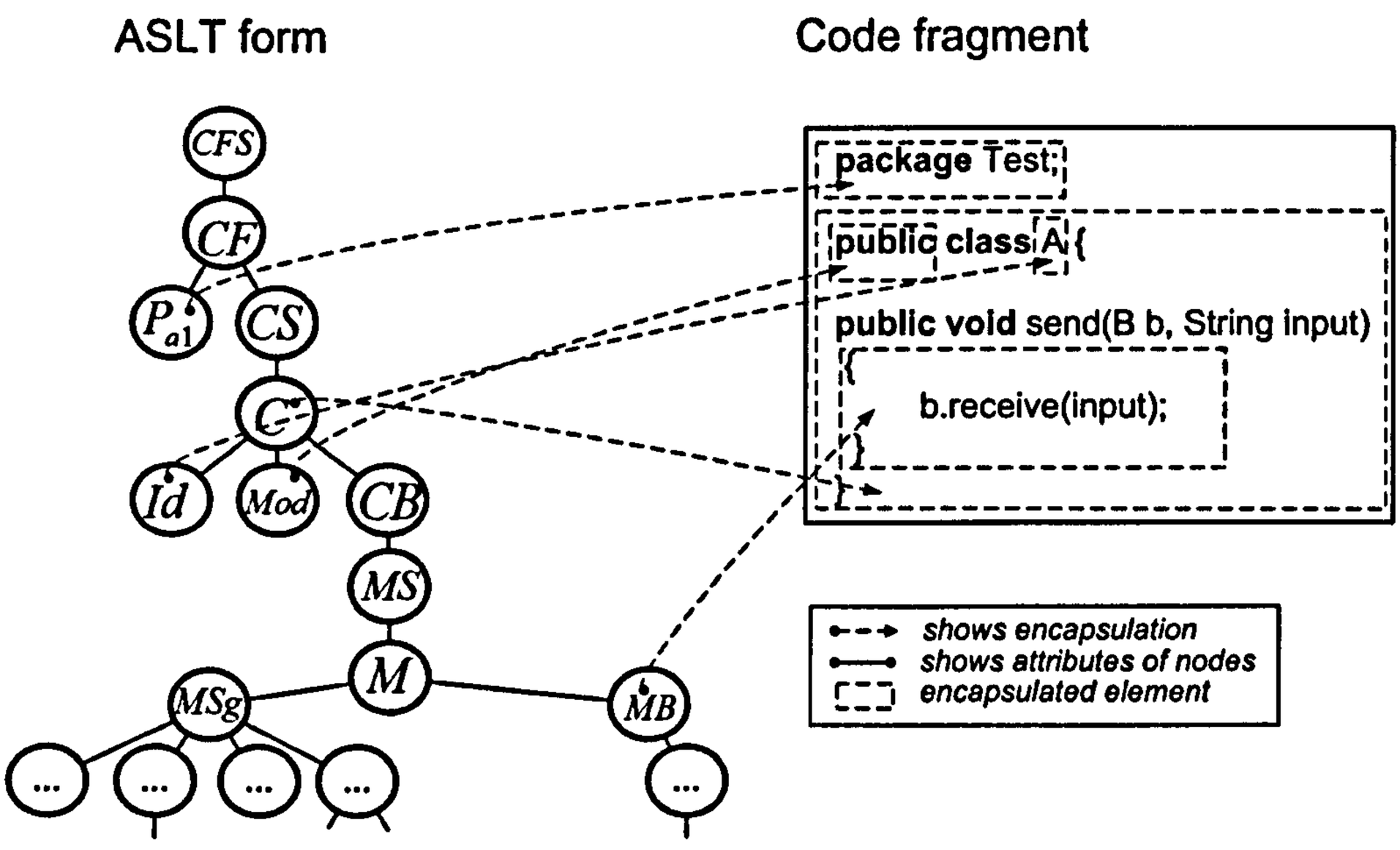


Figure 4.4: ASLT form of a code fragment

Each node is signed with its type and identifier. For example, the type `P` denotes a package construct. It has an identifier `a1`.

4.4 An ASLT for Java

This section introduces a collaborative work, called ASLT for the programming language Java, which has been designed and implemented by Wolke in [93]. For other object oriented programming languages the ASLT will have a very similar architecture. ASLT for Java programming langage is defined by atoms and atomic operations. Each type of

an atom is defined as an object (known from the object-oriented technology). Atoms aret nodes in the ASLT. Each object describing atom type defines possible children the node in the tree may have as well as methods to work with the node. Instances of atoms represent concrete constructs from the program code. Atoms plugged together forming an ASLT that describes a program code fragment at the Atomic Level.

Every language structure like package, class, method, loop, statement or expression is managed by a node in the tree. ASLT for Java implements all atoms for the Java programming language in the Java programming language. The ASLT represents a Java program as a tree, nodes of which are Java objects. The existing Java implementation can be used to easily implement an ASLT for the other object-oriented programming languages since many structures will be similar. An extract of the nodes that the Java ASLT can have is shown in the UML class diagram in Figure 4.5.

The ASLT for Java defines more than two hundred node types, and hence more than two hundred class definitions. Figure 4.5 shows only the node types of the uppermost layer. These are responsible for the coarse-grained structuring of Java source codes. There are also node types for sub-structures, that represent for example individual statements (like a for-loop or a if-else statement). The complete Java ASLT with all its node types or classes can be obtained from [93]. In addition to classes, the Java ASLT API[2] as well as the documentation can be found there.

The ASLT API offers the atomic manipulation of source code in a type safe way. According to the specification of the underlying programming language only special nodes can be attached to other nodes.

The ASLT for Java treats the source code in a different manner as it is done by usual text files. Moreover, the ASLT is responsible for the administration of the source code. Most of the node types represent constructs of the Java language and are responsible for their administration. The object definitions of atoms provides methods for reading the information of the structure or changing it. However not all atoms in the ASLT represent language constructs in the encapsulated program code fragment. Some atoms are needed for grouping purposes. For example, the atom `ASLTJavaProject` represents a project under which different atoms, which represent a files, can be grouped. The atom `ASLTJavaProject` has an attribute denoting a project name and a child atom of type `ASLTJavaSourceCodeFileSet`. The `ASLTJavaSourceCodeFileSet` atom is for the administeration of Java source code files. Each of them is represented by an atom

[2]An application programming interface (API) is a source code interface that an operating system or library provides to support requests for services to be made of it by computer programs

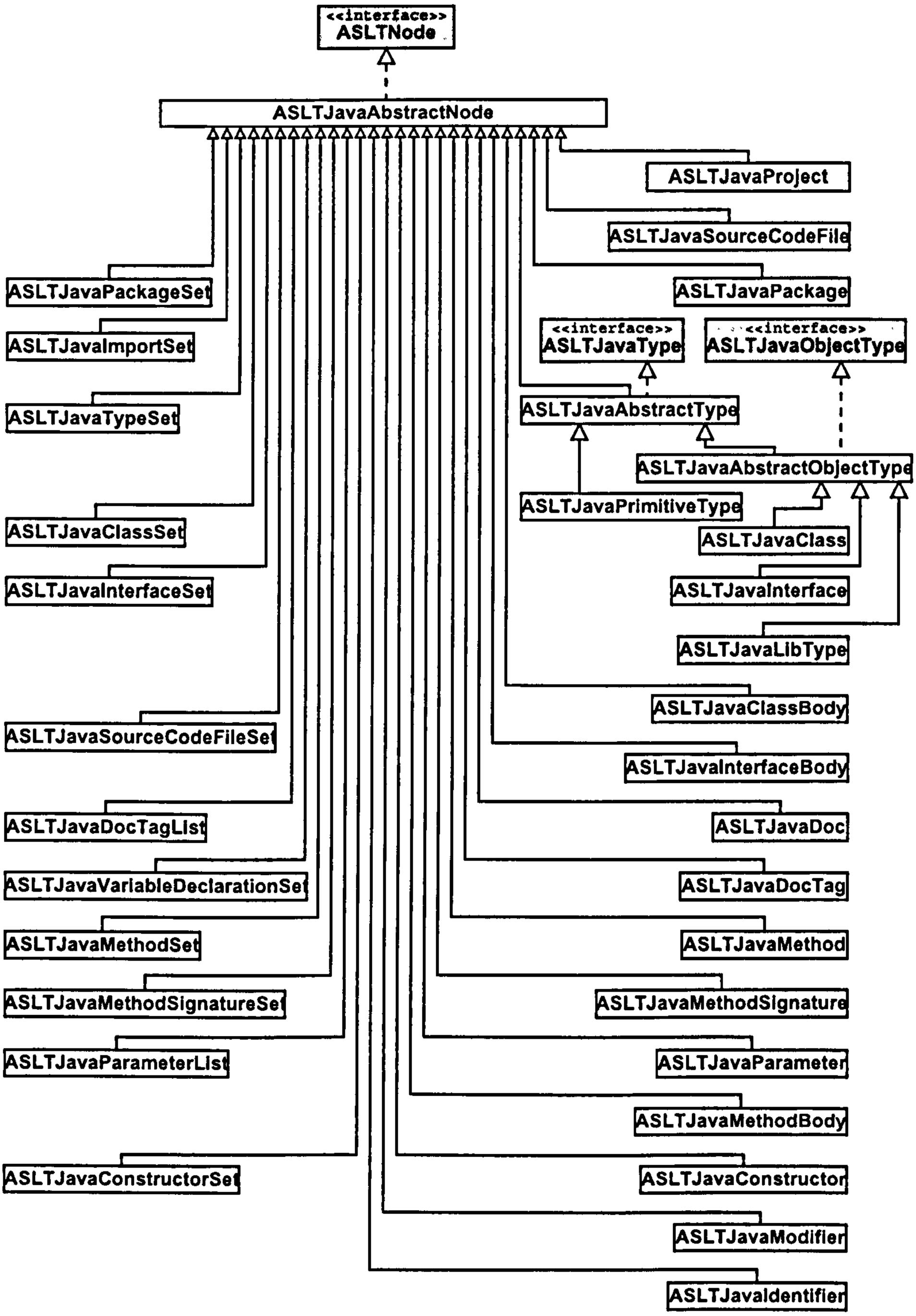


Figure 4.5: Node Types of the Java ASLT (Extract) [93]

of type ASLTJavaSourceCodeFile. Figure 4.6 shows a UML class diagram of the class ASLTJavaSourceCodeFile.

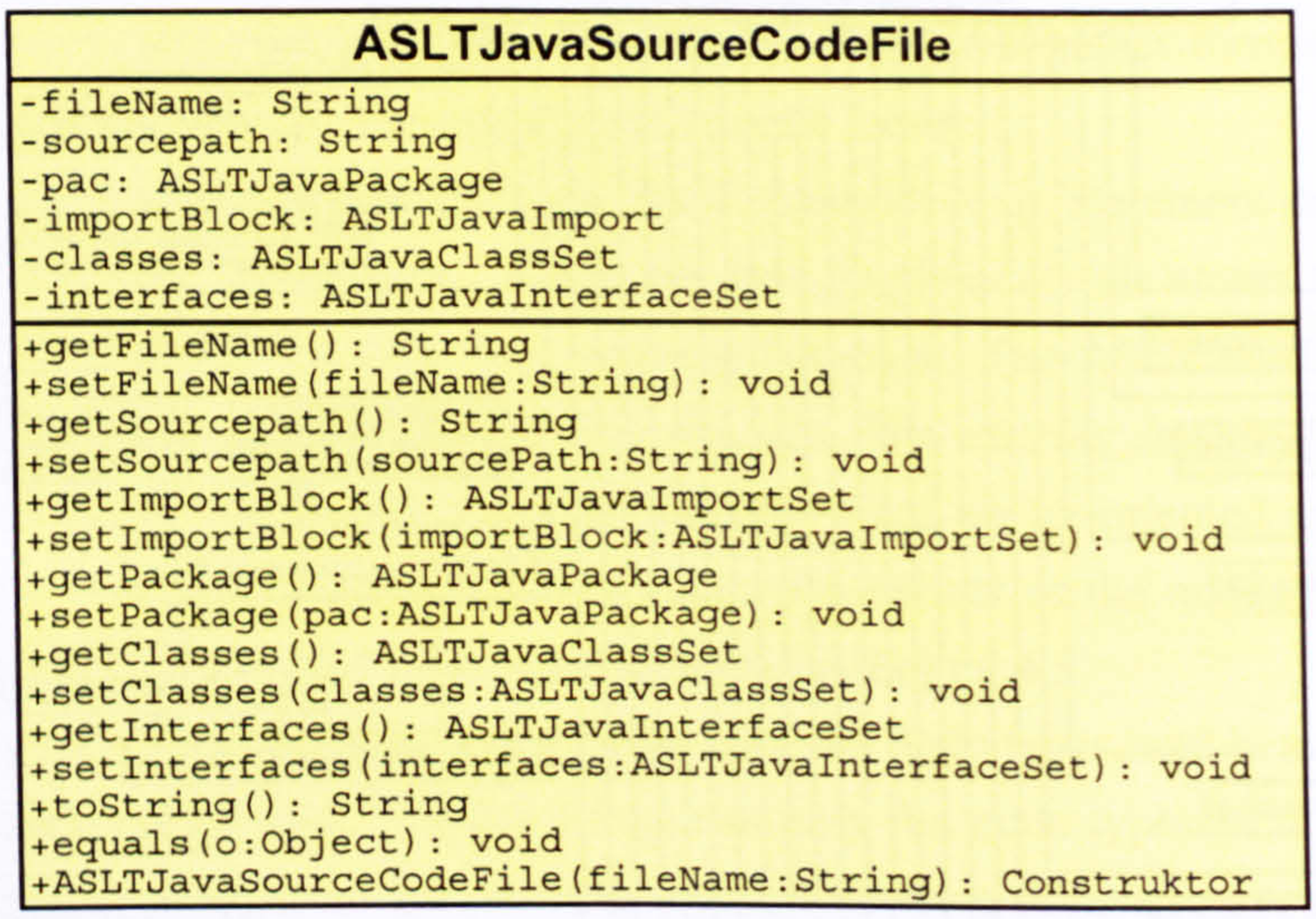


Figure 4.6: UML Class Diagram of Node Type ASLTJavaSourceCodeFile [93]

An instance of ASLTJavaSourceCodeFile groups all the information that belongs to a source code file in Java. The file name and the path to the source code file are specified directly as attributes in the atom. The package name, the import block as well as the defined classes and interfaces are managed by children atoms since they form hierarchical structures themselves. Since a source code file in Java can contain several classes and interfaces, the atom types with the suffix "Set" are used in order to guarantee uniqueness of the classes and interfaces within a source code file. An atom ASLTJavaClassSet manages a set of atoms of type ASLTJavaClass. Each instance of ASLTJavaClass represents a Java class. Figure 4.7 shows a UML class diagram of the node type ASLTJavaClass.

All the elements of a Java class are expressed as children atoms. The individual elements of the class definition are each represented by a separate child atom. The class body, for example, is represented by an atom ASLTJavaClassBody. The atom ASLTJavaClassBody can be extended by children nodes for methods, variables, constructors, inner classes and so on.

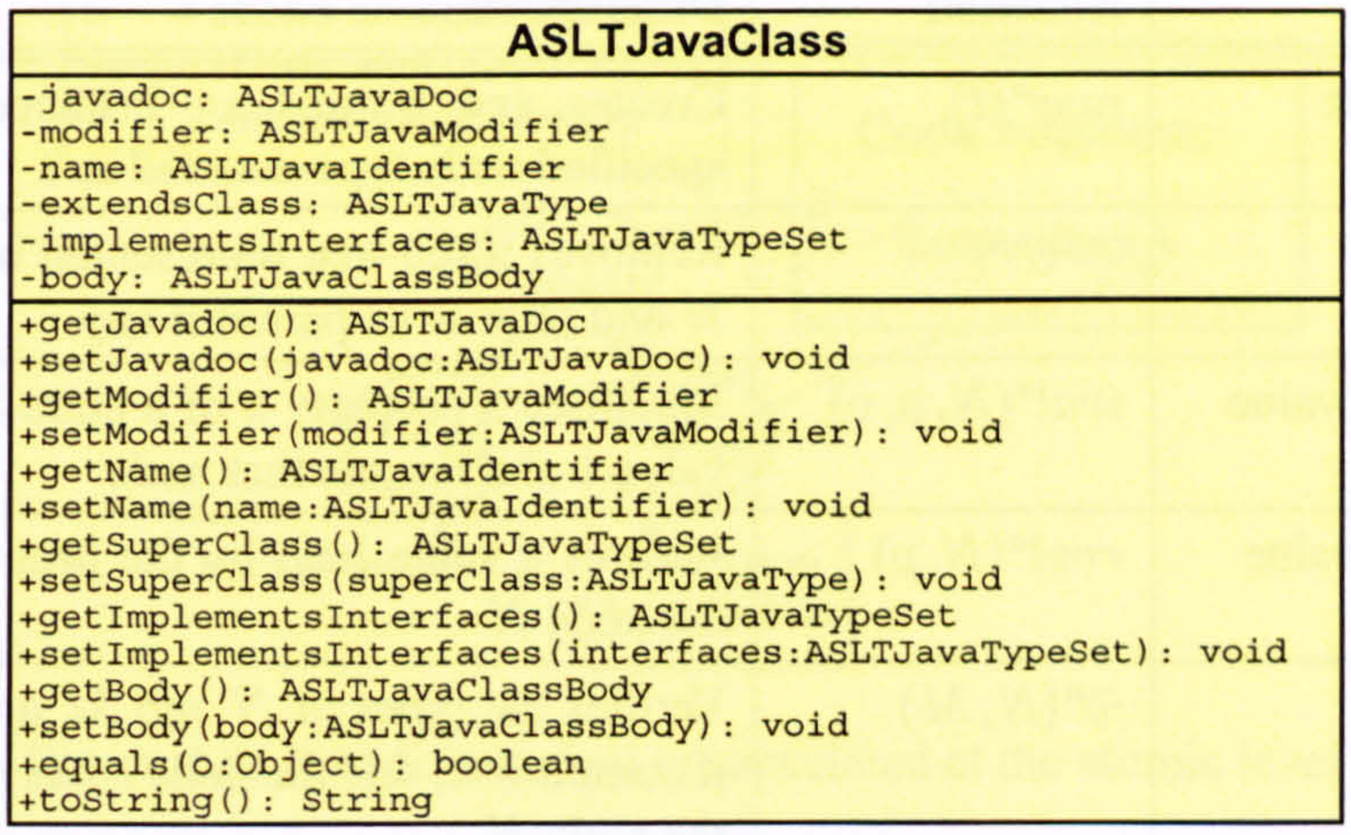


Figure 4.7: UML Class Diagram of Node Type ASLTJavaClass [93]

The entire program code fragment that is written in the Java programming language can be represented in the ASLT form. Further we exaplain atomic operations to modify ASLT instances.

4.5 Atomic Operations

The ASLT API provides routines to access the ASLT data structure, analyse and manipulate it. In this thesis we define a term *atomic operations* to denote these routines. An atomic operation is a rule to analyse and transform code fragments at the level of language constructs, i.e. search for an element type, delete an element and create an element. More complex operations can be constituted with atomic ones.

Table 4.1 lists and describes few basic atomic operations.

The listing depicted in Figure 4.8 shows a sequence of atomic operations to compose a part of class code fragment. Operators of this example are described in table 4.1.

The step (1) results an instance `class1` of the atom `Class`. This instance represents a class code fragment. At the step (2) and (3), in the similar manner, the instances `id1` and `mod1` of types `Identifier` and `Modifier` are defined. The instance `id1` represents an identifier and the instance `mod1` represents a modifier. At the steps (4), (5) and (6), the created instances are initialised with some values. The identifier gets a value "MyClass" for its attribute `name` at the step (4). At the steps (5) the modifier's attribute `static` is

Name	Notation	Description
Instantiate	$new^a(t)$	Creates and returns an instance of the specified node type denoted as t
Remove	$\oslash^a(N)$	Removes a sub-tree represented by a node N and returns its parent if any
Initialise value	$ival^a(N,p,v)$	Initialises a property p of a node N with a value v and returns that node
Request value	$rval^a(N,p)$	Returns a value held by the property p of the node N
Attach	$\oplus^a(N,M)$	Verifies the types of N and M nodes and, if compatible, sets the node M as a child of the node N
Detach	$\ominus^a(N,M)$	Removes a parent-child relationship, if present, between nodes N and M
Walk Down	$walk\downarrow^a(N,t)$	Resturns a child specified by the type t of the parent N
Walk Up	$walk\uparrow^a(N)$	Resturns a parent of a child N
Clone	$clone^a(N)$	Returns a copy of a node N
Search	$find^a(N,M)$	Starting from the node N it searches a match for a node described by the node M. Returns nodes which have been found

Table 4.1: Atomic operations

```
(1) class1 = new^a(Class)
(2) id1 = new^a(Identifier)
(3) mod1 = new^a(Modifier)
(4) id1 = ival^a(id1,name,"MyClass")
(5) mod1 = ival^a(mod1,static,"true")
(6) mod1 = ival^a(mod1,modtype,"public")
(7) class1 =⊕^a(class1,id1)
(8) class1 =⊕^a(class1,mod1)
```

Figure 4.8: An example of a composition at the atomic level

assigned with the value "true". Afterwards, at the step (6), another its attribute `modtype` receives the value "public". Steps (7) and (8) compose defined instances together. The step (7) results attachment of an identifier to the class. At the step (8), the modifier is attached to the class. The processing of the sequence of steps results a tree that is depicted in Figure 4.9.

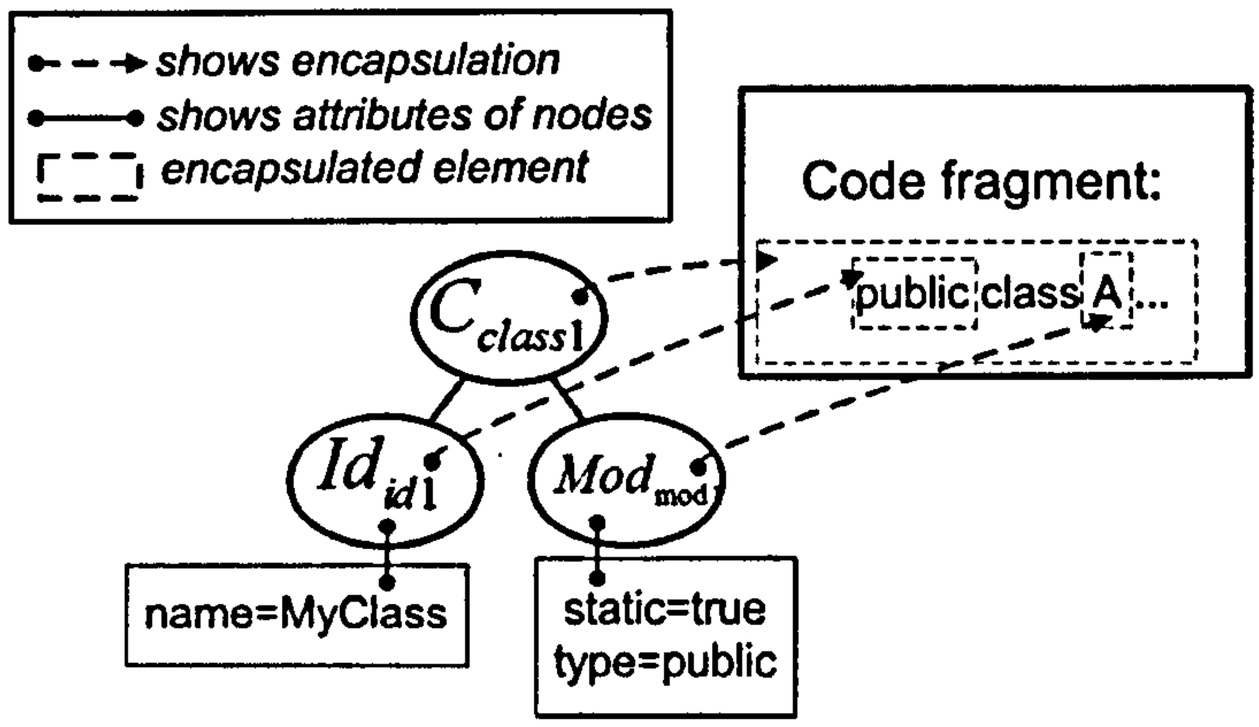


Figure 4.9: A code fragment encapsulated at the atomic level

The figure shows the ASLT (at the left side) that encapsulates a code fragment (depicted at the right side). The atom that represents the class contains two children atoms. These are the `modifier` and the `identifier`.

4.6 Summary

In this chapter we have shown the main concepts defined at the Atomic Level of composition. We presented the collaborative ASLT approach, as well as a scope of using ASLT within the frame of this thesis. We introduced the notion of atoms and atomic operations thus classifying and adapting the ASLT terminology to the terminology defined in this thesis.

Figure 4.10 shows the current progress of describing the levels of composition in the Neurath Composition Framework. With the bold line we marked the Atomic Level described in this chapter.

The concepts defined at the Atomic Level of composition form basis for next level of composition. At the next Template Level a program code is composed with templates. Next section explains concepts defined at the Template Level.

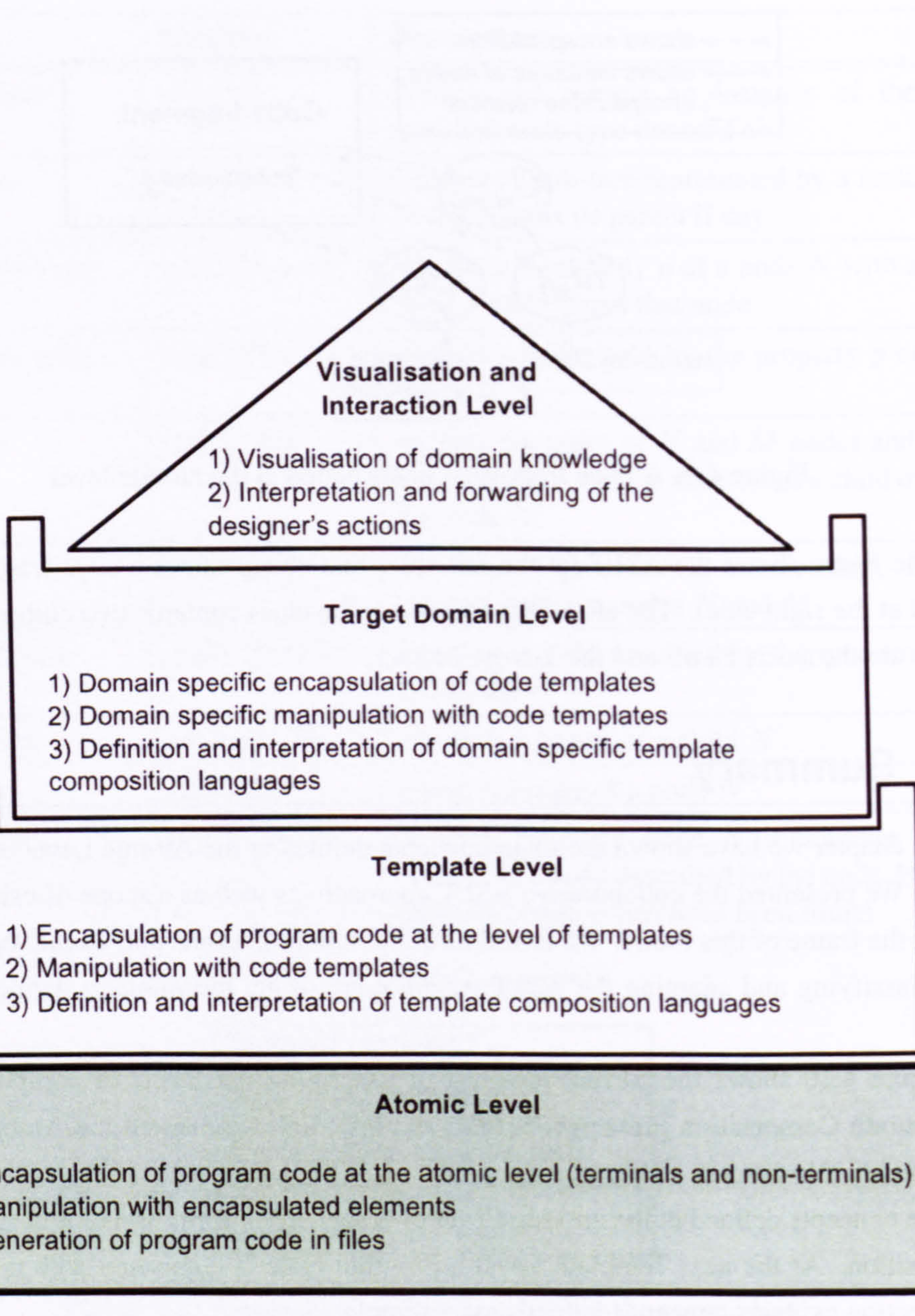


Figure 4.10: Levels of composition within the Neurath Composition Framework

Chapter 5

Template Level

This chapter presents a composition system to build a program code with templates. The concepts that form the composition system are united within a Template Level of composition. We explain the concepts defined at this level. We introduce parts of the composition system such as a component model, composition technique and composition language. The component model is presented with Parametric Code Template components. The composition technique, that is presented with molecular operations, is a strategy to combine and configure PCTs. Further, we describe a simple composition language to form composition expressions. Afterwards, an implementation environment of the concepts defined at the Template Level is briefly described. The chapter is concluded with summary.

5.1 Introduction

Compared to the atomic composition, that has been discussed in Section 4, a molecular compostion works with groups of atoms and defines the dependencies between these groups. Each atom represents a distinct monolith construct within a program code. Groups of atoms together with their dependencies form a *molecule* or *template*. In terms of the program code a template is a group of program code fragments that can be applied together in order to implement a specified feature or behaviour.

We define a molecular composition as a process of composing a program code with templates. Templates are described by a component model that is called Parameteric Code Template component model. This model specifies how code templates are encapsulated, managed and cooperate with other templates. Additionally, we define *molecular operations*, which are manipulation rules that can be applied to one or more templates. Templates and molecular operations can be applied to develop and maintain a software system represented by fragments of program code written in some programming language (for example Java).

Concepts related to the molecular composition are defined within the Template Level of composition. Further, we explain these concepts and their collaboration. Section 5.2 introduces the concepts defined at the Template Level of composition. Afterwards, Section 5.3 presents a meaning of the template that is relevant to the context of the thesis. A PCT component model and molecular operations are described in Sections 5.4 and 5.5. Section 5.6 introduces a simple template composition language. An implementation environment for the concepts from the Template Level are briefly presented in Section 5.8. Finally, a summary is given in Section 5.9.

5.2 Architecure

The Template Level defines concepts to solve an issue of composing a software system that is represented by a program code fragment, with templates. We recongnise the following main questions, answers to which characterise a template-based composition process:

1. How templates are encapsulated?

2. How templates can be configured?

3. How templates interact with other templates?

4. How new templates are defined?

We try to answer these questions and provide a template composition system that inculdes the following:

- A template based composition language to form composition expressions, called PCT-L.
- A composition technique, called *molecular operations*.
- A component model, called *PCT component model*.

Main components of the template composition system are PCTs and molecular operations. They can be composed in form of expressions defined by the PCT-L. Being processed expressions result in a transoformation of a program code.

There are two main phases defined. The first phase is the composition system definition phase. At this phase a template composition language is provided by defining PCTs and molecular operations. The second phase is the design phase. During this phase the template composition language is used to design a software system. In the following sections we are going to give more details regarding both phases.

5.2.1 Composition System Definition Phase

At this phase the definition of the template composition language PCT-L takes place. Figure 5.1 depicts basic concepts which are playing a central role during the composition system definition phase. At the input for the Template Level the domain requirements are specified. These requirements are generated by the Domain Expert who is going to use the PCT-L at other levels of composition when designing a software system within a specific application domain. The domain requirements are processed into the template composition language PCT-L.

The processing of the domain requirements is done by the Software Architect, who works on the language domain and is an expert in templates, software architectures, programming and so on in that fashion. In order to define PCT-L he operates with concepts defined at the Atomic Level and existing definitions at the Template Level. He may define new PCT components and molecular operations with help of atoms and atomic operations.

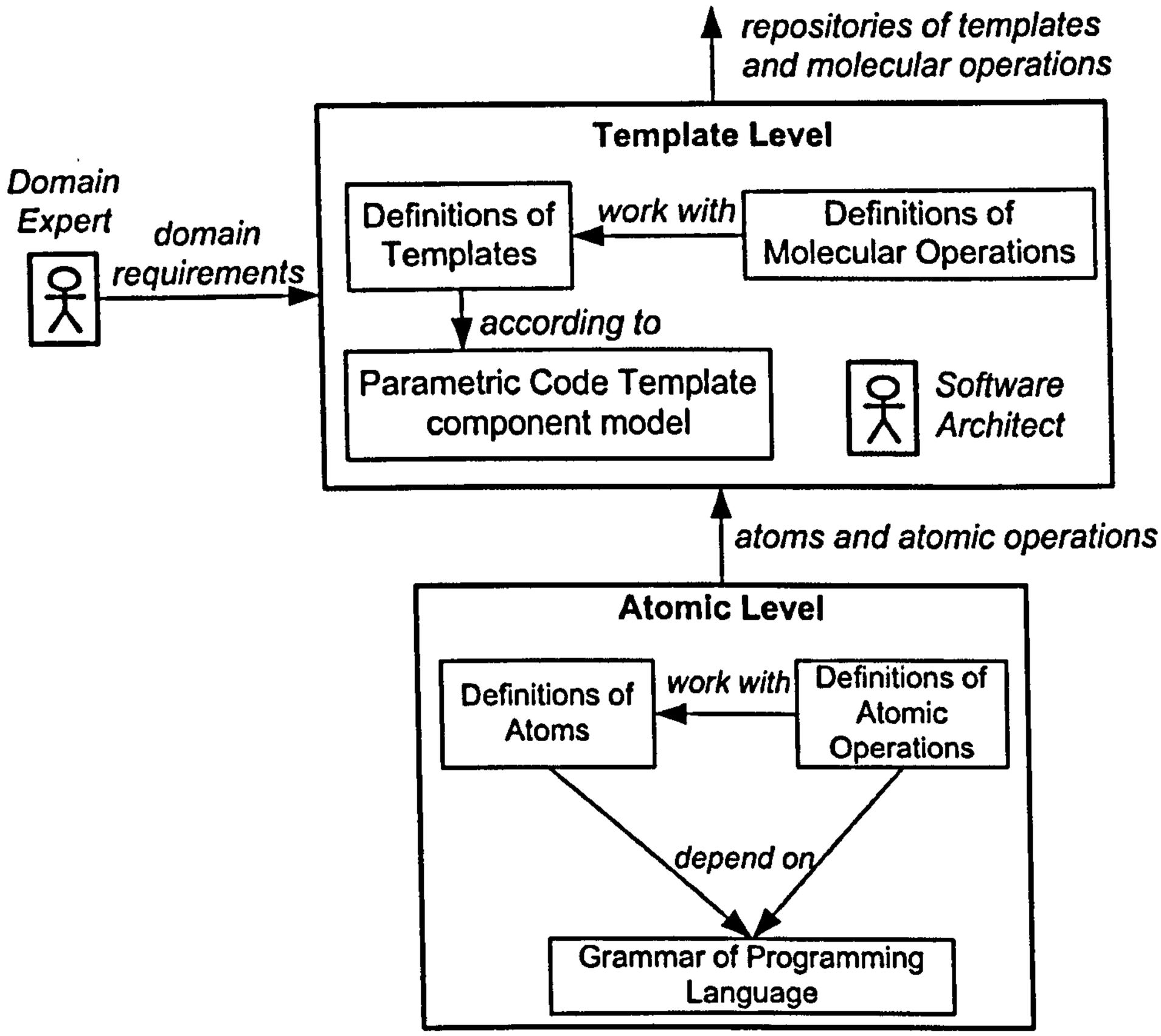


Figure 5.1: Concepts defined at the Template Level of composition during the composition system definition phase

PCT components are formed according to the PCT component model. Molecular operations are rules of how templates may be transformed. The production of the Software Architect that works at the Template Level during the composition system definition phase are repositories of templates and molecular operations suited to design software systems within specified application domain.

5.2.2 Design Phase

At the design phase the PCT-L is used to compose a software system within a specified application domain. The composition is seen as a sequence of molecular operations applied to PCTs.

At the design phase the following routines are performed:

1. Formation of sentences of PCT-L. These are expressions that consist of molecular operations as operators and PCTs as operands.

2. Processing of formed sentences of PCT-L. This initiates a transformation of a target program code that represents a software system.

3. Forwarding atomic manipulation requests to the concepts defined at the Atomic Level.

4. Description of the state of the designed system in terms of PCTs. The designed system is seen as one big template with number of nested templates.

Figure 5.2 depicts basic concepts which are playing a central role during the design phase. At the design phase a program code is composed with templates. The specification of how templates will be manipulated is defined in form of a PCT-L expressions, which are sentences of the template composition language formed at the composition system definition phase. At the Template Level expressions can be effectively formed by Software Architect. Basically, operators of the expression are molecular operations and operands are PCTs. PCT-L expressions are processed according to the semantics defined in the molecular operations and PCTs. Being executed PCT-L expressions results in a new transformed template, shown in the figure as a cloud denoted with Template Model, that carries atomic structures (ASLT) and therefore a program code, representing a new state of the designed software system.

5.3 Templates

We define a template as a group of program code fragments that can be combined (merged) together in order to implement a specified feature or behaviour. Additionally, we define special requirements to templates:

- *Modularity.* A template is a module or unit of composition that has a well-defined interface by which it can be configured and analysed.

- *Compositionality.* Templates can be composed (combined) together in order to form new templates.

- *Parameterisation.* The structure of the template, as well as the values carried by the structure, are subjects or parameterisation. With the help of parameters provided a template can be configured.

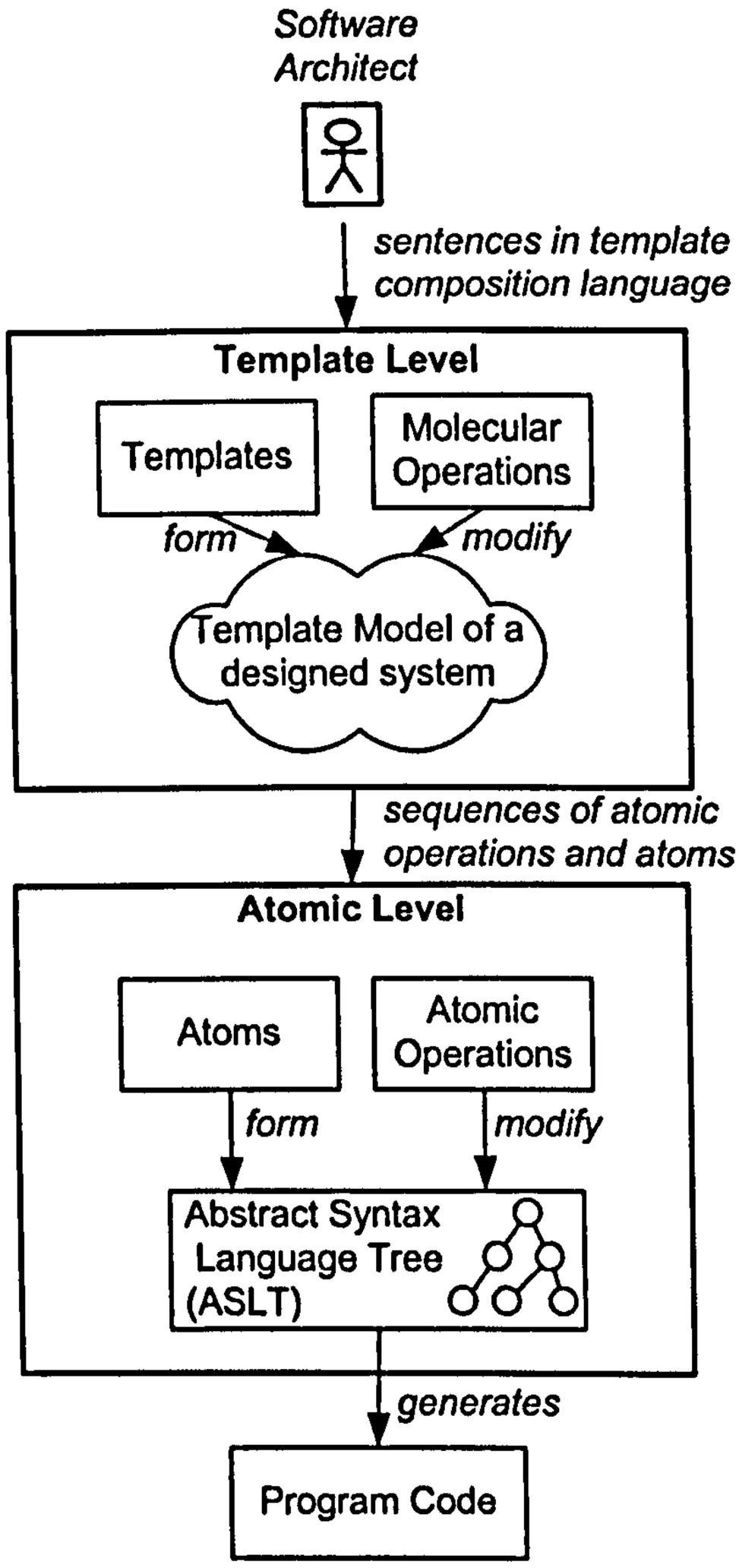


Figure 5.2: Concepts defined at the Template Level of composition during the design phase

- *Reactivity (awareness).* Changes occured within a template or outside of it may initiate a reaction of that template in order to adapt itself to these changes.

To demonstrate what templates are lets consider the following examples. Listings 5.1 and 5.2 show code fragments that can be considered as templates. Lets call them `Class` and `Method` respectively.

```
1 public class SomeClass{}
```

Listing 5.1: Template Class

```
1 public void someMethod(){}
```

Listing 5.2: Template Method

Listing 5.3 shows the result of the composition of the templates `Class` and `Method`. The resulted code fragment is a class specification, that contains a method.

```
1 public class SomeClass{
2   public void someMethod(){}
3 }
```

Listing 5.3: Composition of templates Class and Method

Listing 5.4 shows the *parameterisation* of the template `Property`. By this template we mean a bean property design pattern in Java programming language that is commonly used. According to this pattern, a property with some name and of some type is declared, whose value can be set or retrieved. This pattern is used so that the program code can be executed when setting or retrieving property values. This results in that the property value does not need to be stored in a field, can be delivered lazily, setting the property can fire events etc.

The template contains three fragments of code, which are a variable declaration and two methods. Those three fragments are often used together in a class to define a property that the instances of that class will have. The template specifies the two parameters `type` and `property name`. In the program code below we denote parameters of templates with `<parameter name>`.

```
1 private <type> <property name>;
2 public void set<property name>(<type> <property name>){
3   this.<property name> = <property name>;
4 }
5 public <type> get<property name>(){
6   return <property name>;
7 }
```

Listing 5.4: Parameterised template Property

Listing 5.5 defines a template `Property` with the two parameters `type` and `property` name assigned with the values `int` and `property`.

```
1 private int property;
2
3 public void setproperty(int property){
4   this.property= property;
5 }
6
7 public int getproperty(){
8   return property;
9 }
```

Listing 5.5: Template Property with parameters set

The example also shows that by the parameters it is possible to establish dependencies between parts of a template.

The next example shows the *reactivity* of templates. Listing 5.6 shows a template with a code fragment that prints the value of the declared variable. Such a code fragment can be part of the class specification. Imagine that a template is defined as if a new variable is declared, the method will be automatically extended with specification to print the value of the new variable. This is shown in Listing 5.7

```
1 private int property = 10;
2 public void printInfo(){
3   System.out.println(property);
4 }
```

Listing 5.6: A template that reacts on a new variable declaration which is added (the reaction specification is not shown in the listing)

```
1 private int property = 10;
2 private String name = "Newton";
3
4 public void printInfo(){
5   System.out.println(property);
6   System.out.println(name);
7 }
```

Listing 5.7: The result of adding the new variable declaration into the template

To generalise code templates we propose a component PCT model to encapsulate templates according to defined requirements and a composition technique, called molecular operations, to manipulate with templates.

5.4 Parametric Code Templates

PCTs are components that encapsulate code templates which are accessible by the interface that a PCT provides. PCTs can be seen as managers of fragments of a program code. They have got all of the necessary knowledge about that fragment. Each PCT definition represents a type or a class and can be instantiated. Each instance of PCT manages a concrete instance of code fragment (template).

Figure 5.3 depicts the general idea of the PCT component model.

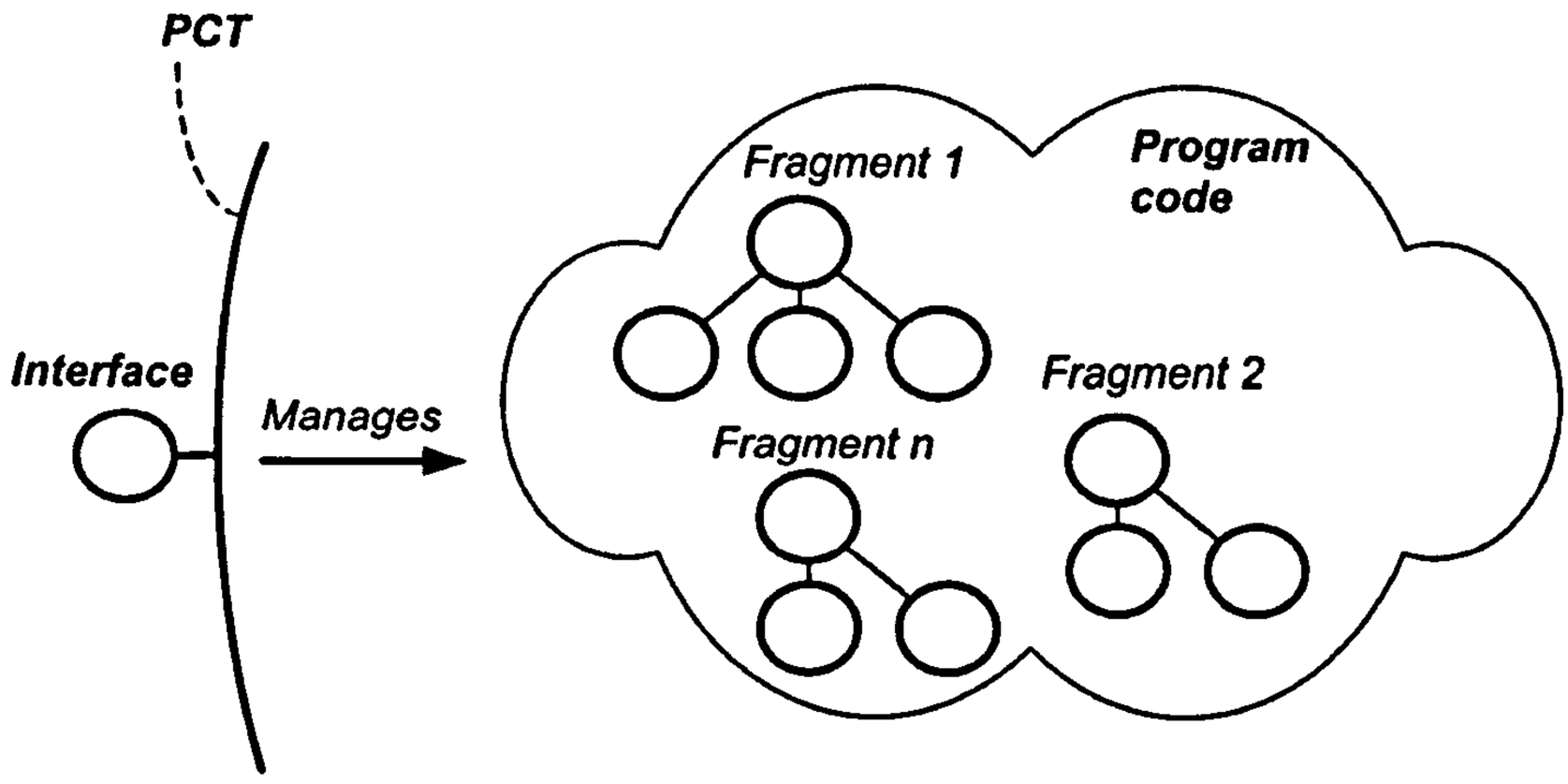


Figure 5.3: The basic idea of a PCT

The program code, which is encapsulated by a PCT, is managed by the defined interface. The following management routines are feasible:

1. It contains one or more fragments of a program code. New fragments can be added and already encapsulated fragments can be removed.

2. Any construct of the program code can be parameterised via a variable (parameter). *1:n* relationships may exist, where *n* is a natural number. New parameters can be defined and existing ones removed. Parameters can be set with a concrete value(s) and their value(s) can be requested.

3. The structure of a PCT (fragments and their interrelationships) is a subject of parameterisation as well.

4. PCT provides analysis (or reflection) mechanisms. For example, specific templates that a PCT is holding can be requested.

5. It may specify a reaction mechanisms on certain events.

5.4.1 Architecture of PCT

In order to perform specified management routines, the PCT component model defines the following concepts:

1. *A PCT module.* It is a container where code fragments, forming a template or other PCT modules, are kept. In case of holding only one code fragment, the PCT module is referred to as *PCT leaf*, otherwise the PCT module is called a *PCT container.* All PCT modules held by a PCT module are called *composites.* A PCT module specification is denoted with a name called *type name.*

2. *An interface* to the PCT module through which a template is managed.

3. *Parameters.* Variables mapped to the concrete constructs that are in code fragments. There could exist 1:1 and 1:n relationships between variable and constructs.

4. *Managment bahaviour.* This behaviour is represented by methods that define configuration schemes for the PCT. These methods are accessible through the interface.

5. *Awareness constraints.* Event-based mechanism to provide the reactivity of PCTs. Awareness constraints are the specification of reactions on various events.

6. The PCT component model uses an ASLT [93] to describe code fragments.

7. *Relations.* Relationships between code fragments may be defined. For example, if one code fragment is contained within another code fragment, a "merged" relationship exists.

Figure 5.4 depicts an example of a PCT and describes all items a PCT is constituted with. The (a) part shows the PCT named `PCT Class`. It contains only one code template represented by atomic elements (ASLT structure). The `PCT Class` defines

the parameters `param1` and `param2`. They are mapped to some constructs of the program code represented by the nodes of the ASLT. The PCT also defines a management behaviour `addStatement()`. The rectangle with contraints denote design context reactivity specifications for the PCT.

The (b) part depicts the PCT container named `PCT Result`. It contains two composites. One of them is an instance of `PCT Class`. The lines with arrows (both sides are arrowed) inside the `PCT Result` represent the mappings between the parameters which are defined by the `PCT Result` and the parameters defined by the composites. The figure also shows the "merged" relationship between the `PCT Class` and the `PCT Property`. This relationship means that the templates are connected at the ASLT level. Two PCTs are not related if any change in one PCT causes no change in the other one.

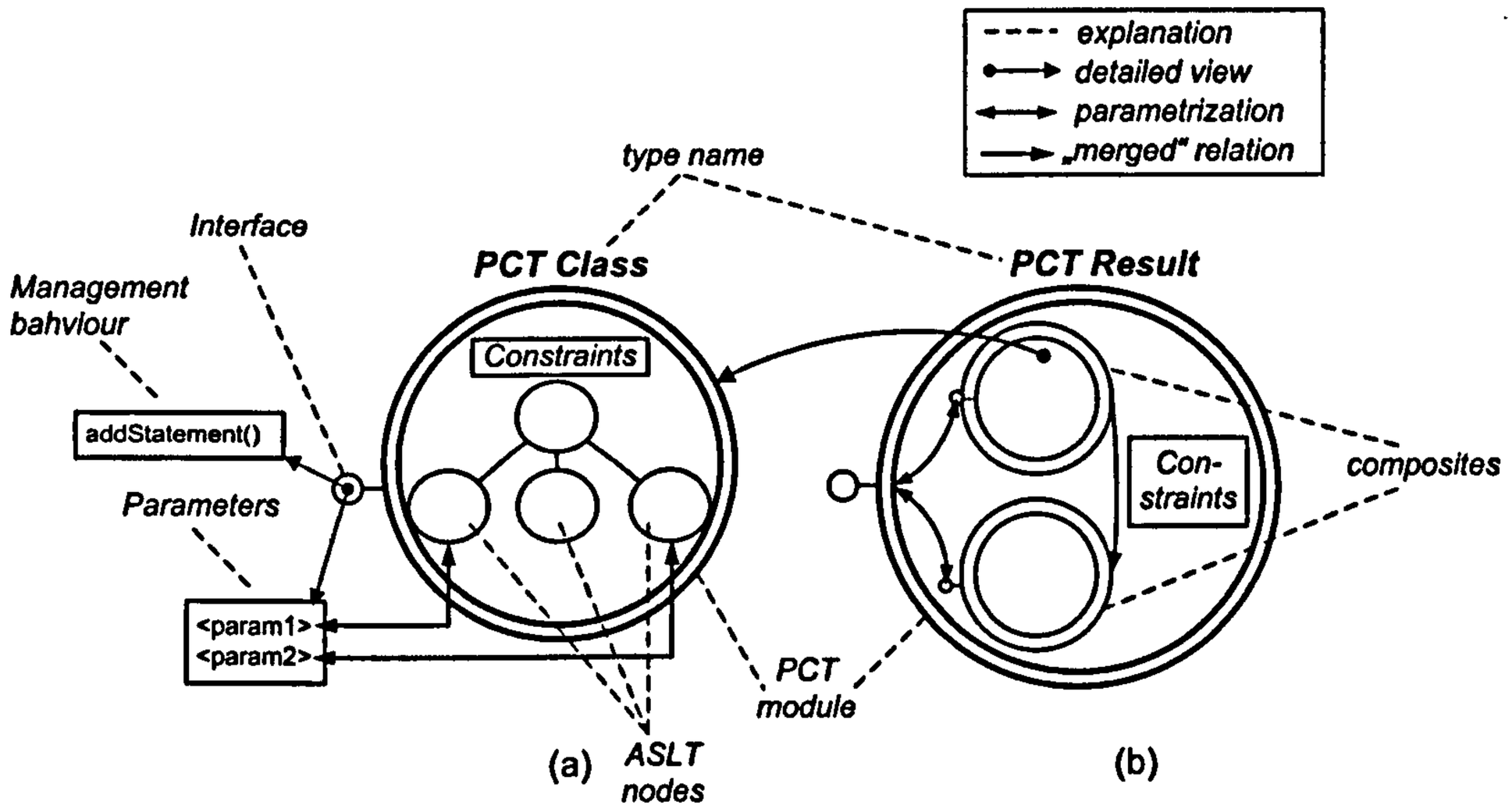


Figure 5.4: Examples of PCTs: (a) - a PCT-leaf and (b) - a PCT container that contains composites

5.4.2 Example of a PCT

This section demonstrates the encapsulation of a code fragment according to the PCT component model. The target code fragment is shown in Listing 5.8.

```
1 public class <className>{
2
3   private <type> <property name>;
```

```
4
5   public void set<property name>(<type> <property name>){
6     this.<property name> = <property name>;
7   }
8
9   public <type> get<property name>(){
10    return <property name>;
11  }
12 }
```

Listing 5.8: A code fragment encapsulated by a PCT

The code fragment is a program written in Java programming language. It specifies a class with some name, that contains one property that has some name and is of some type.

A PCT for the code fragment may contain the following items:

1. A PCT module that manages a code fragment that defines a class, which type name is `PCT Class`. Its interface defines the parameter `cName`.

2. A PCT module that manages a code fragment that defines a property, which type name is `PCT Property`. Its interface defines the parameters `pType` and the `pName`.

3. A PCT module that manages a code fragment that consists of fragments encapsulated by the `PCT Class` and the `PCT Property`. The type name of this PCT module is `PCT Result`. Its interface defines the parameters `cName`, `pType`, `pName`. These parameters are mapped to the corresponding parameters of the composites.

Figure 5.5 schematically shows three code fragments encapsulated by `PCT Result`, `PCT Class` and `PCT Property`. The arrows between two code fragments mean that these fragements are merged.

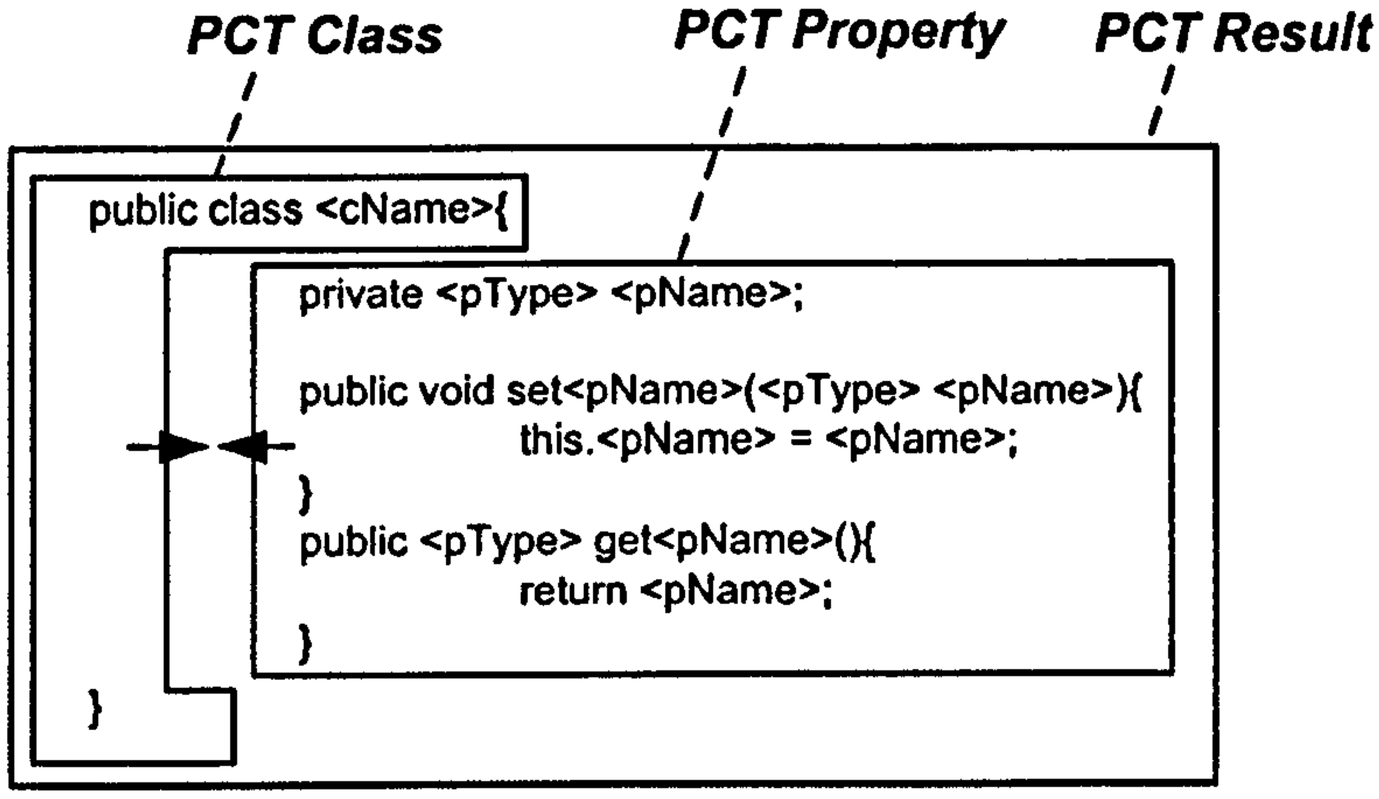


Figure 5.5: Code fragments encapsulated by the PCT Result

According to the simple visual notation used in Figure 5.4, the architecture of the defined PCTs will be depicted as in Figure 5.6.

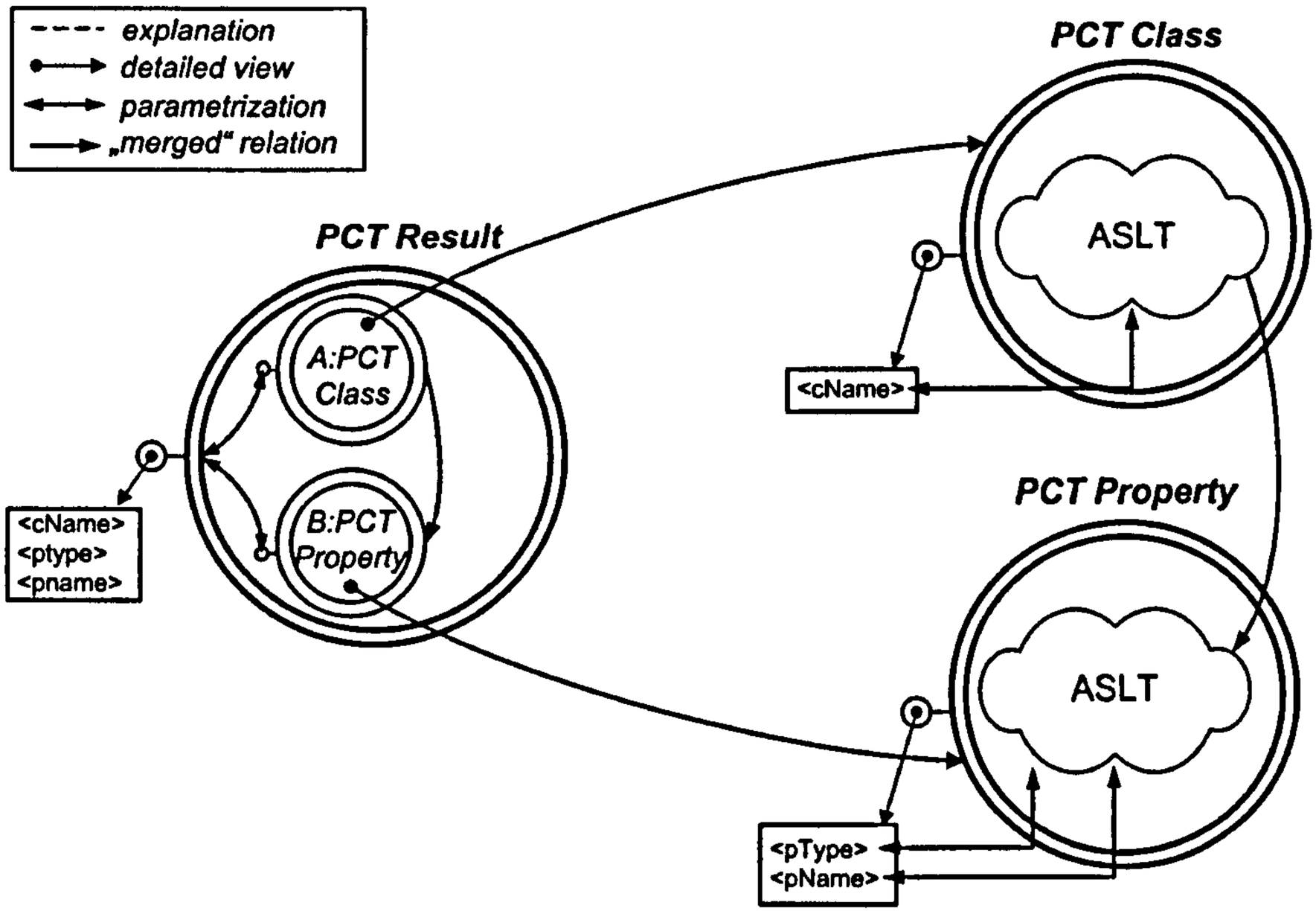


Figure 5.6: Structure of the sample PCT Result

The figure does not show details held by ASLT structures. Instead we further provide a Figure 5.7 to show how the parameters of `PCT Class` and `PCT Property` are mapped to the carried ASLT structure.

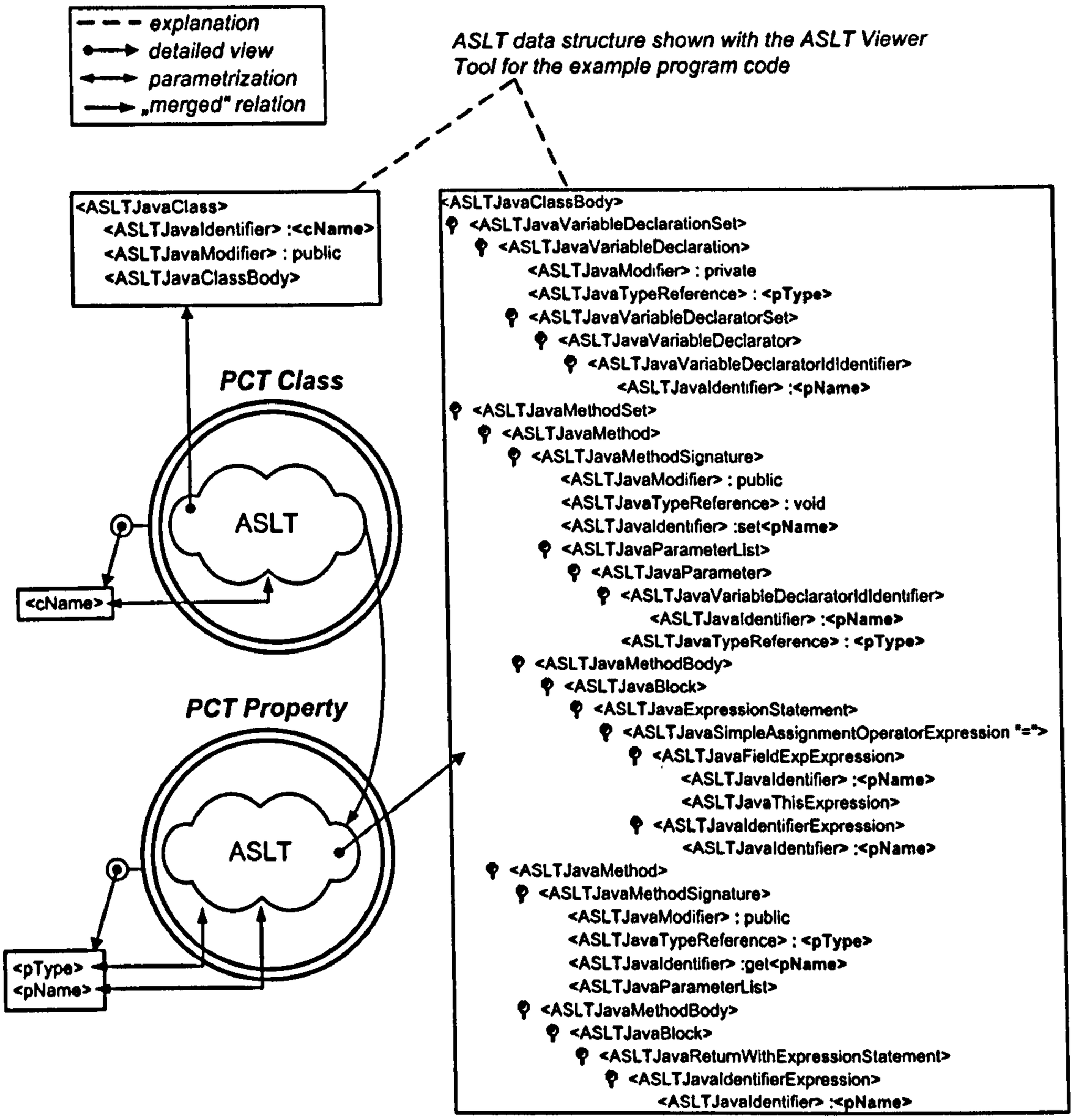


Figure 5.7: An architecture of the PCT Result

Figure 5.7 shows two templates (PCT leaves) `PCT Class` and `PCT Property` that encapsulate code fragments of the example. Code fragments are shown at the atomic level as ASLT data structures. The bold text pieces `cName`, `pType` and `pName` denote parameterised atoms in the ASLT.

5.4.3 Template Architecture Diagram

We defined the template architecture diagram which is needed to show the organisation of PCTs. This includes information about the composites of the PCT, relationships between them, as well as parameters and their relationships.

Figure 5.8 shows an example of the template diagram. The figure shows the PCT called `SomePCT`. It contains three composites which are `ClassPCT`, `PropertyPCT` and `StatementPCTLeaf`. A diamond-like element, that contains a string inside, denotes a parameter. `SomePCT` defines one parameter `className`. The relationship without an arrow shows mapping to parameters of composites defined inside. The arrow with an arrow denotes the "merged" relationship between the composite, which means that their atomic structures are merged.

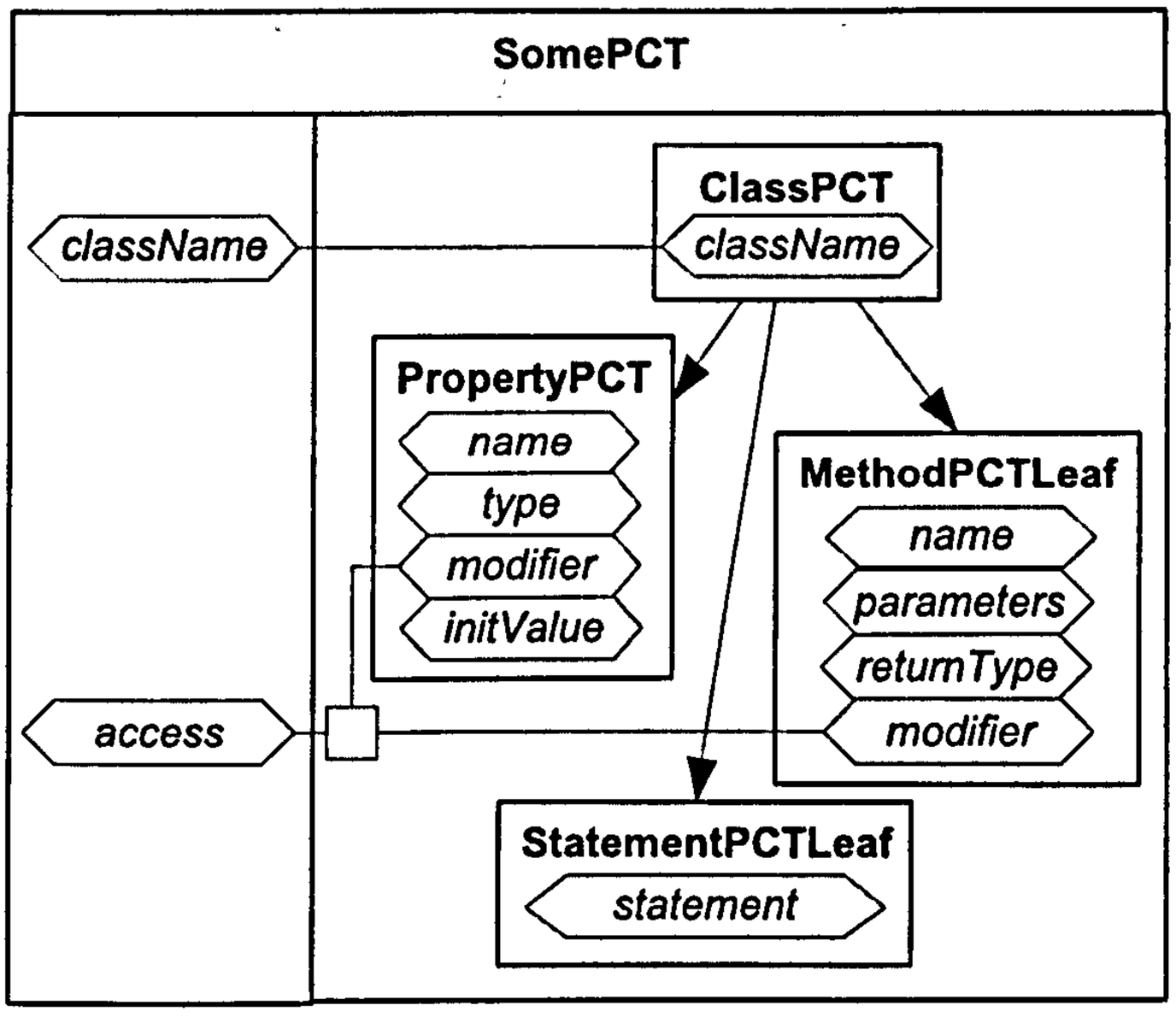


Figure 5.8: An example of template diagram

These kinds of diagrams are used in Chapter 8.

Figure 5.9 shows an extract of the PCT Class hierarchy.

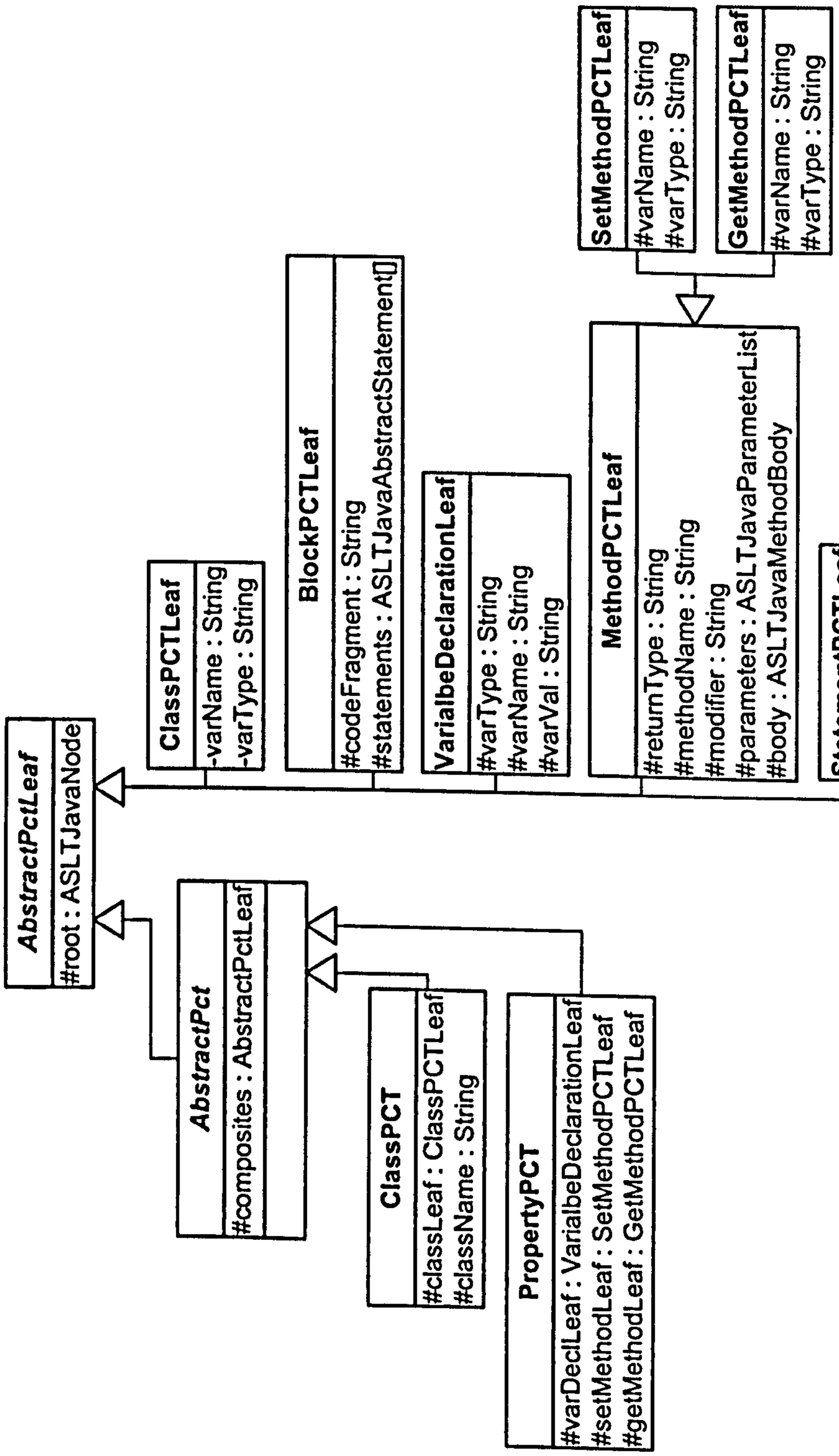


Figure 5.9: UML class diagram: PCT Class hierarchy (extract)

5.4.4 PCT Class Hierarchy

PCTs are defined according to the object-oriented technology, when a system is built as a group of interacting objects. Each object represents some entity of interest in the system that is being modeled, and is characterised by its class, its state, and its behaviour. We have introduced object-oriented systems in the literature review 2.6.2.

PCT types are defined as classes. We apply the concept of inheritance when new classes are formed by using classes that have already been defined. The most common characteristics are defined in the very base class (super class). Further PCT types are specified by extending this base class or other alerady defined PCTs, forming the PCT class hierarchy. Iheritance provides the support for representation by categorisation. An advantage of inheritance is that modules with sufficiently similar interfaces can share a lot of code, reducing the complexity of the program.

To simplify, we only show relevant attributes of classes and do not show behaviour aspects. PCT leaves are defined by the class `AbstractPctLeaf`. These are `ClassPCTLeaf`, `BlockPCTLeaf` and so on. PCTs that can contain other PCTs are defined by the class `AbstractPct`. Those PCTs are used to form code templates that consist of other ones. The figure depicts two of this kind, which are `ClassPCT` and `PropertyPCT`.

5.4.5 PCT Leaves

Basically, all PCT leaves are defined by the class `AbstractPctLeaf`. All PCT leaves have the following common features:

1. They hold an ASLT structure that describes a program code fragment. This structure can be assigned or requested.

2. They define an *initialisation pattern* that should have been used by all derived leaves. The initialisation pattern is a mechanism that we have defined to correctly initialise a PCT leaf. The initialisation assumes the creation of an intial atomic structure (ASLT) that is carried by the leaf, definition of parameters and the specification of connections between them and the atomic structure (ASLT).

3. They own a reference to the PCT (non-leaf) which is a parent. A parent PCT represents a template, part of which are children PCTs.

4. They define some additional behaviour that is a subject of extension. This includes the ability to clone, to generate the code, to persist itself or to load a persisted object.

5. They define a reaction on a composition event, when a leaf is added into the container as a composite. This behaviour is part of an *injection pattern*. The reaction on a composition event is defined by the abstract method `injectInto(AbstractPct pct)`. This method is automaitically called when the PCT is added into the PCT container.

Figure 5.10 depicts a UML class diagram that describes the class `AbstractPctLeaf` with its most relevant attributes, bahaviours and relationships.

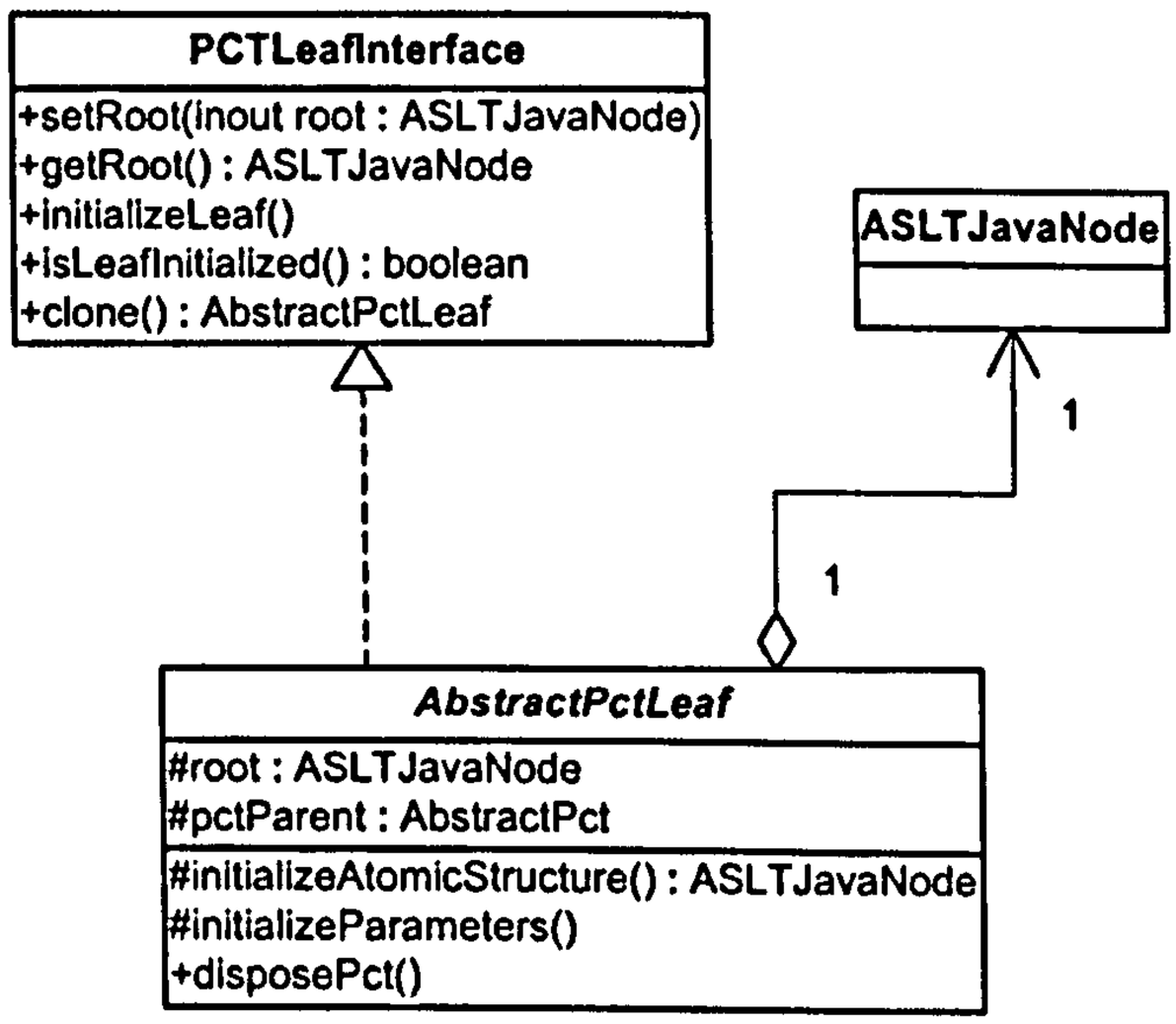


Figure 5.10: UML class diagram: PCT leaf

The basic common behavour of PCT leaves is expressed through the methods defined in the interface `PCTLeafInterface` which is implemented by the abstract class `AbstractPctLeaf`. It is shown that each PCT leaf holds a root node of the type `ASLTJavaNode`. This is the root of an ASLT that describes a program code fragment carried by a PCT leaf. The methods `setRoot()` and `getRoot()` are defined to assign new root or to request a defined one.

5.4.6 Leaves Initialisation

The methods `initializeLeaf()`, `initializeAtomicStructure()`, `initializeParameters()` are related to the already mentioned initialisation pattern. This pattern standardises the initialisation during the instantiation of a leaf. If implemented correctly the pattern is activated during the leaf instantiation and does the following:

1. Starts an initialisation of an atomic structure (ASLT). Initialisation means the generation of an ASLT structure for the encapsulated code fragment. This initialisation, specified with the method `initializeAtomicStructure()`, has to be defined by each derived leaf.

2. Saves the reference to the initialised ASLT.

3. Starts an initialisation of parameters. Variables are defined and connected to the ASLT nodes that are targets of parameterisation. With the connection, a value that is set to a variable, will be correctly passed to the connected node(s) in the ASLT and assigned to some attributes. This initialization, specified with the method `initializeParameters()`, has to be defined by each derived leaf.

Figure 5.11 depicts a UML activity diagram. It shows dynamic aspects of the system concerning the initialization pattern discussed above. The process of initialization is started when the class is instantiated.

With the method `isLeafInitialized()` the leaf can be requested, if it has been initialised.

5.4.7 Derived PCT Leaves

Derived PCT leaves extend already defined ones (see Figure 5.12).

When defining a new PCT leaf, there are some rules that exist to provide initialization and management mechanisms as well as certain reaction mechanisms. With those mechanisms all leaves can be handled in a standard way. Derived PCT leaves have to meet the following requirements:

1. Call an initialization routine of the super class.

2. Override a method `initializeAtomicStructure()`.

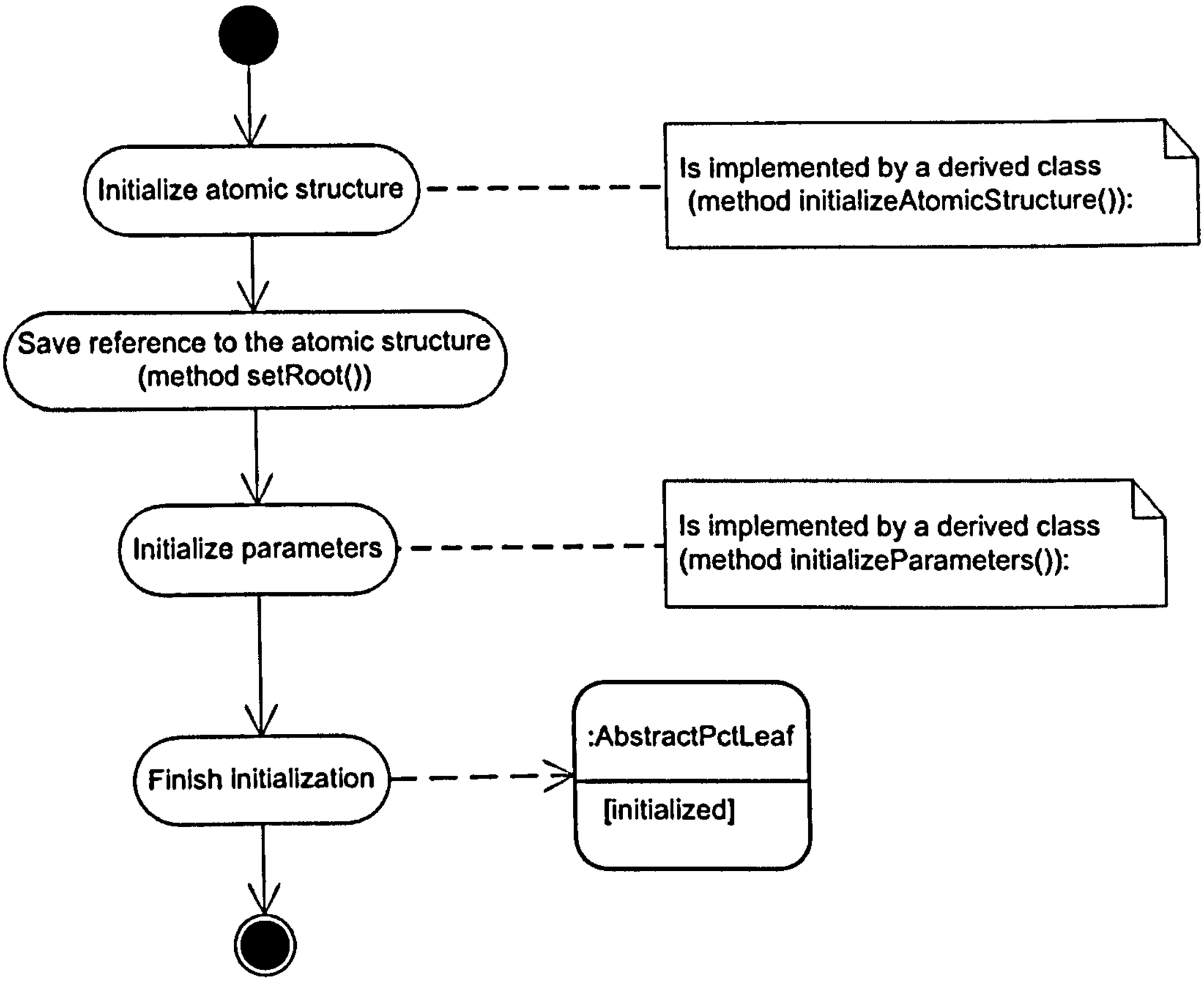


Figure 5.11: UML Activity diagram: Initialization of a PCT leaf

3. Override a method `initializeParameters()`.

4. Define parameters, methods to access them (assign and request a value) and establish connection to one or more objects of the atomic structure (ASLT). These objects can for example be attributes of nodes, nodes themselvs, groups of nodes and groups of attributes.

5. Define management behaviour when the PCT leaf is able to configure its structure in some way.

6. Specify a reaction on the merging with another PCT, by overriding the method `injectInto()`.

When these requirements are met, the derived PCT leaf will have the initializaion behaviour during the instantiation as shown in the activity diagram (see Figure 5.13).

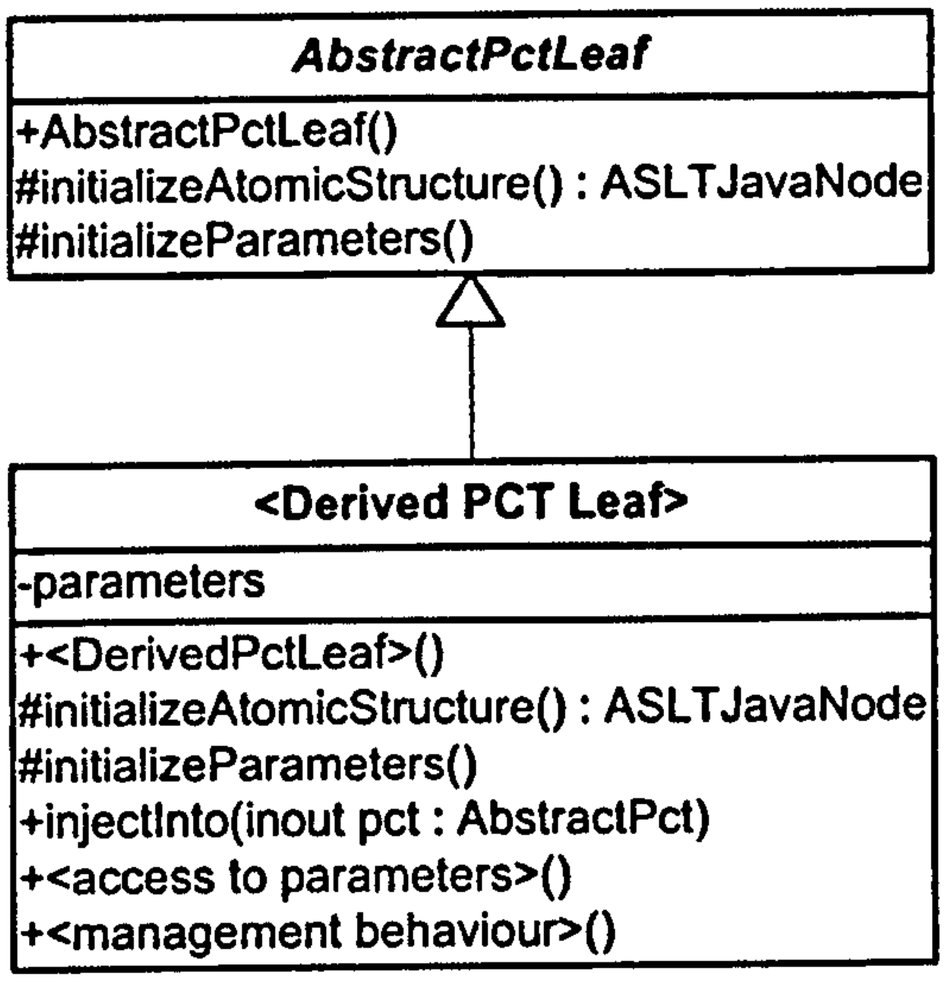


Figure 5.12: UML Class diagram: Derived PCT leaf

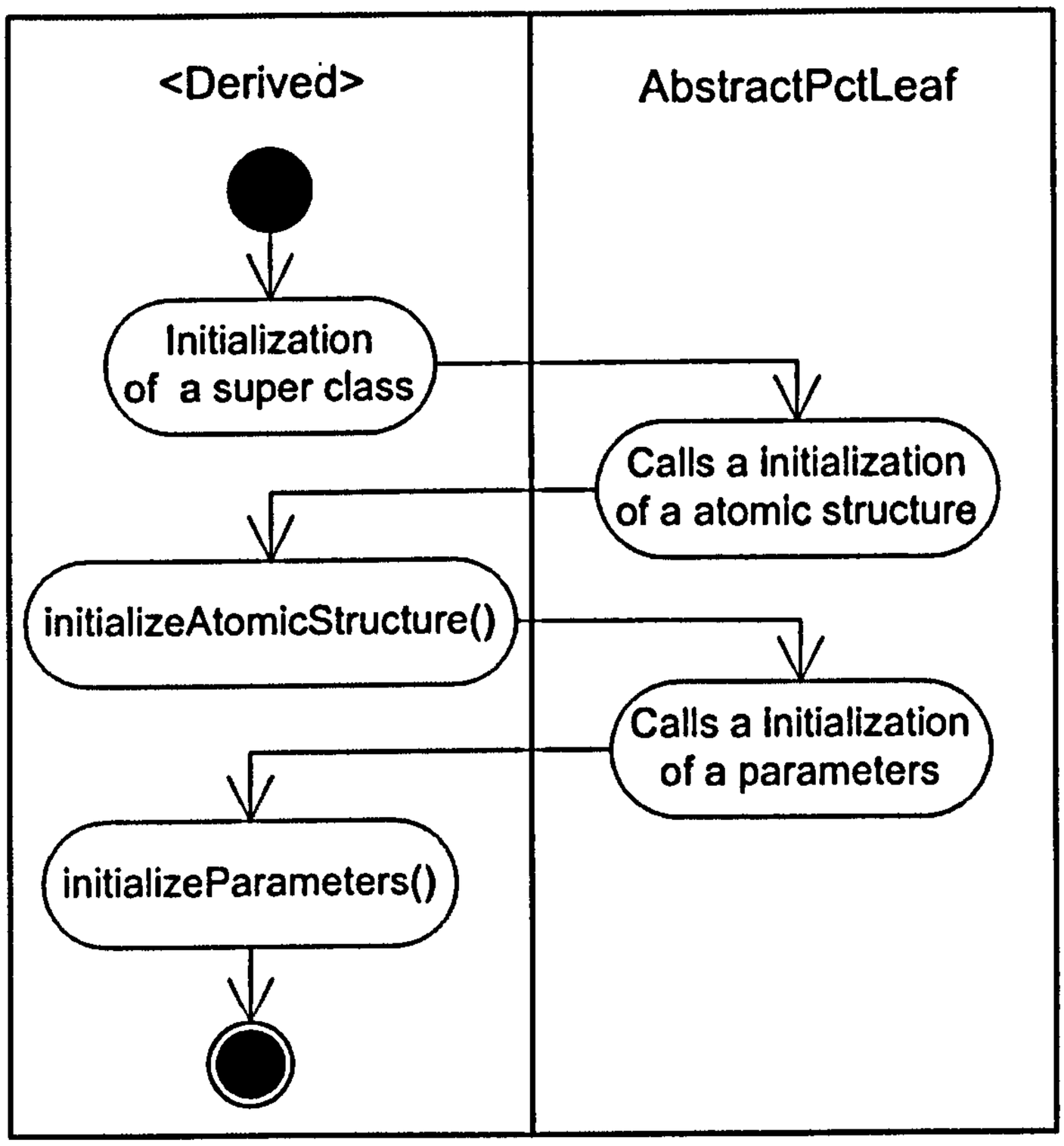


Figure 5.13: UML Activity diagram: Initialization of a derived PCT leaf

The ASLT structure and the values carried by that structure are subjects of parameterization. One or more values caried by the ASLT can be parameterised through a variable. We call this variable a *parameter* of the PCT leaf. Figure 5.14 shows some PCT leaf, which carried ASLT structure is parameterised with the parameters (variables) a, b, c.

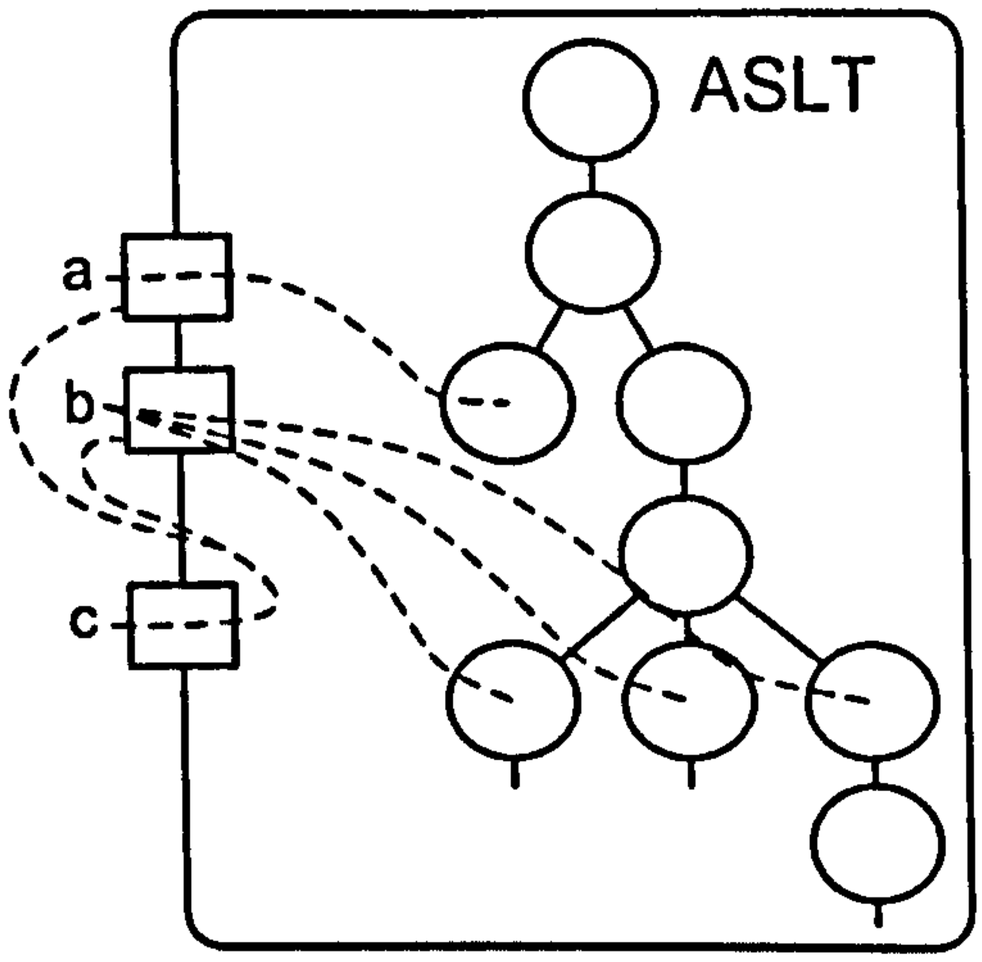


Figure 5.14: Parameters in the PCT leaf

These variables represent three different types of parameterization. With the parameter a an attribute of one node can be assigned with a value. This is a "1 to 1" relationship. The parameter b has got a "1 to many" relationship. The parameter c represents complexer parameterization, when assigning the value to c causes assigning the same or changed value to a and b. There could be more strategies defined of how the values carried by ASLT nodes can be parameterised.

The parameterization of the ASLT structure carried by PCT leaf is defined with so-called *management behaviour* (or also *management methods*). These are methods that specify configuration routines of the ASLT which are typical for the PCT leaf.

5.4.8 PCT Containers

PCT containers can contain other PCTs (that are leaves as well). PCT containers are necessary to specify more complexer templates formed with other existing templates. All PCT containers are defined by the class `AbstractPct` that specifies the following behaviour:

1. Holds other PCTs, both containers and leaves. All contained PCTs are called composites. Composites can be flexibly added and removed. There is a so-called "add and merge" behaviour defined when the reaction request is addressed to the composite which is being added. The "add and merge" behaviour is part of a so-called `injection pattern`.

2. Does not hold an atomic structure (ASLT). But composites that are leaves may do.

3. Specifies analysis behaviour. For example the search of composites by their type.

Figure 5.15 depicts a UML class diagram that describe the class `AbstractPct`. This class describes the PCT containers.

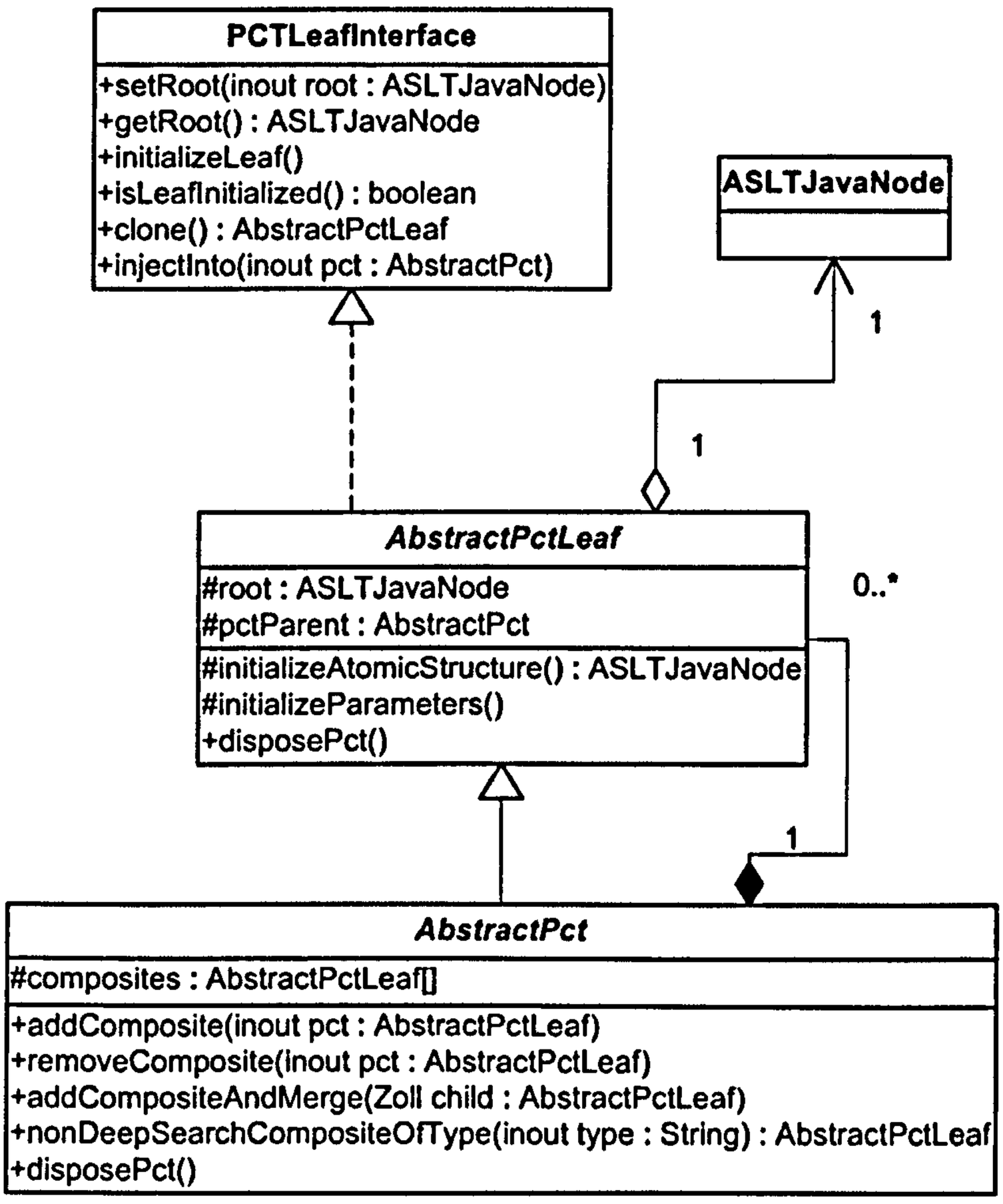


Figure 5.15: UML class diagram: PCT container

As the diagram shows, PCT containers extend PCT leaves. The methods `addComposite()` and `removeComposite()` are used to control the amount of composites that are held by a PCT container. The method `addCompositeAndMerge()` is not only needed to add a new composite but also to intiate a reaction on the composition event. The method `nonDeepSearchCompositeByType()` represents an analysis mechanism that can be used to investigate a container. Finally, the method `disposePct()` is overidden. This method is needed to destroy a PCT. It initiates a process of disposal of its composites.

5.4.9 Merging Composites

Composites may be related or not. If composites are not related, there is no existing dependency between these composites. So, if one PCT is deleted it does not cause changes in the program code held by another PCT. A relationship between composites, if existing, can be of very different nature. Such relationship is established automatically when a PCT is being added as a composite into a container. When that happens, a composite which is being added, receives a so-called "Merge" event. By the "Merge" event a specification of the environment, wherein the composite is added, is passed. The composite may define a reaction on this event. Typically, reaction means an integration into the environment wherein the composite is added. The process of adding a composite and merging is shown in Figure 5.16.

The UML activity diagram shows a workflow which is involved when a PCT is added into the PCT container using an "Add and merge" scheme. This scheme is specified by the method `addCompositeAndMerge()` within a PCT container.

5.4.10 Derived PCT Containers

The basic behaviour of a PCT container is defined with the class `AbstractPct`. Derived classes extend this super class, forming a PCT class hierarchy (we have specified it in Section 5.4.4). Figure 5.17 depicts a UML class diagram that shows method signatures which are specifying the behaviour the derived PCT container class has got.

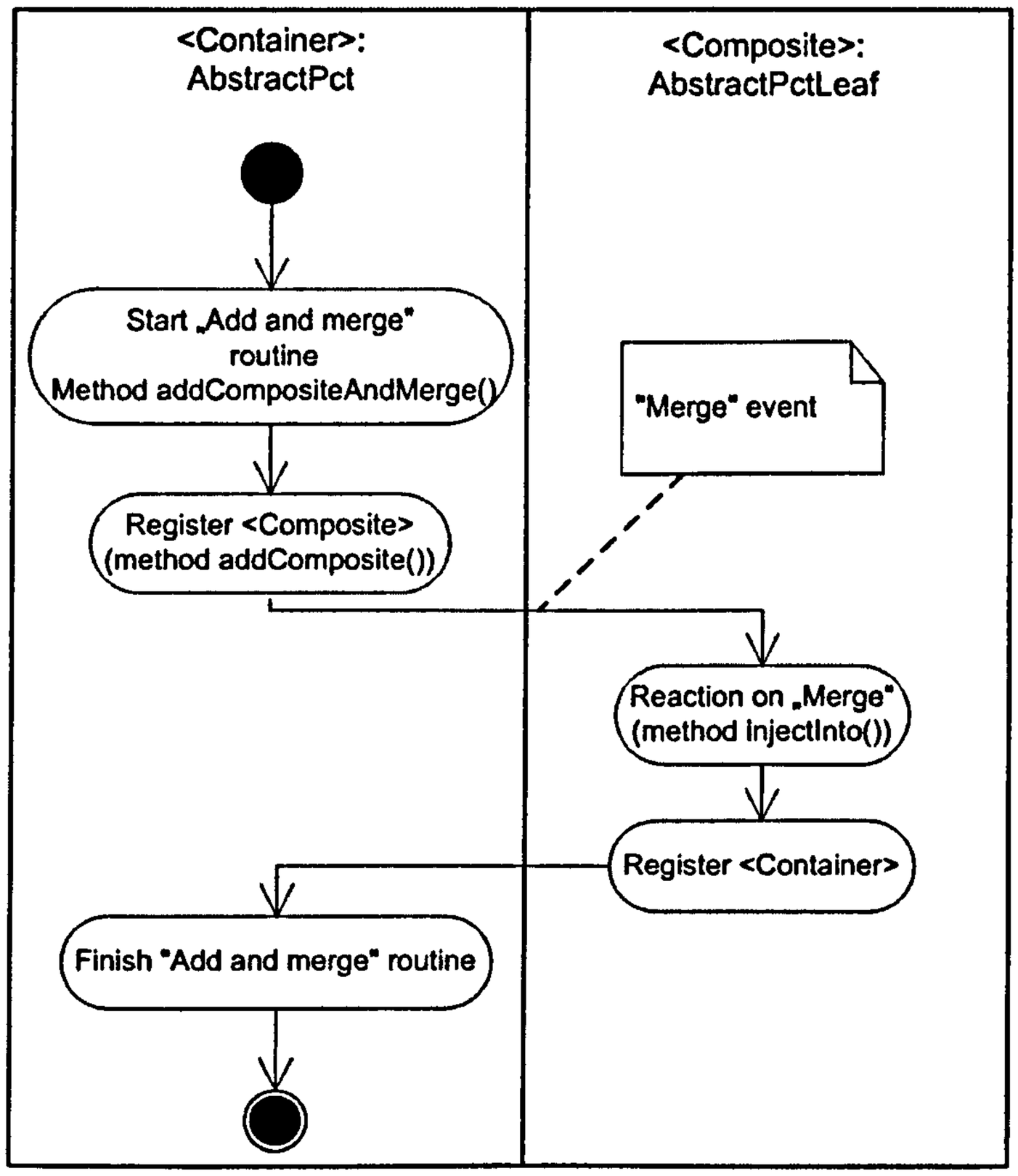


Figure 5.16: UML Activity diagram: Merging Composites

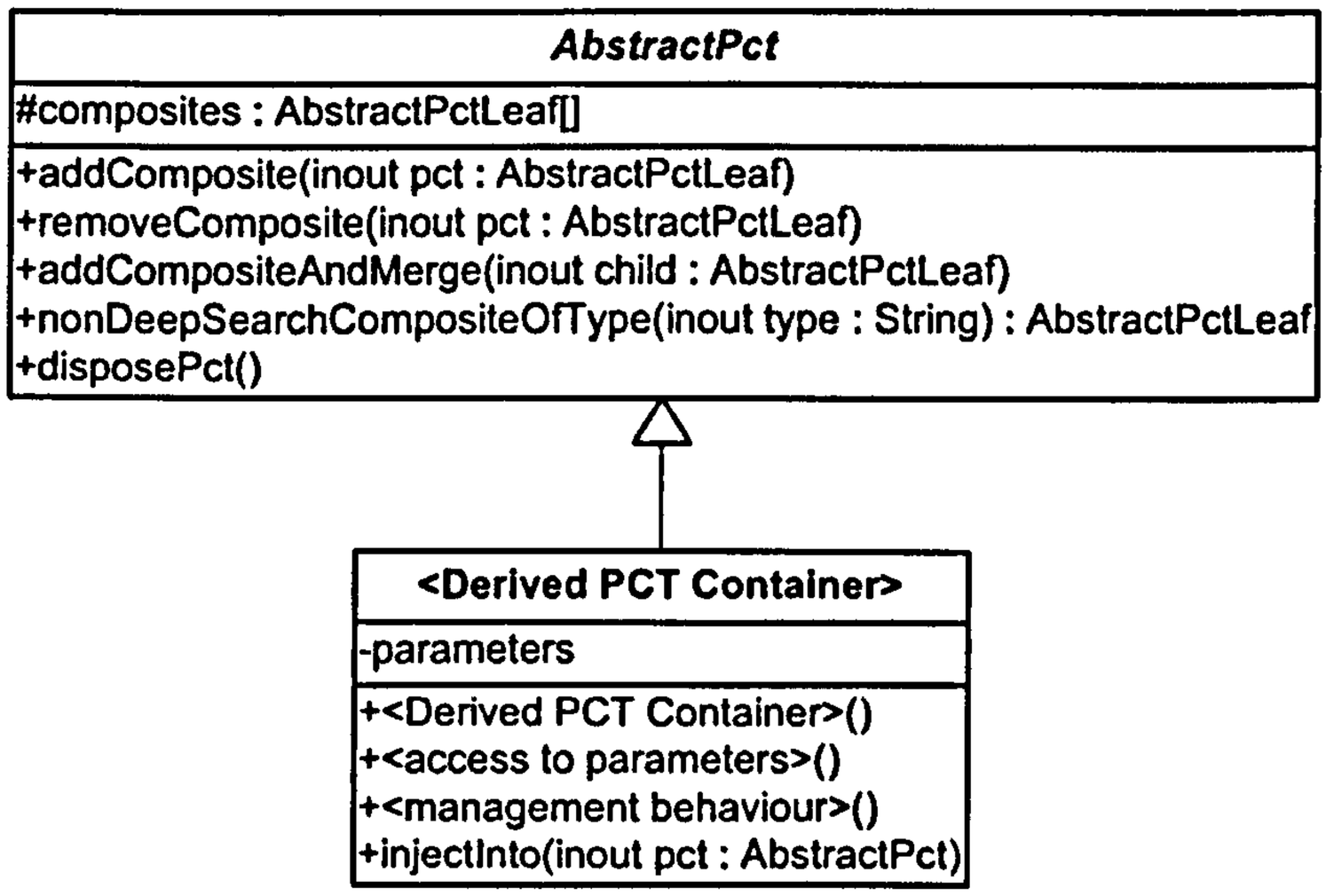


Figure 5.17: UML Class diagram: Derived PCT container

Each method denoted with <. . .> represents a stereotype of behaviour that has to be implemented. The meaning of stereotypes are sprecified in Table 5.1.

Stereotype	Meaning
`<Derived PCT Container>`	Represents a type name of the new PCT container. The method signature `<Derived PCT Container>()` represents a constructor
`<access to parameters>`	Represents methods through which parameters can be assigned with a value or their actual value can be requested
`<management behaviour>`	Represents methods through which the structure of the PCT can be configured

Table 5.1: Derived PCT container: behaviour stereotypes

The parent PCT container may contain such composites as PCT leaves and PCT containers. The parameters of the parent container are mapped to the parameters defined by its composites. Relationships between the parent's and the composite's parameters are typically "one to one" and "one to many". This is shown in Figure 5.18.

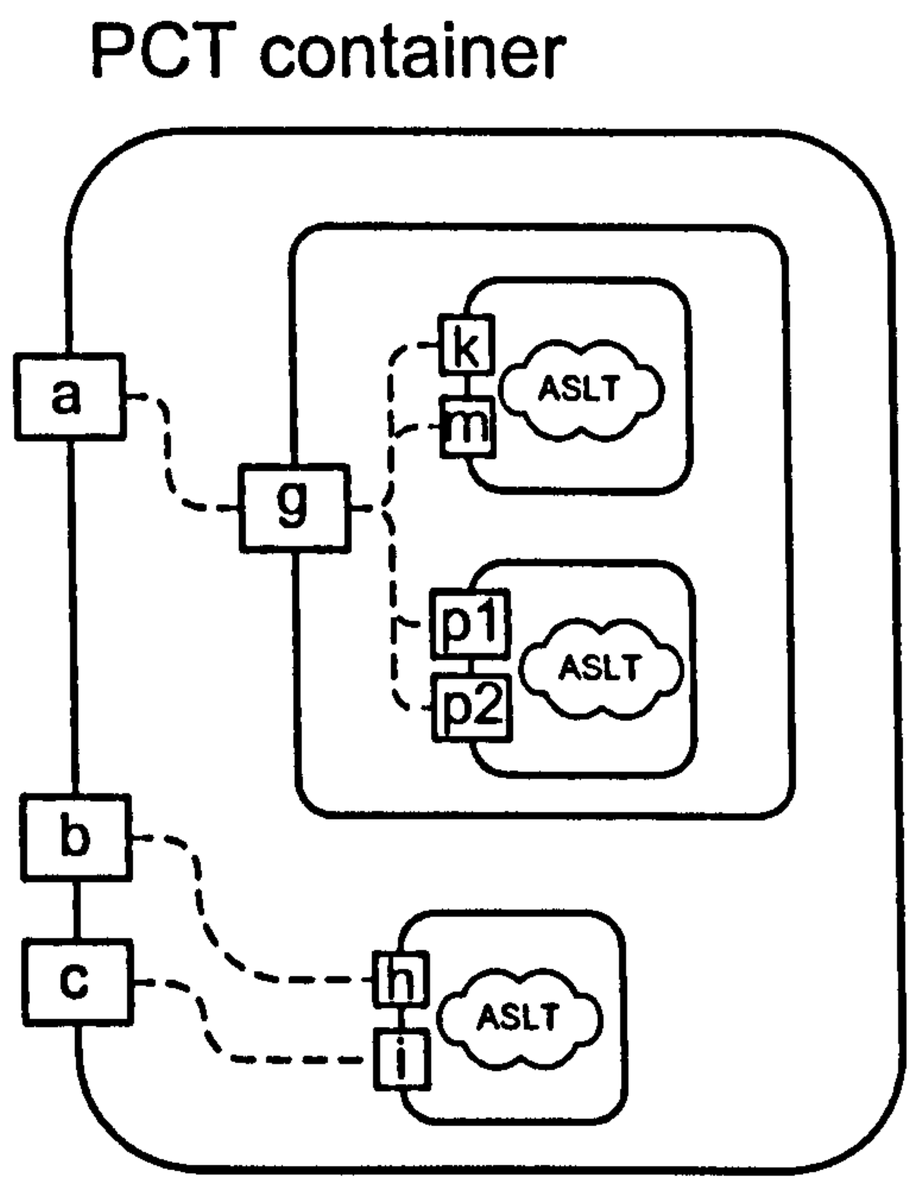


Figure 5.18: Parameters in the PCT container

An access to the parameters is managed by the methods `setXXX()` and `getXXX()`, where `XXX` is the name of the property, the first letter of which is in the upper case.

The management behaviour is defined by additional methods. There is a wide spectrum of what management could be. The configuration of a PCT is the common for all management routines. A principle example is given in Figure 5.19.

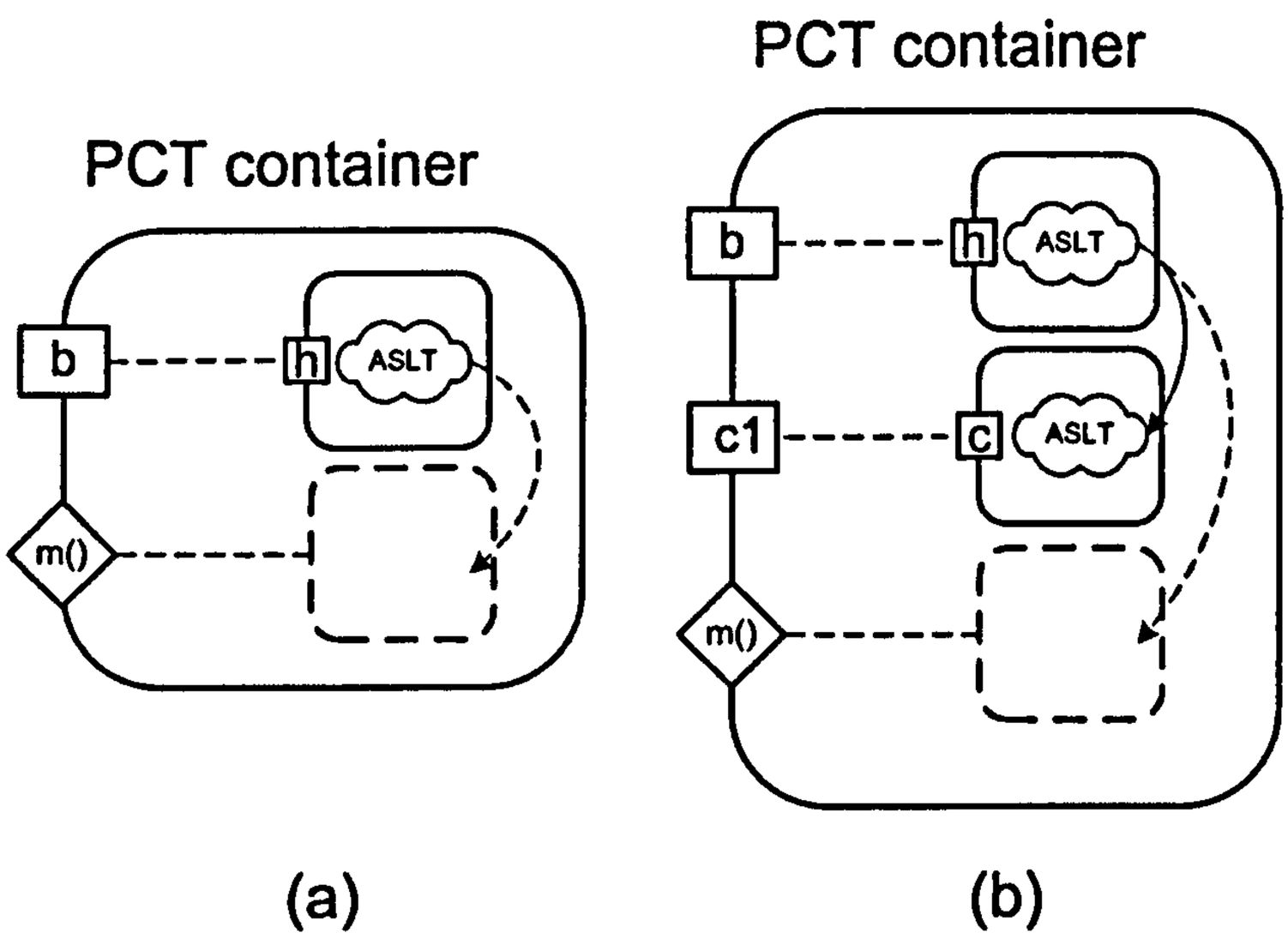


Figure 5.19: Management behaviour in the PCT container. (a) - PCT container with *m()* management behaviour defined; (b) - a structure of the PCT container after method *m()* is called

The figure shows a structure of the PCT container before calling the method `m()` and afterwards. Part (a) shows the PCT container that has a parameter `b` mapped to the parameter `h` of its composite. The method `m()` defines a management behaviour. If called, it generates a composite, merges with the composite initialy presented, and extends an interface with a new parameter mapped to the newly generated composite.

5.5 Molecular Operations

PCTs are half-passive entities. Passive means that they themselfs cannot cause a transformation of the target software system. The reason for the prefix "half" is that reaction mechanisms can be specified for a PCT by demand so that the PCT becomes able auto-

matically reacting on certain events coming from the environment (other PCTs). Events are initiated by applying molecular operations.

A Molecular Operation (MO) is a rule to manipulate with one or more PCTs. A manipulation typically is a mixture of the analysis and transformation routines.

Analysis routines are needed to request a strucuture of PCTs in order to perform a certain act of manipulation. Analysis routines take place in forming pre- and post-conditions. We use the term transformation to denote the process of changing one code fragment into another.

Add/remove/set/request parameters and composites, instantiate/remove/merge/search PCTs are the examples of what molecular operations can do. Molecular operations may be formed with defined atomic operations and other molecular operations. We specify some examples of molecular operations in Table 5.2.

Name	**Notation**	**Description**
Instantiation	$new^m(t)$	Creates and returns a new instance of a type t
Set Parameter	$setP^m(N, p, v)$	Sets the value v to the parameter p of the PCT N
Merge	$\oplus^m(N, M)$	Merges (binding at the atomic level) the PCT N and the PCT M according to their reaction on the "merge" event.
Add/remove Composite	$+c^m(N, M)$, $-c^m(N, M)$	Adds/Removes the specified composite M into/from the PCT N
Search first composite by type	$?c^m(N, t)$	Searches within the PCT N for a composite of type t

Table 5.2: Molecular operations

5.5.1 Architecture of a Molecular Operation

We show the basic architecture and behaviour of molecular operations by using object-oriented technology. A molecular operation is defined as a type in the class `AbstractOperation`. This class collects features that are common for all derived molecular operations. Figure 5.20 depicts a UML class diagram of `AbstractOperation`.

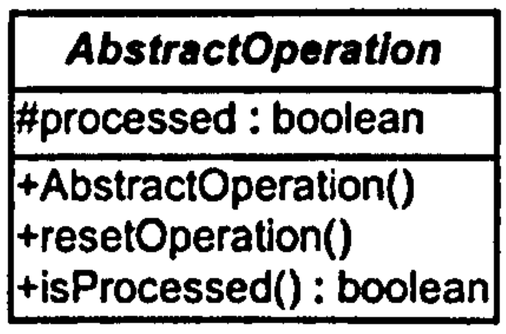


Figure 5.20: UML Class diagram: abstract class *AbstractOperation*

Basically, the class `AbstractOperation` defines a quite minimal behaviour. It is expressed by two methods. The first one is the `resetOperation()` that brings the molecular operation into the initial state. The second one is `isProcessed()`, that can be used to request the molecular operation if it has already been processed.

5.5.2 Molecular Operations Class Hierarchy

In the same fashion as PCTs, molecular operations are defined according to the object-oriented technology, when a system is built as a group of interacting objects. We explain the class hierarchy of molecular operation in the same fashion as we have done for PCTs in Section 5.4.4.

Object-orientation assumes, that each object represents, some entity of interest in the system being modeled, and is characterised by its class, its state, and its behaviour. We have introduced object-oriented systems in the literature review 2.6.2.

Molecular operations are defined as classes. We apply the concept of inheritance when the new classes are formed by using classes that have already been defined. The most common characteristics are defined in the very base class (super class). Further types of molecular operations are specified by extending this base class or other alerady defined ones, forming the class hierarchy of molecular operations. Iheritance provides the support for the representation by categorization. An advantage of inheritance is that modules with sufficiently similar interfaces can share a lot of the code, reducing the complexity of the program. Figure 5.21 shows an extract of the PCT class hierarchy.

5.5.3 Derived Molecular Operations

Derived molecular operation extend already defined ones (see Figure 5.22).

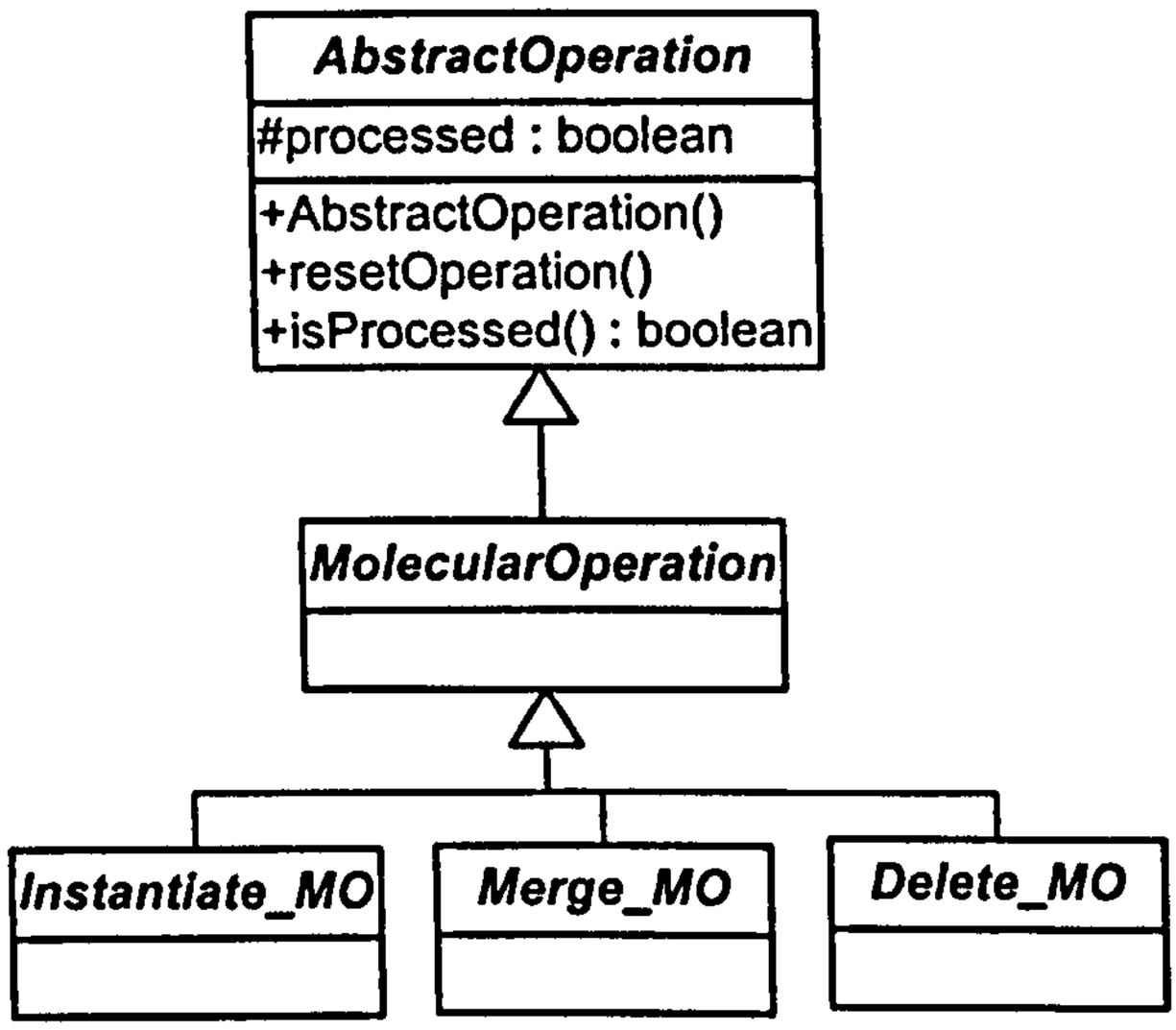


Figure 5.21: UML Class diagram: class hierarchy of molecular operations

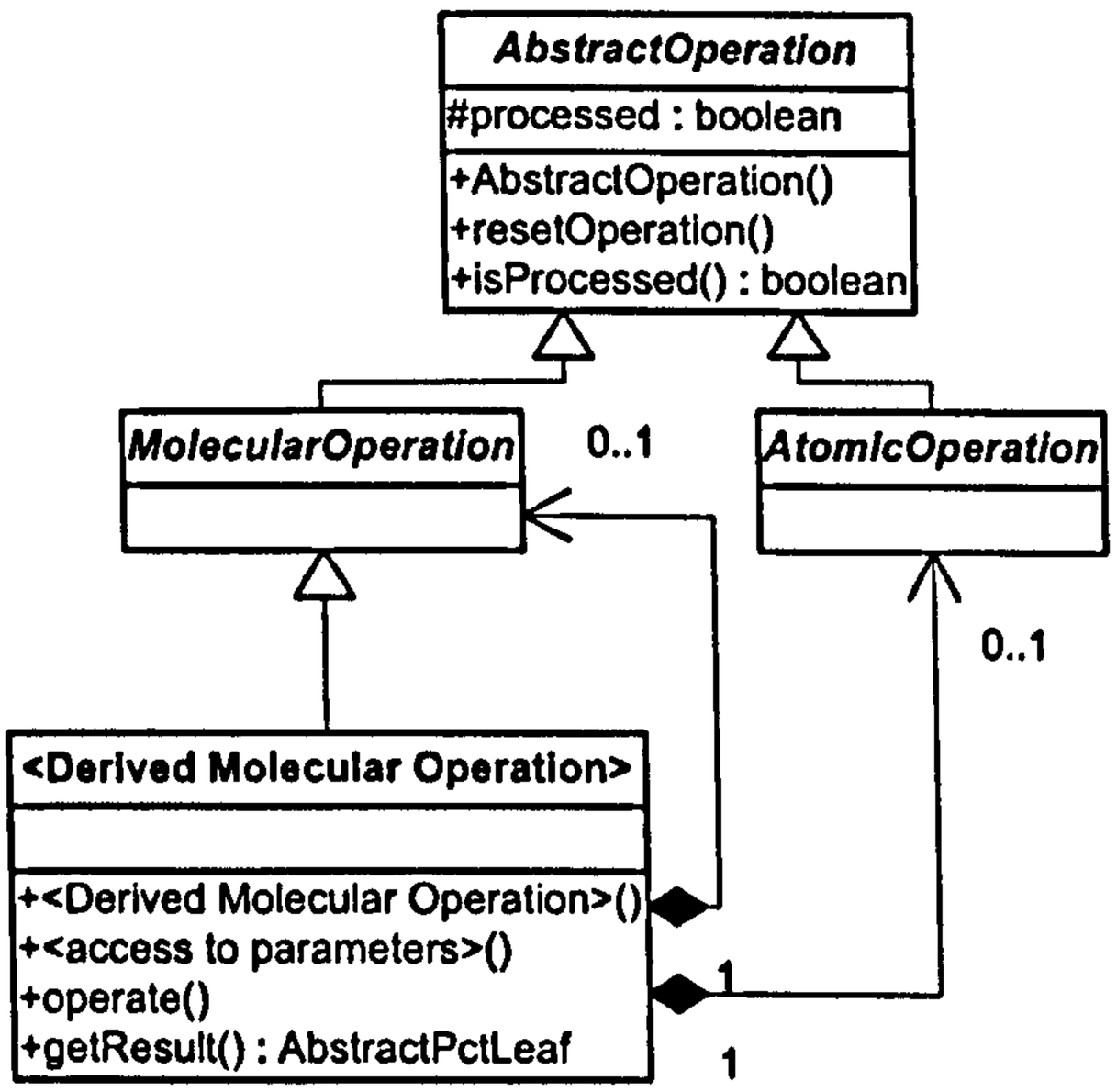


Figure 5.22: UML Class diagram: derived molecular operations

The figure shows that the category of molecular operations, represented by the class `MolecularOperation`, is separated from the category of atomic operations, represented by the class `AtomicOperation`. The derived class that represents a new molecular operation extends the class `MolecularOperation`. It may be composed with

other already defined atomic and molecular operations. This is shown with composition dependencies.

In the derived class, denoted as `<Derived Molecular Operation>`, each method denoted with `<...>` represents a stereotype of behaviour to be implemented. The meaning of stereotypes are the following:

- `<Derived Molecular Operation>`. Represents a type name of the new molecular operation. The method signature `<Derived Molecular Operation>()` represents a constructor.

- `<access to parameters>`. Represents methods through which parameters of the molecular operation can be assigned with values or their actual value can be requested.

The method `operate()` is the core method of the molecular operation. This method implements a manipulation rule of the molecular operation. With the method `getResult()` the PCT, that is the result of the operation, may be requested.

Further we introduce three basic molecular operations that are used at the Template Level of composition.

5.5.3.1 "Instantiate" Molecular Operation

The molecular operation "Instantiate" is one of the basic operations in the template-based composition. It is used to create exemplars of PCTs according to their type specification. Figure 5.23 shows the specification of the class representing the "Instantiate" molecular operation.

The operation is defined by the class `Instantiate_MO`. The class has a dependency relationship with the class `AtomicOperation` that represents atomic operations. This relationship means that atomic operations are involved during the processing of the "Instantiate" molecular operation. Fields and methods of the class `Instantiate_MO` are described in Table 5.3.

Fields/Methods	Description
`pctTypeName`	The parameter of the molecular operation that represents a name of the PCT type that is to be instantiated with the molecular operation
`instance`	An exemplar that is created when the operation is processed

Fields/Methods	Description
`Instantiate_MO()`	A constructor with no arguments. May be called when an exemplar of the operation is created
`Instantiate_MO (pctTypeName:String)`	A constructor with the argument `pctTypeName` of type `String`. The argument represents a name of the PCT type which is supposed to be instantiated with the operation
`setPctTypeName (pctTypeName:String)`	A method to assign the value to the `pctTypeName`
`getPctTypeName():String`	The method returns the value of the parameter `pctTypeName`
`operate()`	Generates an exemplar of PCT type specified by the parameter `pctTypeName`
`getResult()`	Returns the result of the operation hold by the variable `instance`

Table 5.3: Fields and methods of the class *Instantiate_MO*

The method `operate()` that contains the core semantics of the "Instantiate" operation is shown in Listing 5.9.

```
1 public operate(){ ...
2 (1) Class theClass = Class.forName(getPctTypeName());
3 (2) Object instObj = theClass.newInstance();
4 (3) instance = (AbstractPctLeaf) instObj;
5 (4) processed = true; ...}
```

Listing 5.9: Method operate() of the "Instantiate" molecular operation

The method `operate()` defines four main steps. At step (1) and (2) the reflection API of the Java programming language is used in order to instantiate an exemplar `theClass` of a type specified by `pctTypeName` variable. The value of that variable is requested with the help of method `getPctTypeName()`. At step (3) a reference to the created exemplar is casted into the correct type `AbstractPctLeaf` and saved into the variable `instance`. At step (4) the state of the operation, represented by the variable `processed`, is set to "processed".

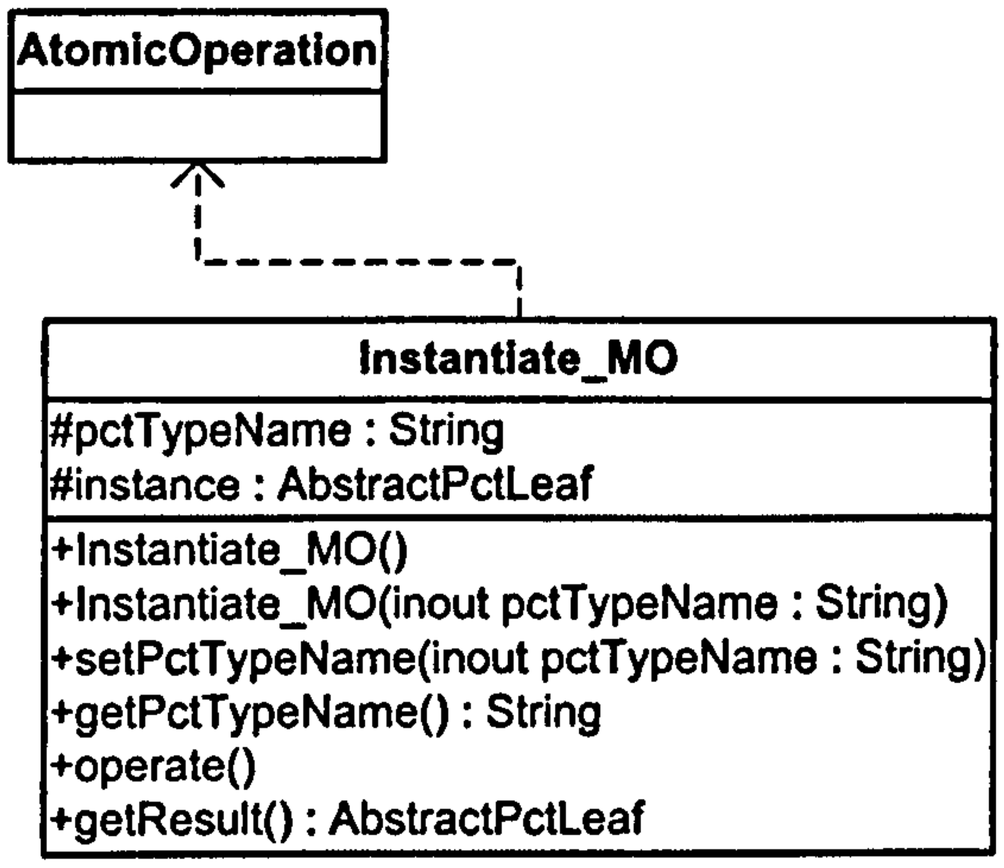


Figure 5.23: UML Class diagram: "Instantiate" molecular operation

The method `operate()` does not directly use any atomic operations. However they are used when a PCT is instantiated. When that happens, its constructor is automatically called. According to the initialization pattern described in Section 5.4.6, an atomic structure is initialised. At this time the atomic operations, such as an instantiation of an ASLT node, are used.

5.5.3.2 "Merge" Molecular Operation

The molecular operation "Merge" is one of the basic operations in the template-based composition. It is used to compose two PCTs. One of it is a container and the other one is a composite. Besides, the molecular operation initiates a "Merge" event, causing a reaction of a composite on the new parent environment where it is placed. Figure 5.24 shows the specification of the class representing the "Merge" molecular operation.

The operation is defined by the class `Merge_MO`. The class has a dependency relationship with the class `AtomicOperation` that represents atomic operations. This relationship means that atomic operations are involved during the processing of the "Merge" molecular operation. Fields and methods of the class `Merge_MO` are described in Table 5.4.

Fields/Methods	Description
`compositePct`	The parameter of the molecular operation that represents a composite that the operation is adding into the parent PCT container

Fields/Methods	Description
`parentPct`	The parameter of the molecular operation that represents a parent container wherein the composite is going to be added
`Merge_MO()`	A constructor with no arguments. May be called when an exemplar of the operation is created
`Merge_MO (parentPct:..., compositePct:...)`	A constructor with the arguments `parentPct` and `compositePct` representing a parent container and a composite respectively
`setParentPct (parentPct:...)`	A method to assign the value to the parameter `parentPct`
`setCompositePct (compositePct:...)`	A method to assign the value to the parameter `compositePct`
`getParentPct()`	Returns a value of the parameter `parentPct`
`getCompositePct()`	Returns a value of the parameter `compositePct`
`operate()`	Adds a composite `compositePct` into the PCT parent container `parentPct` and initiates a notification of the `compositePct` with the event "Merge". This may cause a reaction on the composite, if defined. Section 5.4.9 gives some additional information on merging.
`getResult()`	Returns the result of the operation held by the parent container (variable `parentPct`)

Table 5.4: Fields and methods of the class *Merge_MO*

The method `operate()`, that contains the core semantics of the "Merge" operation, is shown in Listing 5.10.

```
1 public operate(){...
2 (1) parentPct.addCompositeAndMerge(compositePct);
3 (2) processed = true; ...}
```

Listing 5.10: Method operate() of the "Merge" molecular operation

The method `operate()` defines two main steps. Step (1) gives a command to the parent PCT container (variable `parentPct`) to add a composite (variable `compositePct`) and to notify this composite with the "Merge" event. This will cause a reaction on the

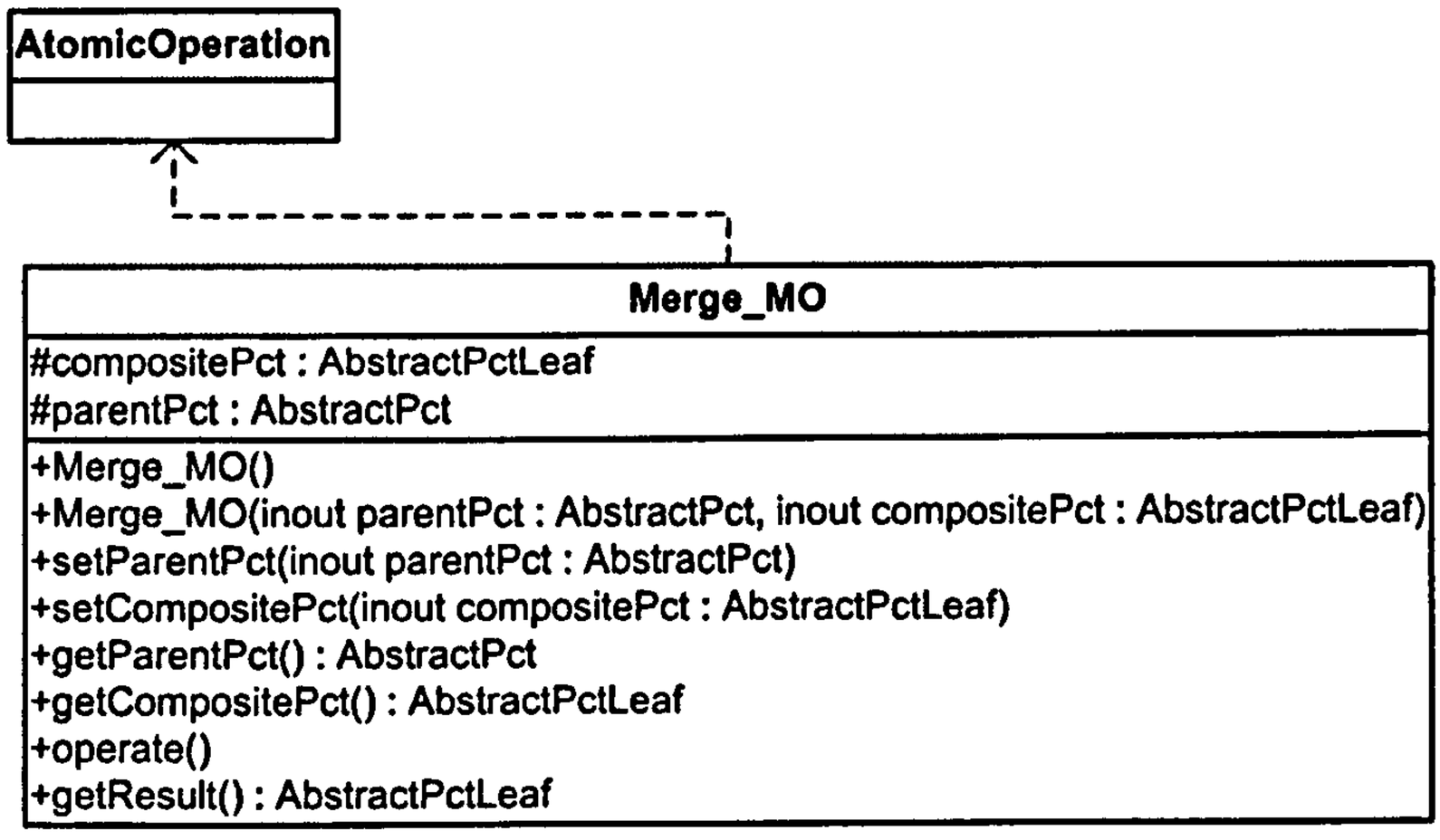


Figure 5.24: UML Class diagram: "Merge" molecular operation

`compositePct` (if this composite defines a reaction). We have already introduced this in Section 5.4.9. At step (2) the state of the operation, represented by the variable `processed`, is set to "processed".

The method `operate()` does not directly use any atomic operations. However they may be used when the composite is reacting on the "Merge" event.

5.5.3.3 "Delete" Molecular Operation

The molecular operation "Delete" is one of the basic operations in the template-based composition. It is used to remove a specified PCT instance. The molecular operation initiates a "Dispose" event that is sent to the instance that has to be removed. This instance reacts on the event by performing a correct removing itself. Figure 5.25 shows the specification of the class, representing the "Delete" molecular operation.

The operation is defined by the class `Delete_MO`. The class has a dependency relationship with the class `AtomicOperation` that represents atomic operations. This relationship means that atomic operations are involved during the processing of the "Delete" molecular operation. Fields and methods of the class `Delete_MO` are described in Table 5.5.

The method `operate()` that contains the core semantics of the "Delete" operation is shown in Listing 5.11. The method `operate()` defines two main steps. At step (1) the "Dispose" event is sent to the PCT instance (variable `targetPct`) which is going

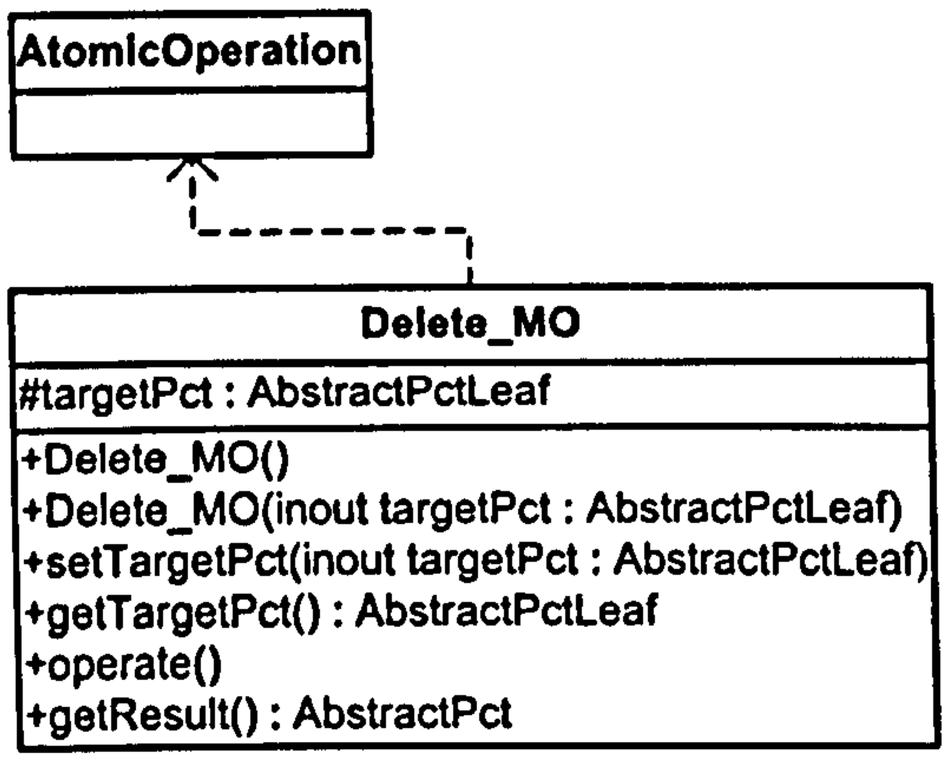


Figure 5.25: UML Class diagram: "Delete" molecular operation

Fields/Methods	Description
`targetPct`	The PCT instance which is to be removed
`Delete_MO()`	A constructor with no arguments. May be called when an exemplar of the operation is created
`Delete_MO( targetPct:...)`	A constructor with the argument `targetPct` that represents a PCT instance to be removed
`setTargetPct (targetPct:...)`	A method to assign the value to the parameter `targetPct`
`getTargetPct()`	Returns a value of the parameter `targetPct`
`operate()`	Sends a "Dispose" event to the PCT instance
`getResult()`	Returns a reference to the PCT container, that the removed composite has had before removing

Table 5.5: Fields and methods of the class *Delete_MO*

to be removed. The sending of a "Dispose" event is expressed via calling the method `disposePct()` that is part of the PCT specification. This will cause a reaction on the `targetPct` (if this composite defines a reaction) on the event. At step (2) the state of the operation, represented by the variable `processed`, is set to "processed". The method `operate()` does not directly use any atomic operations. However they may be used when the composite is reacting on the "Dispose" event.

```
1 public operate(){...
2 (1) targetPct.disposePct();
3 (2) processed = true; ...}
```

Listing 5.11: Method operate() of the "Delete" molecular operation

5.6 Expressions

Sentences of PCT-L are expressions that consist of molecular operations as "operators" and templates as "operands". Expressions are described and processed with the help of an abstract syntax tree-like datastructure, further referred to as expression tree. Figure 5.26 shows schematically how such a tree is defined.

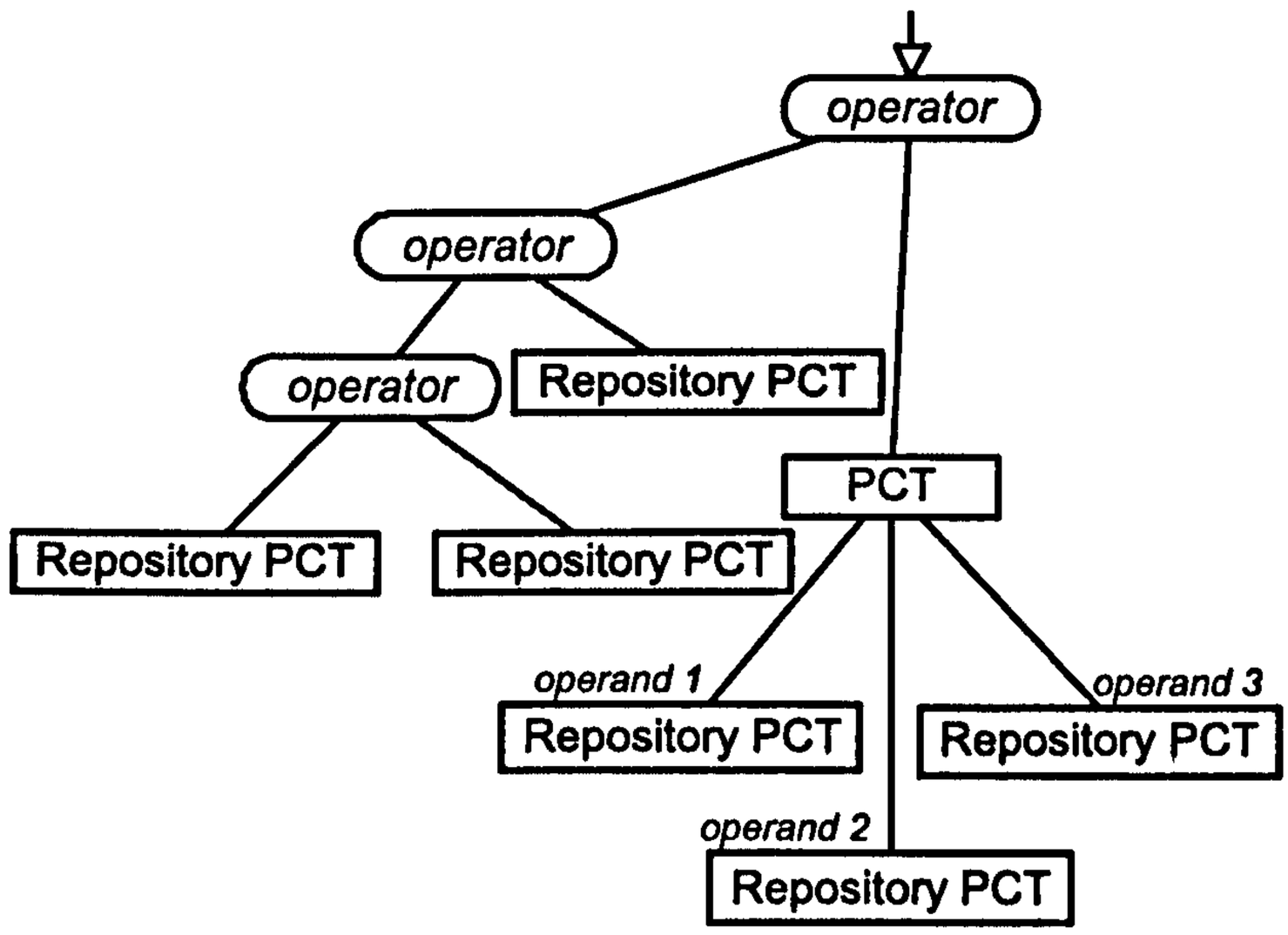


Figure 5.26: An expression tree

With red colour the predefined templates from the common repository are marked. The expression tree shows the order in which operators are applied to group of operands as well as order within these groups. The arrow on the top of the expression means the resulted value. Consider the example of an expression shown in Figure 5.27.

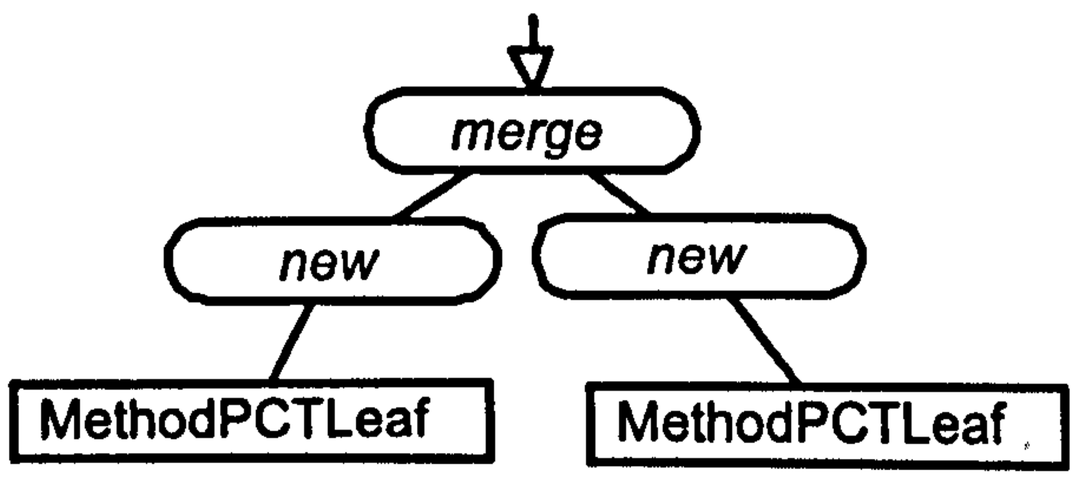


Figure 5.27: An expression shown as a tree

The depicted example consists of three molecular operations: one *merge* and two *new*. During processing of the expression, firstly the *new* operations are processed in order to obtain input for the *merge* operation.

We use object-oriented technology to describe expressions. Each node in the expression tree is represented by an object that is an exemplar of the class `AbstractPctlExpressionNode`. Figure 5.28 shows a specification of this class.

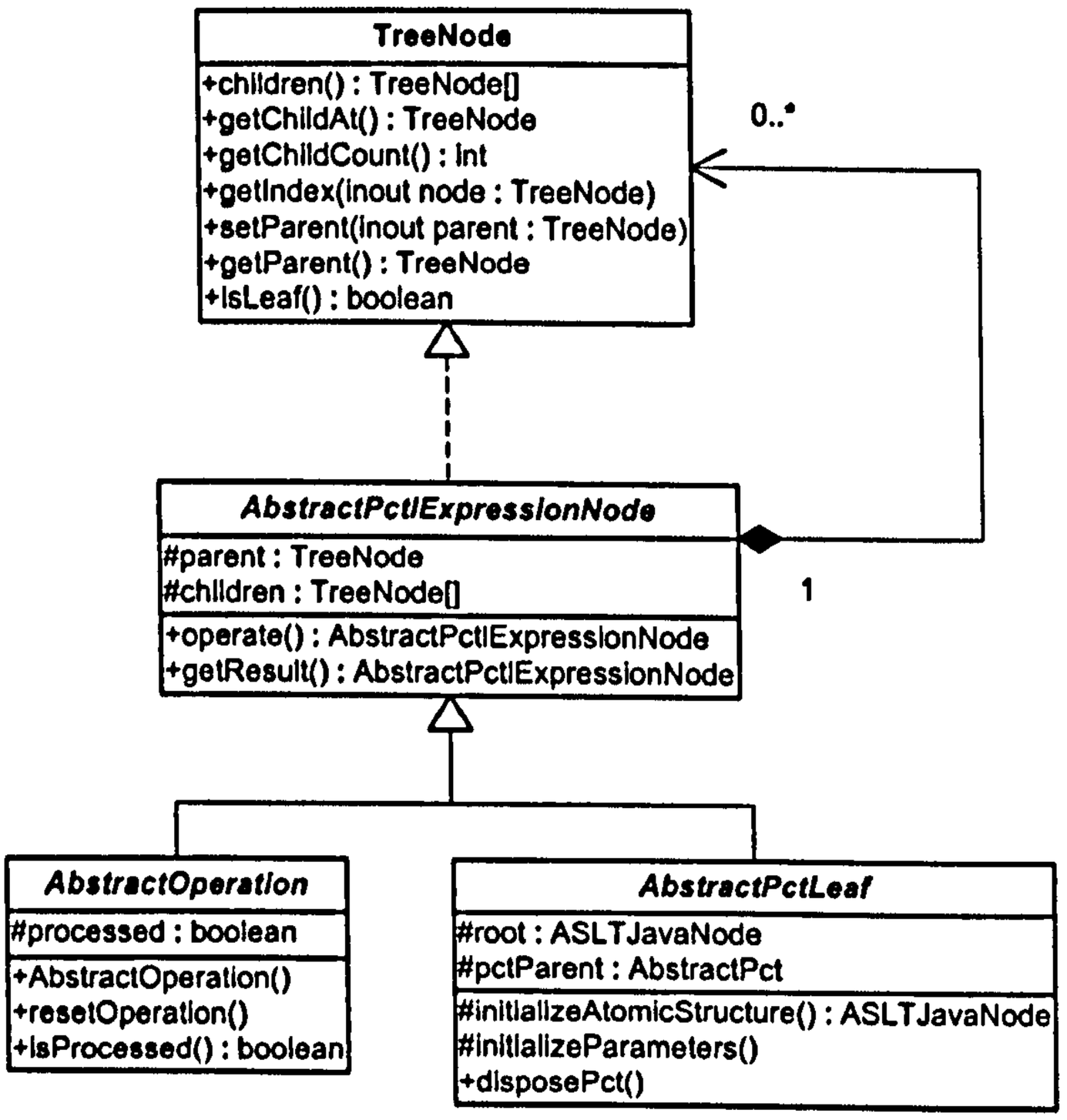


Figure 5.28: UML Class diagram of the class AbstractPctlExpressionNode representing nodes in the expression tree

According to the UML Class diagram, each node in the exression tree has the following behaviour:

1. Each node is a tree node. This is shown with the "realization" relationship between the class `AbstractPctlExpressionNode` and the interface `TreeNode`. That means:
   (a) A node may have multiple children nodes (methods `children()`, `getChildAt()`, `getChildCount()` and `getIndex()`). A composition relationship shows that the children of a node have tree node behaviour (the class `TreeNode`).
   (b) If a node is not a root of the expression tree, it has one parent node (accessed via methods `setParent()` and `getParent()`).
   (c) A node may be defined as a leaf (this is requested with the method `isLeaf()`).
2. Each node in the tree, if not a leaf, represents a part of the whole expression. Each node is able to initiate a processing of a part of expression it holds (the method `operate()`). The result is held by each root of a sub-tree, that represents an expression and may be accessed via the method `getResult()`.
3. Nodes in the expression tree are molecular operations and PCTs. This is shown with the generalization relationships

An expression defined via linked objects (nodes) in tree form can be executed by calling `operate()` method of the root object (node). The way of how expression is processed can be described with the flowchart depicted in Figure 5.29.

Initially, all preconditions are checked which may include checking if operation has been already processed, if all parameters are of correct types and have correct states. If all requirements are met, then all sub-trees are recursively executed by calling the `operate()` method for each sub-root. Once, all sub-expressions are calculated the operation starts working with its parameters. Typically these parameters denote PCTs which should be reconfigured. The operation analyses and re-configures them. If there are no failures occurred, the flag that denotes completeness of processing is set.

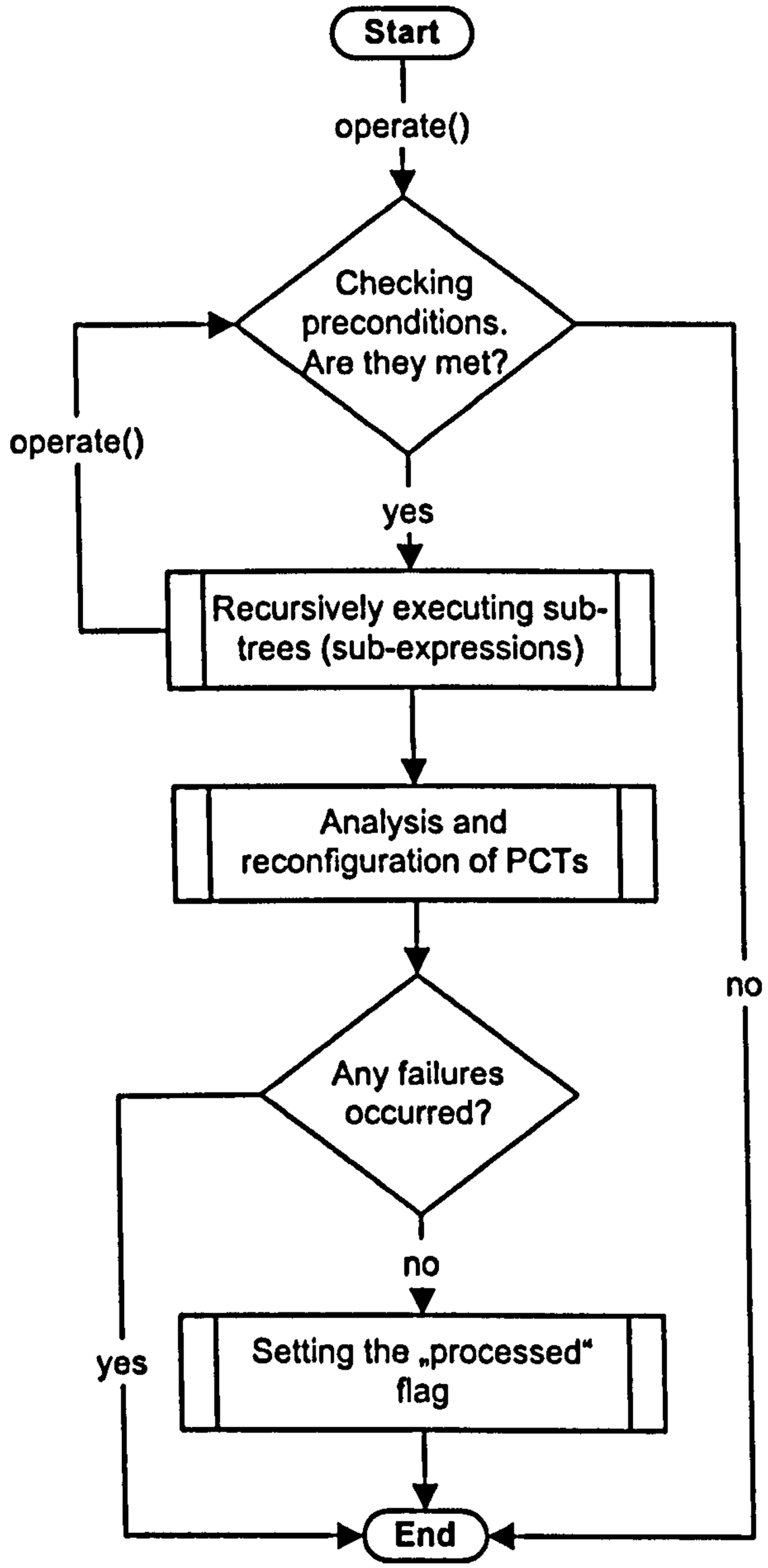


Figure 5.29: A flowchart of an algorithm for processing a PCT-L expression

5.7 Development of PCTs and MOs

PCTs and MOs specify design strategies of a programmer. Few examples follow:

- A `Property` PCT and `Delete` MO specify a strategy to extract a property, defined via a variable declaration and variable access methods, from a specified class.
- A `Constructor` PCT and a `Delete` MO specify a strategy to extract a constructor from a specified class.
- A `Thread` PCT and a `Merge` MO specify a strategy to embody a thread mechanism into a specified class.

Normally for a given programming language, in our case Java, there is a core repository of PCTs and MOs. Typically the repository is formed at least with the following elements:

- `Instantiate`, `Delete` and `Merge` molecular operations and
- PCTs representing non-terminals of a given programming language.

We have implemented a basic repository which has more than enough elements needed for building up use case composition systems presented in Chapter 8. Moreover, other composition systems can be defined using the repository. The repository contains 23 PCTs and 10 MOs. Tables A.2 and A.3 from Appendix A list PCTs and MOs from the repository we created for the Java programming language. Complete specifications of PCTs can be found in Appendix A.2. Other specifications, including specifications of MOs, may be downloaded from [96].

By demand a repository can be extended with new PCTs and MOs. To define a new PCT in order to fulfil a given design requirement the following steps have to be gone through:

1. Specify a design requirement.
2. Define a set of fragments of a program code and identify their borders of structural configurability and dependencies between these fragments.
3. Create PCT leaves for fragments which have to be managed at the atomic level. Implement parameters and management behaviour for specified configurability borders. Implement reactivity behaviour for needed events.
4. Create PCT containers for fragments which have to be managed at the template level. Implement parameters and management behaviour for specified configurability frames. Implement reactivity behaviour on needed events.

5. Combine PCT leaves and PCT containers into a final PCT container. Implement parameters and management behaviour for specified configurability frames. Implement initialisation of the final PCT container. Implement reactivity behaviour on needed events.

To define a new MO the following steps have to be gone through:

1. Specify code fragments in form of PCT which are going to be subjects of manipulation for a newly specified MO.

2. Specify pre-conditions chosen PCTs have to meet. There can be a wide range of preconditions specified dictated by characteristics of each PCT including its name, amount of parameters and current state.

3. Specify the core operation algorithm of the MO to manipulate with PCTs.

4. Implement parameters, pre-conditions and the core operation algorithm of the MO according to the MO model that have been described in this section.

A concrete description of PCTs and MOs from the core repository is given in Section A.2 from Appendix A. More complex PCTs and MOs, created via combination of the ones defined in the repository, are described in Section 8.4.2.1 and Section 8.5.2.1.

5.8 Implementation Environment

The concepts defined at the Template Level of composition are practically implemented in the *Template Composition System Library*. The library represents development tools to define PCTs and molecular operations as well as to test them. It was implemented with the Java programming language. We defined the package `neurath.templatelevel` as the main package of the library. It contains four main sub-packages. These are:

1. `atomic` - defines classes representing the atomic level of composition. This includes classes that describe atoms, atomic operations and other classes related to the management of ASLT structures.

2. `pctl` - defines classes representing the molecular level of composition. This includes classes that describe expressions including PCTs and molecular operations. These components are defined by the further sub-packages.

3. `pctl.pcts` - defines classes that describe different PCTs.

4. `pctl.operations` - defines classes that describe different molecular operations.

Figure 5.30 shows main packages that implement the Template Composition System Library. The rectangles labled as "Repository" and "Expression" denote main discussed concepts of the Template Level of composition. Relations between the "Repository" and the sub-packages `pct.operations` and `pctl.pcts` show that the repository is formed with the molecular operations and PCTs. The relation from the "Repository" to the "Expression" reflects the fact that expressions are formed with the elements of the repository which are molecular operations and PCTs.

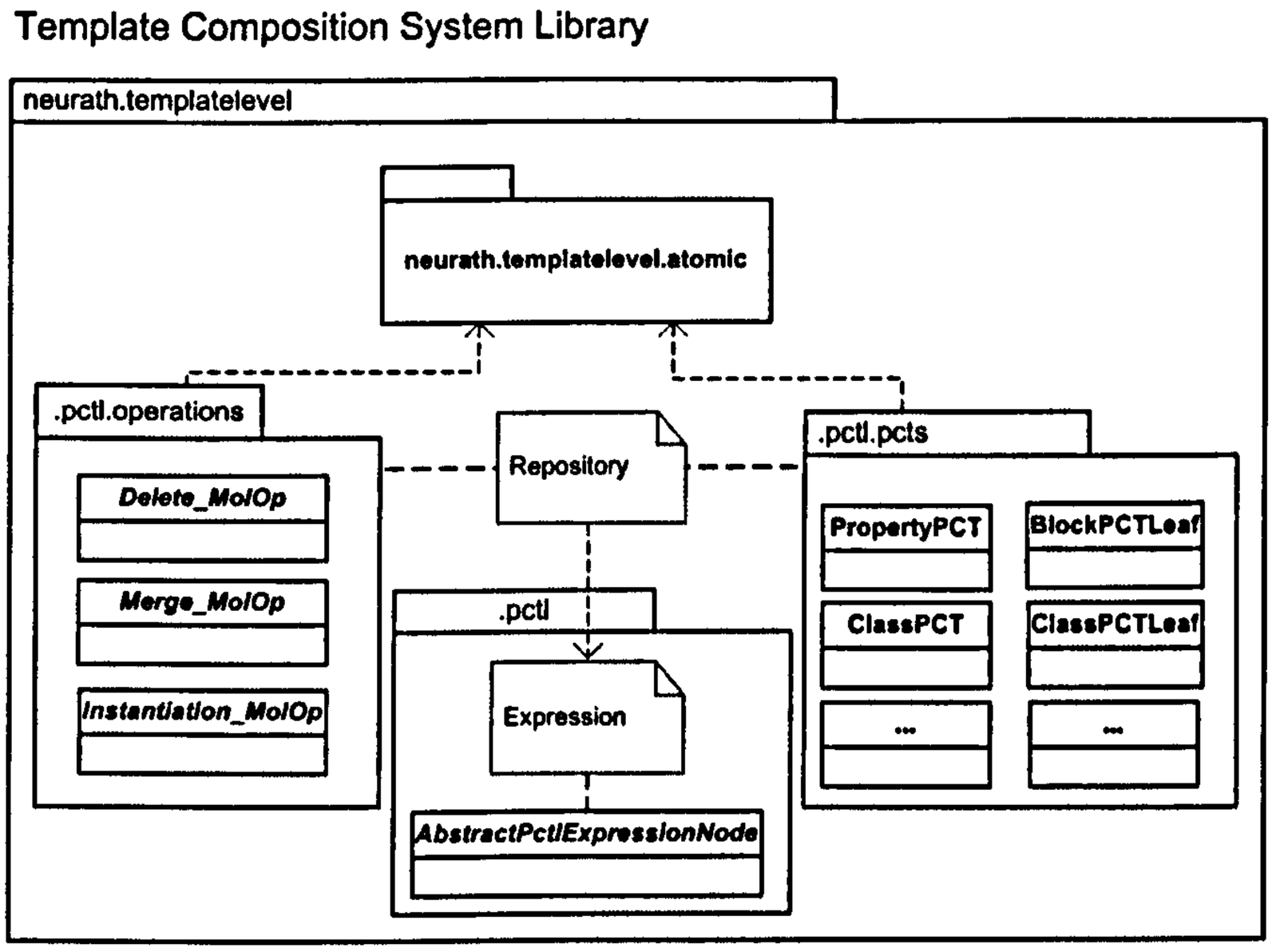


Figure 5.30: Template Composition System Library: main packages

Listing 5.12 shows a program that uses the API of the library. The program creates a simple PCT-L expression. It defines instances of the `Merge_MO` and `Instantiation_MO` operations. These operations were discussed in sections 5.5.3.2 and 5.5.3.1. The operations use the following operands: text values `PropertyPCT` and `ClassPCT`, and instances of PCTs that are results of instantiation operations.

```
1 import neurath.templatelevel.atomic.*;
```

```
2  import neurath.templatelevel.pctl.*;
3  import neurath.templatelevel.pctl.pcts.*;
4  import neurath.templatelevel.pctl.operations.*;
5  ...
6  Merge_MO root = new Merge_MO();
7  Instantiate_MO instOp1 = new Instantiate_MO();
8  instOp1.setTypeName("MethodPctLeaf");
9
10 Instantiate_MO instOp2 = new Instantiate_MO();
11 instOp2.setTypeName("ClassPctLeaf");
12
13 root.setCompositePct(instOp1);
14 root.setParentPct(instOp2);
15
16 AbstractPctLeaf result = root.operate();
17 ...
```

Listing 5.12: Example program to create and process a PCT-L expression

The program defines an expression (depicted in Figure 5.27 in Section 5.6) and then initiates its processing. The result of the processed operation is saved in the variable `result`.

5.9 Summary

This chapter has introduced the Template Level of composition. This level is defined on top of the Atomic Level and represents a composition system to compose program code with templates. Figure 5.31 shows different levels of composition and marks the material that have been described already.

We have presented the meaning of a *template* and defined such core terms as *molecular composition*, *Molecular Operations* and *Parametric Code Templates*. The general concepts defined at the Template Level of composition and their collaborations have been described. It was explained how the concepts work in different phases of the NCF life cycle. The chapter has concentrated in specification details of the PCT component model and the molecular operations composition technique using an object-oriented technology. It was shown how both PCTs and molecular operations are categorised and what

behaviour have objects represented by the main categories. Furthermore, PCT-L expressions were presented, that are formed with PCTs and molecular operations. PCT-L expressions represent basic composition language. Finally, an implementation environment for the concepts defined at the Template Level of composition were introduced. Next, we explain the concepts defined at the Target Domain Level and their collaborations during different phases of the NCF life cycle.

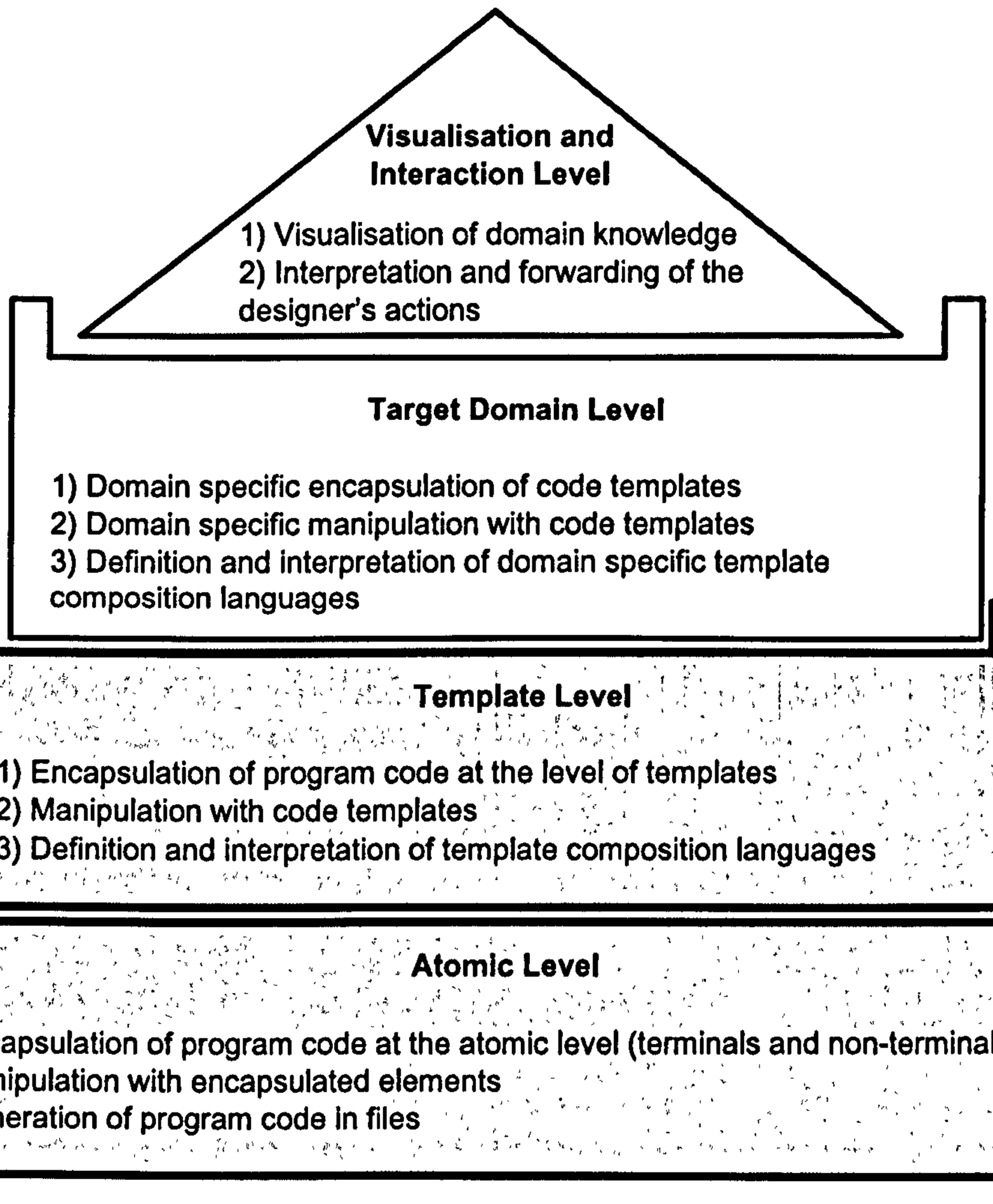


Figure 5.31: Levels of composition within the Neurath Composition Framework

Chapter 6

Target Domain Level

This chapter presents a domain-specific composition system to be used by domain experts to build a program code with templates. The concepts that form the composition system are united within the Target Domain Level of composition. The concepts defined at this level solve the problem of bridging template composition systems and domain experts. We explain these concepts as well as their collaboration. We introduce parts of the composition system, including a component model, a composition technique and a composition language. A component model is presented with Domain-specific Components defined on top of Parametric Code Templates from the Template Level. We describe a strategy to compose Domain-specific Components together with the composition technique presented by Domain-specific Operations defined on top of the molecular operations from the Template Level. Further, we introduce a simple composition language presented by domain-specific expressions. Afterwards, an implementation environment for the concepts defined at the Target Domain Level is briefly described. The chapter is concluded with summary.

6.1 Introduction

The process of molecular composition, under which we mean the process of composing program code templates, is oriented to experts in such fields as software architectures, templates, programming languages and software design. These experts have all necessary knowledge to maintain the molecular composition process. However, we think, template-based code composition could be potentially used by experts from other domains as well. We tell "potentially", because without a special interpretation mechanisms those experts from other domains simply can not understand terminology from the composition system's domain. Domain experts require mechanisms to express the process of composition with the terms and means they could understand. Those mechanisms could "externalise" the process of molecular composition up to the level of domain experts, thus making the composition with code templates applicable by domain experts.

There should be a layer between composition system and domain experts built. Figure 6.1 illustrates such a layer marked with the "?" sign.

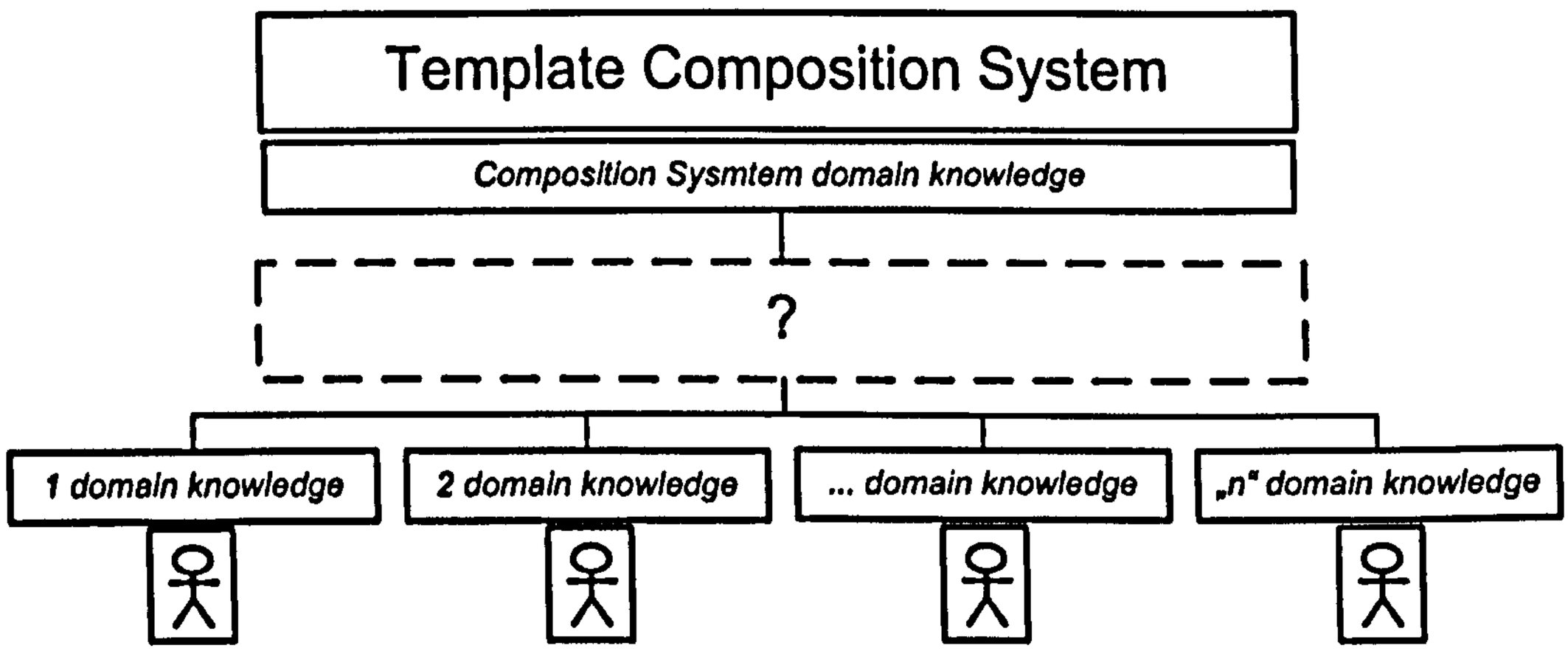


Figure 6.1: Externalisation Layer

It is located between Template Composition System and domain experts from different application domains. More precisely, the layer maps knowledge from different domains onto knowledge from the composition system domain. To describe the "?" layer we define several concepts at the Target Domain Level of composition.

The concepts at the Target Domain Level should solve two main tasks. First, domain-specific languages mapped onto template-based composition languages could be defined. Second, these domain-specific languages could be used, which means formation of sen-

tences and their interpretation. Therefore, in the fashion as in case of the concepts defined at the Template Level of composition, we distinguish between two phases: (1) *composition system definition phase* and (2) *design phase*.

Further, we describe the concepts defined at the Target Domain Level of composition and their collaborations at different phases.

6.2 Architecture

The concepts defined at the Target Domain Level try to solve an issue of bridging (or externalizing) a template-based composition process to various domain experts so that they can use the composition system to design software for their domains. More specifically, we separate the following main issues:

1. Domain-specific encapsulation of program code templates
2. Domain-specific manipulation with program code templates
3. Definition of domain-specific template-based composition languages and interpretation of sentences written on that language

To be more exact, when we tell here "program code templates" we mean components built according to the PCT component model, described in Section 5. To encapsulate program code templates, we define a new component model, called *Domain-specific Components (DSCs)*. The components built according to that model encapsulate PCTs and exact them according to the requirements of an application domain.

When we described molecular composition in Section 5, we introduced molecular operations to manipulate with PCTs. *Domain-specific Molecular Operations (DSOs)* represent domain-specific rules to manipulate with program code templates. Being implemented according to the object-oriented technology, DSOs encapsulate molecular operations and exact them according to the requirements of an application domain.

There are two main phases defined. First phase is the composition system definition phase. At this phase a domain-specific language is provided by defining DSCs and DSOs. The second phase is the design phase. During this phase the domain-specific language is used to design a software system. In the following sections we are going to give more detailed information on both phases.

6.2.1 Composition System Definition Phase

At this phase a DSL is defined on top of the PCT-L. Figure 6.2 depicts basic concepts which are playing a central role during the composition system definition phase. Input material for the concepts at the Target Domain Level are repositories of PCTs and molecular operations defined by a Software Architect at the Template Level of composition. These components are extended with domain-specific notations according to the domain requirements provided by the Domain Expert.

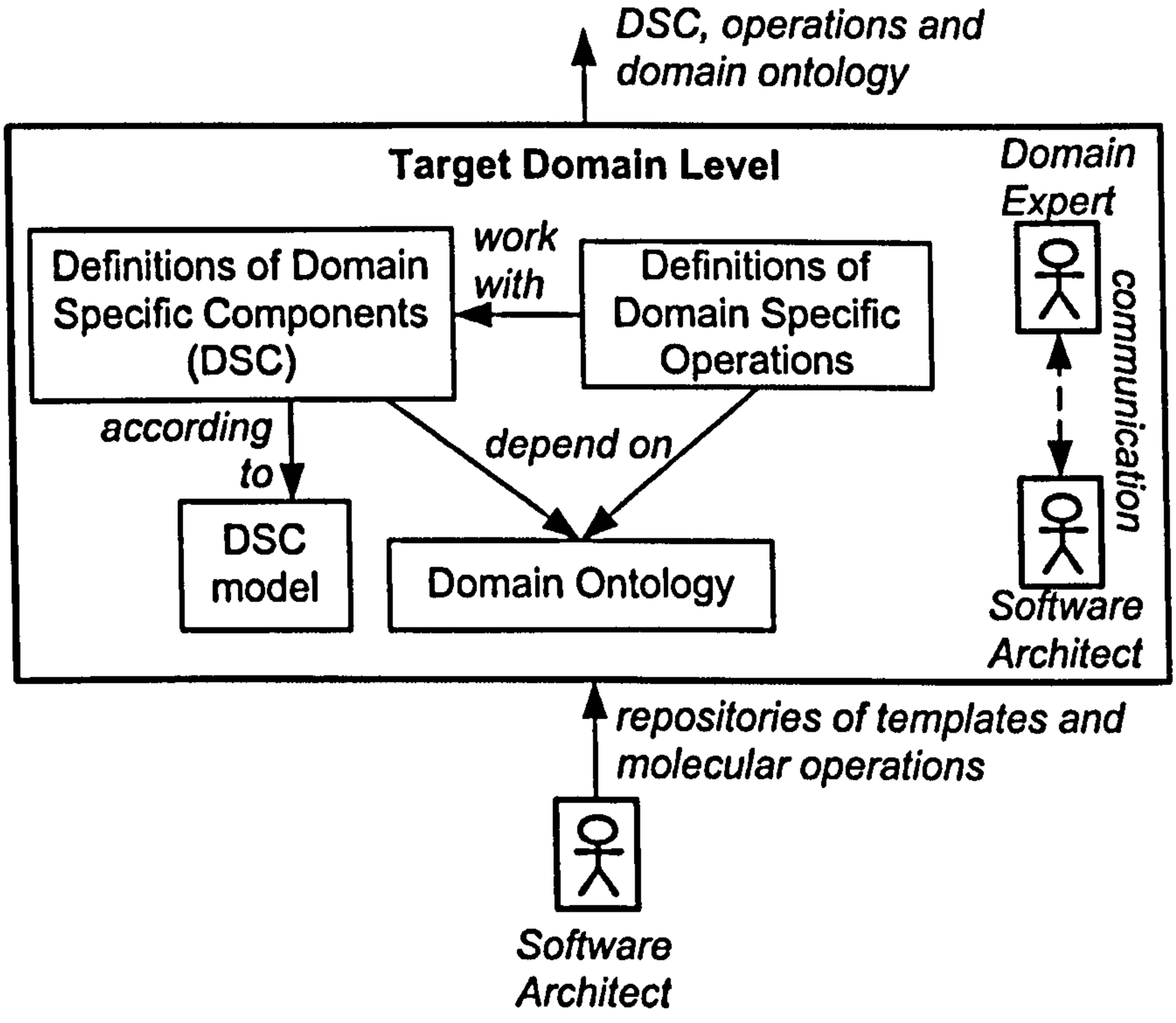


Figure 6.2: Concepts defined at the Target Domain Level of composition during the composition system definition phase

Both the Domain Expert and the Software Architect may work at the Target Domain Level during the composition system definition phase. The Domain Expert may form new DSCs and DSOs using existing ones. The Software Architect works additionally with the material defined at the Template Level. The domain requirements are processed into the template composition language PCT-L.

At the Target Domain Level PCTs are extended by DSCs and molecular operations by DSOs. DSCs are formed according to the DSC model. DSCs and DSOs are influenced by domain ontology. The ontology is a product of domain analysis and it describes domain-specific terms and relationships that may form a software system at design phase. DSCs, DSOs and the domain ontology are main productions at the composition system definition phase.

6.2.2 Design Phase

At the design phase the DSL formed at the composition system definition phase is used to compose a software system within a specified application domain. The composition is seen as a sequence of molecular operations applied to PCTs.

At the design phase the following routines are performed:

1. Formation of sentences of DSL. These are expressions that consist of DSOs as operators and DSCs as operands.

2. Processing of formed sentences of the DSL. This results in sentences in the template composition language PCT-L that are processed at the Template Level.

3. Description of the state of the designed system in terms of an application domain. The designed system is described with domain ontology that incorporates knowledge about what elements exist and how they are related.

Figure 6.3 depicts basic concepts which are playing a central role during the design phase. At the design phase a program code is composed with help of DSCs. They are defined on top of the template specifications, called PCTs, defined at the Template Level. Composition of DSCs is performed with help of DSOs. From one side, the domain-specific composition results in sentences in template composition language PCT-L which are processed at the Template Level. From the other side, the domain-specific composition transforms the description of the state of the designed software system. This description, called Domain Ontology, is defined in domain-specific terms. The states of the designed system, reflected by the Domain Ontology are used at the further level of composition, called Visualisation and Interaction Level.

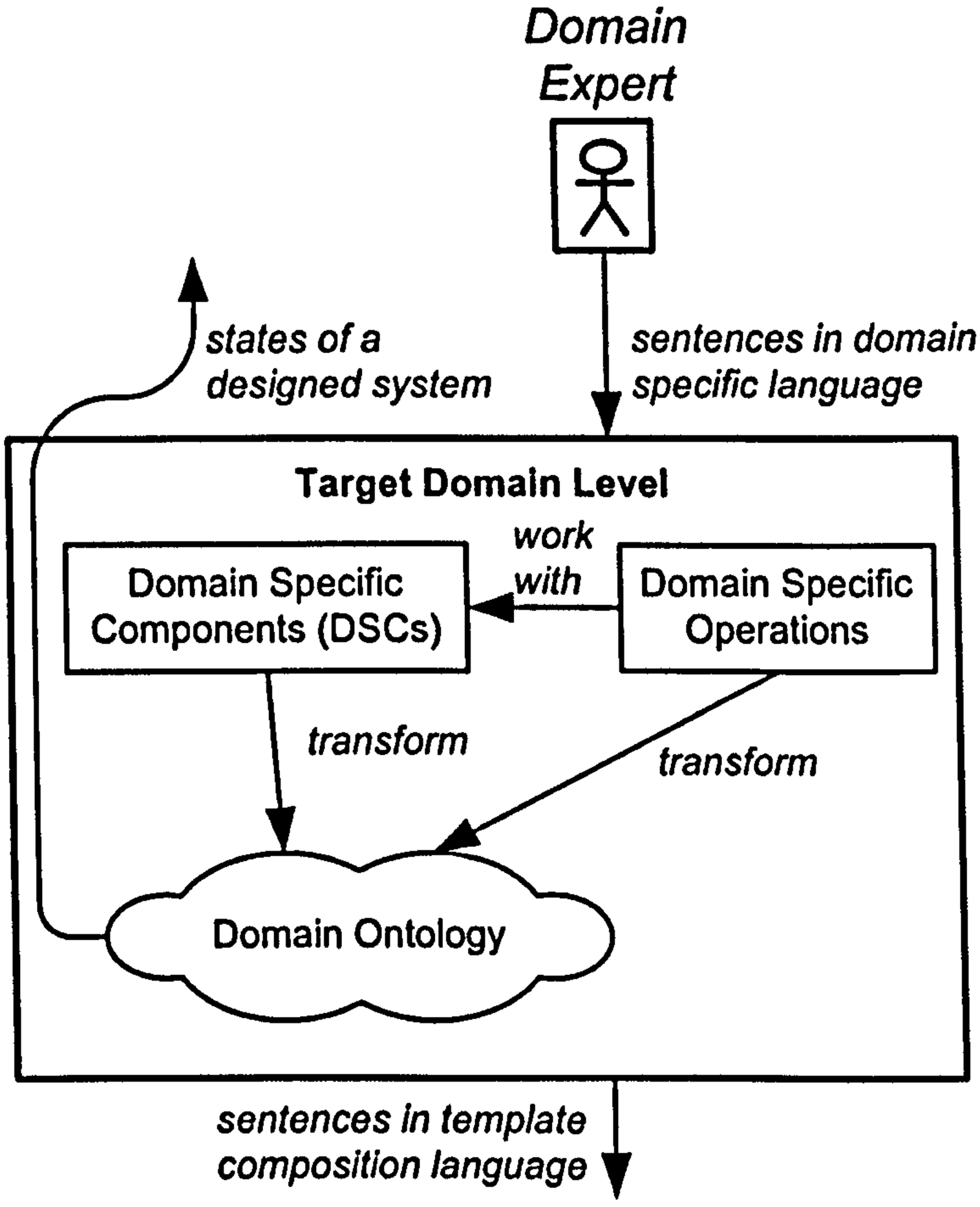


Figure 6.3: Concepts defined at the Target Domain Level of composition during the design phase

6.3 Description of a Target Domain

The concepts defined at the Target Domain Level of composition solve an issue of bridging (or externalizing) a template-based composition process up to the level of domain experts. These experts use a composition system to domain-specifically design software systems for the required application domain. We call this domain a *target domain*.

Before establishing relations between a composition system and a target domain, it is necessary to describe the target domain, by asnwering the following questions:

1. What terms a target domain defines?

2. What are relations between these terms?

3. How and when those relations are established?

We use ontology to describe a target domain. We take the following relevant definition of ontology provided by Hendler in [44]: "Ontology is a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic". The ontology describing a target domain is referred as to *domain ontology*. The domain ontology provides a vocabulary for referring to the terms in a subject area as well as a taxonomy that is a hierarchical categorisation of entities within a domain.

The domain onologies give an answer on the first two questions specified above. Figure 6.4 depicts an example of domain ontology for the "Virtual sensor" domain.

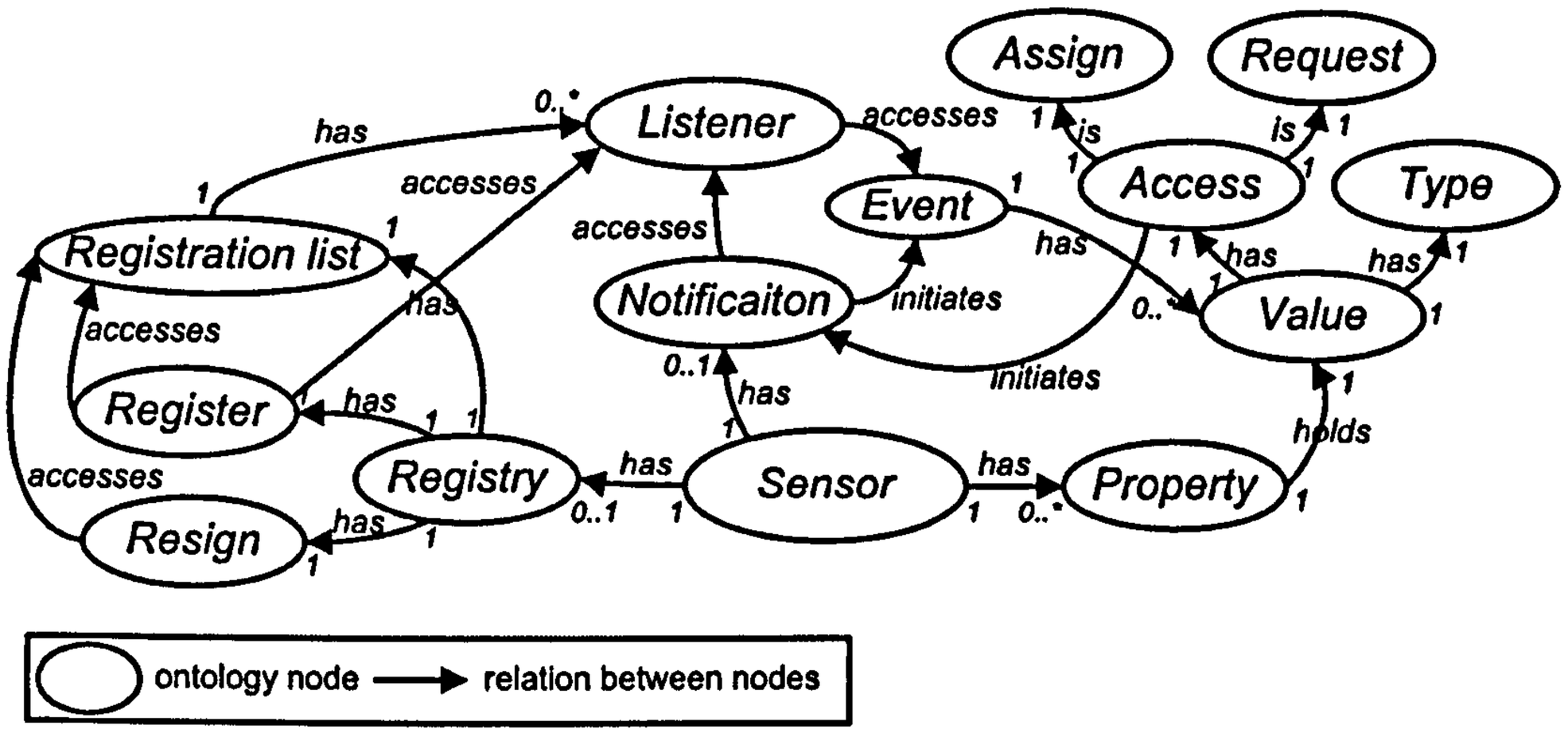


Figure 6.4: An example of the "Virtual sensor" domain ontology

The figure illustrates domain-specific terms and relationships between them. It is possible to recognise such terms as `Sensor`, `Registry`, `Property`, `Notification`, `Listener`, `Value` and many others. The relations between terms show their basic dependencies. For example, the relation `has` between `Sensor` and `Property` denotes that "one Sensor may have zero or more Properties". If we follow further relationships from `Property` to other terms we can see what actually `Property` means.

The domain ontology describes static aspects of a target domain. To describe dynamic aspects to build a systen within a target domain, there is a need to answer the question "How and when terms appeared and relations between them are established?". Rela-

tionships between nodes are created according to the *domain-specific manipulation rules*. Each rule is characterised by a name, input parameters and a rule specification.

The domain-specific terms defined by domain ontology are represented by DSCs. The domain-specific manipulation rules are represented by DSOs. To specify the domain ontology we have defined a concept, called *Simple Knowledge Web Context* (*SkwContext*). Figure 6.5 illustrates a connection between SkwContext and domain-specific components and molecular operations.

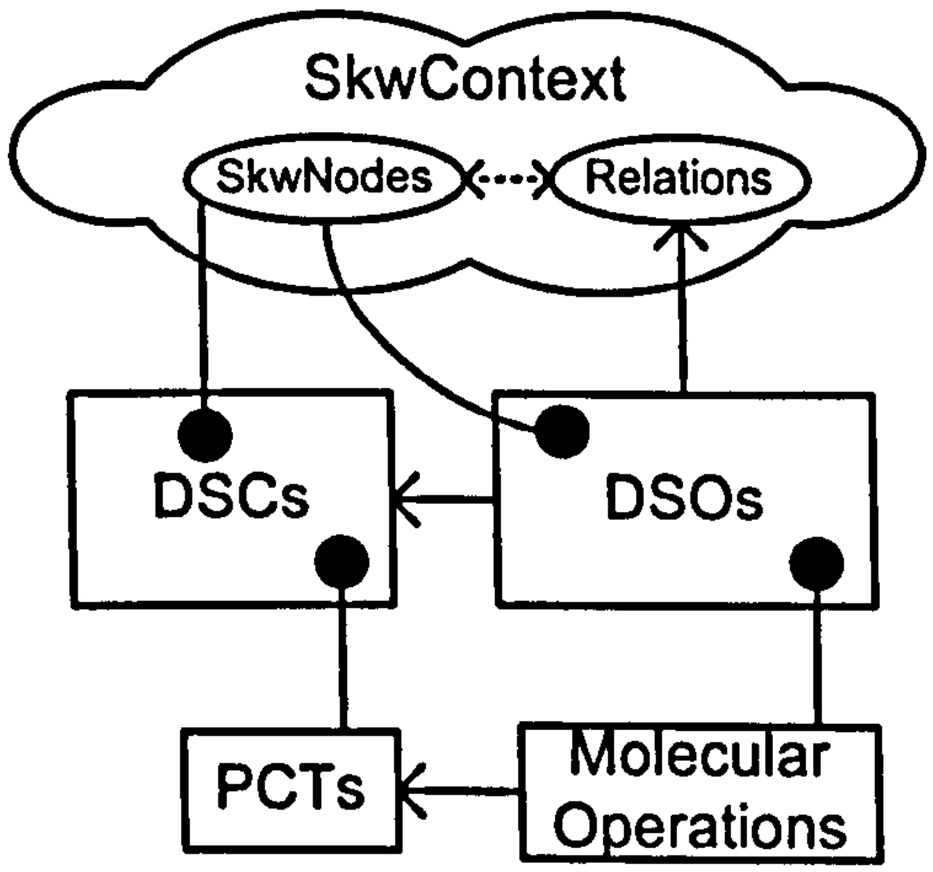


Figure 6.5: Connection between domain-specific components and molecular operations to the SkwContext

The SkwContext defines objects called `SkwNodes` that represent terms of the domain ontology. It defines also objects, called `Relations`, that represent relations between nodes. Further, the figure brings some details regarding the connection of the SkwContext to the DSCs and DSOs. A line with a black circle drawn at one side represents an encapsulation and usage relation. This relation means that an object that has a black circle within it borders encapsulates a connected object and uses that object. In this way DSCs and DSOs are connected with SkwNodes, as well as, with PCTs and molecular operations respectively. A solid line with an arrow represents the usage relation. This relation denotes any kind of manipulation, like creation, deletion, modification and so on. A dashed bi-arrowed line represents a dependency between SkwNodes and SkwRelations and means that: (1) Defined SkwNodes may be related with SkwRelation (2) Each defined relation binds nodes.

Further, we explain more SkwContext, Domain-specific components and Operations.

6.4 Simple Knowledge Web Context

The SkwContext introduces a concept for specification and application of domain ontology. The SkwContext defines a graph-like data structure to hold domain ontology. Nodes, that are called SkwNodes, in the SkwContext represent terms defined by the domain ontology. Relations, that are called SkwRelations, in the SkwContext represent relations between terms. Figure 6.6 depicts a UML class diagram that gives some information about what SkwContext can do. Table 6.1 explains methods defined by the class `SkwContext` representing the SkwContext concept.

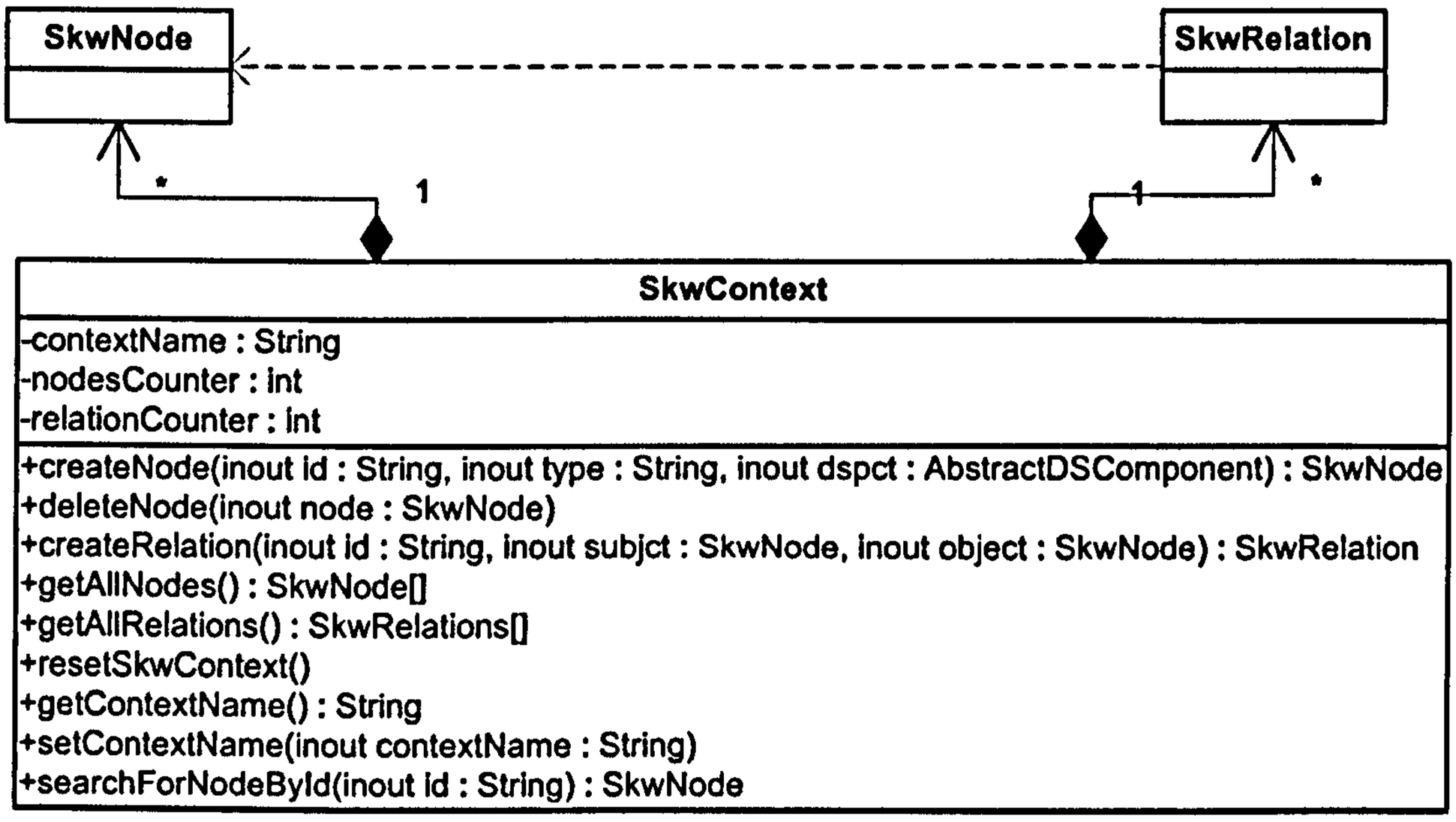


Figure 6.6: UML Class diagram: SkwContext

Method	Explanation
`createNode`	This method creates an SkwNode with the `id` identifier and `type` type. Moreover it relates the node with a DSC specified by the `dspct`
`deleteNode`	Removes a SkwNode specified by the `node`
`createRelation`	This method creates a relation between two nodes, specified by `subject` and `object` respectively. The `id` is an identifier of the relation representing the type of relation

Method	Explanation
`getAllNodes`	This method returns all nodes registered within the SkwContext
`getAllRelations`	This method returns all relations registered within the SkwContext
`resetSkwContext`	Clears up and brings the state of the SkwContext into the initial state with zero registered nodes and relations
`getContextName`	Returns the name of the SkwContext
`setContextName`	Assigns the name `contextName` to the SkwContext
`searchForNodeById`	Searches within a SkwContext for a node identified as `id`

Table 6.1: Functionality of the SkwContext

6.4.1 SkwNodes

Each SkwNode object represents a term defined by the domain ontology that describes a target domain. A SkwNode is a generic type that may define different ontology terms. Every SkwNode object is characterised by the following:

1. It has a distinct *identifier*.
2. It has a *type*.
3. It encapsulates information about SkwNodes related with this SkwNode.
4. It is connected with DSC that implements a domain-specific template for the SkwNode.
5. It may have attributes, which are connected to the attributes of the related DSC.

Domain ontology normally defines multiple terms that actually are described as types or classes. Multiple instances of those types can be created by demand. SkwNodes represent domain-specific terms as types. An identifier of a SkwNode denotes an instance. Each SkwNode may be related to other SkwNodes. We work with directed relation, which connects one SkwNode as *source* with other SkwNode as *destination*. Each SkwNode

holds the information about all sources that have this SkwNode as a destination, as well, as the information about all destinations that have this SkwNode as a source. Moreover, each SkwNode is connected bidirectional to the DSC instance that implements a domain-specific template for the domain-specific term represented by this SkwNode. Another feature of the SkwNode is that it defines the same attributes as the connected DSC does, so that through these the attributes of the DSC may be accessed.

Figure 6.7 illustrates a UML class diagram with the class `SkwNode` defined.

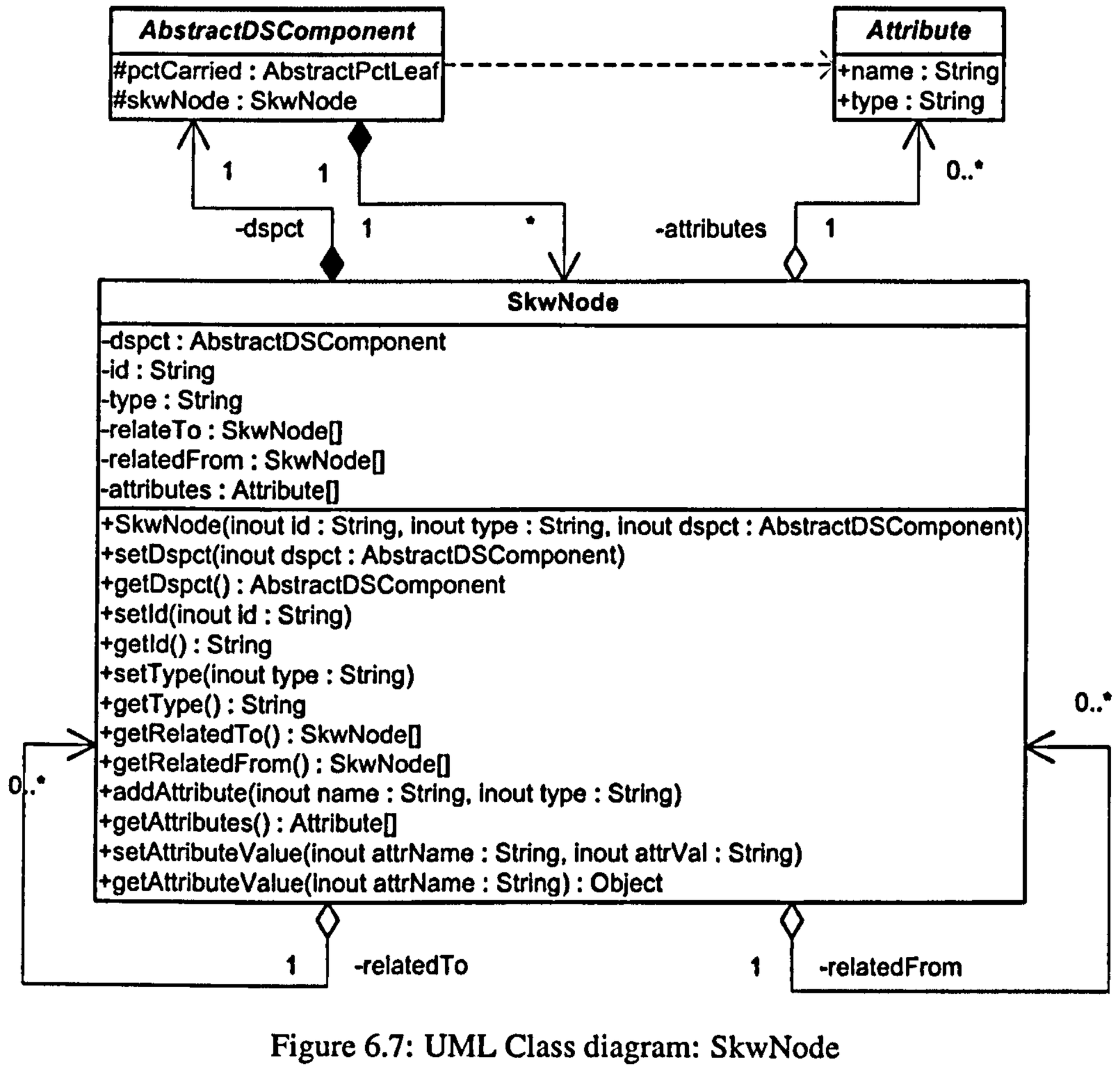


Figure 6.7: UML Class diagram: SkwNode

Table 6.2 explains methods of the class `SkwNode`.

Method	Explanation
`SkwNode`	The constructor that creates an instance of SkwNode. The instance has an identifier `id`, a type `type` and connected to a DSC `dspct`
`setDspct`	Connects this SkwNode to the DSC `dspct`
`getDspct`	Returns the DSC connected to this SkwNode
`setId`	Assigns the identifier `id` to this SkwNode
`getId`	Returns the identifier of this SkwNode
`setType`	Assigns the type `type` to this SkwNode
`getType`	Returns the type of this SkwNode
`getRelatedTo`	Returns all related "destination" SkwNodes
`getRelatedFrom`	Returns all related "source" SkwNodes
`addAttribute`	Registers a new attribute for this SkwNode and binds with the corresponding attribute defined in the connected DSC. The attribute has a name defined by `name` and a type defined by `type`
`getAttributes`	Returns all registered attributes
`setAttributeValue`	Assigns a value `attrVal` to the attribute with a name `attrName`
`getAttributeValue`	Returns a value of the attribute denoted by the name `attrName`

Table 6.2: Functionality of the SkwNode

6.4.2 SkwRelations

SkwRelations are represent relations defined between terms in the domain ontology. A SkwRelation is a generic type that may define different types of relations. It is characterised by the following:

1. It has an identifier that denotes an instance of a SkwNode
2. It has a type
3. It has a source (subject) and a destination (object) SkwNodes

A SkwRelation represents a relation between two terms in the domain ontology. A relation can be of some type that represents the meaning of the relation. SkwRelation is defined with a class `SkwRelation`. Figure 6.8 illustrates a UML class diagram with the class `SkwRelation` defined. Table 6.3 explains methods of the class `SkwRelation`.

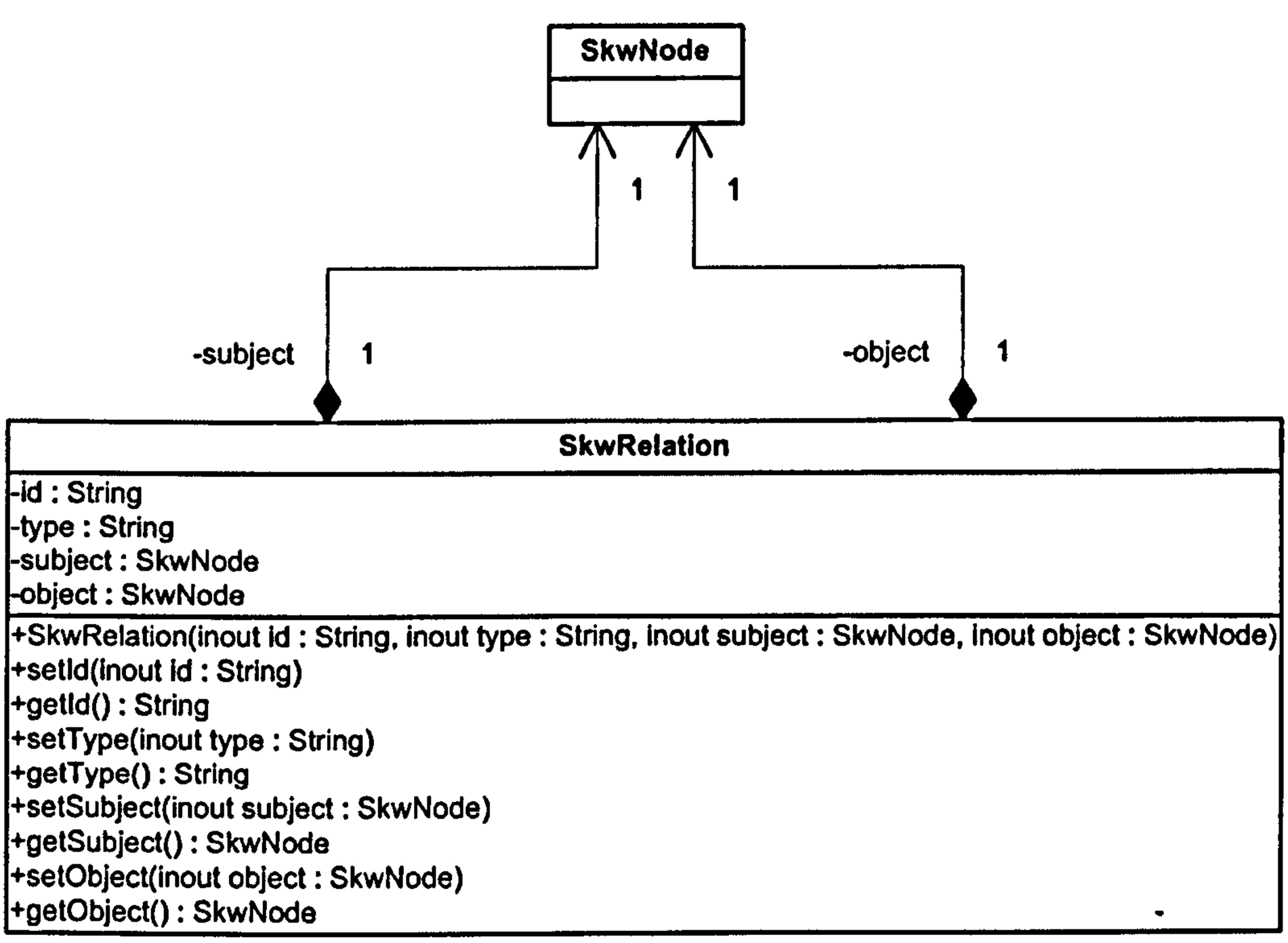


Figure 6.8: UML Class diagram: SkwRelation

Method	Explanation
`SkwRelation`	The constructor that creates an instance of the SkwRelation. The instance has an identifier `id`, a type `type` and connects the SkwNode `subject` with the SkwNode `object`, where `subject` is a source and an object is a destination
`setId`	Assigns the identifier `id` to this SkwRelation
`getId`	Returns the identifier of this SkwRelation
`setType`	Assigns the type `type` to this SkwRelation
`getType`	Returns the type of this SkwRelation

Method	Explanation
`setSubject`	Assigns a source (subject) SkwNode, denoted as `subject`, for this SkwRelation
`getSubject`	Returns a source (subject) SkwNode of this SkwRelation
`setObject`	Assigns a destination (object) SkwNode, denoted as `object`, for this SkwRelation
`getObject`	Returns a destination (object) SkwNode of this SkwRelation

Table 6.3: Functionality of the SkwRelation

6.4.3 Rules of SkwContext

The class `SkwContext` declares the following fields:

1. `contextName`: denotes a name of a SkwContext.
2. `nodeCounter`: a counter of nodes defined by a SkwContext.
3. `relationCounter`: a counter of relations defined by a SkwContext.
4. `nodes`: a list of references to nodes defined by a SkwContext.
5. `relations`: a list of references to relations defined by a SkwContext.
6. `contextListeners`: a list of all listeners of events generated by a SkwContext.

Listing 6.1 shows fields declaration written in Java programming language.

```
1 ...
2 protected String contextName = "SkwDefaultContext.ctx";
3 protected long nodesCounter = 0;
4 protected long relationCounter = 0;
5 LinkedList nodes = new LinkedList();
6 LinkedList relations = new LinkedList();
7 private java.util.LinkedList contextListeners = new java.util.
    LinkedList();
```

Listing 6.1: A specification of a SkwContext: fields declaration

Further, the most relevant parts of SkwContext specification are given. A complete specification of the `SkwContext` class in form of source code can be found in [96].

6.4.3.1 Creation of a Node

The class `SkwContext` creates a node according to an algorithm presented in Figure 6.9.

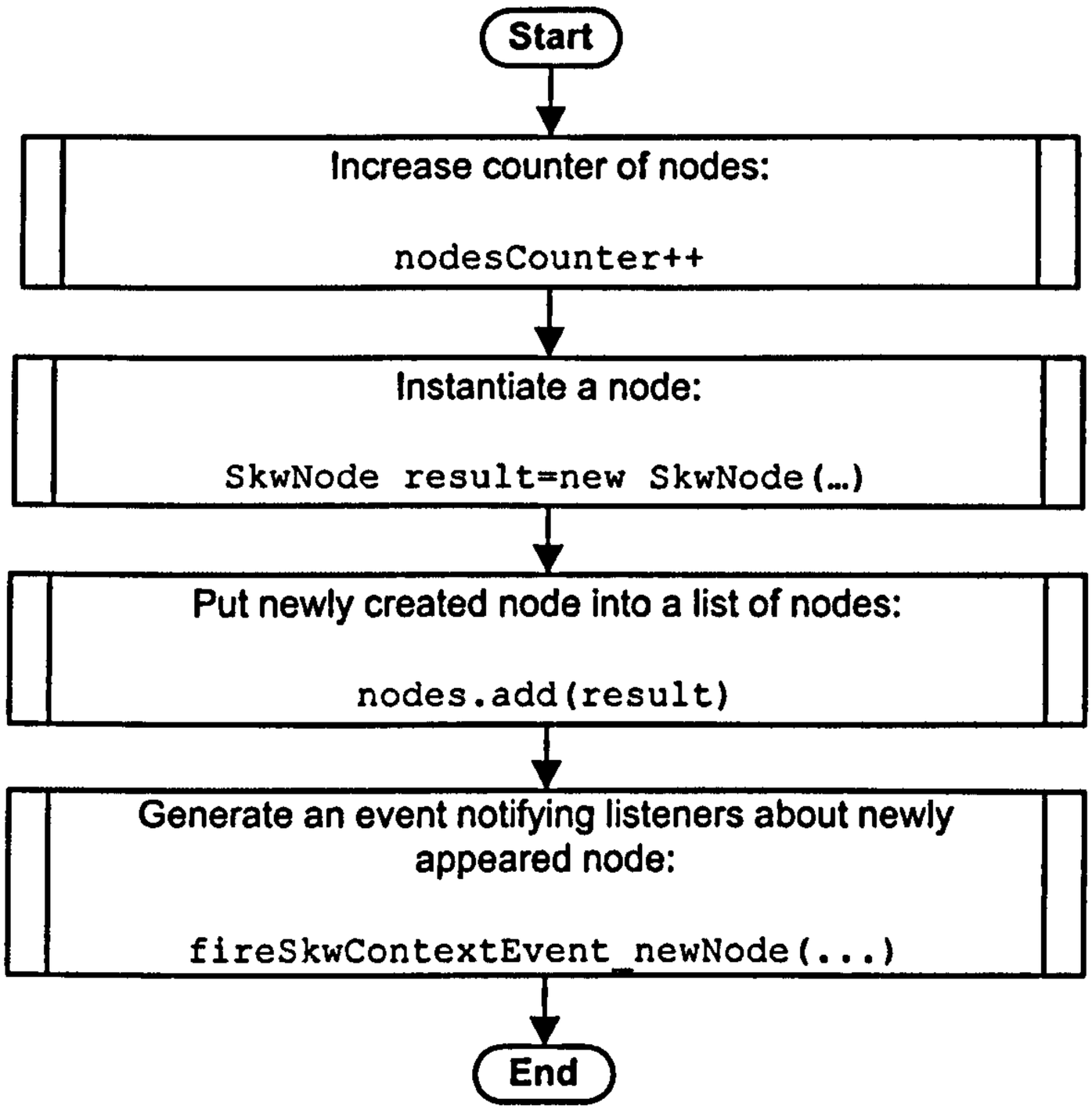


Figure 6.9: A flowchart of creating a new node via SkwContext

The class `SkwContext` provides few ways of creating a node which all are defined according the algorithm. Listing 6.2 contains corresponding specifications.

```
1
2  public SkwNode createNode(String type){
3    nodesCounter++;
4    SkwNode result = createNode(type, null);
5    fireSkwContexEvent_newNode(new SkwContextEvent(this, this,
       null, result));
```

```
6     return result;
7   }
8
9   public SkwNode createNode(String id, String type,
        AbstractDSComponent dspct){
10    nodesCounter++;
11    SkwNode result = new SkwNode(String.valueOf(nodesCounter) ,id
          , type, dspct);
12    nodes.add(result);
13    fireSkwContexEvent_newNode(new SkwContextEvent(this, this,
          null, result));
14    return result;
15  }
16
17  public SkwNode createNode(String type, AbstractDSComponent
        dspct){
18    nodesCounter++;
19    SkwNode result = new SkwNode(String.valueOf(nodesCounter),
          type, dspct);
20    nodes.add(result);
21    fireSkwContexEvent_newNode(new SkwContextEvent(this, this,
          null, result));
22    return result;
23  }
```

Listing 6.2: A specification of a SkwContext: nodes creation

6.4.3.2 Deletion of a Node

Listing 6.3 shows rules which are executed when a node is deleted from the SkwContext.

```
1   public void deleteNode(SkwNode node){
2     nodes.remove(node);
3     fireSkwContexEvent_nodeDeleted(new SkwContextEvent(this,
          this, null, node));
4   }
```

Listing 6.3: A specification of a SkwContext: nodes deletion

6.4.3.3 Creation of a Relation

The class `SkwContext` creates a relation according to an algorithm presented in Figure 6.10.

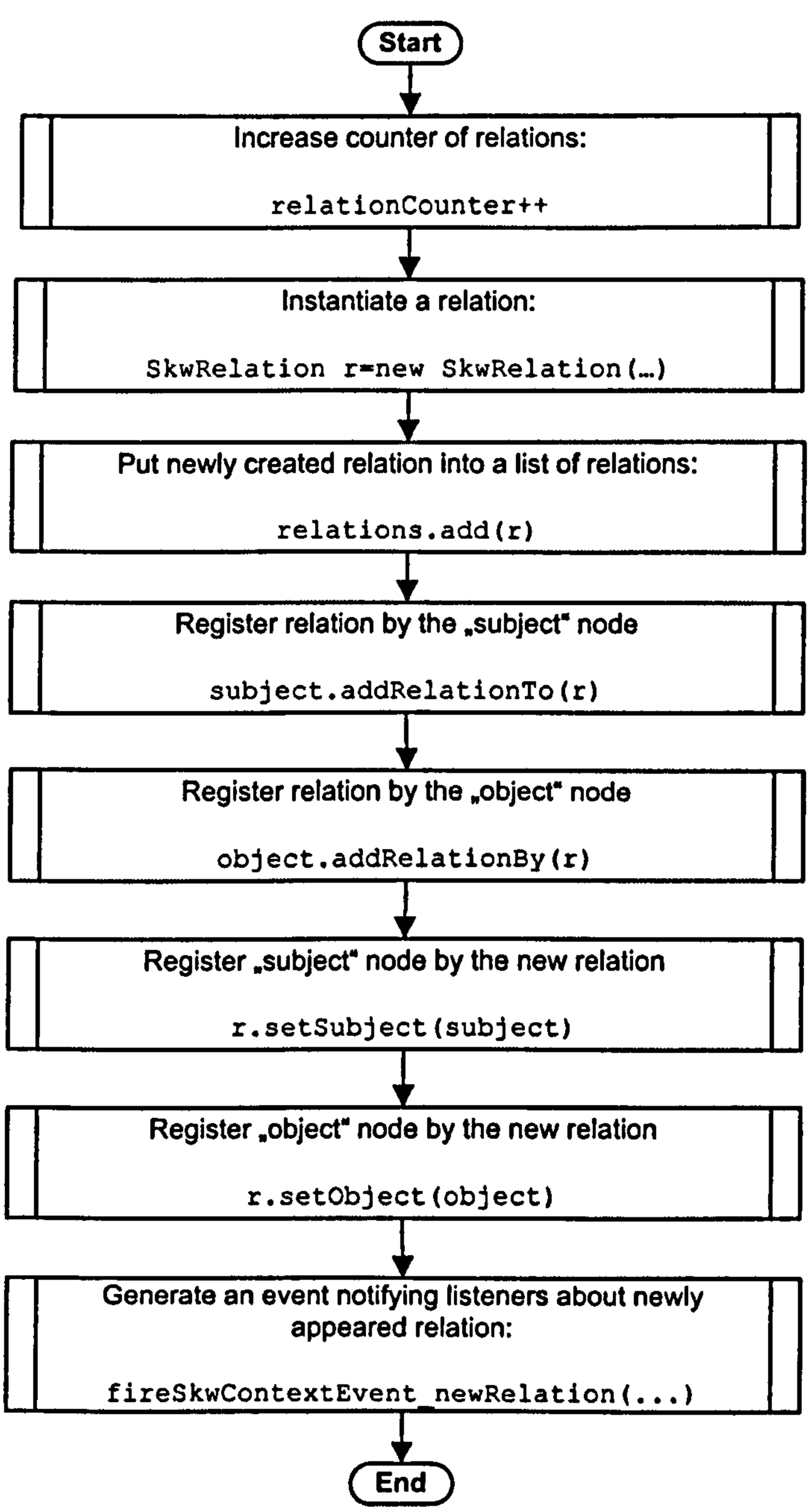


Figure 6.10: A flowchart of creating a new relation via SkwContext

Listing 6.4 shows rules which are executed when a relation is created in the SkwContext.

```
1  public SkwRelation createRelation(String id, SkwNode subject,
       SkwNode object){
2    relationCounter++;
3    SkwRelation r = new SkwRelation(String.valueOf(
       relationCounter), id);
4    relations.add(r);
5    subject.addRelationTo(r);
6    object.addRelationBy(r);
7    r.setSubject(subject);
8    r.setObject(object);
9    fireSkwContexEvent_newRelation(new SkwContextEvent(this, this
       , r));
10   return r;
11 }
```

Listing 6.4: A specification of a SkwContext: relation creation

6.4.3.4 Deletion of a Relation

The class `SkwContext` deletes a relation according to an algorithm presented in Figure 6.11. Listing 6.5 shows rules which are executed when a relation is deleted from the SkwContext.

```
1  public boolean deleteRelation(SkwRelation rel){
2    if (relations.remove(rel)){
3      fireSkwContexEvent_relationDeleted(new SkwContextEvent(this,
         this, rel, null));
4      return true;
5    } else
6    return false;
7  }
```

Listing 6.5: A specification of a SkwContext: relation deletion

6.4.3.5 Requesting all Nodes

Listing 6.6 shows rules which are executed when all existing nodes are requested from the SkwContext.

```
1  public LinkedList getAllNodes(){
2    return nodes;
3  }
```

Listing 6.6: A specification of a SkwContext: requesting all nodes

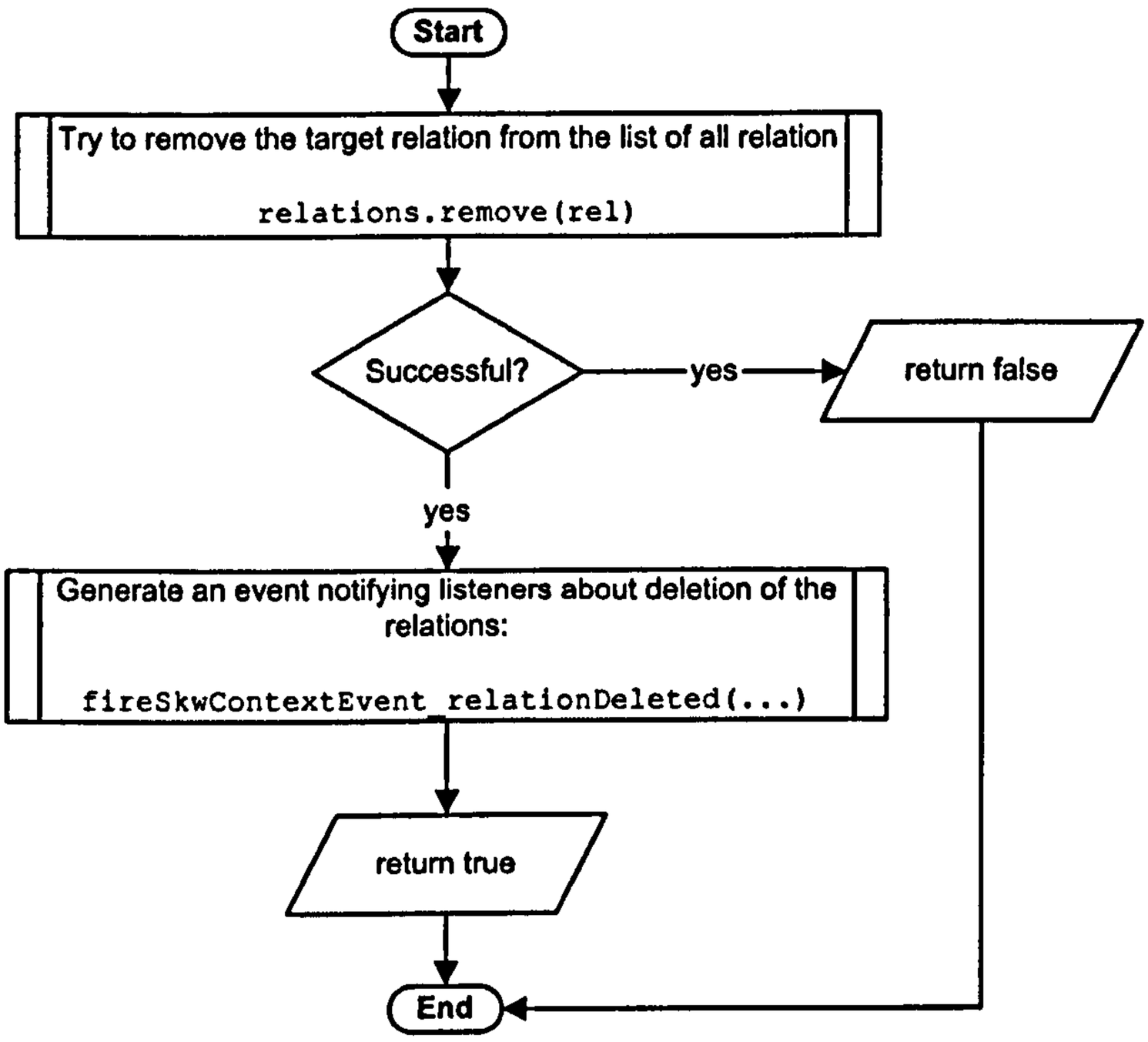


Figure 6.11: A flowchart of deleting a relation via SkwContext

6.4.3.6 Requesting all Relations

Listing 6.7 shows rules which are executed when all existing relations are requested from the SkwContext.

```
1  public LinkedList getAllRelations(){ return relations; }
```

Listing 6.7: A specification of a SkwContext: requesting all relations

6.4.3.7 Resetting SkwContext

Listing 6.8 shows rules which are executed when a SkwContext is reseted.

```
1  public void resetSkwContext(){
2    nodesCounter = 0;
3    relationCounter = 0;
4    nodes = null;
5    nodes = new LinkedList();
6    relations = null;
7    relations = new LinkedList();
8    fireSkwContexEvent_reseted(new SkwContextEvent(this, this));
9  }
```

Listing 6.8: A specification of a SkwContext: resetting a SkwContext

6.4.3.8 Setting/Getting a new Context Name

Listing 6.9 shows rules which are executed when a name of a SkwContext is setted or requested.

```
1  public void setContextName(String contextName){
2    this.contextName = contextName;
3  }
4  public String getContextName(){
5    return contextName;
6  }
```

Listing 6.9: A specification of a SkwContext: setting/requesting a name of a SkwContext

6.4.3.9 Searching for a Node

There are different algorithms to search for a node according to different criterias. Here we present an algorithm to search for a node accoring to its `id` property. Figure 6.12 depicts a flowchart of the algorithm.

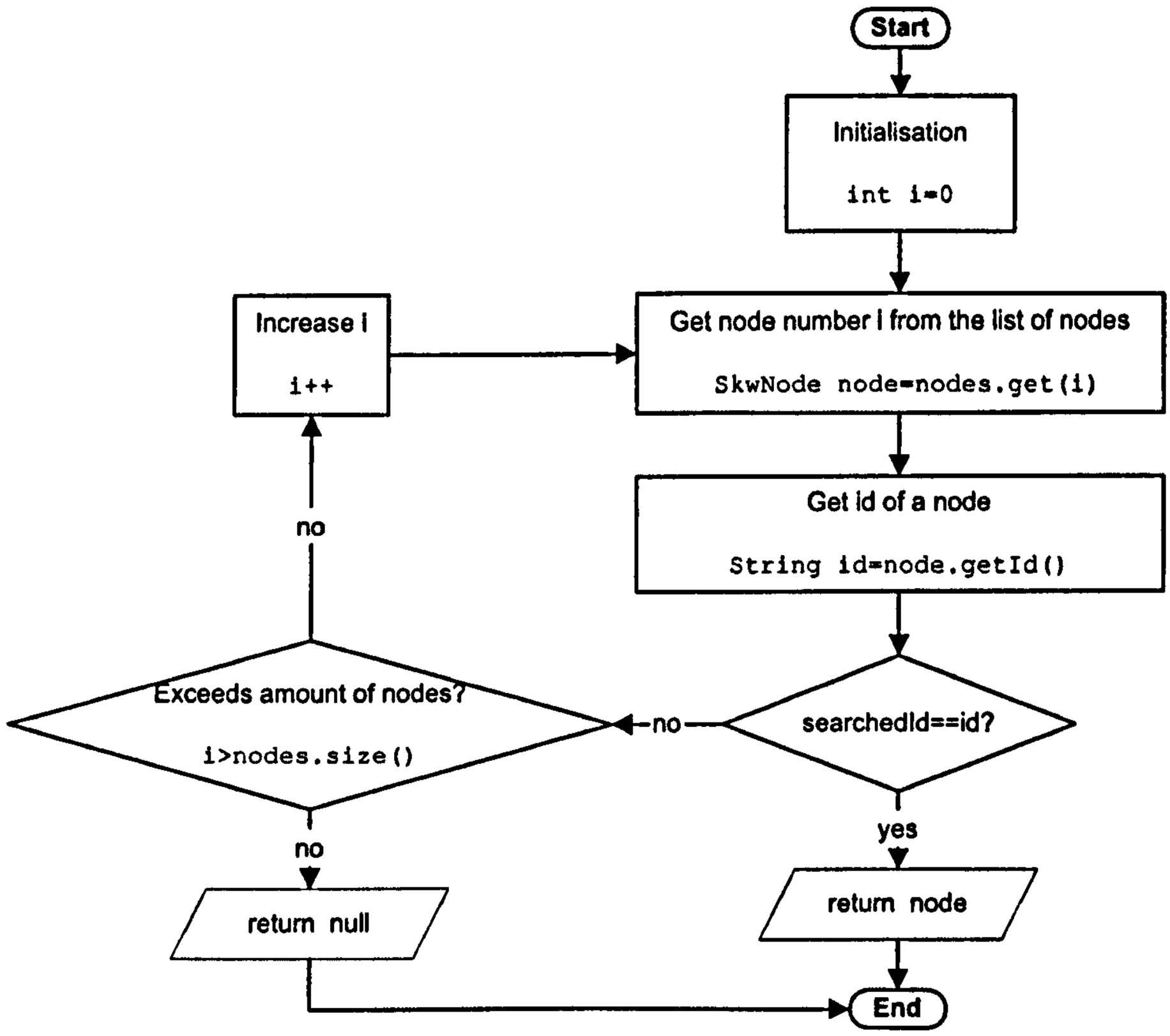


Figure 6.12: A flowchart of a searching for a node in the SkwContext according to the node's `id`

Listing 6.10 shows rules which are executed when a node is searched in a SkwContext according to its `id` property.

```
1  public SkwNode searchForNodeById(String id){
2    for (int i=0; i<nodes.size();i++){
3      SkwNode node = (SkwNode) nodes.get(i);
4      if (node.getId().equals(id)) return node;
5    }
6    return null;
7  }
```

Listing 6.10: A specification of a SkwContext: searching for a node by its `id`

6.5 Domain-specific Components

DSCs represent terms defined by the domain ontology for the target domain. Figure 6.13 shows a DSC mapped to the Term defined in the domain ontology. Each DSC encapsulated a program code template represented by a PCT.

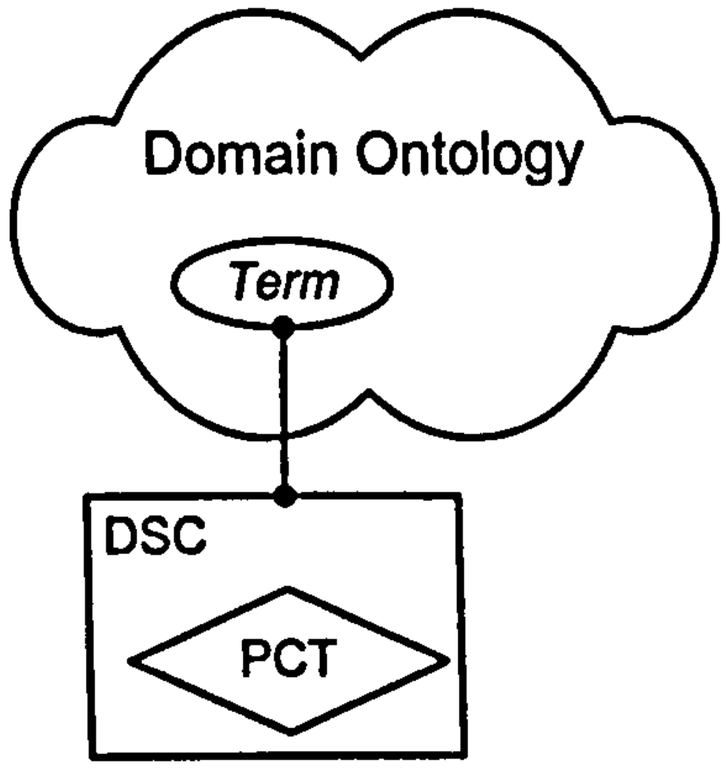


Figure 6.13: Relation between DSC and a term defined by the domain ontology

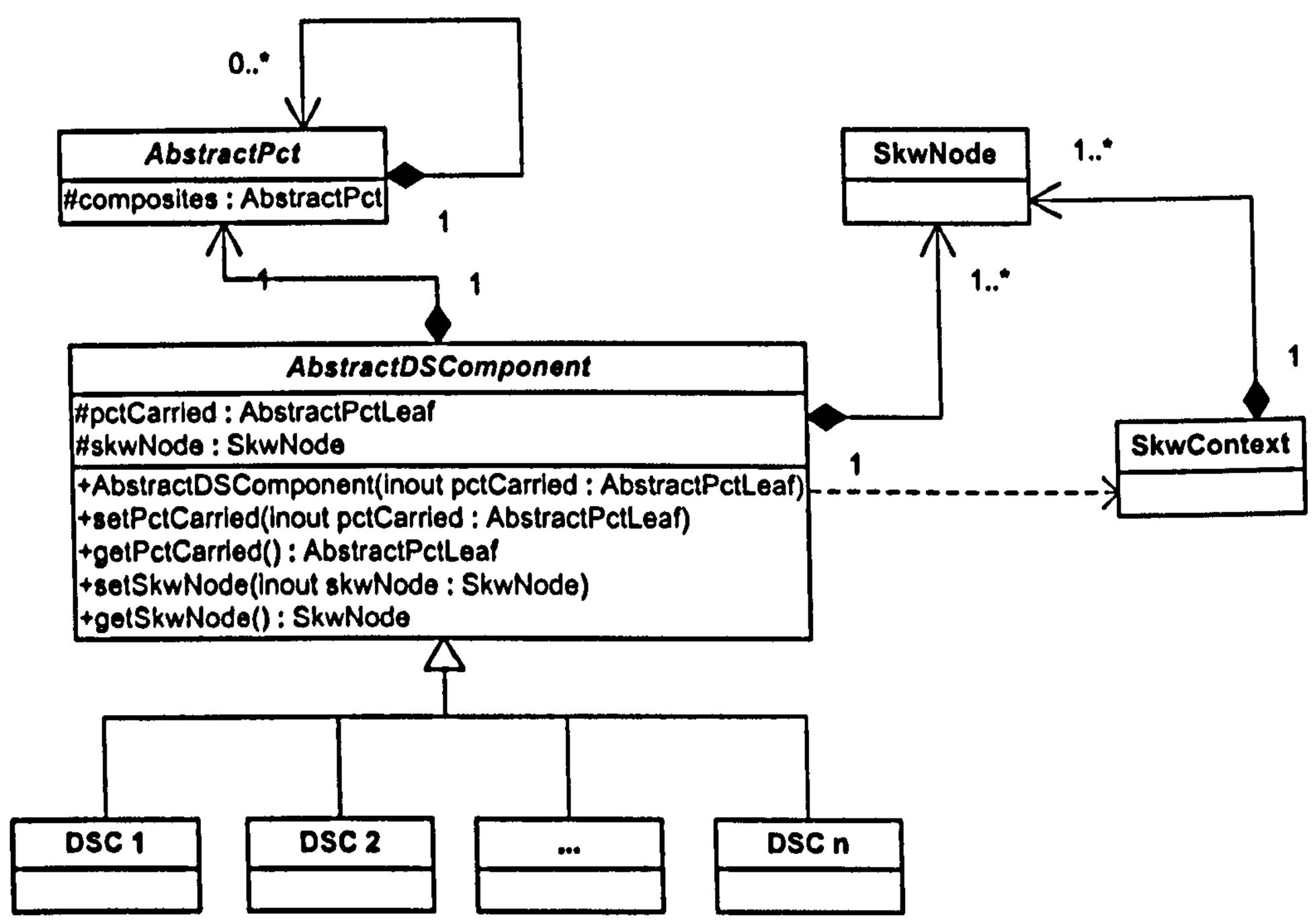


Figure 6.14: UML class diagram: domain-specific component AbstractDSComponent

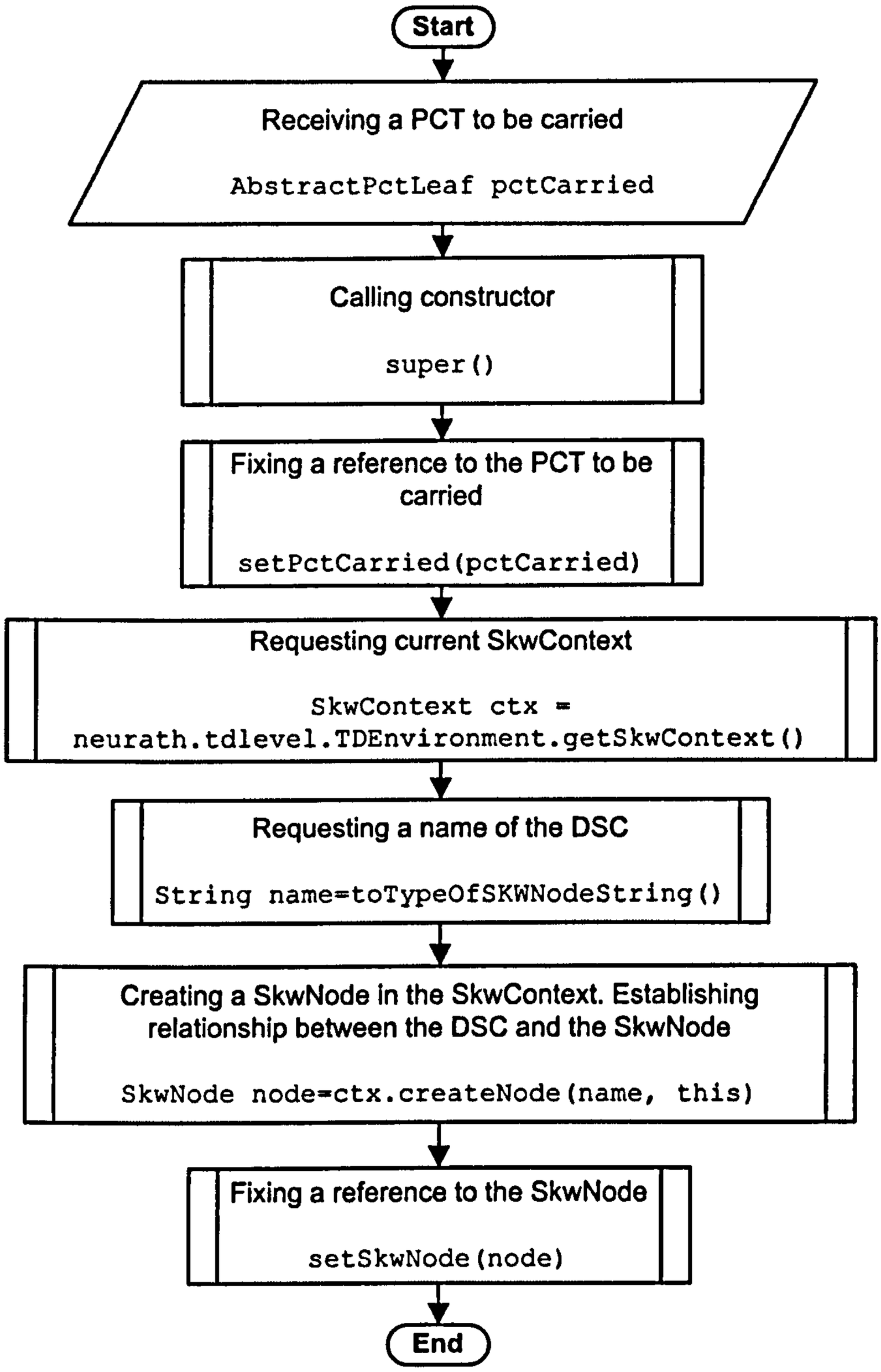


Figure 6.15: A flowchart of an algorithm for an initialisation of the `AbstractDSComponent`

We use object-oriented technology to describe basic architecture and functionality of DSCs. Each DSC is represented by the class `AbstractDSComponent`. Figure 6.14 depicts a UML class diagram that of the `AbstractDSComponent`.

The figure shows the class `AbstractDSComponent`. It holds two attributes `pctCarried` and `skwNode` and defines a constructor and methods to access the attributes. The attribute `pctCarried` and a relation to the class `AbstractPct` define a connection between domain-specific component and an encapsulated program code template. The attribute `skwNode` and relations to the classes `SkwNode` and `SkwContext` represent a connection to the SkwContext which specifies the domain ontology.

Attributes `pctCarried` and `skwNode` are initialised during the instantiation of the `AbstractDSComponent`. Therefore all derived DSCs have to call a constructor of the super class. The initialisation works according to the algorithm depicted in Figure 6.15.

The algorithm works every time an instance of a DSC is created. However, DSCs may extend initialisation with additional specific routines. A complete specification of the class `AbstractDSComponent` in form of source code can be found in [96].

6.6 Hierarchy of Domain-specific Components

DSCs, defined by classes that extend the basic super class `AbstractDSComponent`, form a class hierarchy of domain-specific components. Figure 6.16 depicts an example of the class hierarchy formed by DSCs describing a part of the "House Automation" target domain.

The class hierarchy defines two main classes (terms) `Device` and `Property`. Derived classes extend those two forming the hierarchy further. Derived classes are for example `TemperatureSensor` and `WindSensor`.

6.6.1 Derived DSCs

Derived classes are created by a Software Architect during the composition system definition phase. They are created according to the requirements defined by domain experts that are going to use those classes at the design phase. Additionally, the extension of the base class `AbstractDSComponent` by derived classes is characterised by the following:

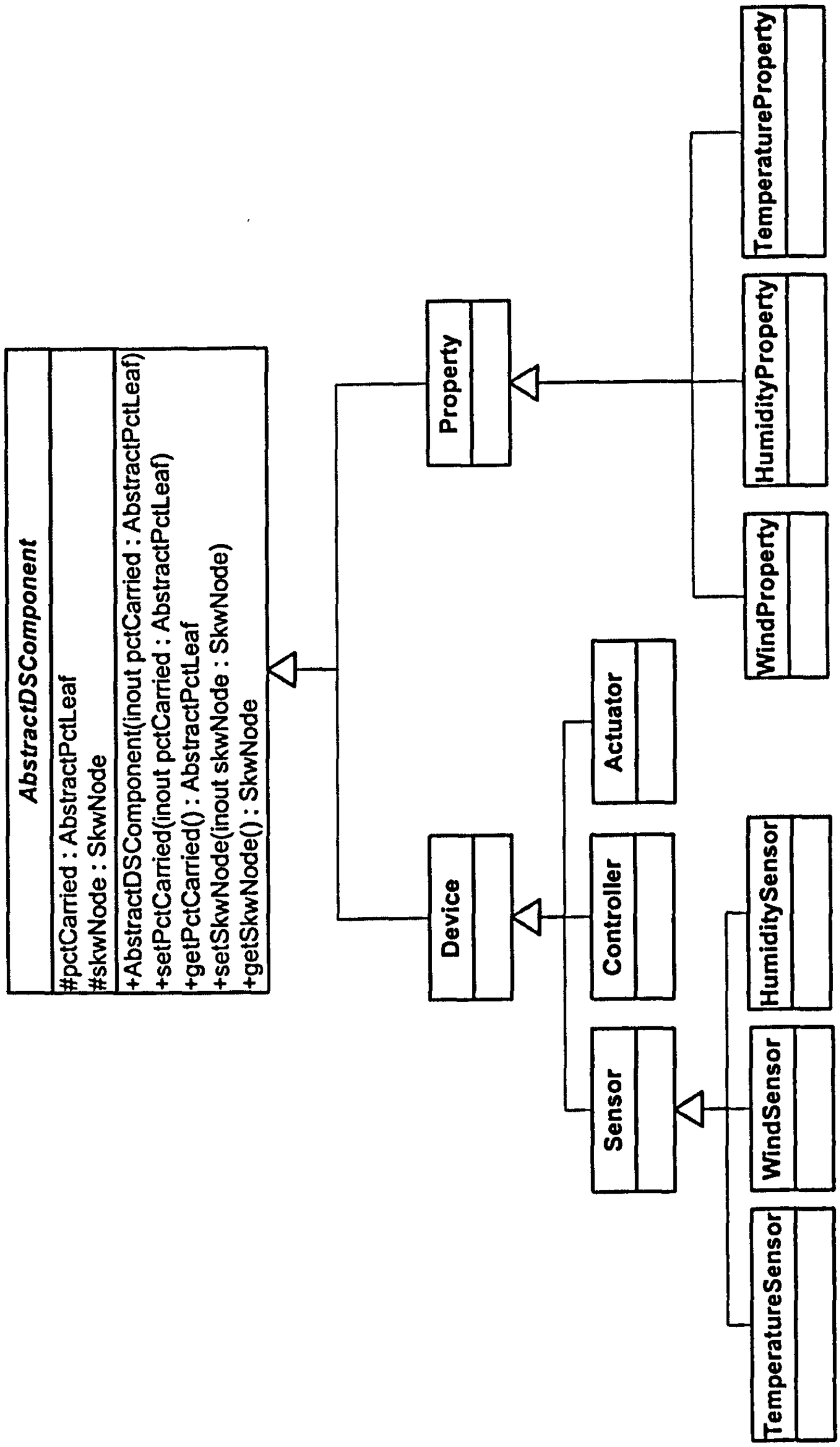


Figure 6.16: UML Class diagram: a class hierarchy of DSCs for a part of the "House Automation" domain

1. **Connection with the PCT.** It is defined what concrete PCT is held by the derived DSC. More speicifically, a instance of PCT is created and saved. Additionally, further connection aspects are defined with the item **Attributes** below.

2. **Connection with the SkwContext.** It is defined how a creation of the derived DSC modifies a SkwContext. In other words, it should be defined what this creation means in terms of the target domain. A frequent example is the following: the creation of the DSC means (causes) a creation within the SkwContxt of a SkwNode of special type. Additionally, further aspects about the connection with the SkwContext are described with the further item **Attributes.**

3. **Attributes.** The attributes of DSCs are defined and access methods to them are provided. Normally, a creation of a DSC means a creation of a SkwNode as each DSC is represented by its domain-specific term. The attributes defined in the DSC, that are interesting for a domain expert to manipulate with, are represented by the attribute definitions within the related SkwNode. Additionally to the attributes, the DSC defines access methods to request values of attributes and to assign values to attributes. Access methods contain routines to work with the related PCT.

Figure 6.17 schematically shows characteristics of a derived DSC.

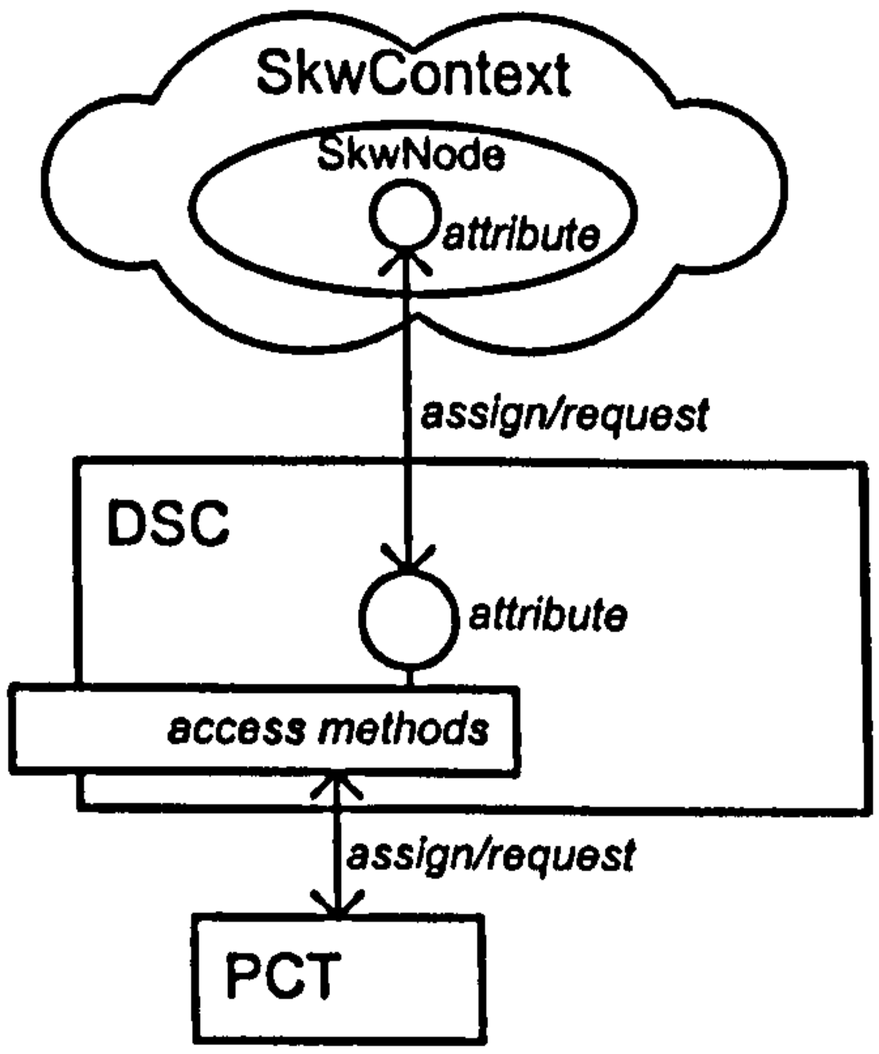


Figure 6.17: Characteristics of a derived DSC component

6.6.2 Rules of Derived DSCs

A specification of a DSC is based on two main parts: (1) initialisation of a DSC and (2) specification of attributes. Further both are described. A concrete example of specifications of derived DSCs can be found in Section C.1.2 from Appendix C and Section D.1.2 in Appendix D.

6.6.2.1 Initialisation

Figure 6.17 depicts an algorithm showing rules specified by each derived DSC.

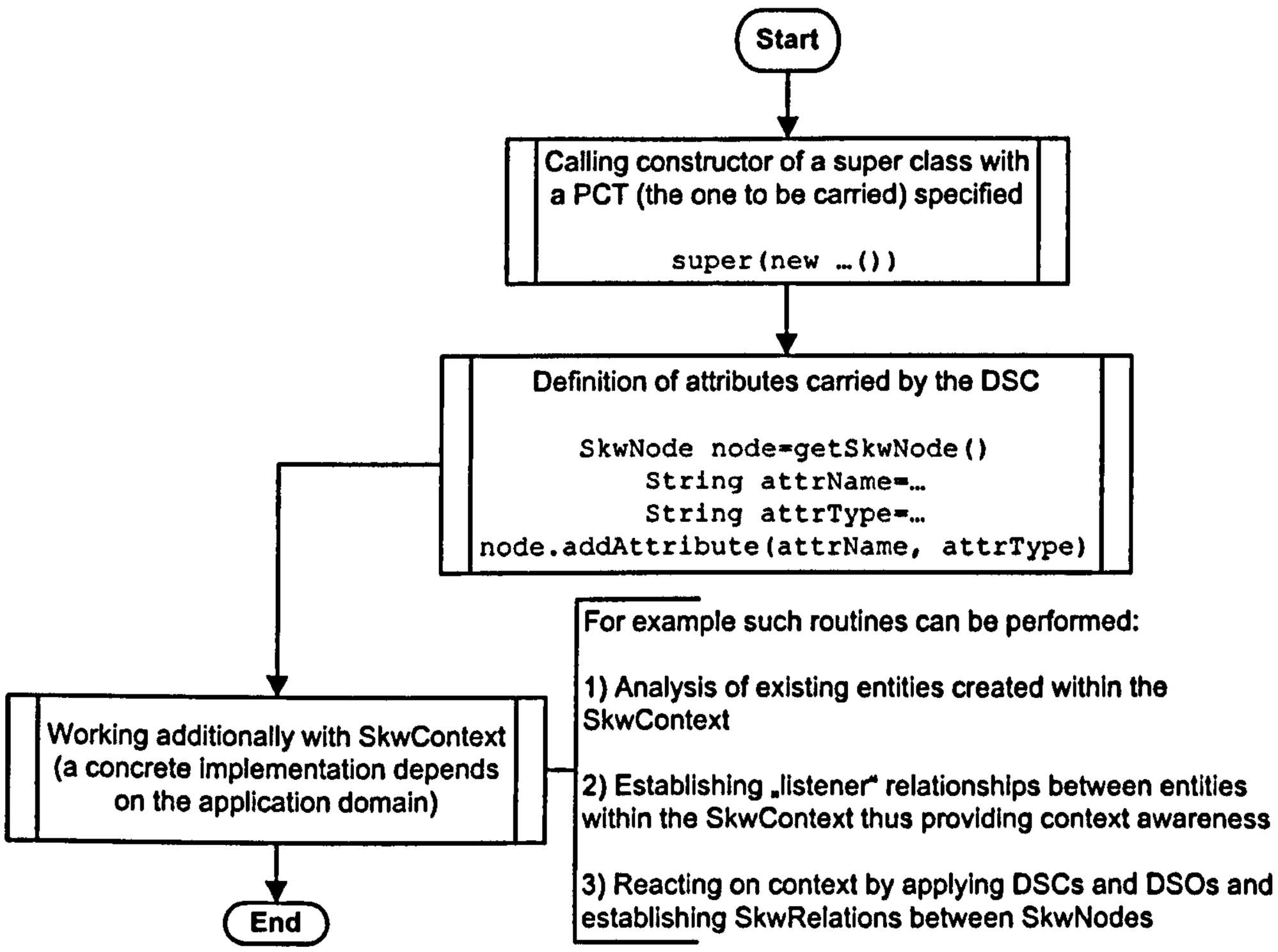


Figure 6.18: A flowchart for an algorithm for initialisation of a derived DSC

6.6.2.2 Working with Attributes

Each DSC has a set of attributes defined. During the initialisation of a DSC these attributes are defined via a name and a type. A type is every time "`String`". It was taken as universal type for holding string values translatable into other types, like Java types

`int` or `double`. However, the developer of DSCs must take care about the correct specification of setter and getter methods for each attribute. A setter method accesses a PCT carried by the DSC and sets a concrete value of an attribute the setter method is defined for. A getter method accesses a PCT carried by the DSC and requests a values associated with the attribute the getter method is defined for.

6.7 DSC-Architecture Specification

The DSC-Architecture specification shows how components defined at different levels of composition are related. Figure 6.19 shows an example of the DSC-Architecture specification.

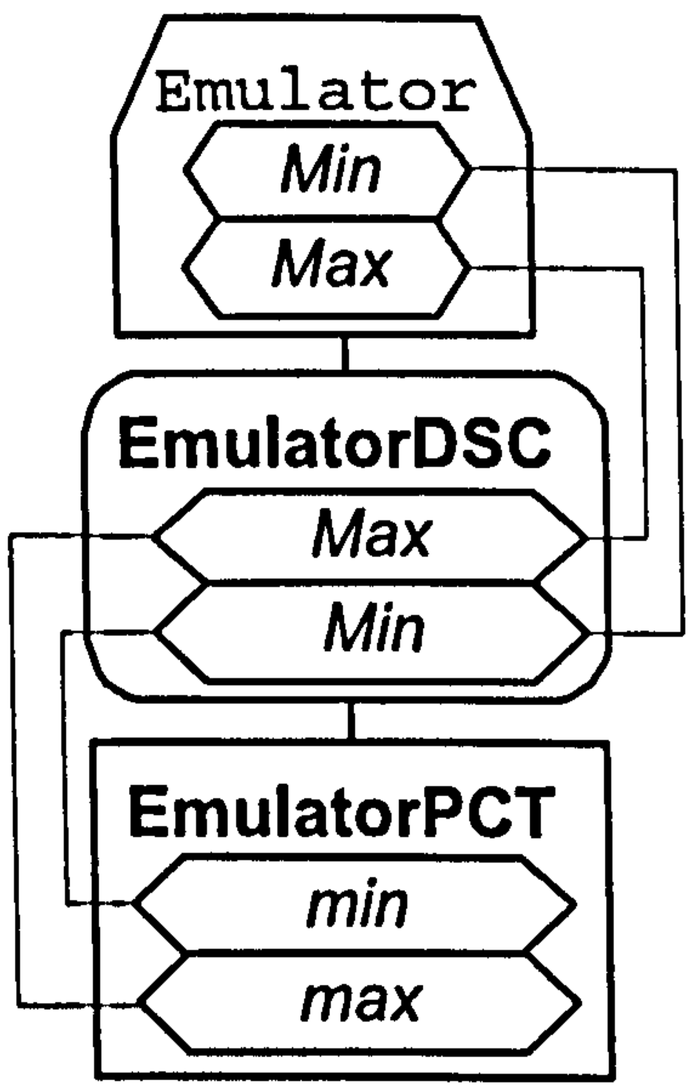


Figure 6.19: An example of the DSC-Architecutre specification

There are three components shown - `Emulator`, `EmulatorDSC` and `EmulatorPCT`. The top element `Emulator` represents a `SkwNode` component. The diamond-like elements denote its attributes, which in this case are `Min` and `Max`. These are connected the attributes defined for `EmulatorDSC` which represents a domain-specific component. Both the DSC and the `SkwNode` represent the Target Domain Level of composition. The component `EmulatorPCT` denotes a PCT and represents a Template Level of composition. The DSC-diagram shows what parameters of a PCT are

connected to what attributes of a DSC, and also what attributes of a DSC are connected to what attributes of SkwNode.

These kinds of diagrams are used in Chapter 8.

6.8 Domain-specific Operations

Domain-specific operations are domain-specific rules to manipulate with domain-specific components. We define model for DSOs to describe operations. Each DSO is a combination of molecular operations, defined at the Template Level of composition (see Section 5), and other DSOs.

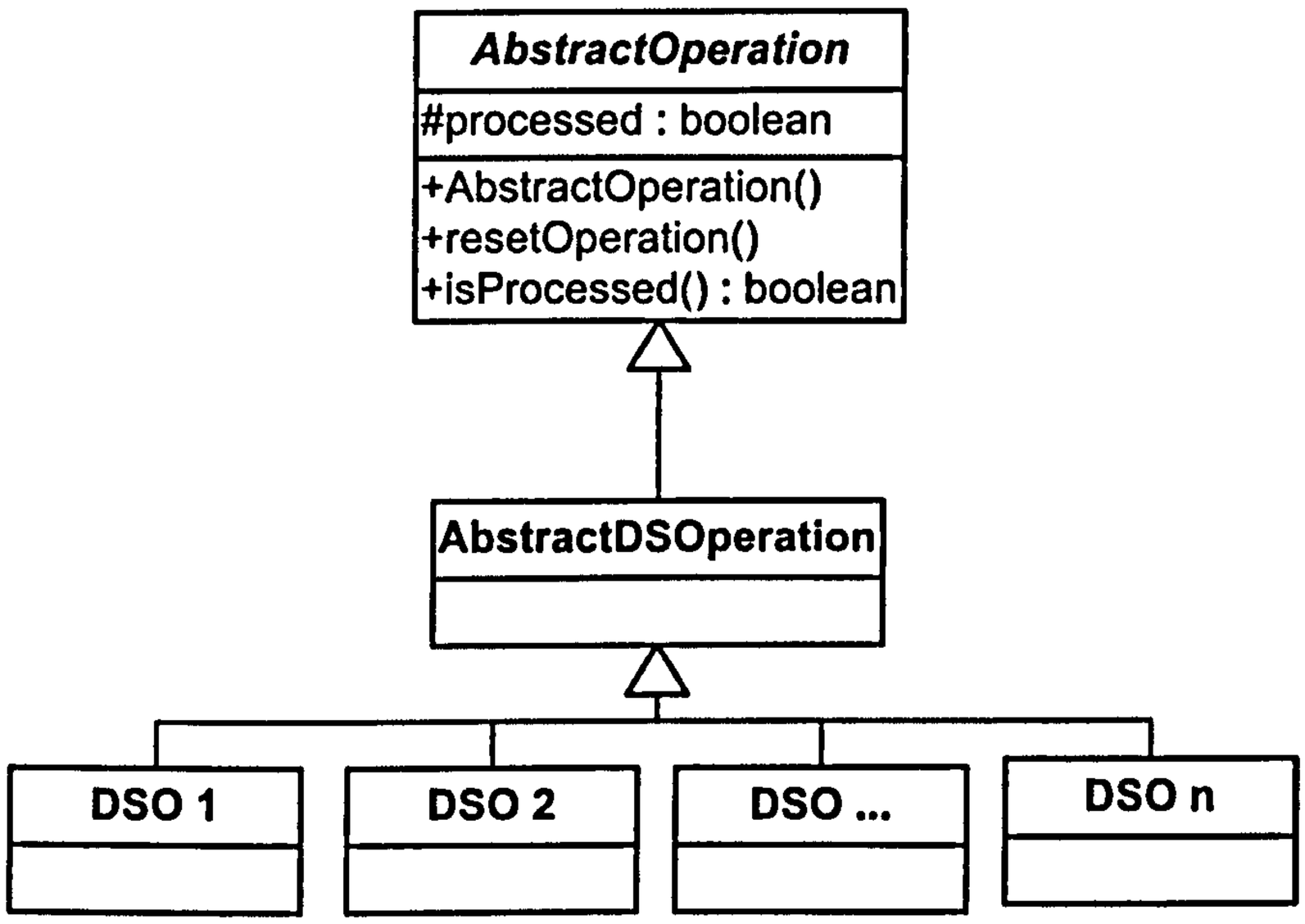


Figure 6.20: UML class diagram: domain-specific operation `AbstractDSOperation`

Figure 6.20 depicts the UML class diagram showing the super class `AbstractDSOperation` and derived classes. Additionally, it shows that the class `AbstractDSOperation` extends, practically without changes, the class `AbstractOperation`. This means that DSOs are defined in a similar way as molecular operations defined at the Template Level. We have defined the class `AbstractDSOperation` to avoid mixing the concepts defined at the Template Level of composition and the concepts defined by the Target Domain Level of composition. See section 5.5 for more details about the molecular operations.

6.9 Hierarchy of Domain-specific Operations

DSOs are defined by classes that extend the basic super class `AbstractDSOperation`. Derived DSOs form a class hierarchy of domain-specific operations. Figure 6.21 depicts an example of the class hierarchy formed by DSOs describing a part of the "Control System" target domain.

The class hierarchy defines several domain-specific operations represented by the derived classes `Add`, `AddDevice`, `AddProperty`, `Create` and `Delete`. The derived classes are created by a Software Architect during the composition system definition phase. They are created according to the requirements defined by domain experts that are going to use those classes at the design phase. Newly defined DSOs should meet certain requirements and are created according to a certain schema.

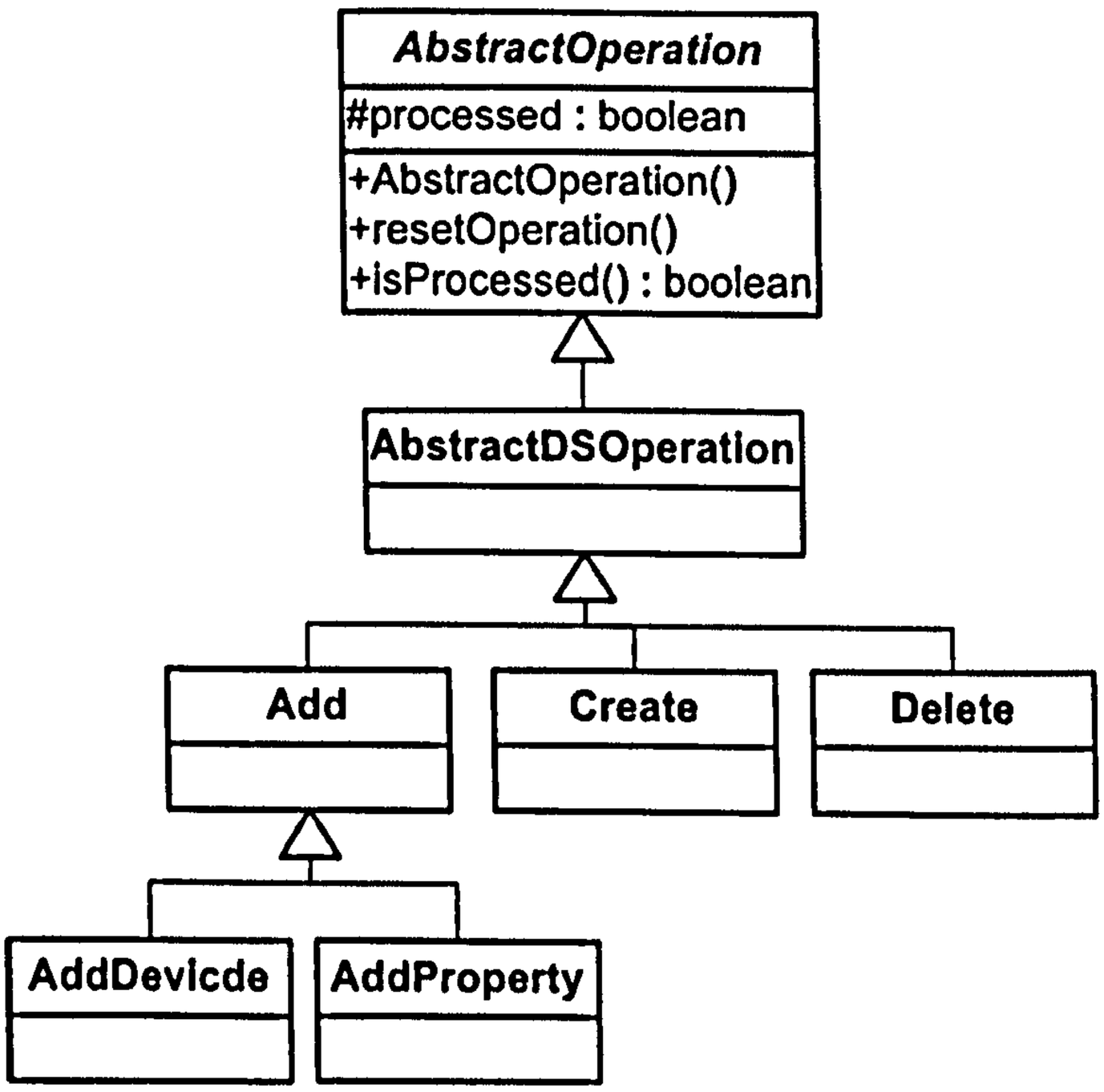


Figure 6.21: An example of the DSO class hierarchy describing a part of the "Control System" domain

Therefore DSOs are characterised by the following:

1. They define parameters (operands) and facilities to assign a value to each parameter, as well as, to request a value of each parameter

2. They define a manipulation rule with operands. Any other operations may be used within the manipulation rule. The rule is processed when the operation is applied to assigned parameters.

3. They define how a SkwContext is changed when the manipulation rule is applied

Figure 6.22 illustrates a UML class diagram showing characteristics of a DSO.

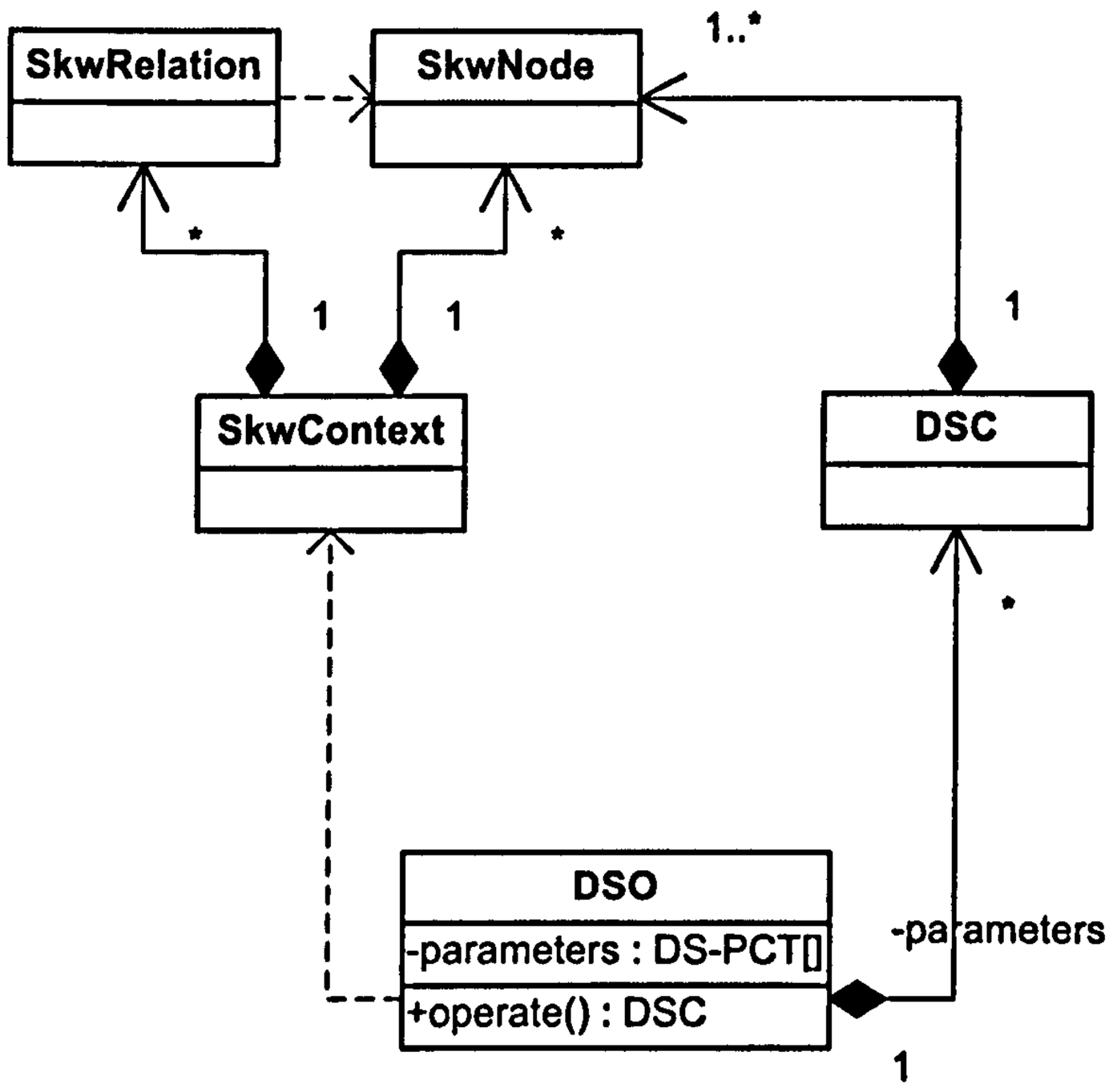


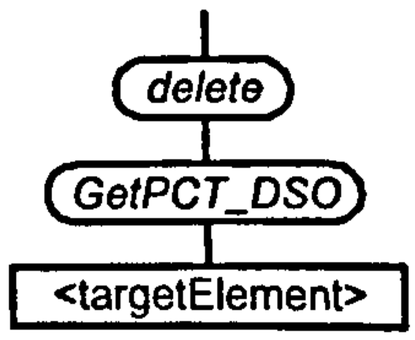
Figure 6.22: UML class diagram: characteristics of a DSO

6.10 DSO-Specification

We have standardised the form in which the DSOs can be specified. This form is called DSO-Specification. Figure 6.23 depicts an example of a DSO-Specification.

```
Delete_DSO(targetElement:AbstractDSComponent)=

(1)
```




```
(2) SkwNode node=targetElement.getSkwNode();
    SkwContext.deleteNode(node);
```

Figure 6.23: An example of a DSO-Specification

The DSO-Specification consists of three parts. The first one is the header of the operation which is located at the top. In case of this example, the header is:
`Delete_DSO(comp:AbstractDSComponent)=`
It denotes the name of the operations as well as input parameters and their types. The second part of the specification is the expression of the operation. Instead of some operands the parameters are used that have the syntax like this: <*parameter name*>. The third part is the description changing the domain ontology described by the SkwContext.

6.11 Domain-specific Expressions

During the design phase domain experts directly use domain-specific components and operations. Frequently they form sequences of DSOs applied to DSCs, thus creating expressions. These expressions represent sentences of a simple domain-specific language. Expressions are formed similar to ones defined at the Template Level of composition (described in Section 5.6).

Expressions consist of DSOs objects as "operators" and DSCs objects as "operands". Figure 6.24 shows an example of an expression in a tree form and as a formula.

Exaclty as in case of molecular operations and program code templates, for DSCs and DSOs we use object-oriented technology to describe expressions. Each node in the expression tree is represented by an object that is an exemplar of the class `AbstractPctlExpressionNode`. Figure 6.25 shows a specification of this class.

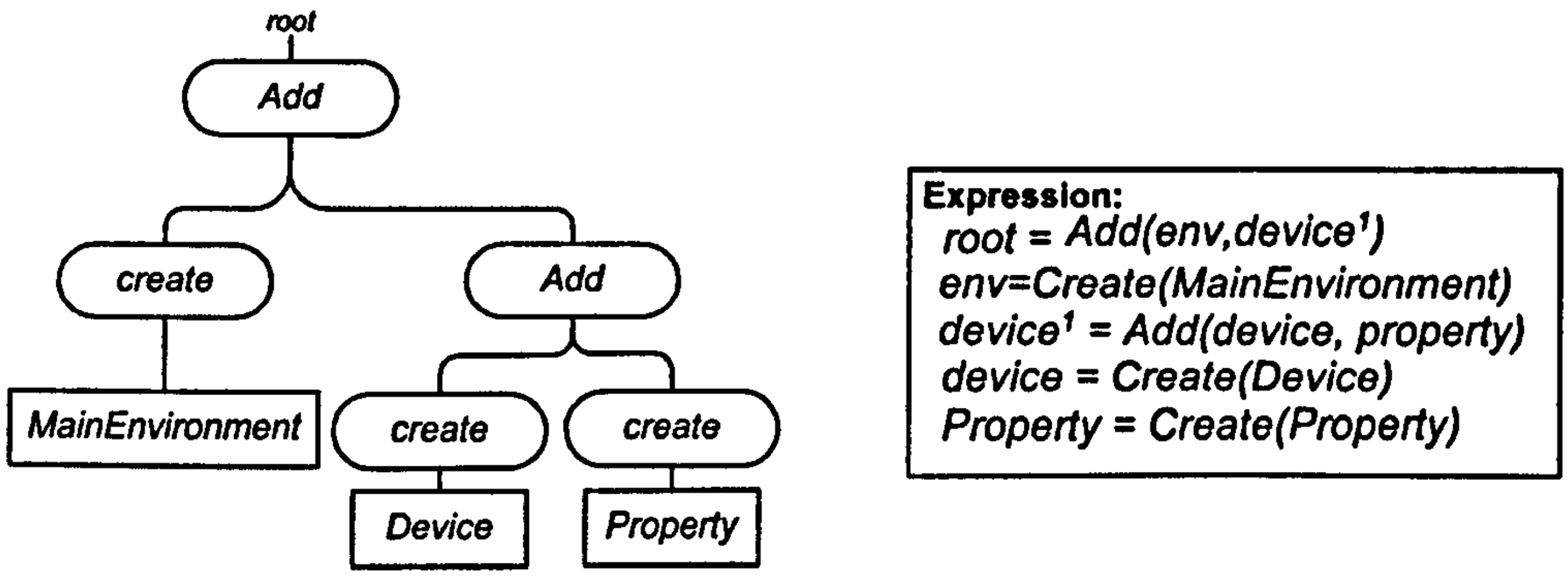


Figure 6.24: Domain-specific expression in a tree form (left) and as a formula (right)

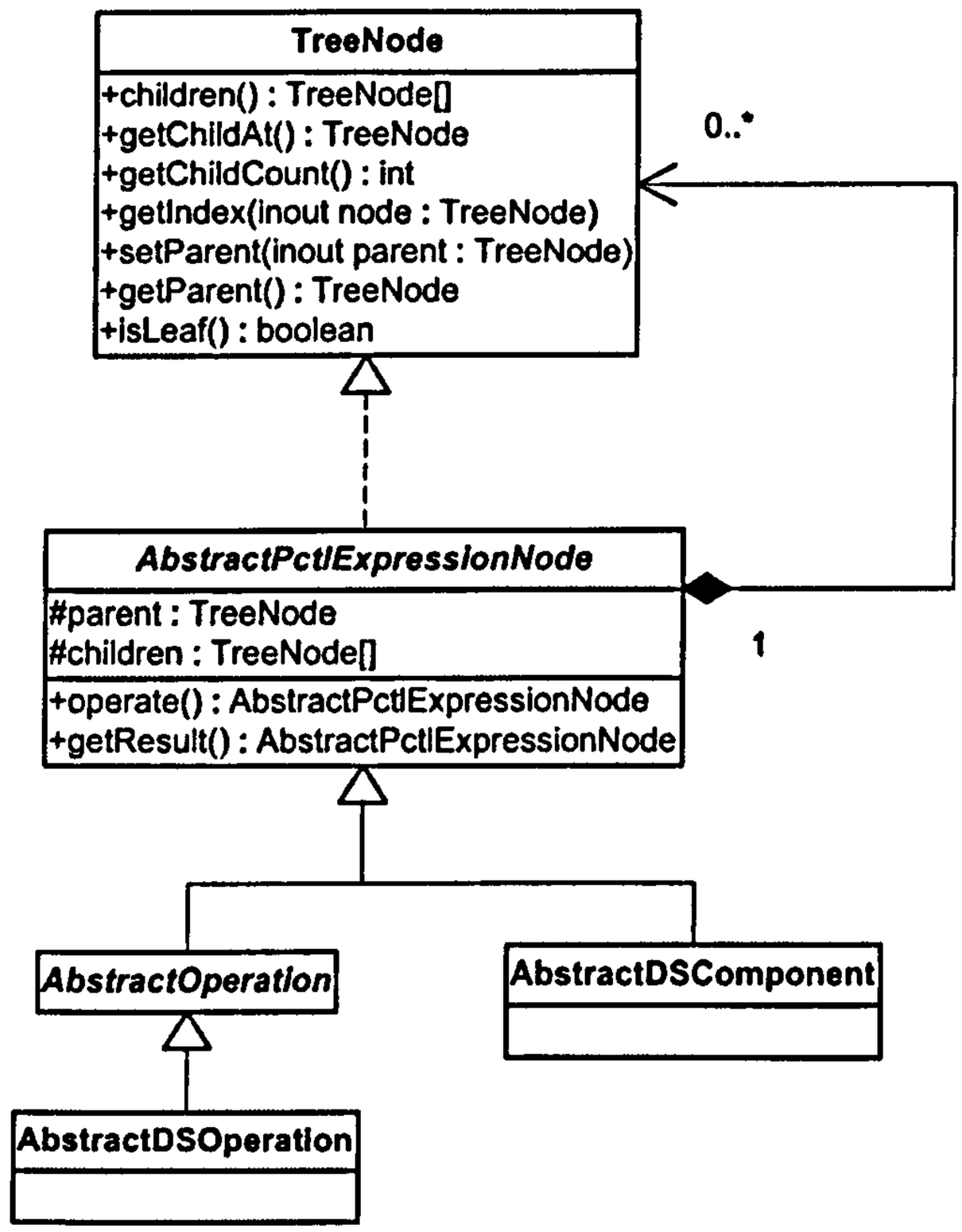


Figure 6.25: UML Class diagram of the class AbstractPctlExpressionNode

Accodring to the UML Class diagram, each node in the exression tree has the following behaviour:

1. Each node is a tree node. This is shown with the "realisation" relationship between the class `AbstractPctlExpressionNode` and the interface `TreeNode`. That means:

   (a) A node may have multiple children nodes (methods `children()`, `getChildAt()`, `getChildCount()` and `getIndex()`). A composition relationship shows that the children of a node have tree node behaviour (the class `TreeNode`).

   (b) If a node is not a root of the expression tree, it has one parent node (accessed via methods `setParent()` and `getParent()`).

   (c) A node may be defined as a leaf (this is requested with the method `isLeaf()`).

2. Each node in the tree, if not a leaf, represents a part of the whole expression. Each node is able to initiate a processing of a part of expression it holds (the method `operate()`). The result is held by each root of a sub-tree, that represents an expression and may be accessed via the method `getResult()`.

3. Nodes in the expression tree are DSOs and DSCs. That is shown with the generalisation relationships.

An expression defined via linked objects (nodes) in tree form can be executed by calling `operate()` method of the root object (node). The way of how expression is processed can be described with the flowchart depicted in Figure 6.26.

Initially, all preconditions are checked which may include checking if operation has been already processed, if all parameters are of correct types and have correct states. If all requirements are met, then all sub-trees are recursively executed by calling the `operate()` method for each sub-root. Once, all sub-expressions are calculated PCTL expressions are formed. These are then executed, thus causing re-configuration at the template level of composition. A corresponding re-configuration is also made at the target domain level of composition, namely in the SkwContext. If there are no failures occurred, the flag that denotes completeness of processing is set.

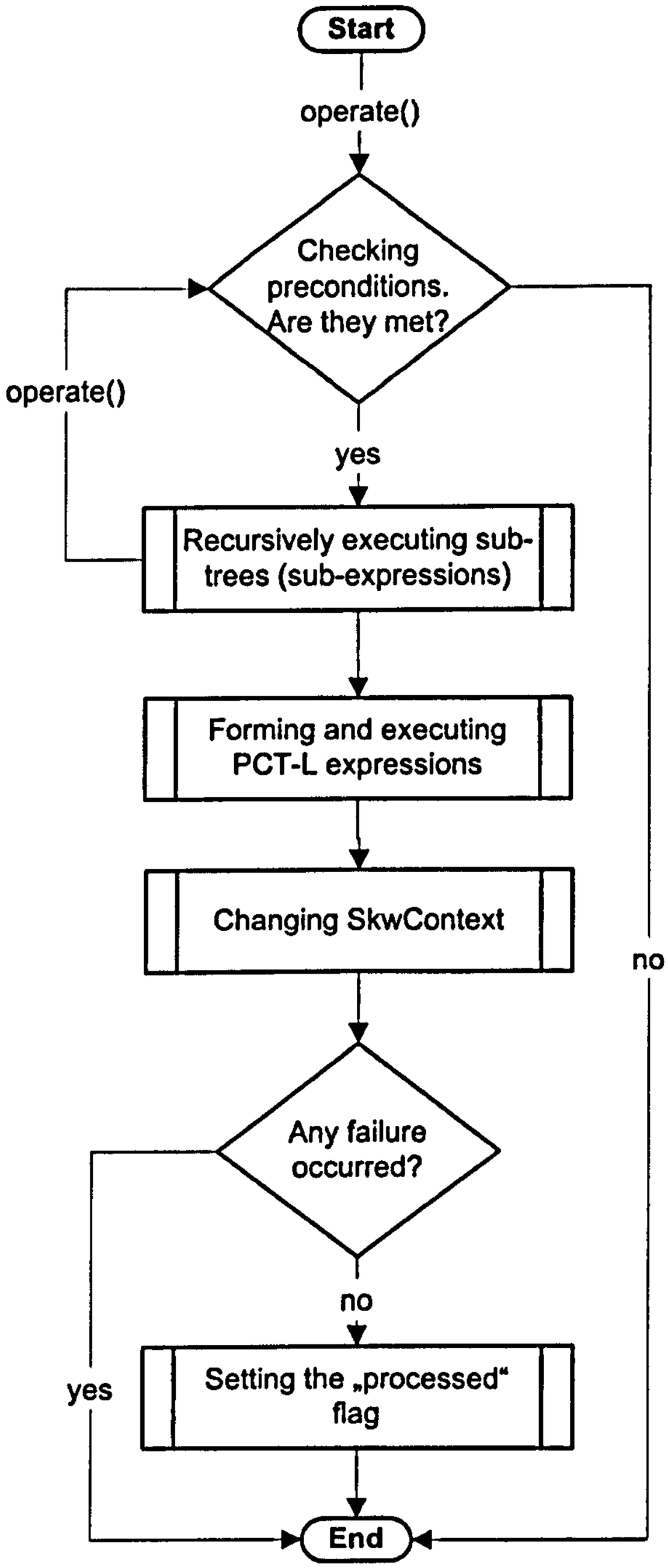


Figure 6.26: A flowchart showing how a domain-specific expression is processed

6.12 Defined DSCs and DSOs

One DSC and few DSO are often reused in different application domains. These are `MainContainerDSC`, `Instantiate_DSO`, `Delete_DSO`, `Merge_DSO` and `GetPCT_DSO`. Further, we provide specifications of each of them.

6.12.1 MainContainerDSC

The `MainContainerDSC` is the very top PCT container that contains other PCTs taking part in configuration of a program code of a software system. It is characterised by the following:

1. It carries `EmptyBasePCT`.

2. It carries the SkwNode named `MainContainer`.

A concrete specification of the class `MainContainerDSC` in form of source code can be found in [96].

6.12.2 GetPCT_DSO

The `GetPCT_DSO` requests a DSCs for a carried PCT. Figure 6.27 depicts a specification of the `GetPCT_DSO`.

```
GetPCT_DSO(comp:AbstractDSComponent)=

(1)comp.getResult()
(2) N/A
```

Figure 6.27: DSO-Specification of the `GetPCT_DSO`

6.12.3 Instantiate_DSO

The `Instantiate_DSO` creates an instance of type denoted by the parameter `dsComponentTypeName`. Figure 6.28 depicts a specification of the `Instantiate_DSO`.

```
Instantiate_DSO(dsComponentTypeName:String)=

(1)Class theClass=Class.forName(pctTypeName);
   Object instObj = theClass.newInstance();
   instance = (AbstractPctlExpressionNode)instObj;
   return instance;

(2) Automatically done according to rules specified in
    the AbstractDSComponent
```

Figure 6.28: DSO-Specification of the `Instantiate_DSO`

A complete specification of the class `Instantiate_DSO` in form of source code can be found in [96].

6.12.4 Delete_DSO

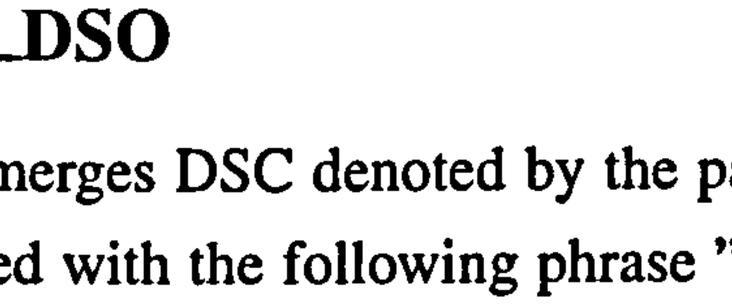
The `Delete_DSO` deletes specified DSC denoted by the parameter `targetElement`. A specification of the `Delete_DSO` has been already depicted in Figure 6.23. A complete specification of the class `Delete_DSO` in form of source code can be found in [96].

6.12.5 Merge_DSO

The `Merge_DSO` merges DSC denoted by the parameters `parent` and `chlid`. Merging can be described with the following phrase "put a child inside a parent". This DSO initiates merging of PCTs carried by DSCs denoted by the `parent` and `child` parameters. Moreover, it produces a "has" relationship between SkwNodes related to the DSCs being merged. The "has" is directed from `parent` to `child`. Figure 6.29 depicts a specification of the `Merge_DSO`.

```
Merge_DSO(parent,child:AbstractDSComponent)=

(1)
```




```
(2) SkwNode node1=parent.getSkwNode();
    SkwNode node2=child.getSkwNode();
    SkwContext ctx=getSkwContext();
    SkwRelation relation=ctx.createRelation("has",node1, node2);
```

Figure 6.29: DSO-Specification of the `Merge_DSO`

A complete specification of the class `Merge_DSO` in form of source code can be found in [96].

6.13 Implementation Environment

The concepts defined at the Target Domain Level of composition are practically implemented in the *Domain-specific Composition Library*. The library was implemented with the Java programming language. We defined the package `neurath.tdlevel` as the main package of the library. It works with the following main sub-packages.

1. `skw` - defines classes that implement SkwContext to describe domain ontology.

2. `dspctl` - defines classes representing the domain-specific language, elements of which are DSCs and DSOs. The core functionality of these elements are presented by the classes `AbstractDSComponent` and `AbstractDSOperation`.

3. `Package for a target domain` - defines classes that describe domain-specific composition language for a target domain.

Figure 6.30 shows main packages that implement the Domain-specific Composition Library. The relation from the package `Package for a target domain` to the package `neurath.templatelevel` shows a cooperation between the Domain-specific Composition Library and the Template Composition System Library that implements concepts defined at the Template Level of composition. The notice `Expression` at the right-bottom corner means that from the DSCs and DSOs defined for a target domain expressions are formed during design time.

Listing 6.11 shows a program that uses the API of the Domain-specific Composition Library.

```
1
2 import neurath.tdlevel.dspctl.*;
3 ...
4 Instantiate_DSO iOp1 = new Instantiate_DSO("MainEnvironment");
5 Instantiate_DSO iOp2 = new Instantiate_DSO("Device");
6 Instantiate_DSO iOp3 = new Instantiate_DSO("Property");
7
8 Merge_DSO mOp1 = new Merge_DSO(iOp2, iOp3);
```

```
9  Merge_DSO mOp2 = new Merge_DSO (iOp1, mOp1);
10
11 AbstractDSComponent result = mOp2.operate();
12 ...
```

Listing 6.11: Example program to create and process an expression of domain-specific composition language

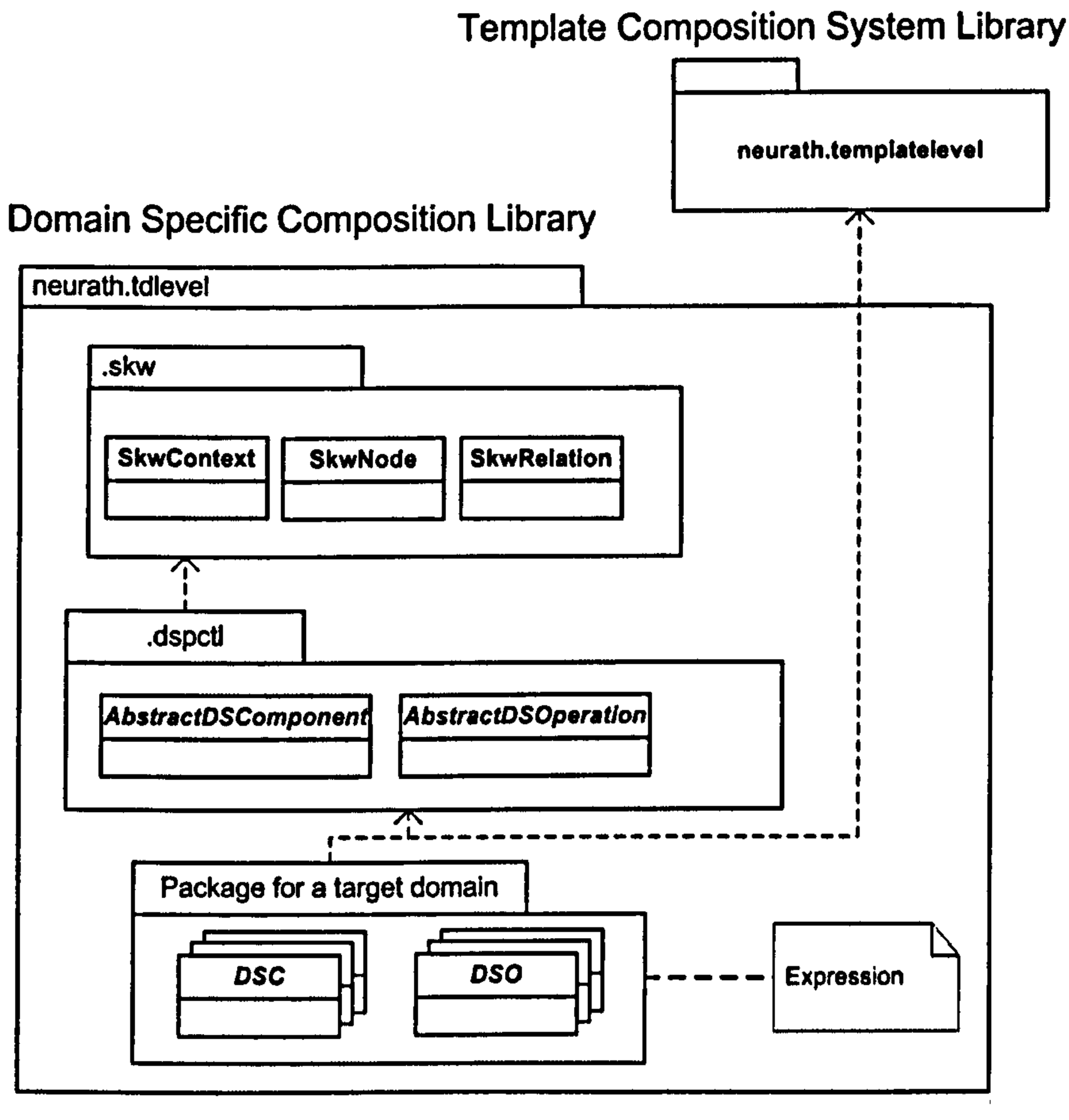


Figure 6.30: Domain-specific Composition Library: main packages

The program describes an expression depicted in Figure 6.24. First instances `iOp1`, `iOp2` and `iOp3` of the operation `Instantiate_DSO` are created. Each instance gets a value as an operand. After this, an instance of the `Merge_DSO` operation is created. This instance takes two operands `iOp2` and `iOp3` which represent a `Device` instance and a `Property` instance. In a similar way the `Merge_DSO` operation denoted by the

instance `mOp2` is defined. After this, the very root represented by the object `mOp2` of the formed expression tree is used to initiate a processing of the expression by calling the method `operate`. The method will recursively initiate a processing all sub-trees (sub-expressions). After processing the expression the result is returned into the variable `result`.

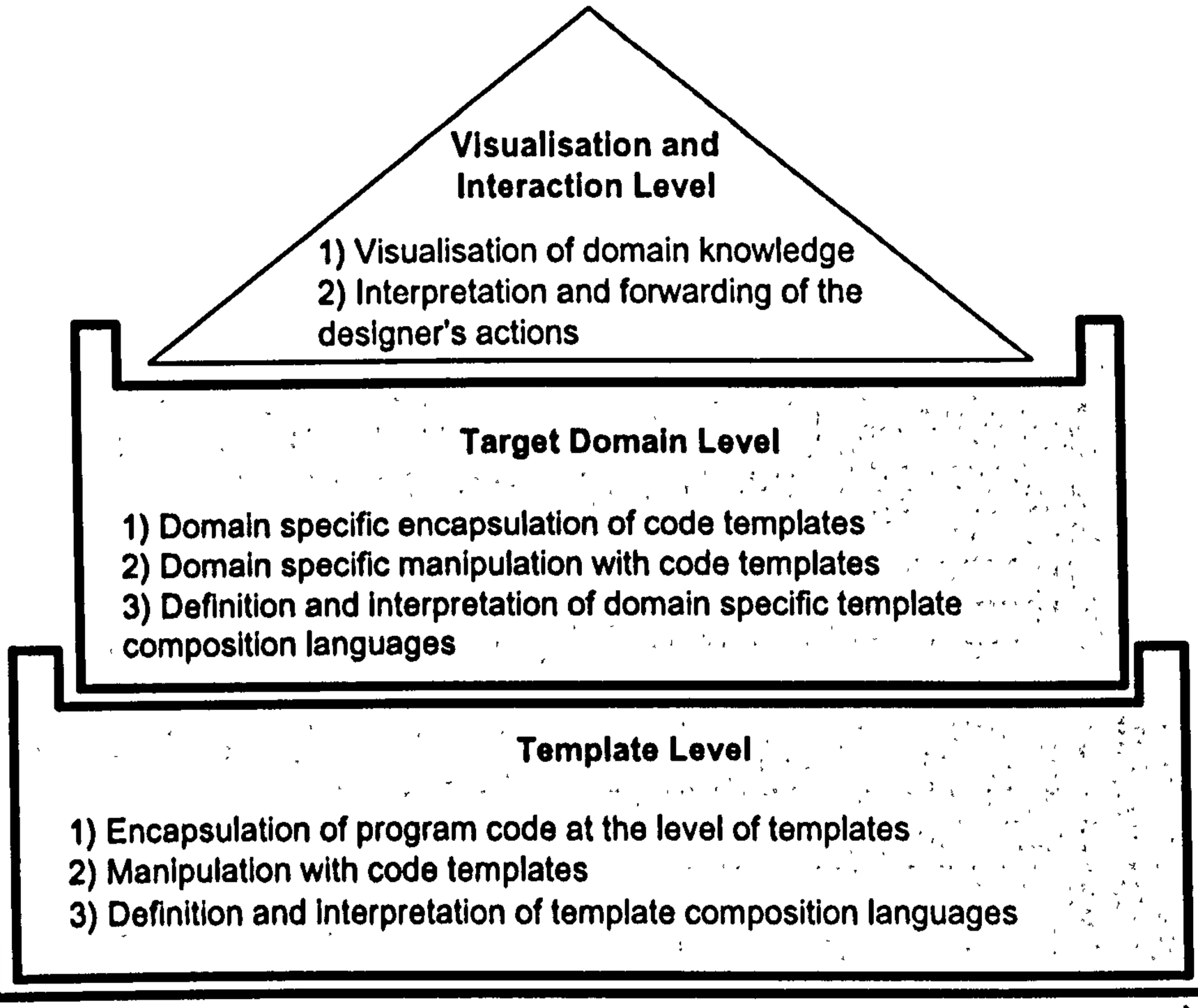


Figure 6.31: Levels of composition within the Neurath Composition Framework

6.14 Summary

This chapter has introduced concepts defined at the Target Domain Level of composition. These concepts represent a domain-specific composition system to compose pro-

gram code with templates by domain experts. We have presented the strategy of bridging a template composition system up to the level of domain experts. Basically this strategy means encapsulation of PCTs and molecular operations with domain-specific components, called DSCs, and domain-specific operations, called DSOs. DSCs and DSOs form simple domain-specific composition language. These components are connected to the SkwContext which describes the state of the system being built statically and dynamically. Moreover, the chapter shows an organisation of an implementation environment for the described concepts.

Figure 6.31 shows different levels of composition and marks the material that has been already described. The next chapter introduces the Visualisation and Interaction Level of composition to improve the externalisation of template composition systems.

Chapter 7

Visualisation and Interaction Level

This chapter presents an extension to domain-specific composition systems defined at the Visualisation and Interaction Level of composition. This extension is a DSVI which is needed to improve domain-specific interaction with a composition system. The chapter shows the organisation of DSVIs. It explains the main workflow of the process initiated by design actions performed by the domain expert. The chapter concentrates on two main aspects of the interaction. These are the interpretation of designer's actions and the interpretation of the state of a designed software system. Moreover, we introduce a basic architecture of an implementation environment for the concepts defined at the Visualisation and Interaction Level of composition. The chapter concludes with summary.

7.1 Introduction

In Section 6 we presented the concepts defined at the Target Domain Level of composition. These concepts are needed to define and use domain-specific composition languages (DSLs) that are based on top of the template composition systems produced at the Template Level. Domain experts apply these DSLs to compose a software system. It is possible to distinguish a bi-directional relation between domain experts and DSLs. From one side, domain experts interact with the language in order to obtain planned results, i.e. program code. From the other side, the state of a resulted in program code is represented and consumed by domain experts to perform further interaction steps.

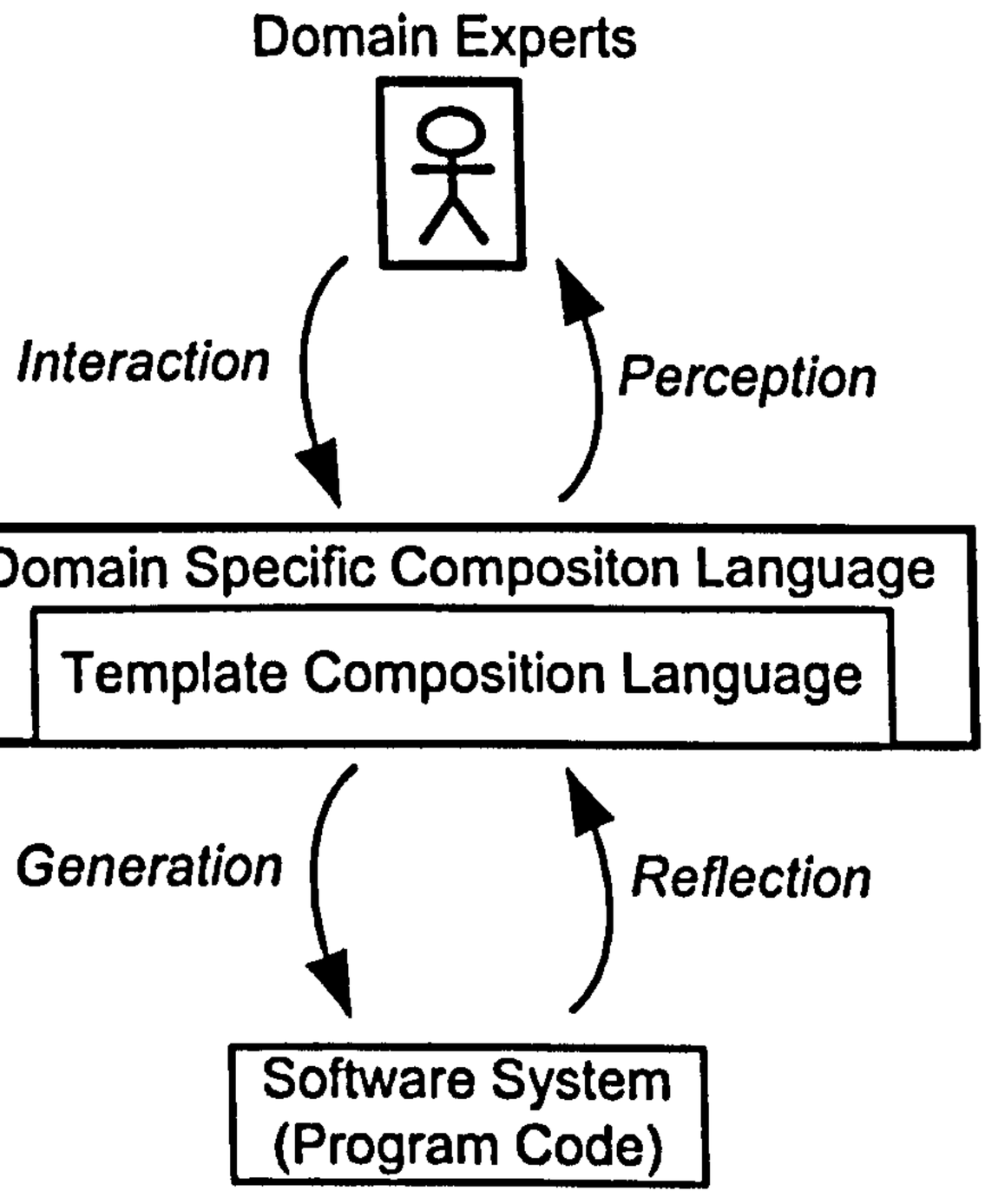


Figure 7.1: Connection between domain experts and a domain-specific composition language

Figure 7.1 shows cooperation between domain experts and a DSL. Domain experts interact with DSL, thus initiating generation of the program code that represents a software system being built. The state of the software system is continuously reflected and interpreted by domain experts, so that they have a feedback on interaction they have performed.

The concepts described at the Target Domain Level define domain-specific composition languages that are text based (a program code) and an interaction is supported by the API of the composition language library. However, these DSLs can be augmented with domain-specific visual notations and interaction mechanisms, thus reducing complexity of a composition process and increasing its efficiency at the design phase. Visual notation and interaction mechanisms form so called DSVI. Figure 7.2 shows cooperation between domain experts and domain-specific composition language via DSVI.

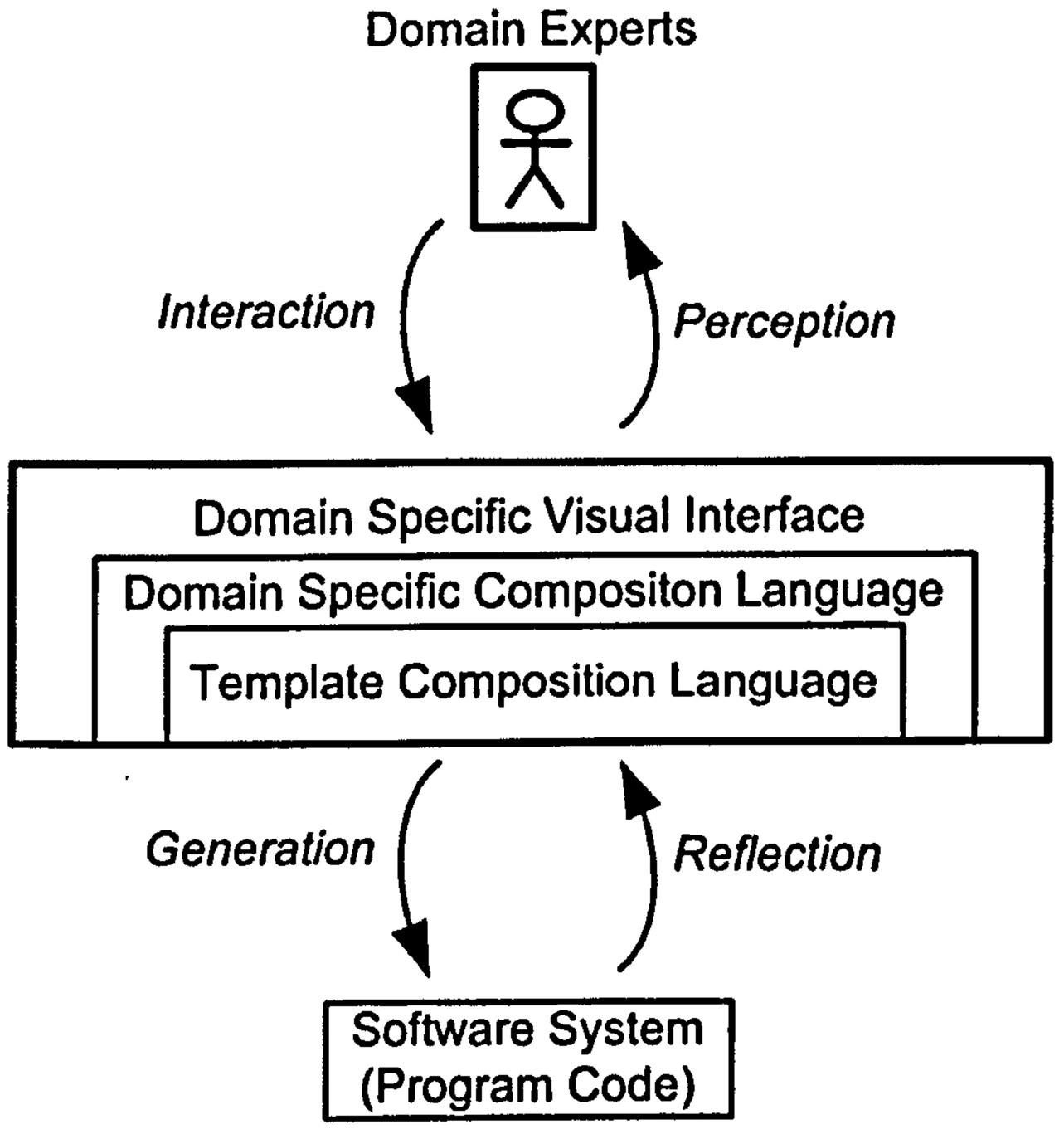


Figure 7.2: Cooperation between domain experts and a domain-specific composition language via domain-specific visual interface

In this chapter we introduce concepts to define and use DSVIs on top of domain-specific composition languages and unite these concepts within the *Visualisation and Interaction Level* of composition. The name of the level is based on two main events continously happening during design phase at this level: (1) receiving of domain-expert's commands by DSVI and responding on these commands and (2) interpretation and perception of state of DSVI

7.2 Architecture

There are two main phases defined. First phase is the composition system definition phase. At this phase a DSVI is defined on top of the domain-specific composition systems (we described them in Section 6. The second phase is the design phase. During this phase a software system is built with help of previously defined DSVI. In the following sections we are going to give more detailed information on both phases.

7.2.1 Composition System Definition Phase

Figure 7.3 illustrates the basic architecture at the Visualisation and Interaction Level of composition from the perspective of the composition system definition phase.

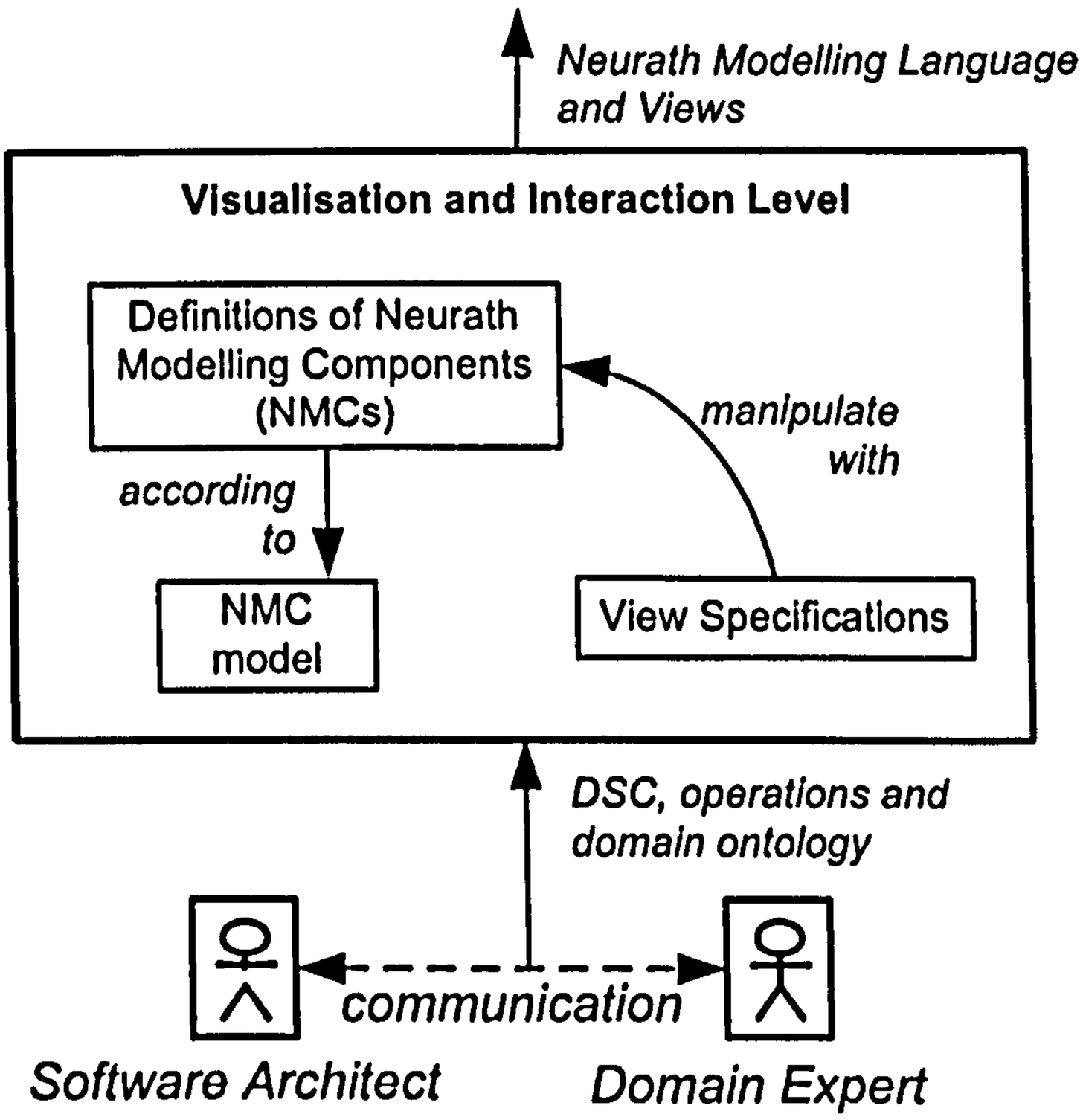


Figure 7.3: An architecture at the Visualisation and Interaction Level (composition system definition phase)

During the composition system definition phase DSVI is defined by the Software Architect according to the requirements specified by the Domain Experts. This interface is defined on top of the domain-specific composition language represented by Domain-specific Components and Domain-specific Operations provided at the Target Domain Level of composition (see Section 6).

The DSVI specification consists of the NML and *Views*. The NML is presented with *Neurath Modelling Components* - visual symbols - that can be combined together. The NML is an interaction mechanism to express the composition steps in a domain-specific way by domain experts. NMCs are defined according to the *NMC model*. The visualisation part is represented by Views. Views specify rules of how system's state is domain-specifically visually represented.

7.2.2 Design phase

During the design phase the DSVI, which is a product of the composition system definition phase, is used by the Domain Experts in order to design a software system. The DSVI is defined on top of a domain-specific language provided at the Target Domain Level. In its turn, this language is defined on top of the template composition system. Therefore, working with DSVI at design time results in a template-based transformation of the designed software system.

From one side, domain experts give commands (actions) to the DSVI that consists of Views and NML, thus resulting in a template-based composition of a software system. From the other side, a state of a software system, which is being built, is visually reflected in terms of the application domain and interpreted by the Domain Expert.

Figure 7.4 shows concepts that take place during design phase. The main input during this phase is actions applied to the DSVI. The DSVI consists of smaller parts that have the same nature as the whole DSVI. The have such a feature as interpretation of the designer's actions into sentences called User Interface Interaction Expressions (UIIEs). These are descriptions of the actions which are needed to generate a expression in the domain-specific composition language (DS-PCTL). These sentences are processed at the Target Domain Level resulting in a new state of a designed system.

The state of the system is visually reflected by NMCs held by Views. As well as in case of NML, actions applied to Views result in transformation of the software system being built and therefore new or modified visual components, which reflect the design state of the software system. The main outputs at the Visualisation and Interaction Level

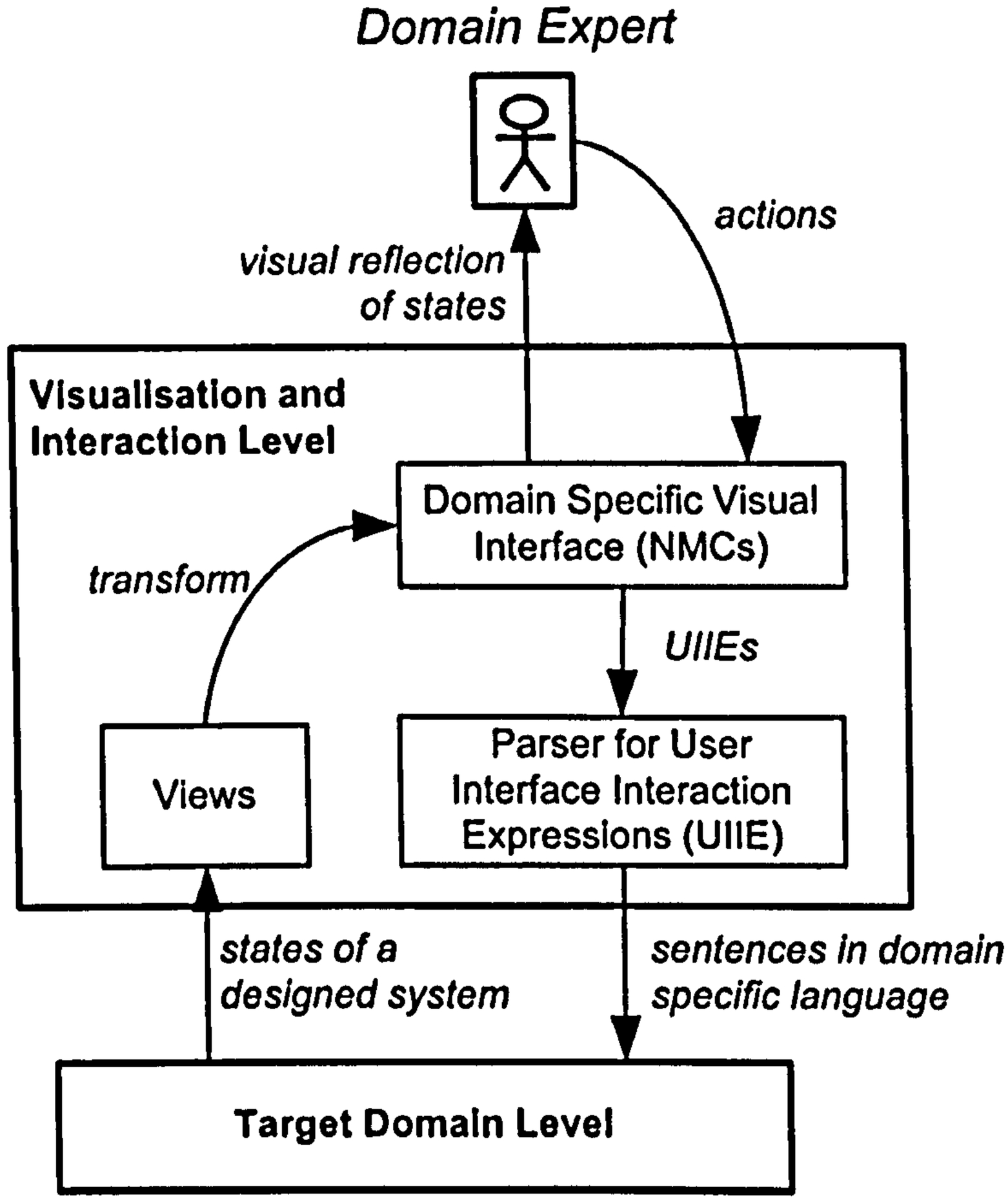


Figure 7.4: An architecture at the Visualisation and Interaction Level (design phase)

during the design phase is the domain-specific visual representation of the system's state. Since Visualisation and Interaction Level involves all the concepts defined at all levels of composition , the other important result is the program code that represents a designed software system.

7.3 Workflow during Design Phase

During the design phase a domain expert visually designs a software system with NML. Figure 7.5 shows a basic workflow diagram for the design process with NML. We distinguish between three main processes.

The first process, shown on the figure as "Interaction with the DSVI", denotes a process of formulation of a DS-PCTL expression by interacting with the DSVI. This interaction means choosing NMCs by the domain expert, for example by clicking a mouse cursor over it, thus resulting in an expression. The expression is a DS-PCTL expression which is an input into the next process.

The second process, shown on the figure as "Transformation of a desiged system", is processing of the input DS-PCTL expression. This results in template-based modifications of a program code of the designed software system. Modified software system is characterised by its changed state. The state model is an input into the third process, called "Reflection of a system's state". During this process the state of the modified software system is reflected visually at the modelling pane. Further we explain each process in detail.

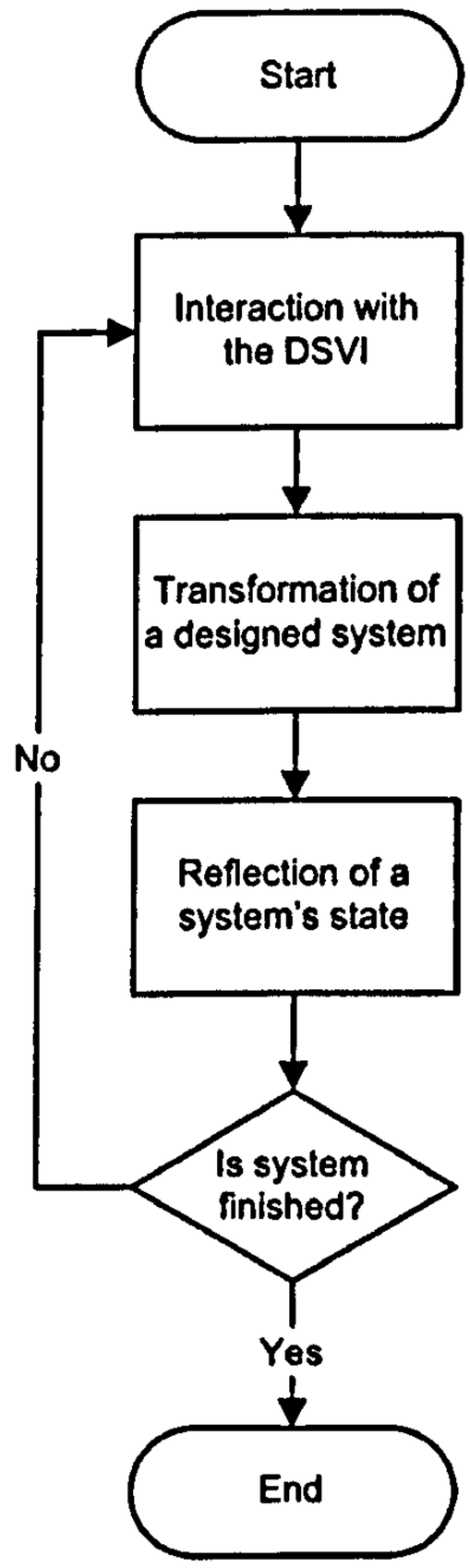


Figure 7.5: Workflow for the design process with NML

7.3.1 Interaction with the DSVI

A domain expert interacts with the DSVI in order to "tell" to the design environment "what should be transformed in the designed software system". Interaction happens by means of NMCs. Shown as visual components, they can be interacted by the domain expert. The interaction can be a mouse click, a keyboard click and so on. What exactly it is depends on the specific implementation of NMCs. The sequence of interactions with NMCs results in so called User Interface Interaction Expression. Generally, UIIE shows a sequence of interactions and elements each interaction is applied to. Figure 7.6 depicts a workflow diagram for the interaction with the DSVI.

The completeness check as well as the translation of the UIIE is performed by UIIE Parser. The output of the parser is a DS-PCTL expression (we have explained it in Section 6.11).

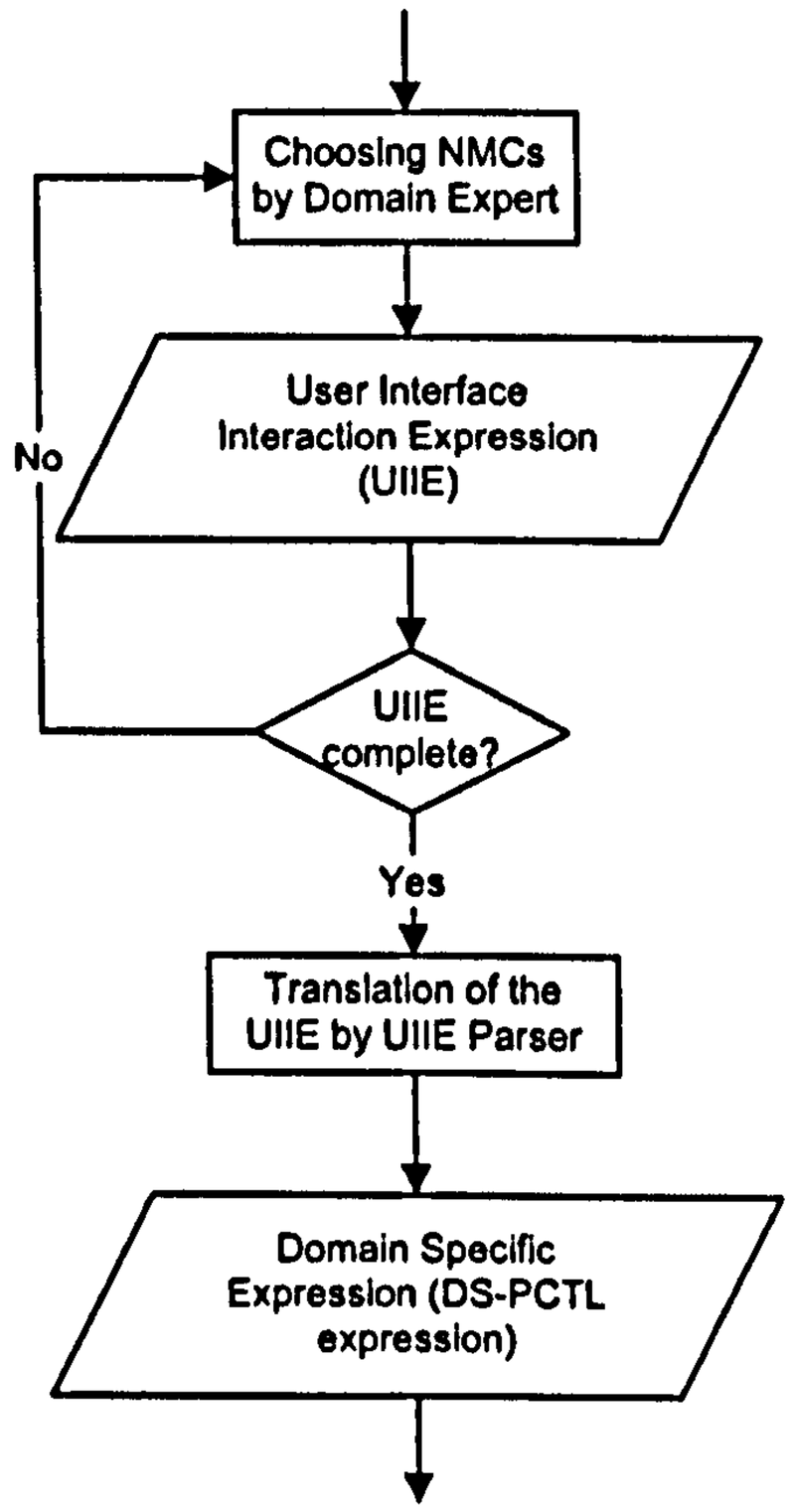


Figure 7.6: Workflow for the interaction with the DSVI

7.3.2 Transformation of a Designed System

A DS-PCTL expression is a rule that defines how the designed system should be transformed in terms of the application domain. These terms have been already mapped onto template-based composition language. Being executed, the DS-PCTL expression causes template-based transformations of the designed system. From one side the result of such transformations is a program code that is a concrete specification of a designed system. From the other side the result is a new state of the designed system, expressed in terms of the application domain. The state is held by the SkwContext described in Section 6.4.

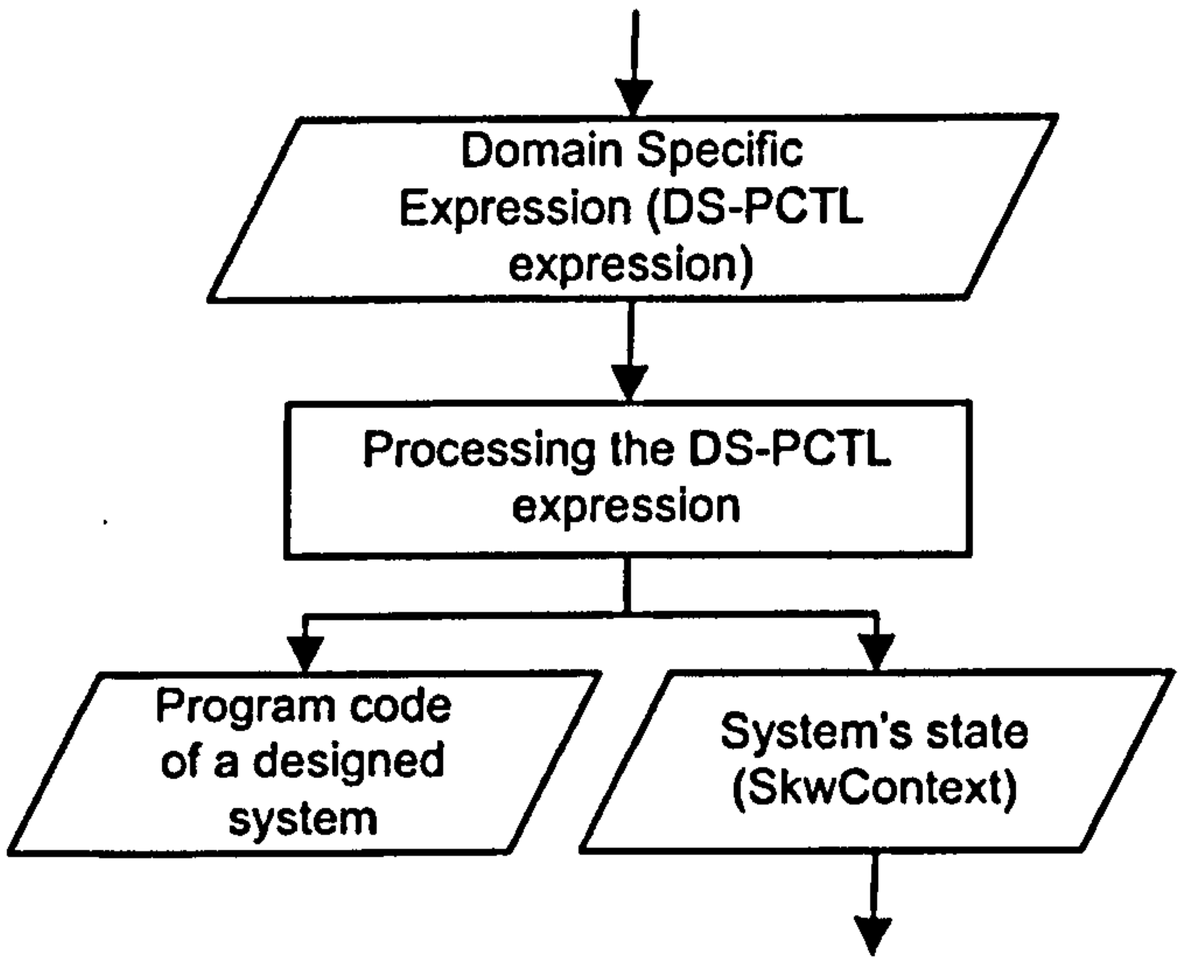


Figure 7.7: Workflow for the transformation of a designed system

Figure 7.7 depicts a workflow reflecting the process of transformation of a designed system.

7.3.3 Reflection of a System's State

System's state is described by SkwContext. The SkwContext defines a graph-like data structure to hold domain ontology. The SkwContext is dynamically changed with the designed system. The changed state should be visually reflected so that domain experts can get the information about new changes and can use this information for further design activities. Figure 7.8 shows a workflow for the process of a system's state reflection.

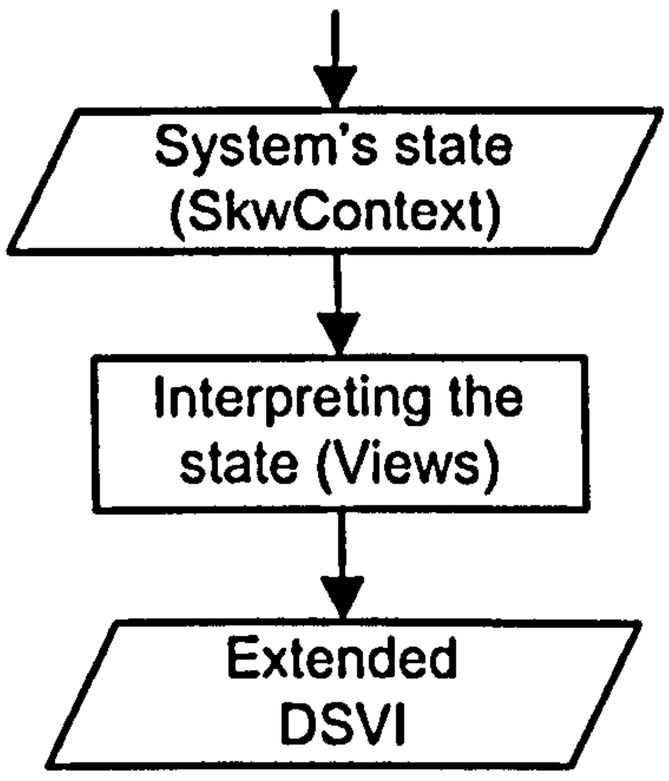


Figure 7.8: Workflow for the state reflection of a designed system

The SkwContext is interpreted by Views. Views result in an extended DSVI as they may generate DSVI parts (new NMCs) for the elements newly appeared in the designed system.

7.4 Domain-specific Visual Interface

The Domain-specific Visual Interface is an interface to the DS-PCTL (Figure 7.9). Domain experts use DSVI to form and execute domain-specific expressions (DS-PCTL expressions) and to obtain information about the state of the designed system. This information is presented in form of a visual domain-specific model.

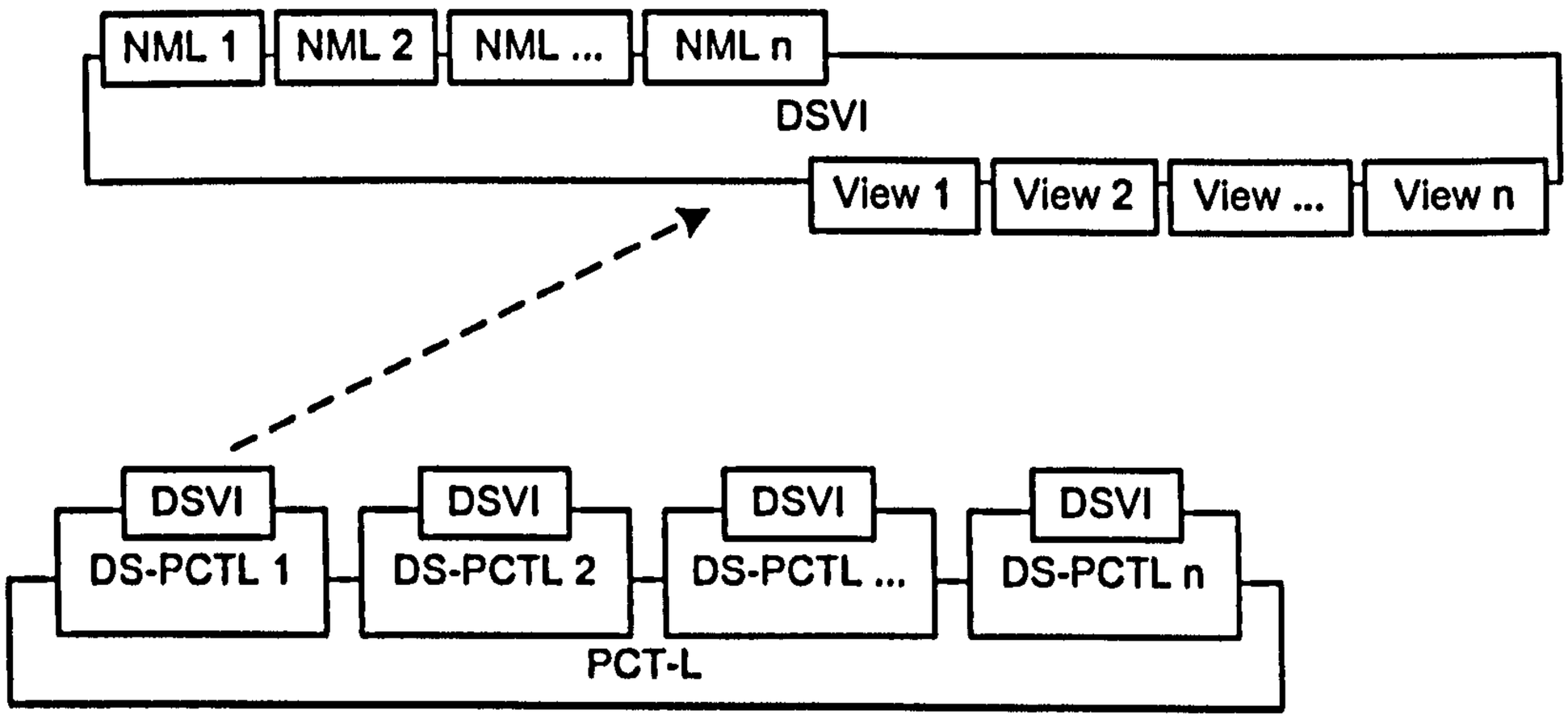


Figure 7.9: Role and Place of the Domain-specific Visual Interface

DSVI is defined with two main kinds of elements. These are Neurath Modelling Languages and Views. A NML is a repository of visual components, which represent building blocks a software system may be designed with. A View is a facility to interpret state of the designed system into DSVI. A product of a View is an extended DSVI. A DSVI may contain several NMLs and Views for the same DS-PCTL. Different NMLs and Views attached to the same DS-PCTL simplify development for various experts for different sub-domains of the main application domain.

7.5 Neurath Modelling Language

The Neurath Modelling Language is needed to build domain-specific expressions at design phase. From the perspective of a domain expert, elements of the NML are graphical symbols that represent DSCs and DSOs. These symbols are Neurath Modelling Components. During design phase NMCs form a domain-specific visual interface. Figure 7.10 shows a domain-specific visual interface reflecting a state of the designed system. The interface is formed with several NMCs.

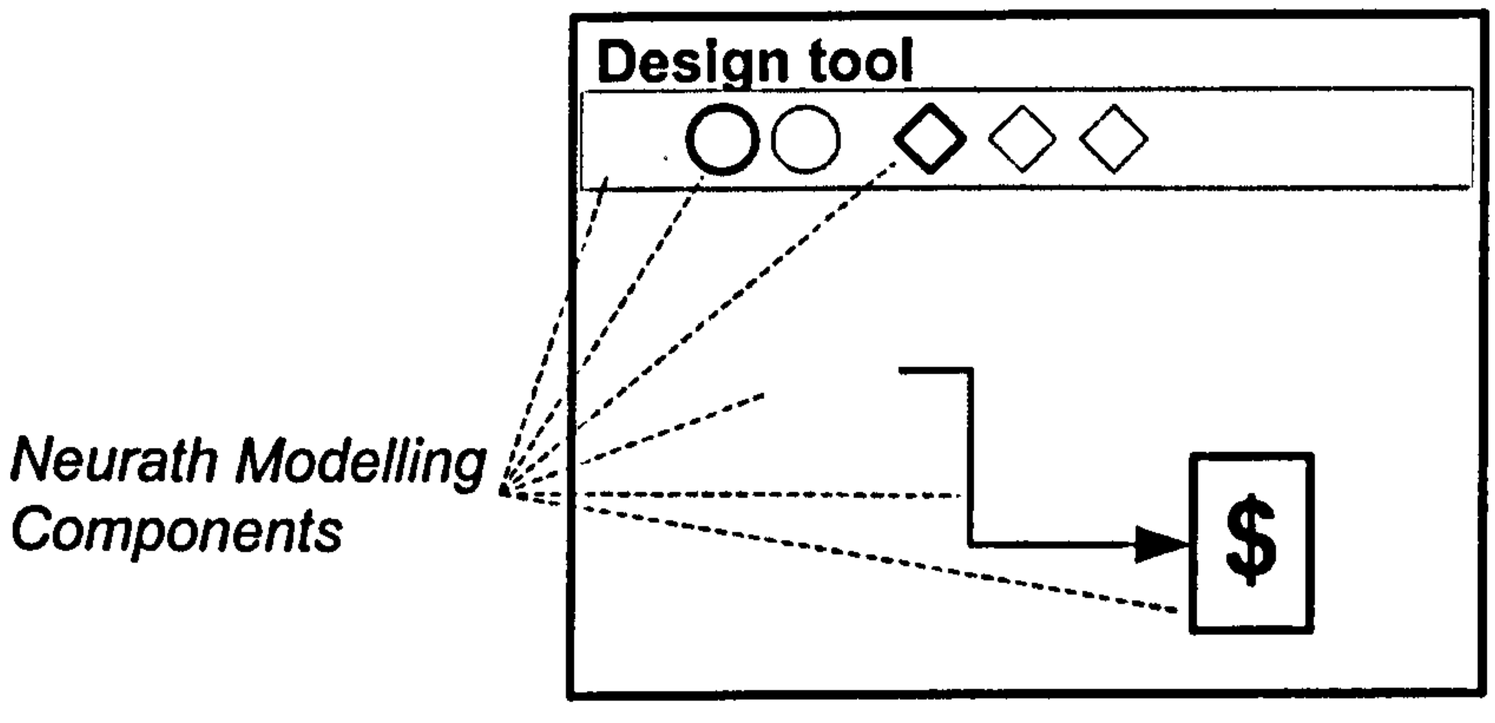


Figure 7.10: Neurath Modelling Components forming domain-specific visual interface

The NMCs at the top represent initial domain-specific elements the system is developed with. Below at the modelling pane there are other NMCs which are results of applying of top-located elements.

Basically, the NML is characterised by the *name* and the *language elements*. The name is an identifier for the NML and practically denotes a name of the domain the NML is used for. Language elements are NMCs that the NML defines. Figure 7.11 shows this.

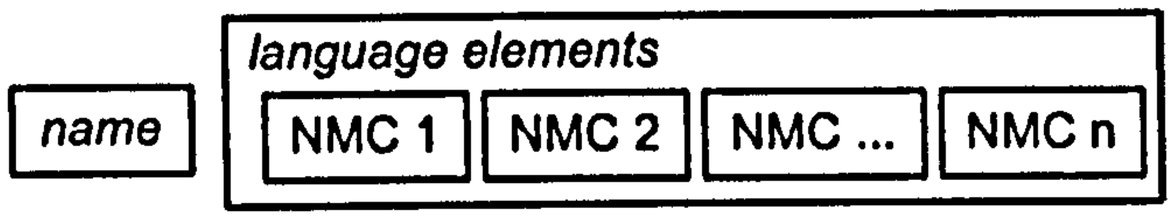


Figure 7.11: Neurath Modelling Language

Figure 7.12 shows an architecture of a NMC.

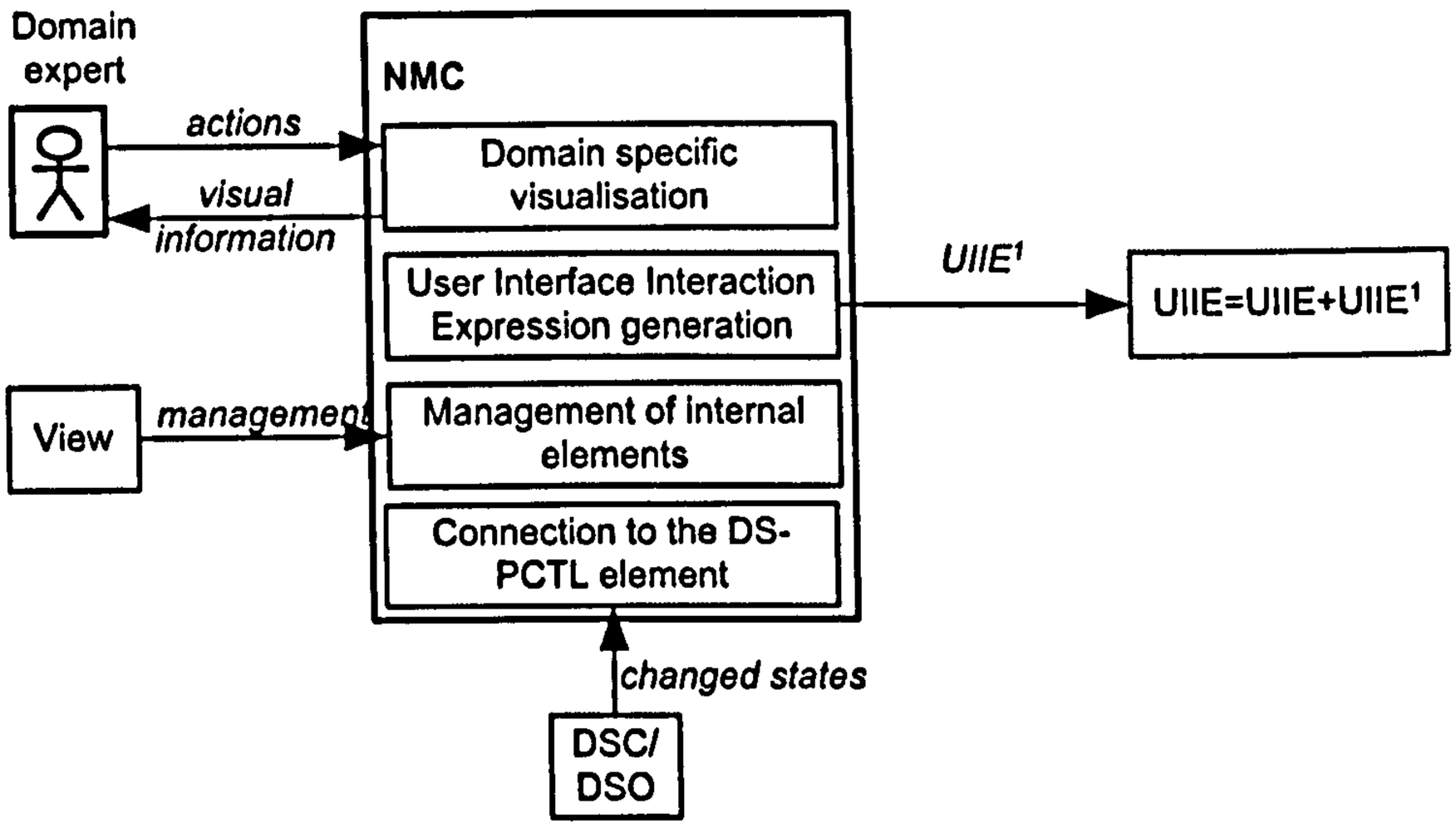


Figure 7.12: An architecture of a NMC

7.5.1 Architecture of Neurath Modelling Components

A NMC consists of domain-specific visualisation, management of internal elements, expression generation routines and connection to the DS-PCTL element (DSC or DSO). Domain-specific visualisation is needed for two tasks:

1. Domain-specifically visually reflect the state of a designed system
2. Provide mechanisms to consume and translates user's actions into domain-specific composition expressions

The mechanisms of managing the internal elements of a NMC assume *layout*, *linkage* and *containment* of these internal elements. The layout, linkage and containment management work with dependencies between the state of a designed system from one side and

the location, graphical linkage relationships and container-containee relationships within NMCs from ther other side. It is the View which initiates the management routines.

User's actions are consumed and translated by a NMC. The result of the translation is a UIIE. This expression describes actions done by domain experts in such terms as `INSTANTIATE`, `COMPONENT INSTANCE`, `COMPONENT TYPE` and `OPERATION`. Basically, the expressions defines what is instantiated and in which sequence. The expressions sequentially coming from different NMCs are collected in the main UIIE.

A NMC encapsulates elements of DS-PCTL (DSC or DSO). This connection is used for two main purposes:

1. Event-based state reflection. As soon as an encapsulated domain-specific component changes its state, the notification message will be sent to the NMC so that with help of the management mechanisms the NMC graphical interface is updated.

2. Generation of domain-specific expressions. At the Target Domain Level the UIIE expression is translated in to the domain-specific expressions specified with the DS-PCTL. During the translation the relationship between NMC and the domain-specific component is used.

7.5.2 Design of Neurath Modelling Components

NMCs are designed quite freely, however according to some minor rules. Figure 7.13 shows a UML class diagram showing some aspects of design. The class `NMC` denotes some NMC. For each NMC the name of the class will be different.

The NMC derives from some GUI components of some GUI library. The NMC is responsible for organisation of its internal structure with these elemens. Additionally it is responsible for listening and processing of the incoming interactions, i.e. mouse click over the NMC. By default, to keep it simple, we define in this thesis only the "mouse click" event. The program code for processing the event includes such routine as generation of the UIIE. This routine however basically is every time the same and can be generated automatically.

The NMC carries a `ViewNode`. This node is part of the tree-like structure, called *View Model*. A View model is a tree-like structure that describes the organisation of the visual interface abstracting it from visualisation library-specific details. A View Model is described later in Section 7.7.2.1. The NMC have to implement an interface

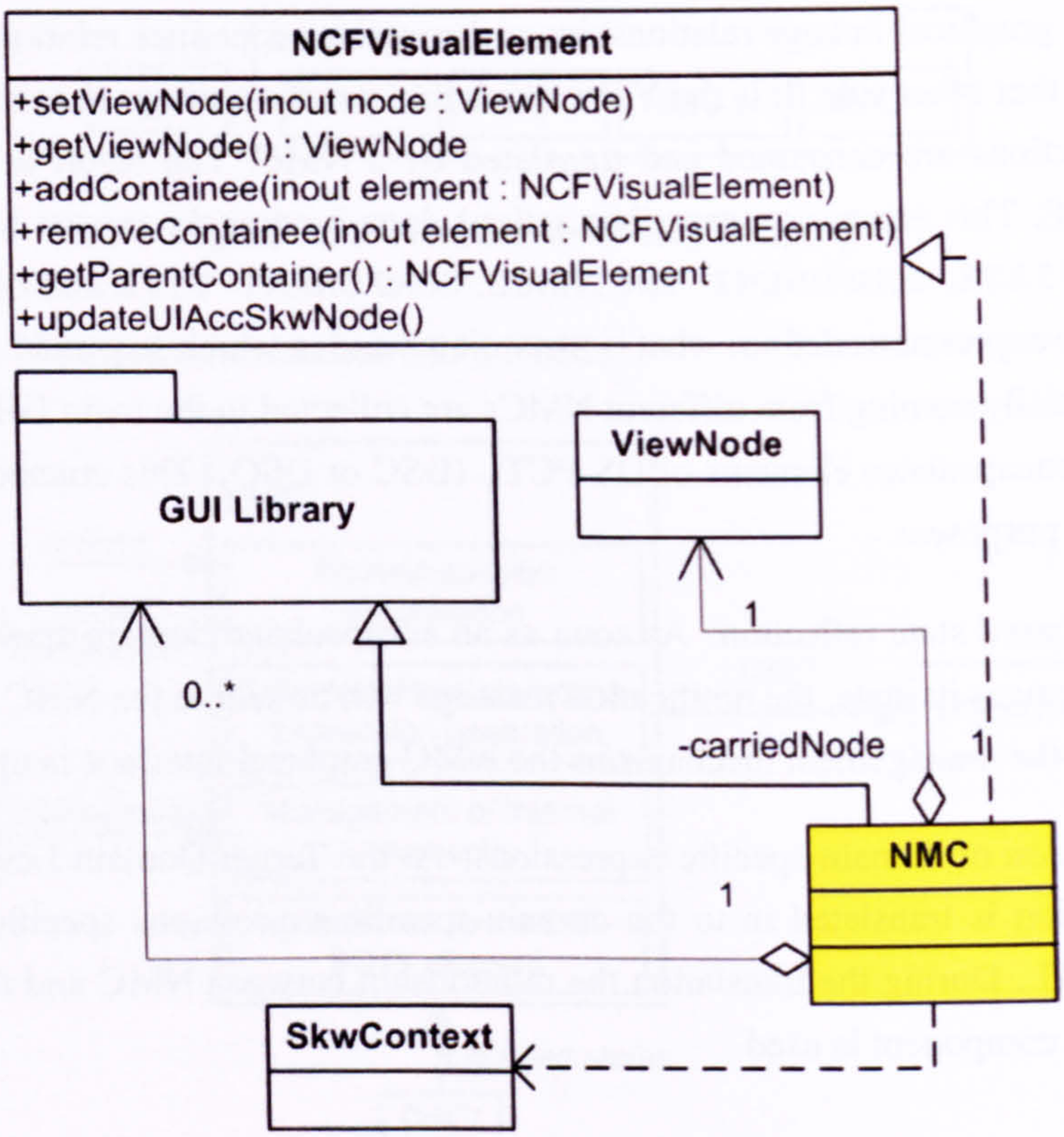


Figure 7.13: A UML class diagram for a Neurath Modelling Component

`NCFVisualElement` which represents the NCF skeleton. It is required to be present in each NMC. Table 7.1 explains methods defined by this interface.

Methods	Description
`setViewNode()`	Relates a `ViewNode` instance with the `NMC`.
`getViewNode()`	Returns related `ViewNode` instance
`addContainee()`	Adds the defined `element` into the GUI containment hierarchy of the `NMC`.
`removeContainee()`	Removes the defined `element` from the GUI containment hierarchy of the `NMC`
`getParentContainer()`	Returns a reference to parent container which is NMC.

Methods	Description
`updateUIAccSkwNode()`	Updates the GUI state according to the state of the domain-specific term which this NMC represents. The domain-specific terms are accessed via the `SkwContext`

Table 7.1: Methods of the interface `NCFVisualElement`

Figure 7.14 shows a sample specification of the NMC called `ConsoleViewerGUI`. The colours mark different aspects of the implementation that have been discussed above.

NMC-SPECIFICATION OF `ConsoleViewerGUI`

```
public class  ConsoleViewerGUI  extends MoveableContainer
implements NCFVisualElement   {
  protected ViewNode carriedNode = null;
    protected JLabel isotype = null;
    protected JLabel className = null;
    protected JTextArea text = null;
    ...
  public ConsoleViewerGUI(  ViewNode node ) {
      super();
      setViewNode(node);
      addMouseListener(this);
      ...
  }
  public void mouseReleased(MouseEvent e){
      ...
  }
  public void updateUIAccSkwNode(){
   text.setText(getViewNode().getSkwNode().
            getAttributeValue("Text").toString());
  }
  ...
  public void addContainee(NCFVisualElement element){   }
  public void removeContainee(NCFVisualElement element){   }
  public NCFVisualElement getParentContainer(){
     return (NCFVisualElement)super.getParent();
    }
}
```

GUI-Library-specific code for appearance
GIU-Library-specific code for interaction
NCF Skeleton
Rest of code

Figure 7.14: An example of the NMC specification

7.6 User Interface Interaction Expression Language

The UIIE is a description of user's actions applied to NMCs. Domain experts (users) sequentially choose different NMCs, thus specifying what transformations of a designed system have to be performed. The UIIE answers the following questions:

1. What actions the designer has performed?

2. What are identifiers of components these actions are applied to?

3. What is the sequence of designer's actions?

The sequence of actions is collected and presented by the main UIIE expression. Figure 7.15 shows how main UIIE expression is formed.

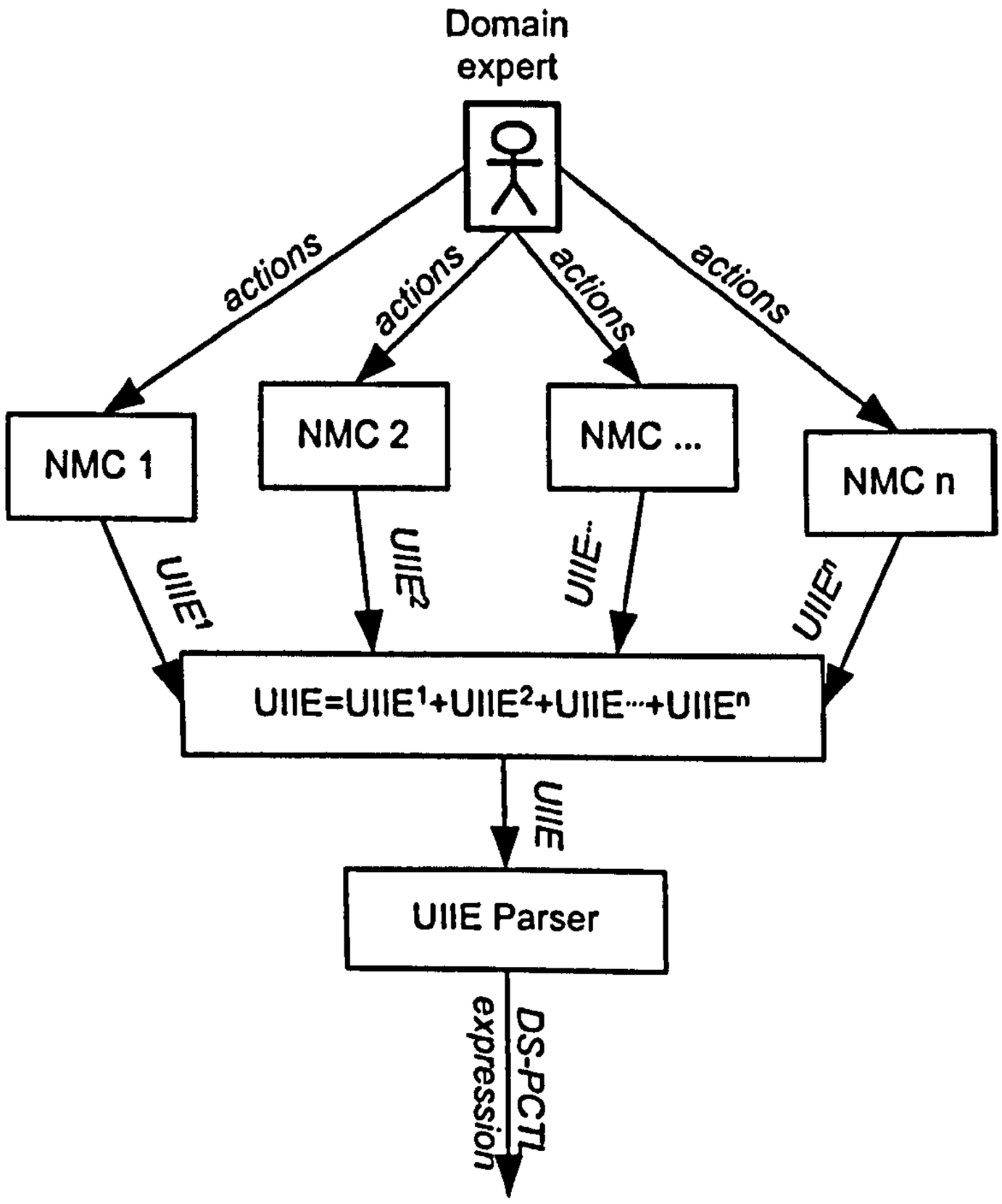


Figure 7.15: Formation of UIIE expressions

The correcteness and completeness of UIIEs are checked by the UIIE Parser. The UIIE Parser translates UIIEs into DS-PCTL expressions.

Lets consider the following example of actions performed by a domain experts and their interpretation by given NMCs. Assume, there are two NMCs have been already defined:

1. A NMC `Device` that represents a DS-PCTL expression: instantiation of a type `Device` with the operation `Create_DSOperation`.

2. A NMC `inst1` that represents a DS-PCT instance of the type `MainEnvironment`.

Figure 7.16 shows schematically a graphical user interface of the design tool domain expert works with.

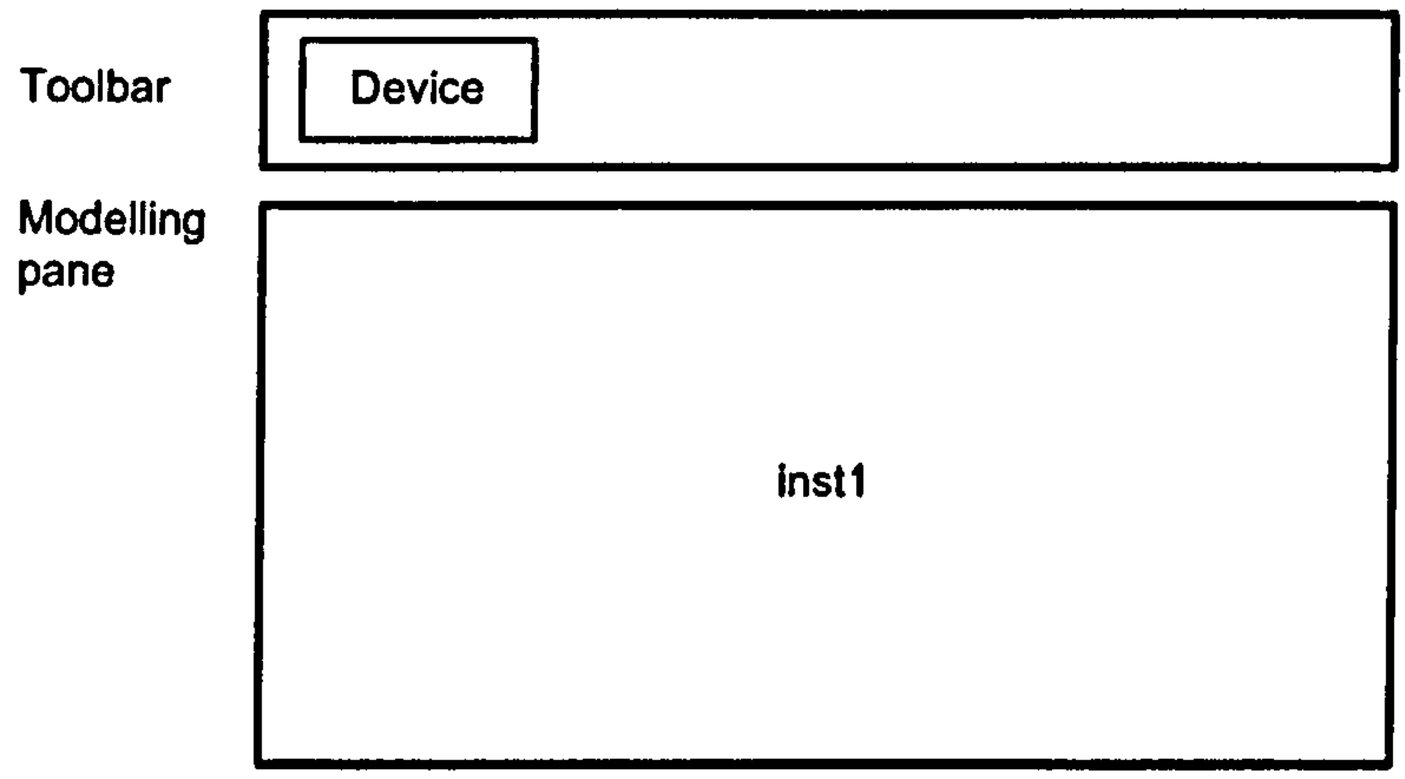


Figure 7.16: Schematically shown graphical user interface of the simple design tool

On the top the toolbar is shown. The toolbar contains initial repository of NMCs that are used to design a system. The figure illustrates the toolbar containing the `Device` NMC. Below the toolbar a modelling pane is shown. The modelling pane reflects the state of the designed system with the help of NMCs. The modelling pane contains the `inst1` NMC. The domain expert performs the following actions:

1. Clicks with the mouse over the `Device`.

2. Moves the mouse to the modelling pane area over the `inst1` NMC.

3. Clicks with the mouse over the `inst1`.

With these actions the domain experts says "Create an instance of Device and put inside the inst1 object". The UIIE in this case will be as shown in Listing 7.1.

```
INSTANTIATE COMPONENT TYPE=Device OPERATION TYPE=
   Create_DSOperation|INSTANCE inst1
```

Listing 7.1: An example of the User Interface Interaction Expression

The UIIE is created according to the language defined. We call this small language the *UIIE language*. Listing 7.2 shows a grammar of this language.

```
Start             ::= Original
Original          ::= clicked_comp_type <SEP> clicked_instance
                    | clicked_oper_type
                    ( <SEP> clicked_parameter )*
clicked_comp_type ::= component_type
clicked_oper_type ::= operation_type
component_type    ::= <INSTANTIATE> <COMPONENT>
                      typeAssign <OPERATION> typeAssign
operation_type    ::= <INSTANTIATE> <OPERATION> typeAssign
                      ( <PARAM> <EQUALS> paramAssign )*
paramAssign       ::= <ID>
typeAssign        ::= <TYPE> <EQUALS> <ID>
clicked_instance  ::= instance
instance          ::= <INSTANCE> <ID>
clicked_parameter ::= <INSTANCE> <ID>

<INSTANTIATE>     ::= "INSTANTIATE"
<COMPONENT>       ::= "COMPONENT"
<TYPE>            ::= "TYPE"
<PARAM>           ::= "PARAM"
<EQUALS>          ::= "="
<INSTANCE>        ::= "INSTANCE"
<OPERATION>       ::= "OPERATION"
<SEP>             ::= "|"
<ID>              ::= ["a"-"z","A"-"Z","_","."]
                      (["a"-"z","A"-"Z","_","0"-"9","."])*
```

Listing 7.2: BNF of the User Interface Interaction Expression Language

7.6. USER INTERFACE INTERACTION EXPRESSION LANGUAGE

Basically, two main cases of designer's actions are possible:

1. Placing an instance on the modelling pane that represents a DSC.
2. Placing an instance on the modelling pane that represents a DSO.

Further we explain both main cases of the designer's actions.

7.6.1 Instantiation of a Component

In case of instantiation of a component the designer takes the NMC that represents a domain-specific component type and then places (it can be done for example by dragging with a mouse) it into the instance on the modelling pane. For the case of `Device` component type this will cause the following expression:

```
INSTANTIATE COMPONENT TYPE=Device OPERATION TYPE=
Create|INSTANCE inst1
```

The expression precisely describes a sequence of actions and elements these actions are applied to. This expression means the following:

1. Instantiation of a `Device` component with the operation `Create`.
2. Adding the newly created instance into the instance `inst1`.

The production of the translation process is the DS-PCTL expression. Table 7.2 shows more detailed description of the translation process and Figure 7.17 shows the resulted in domain-specific expression.

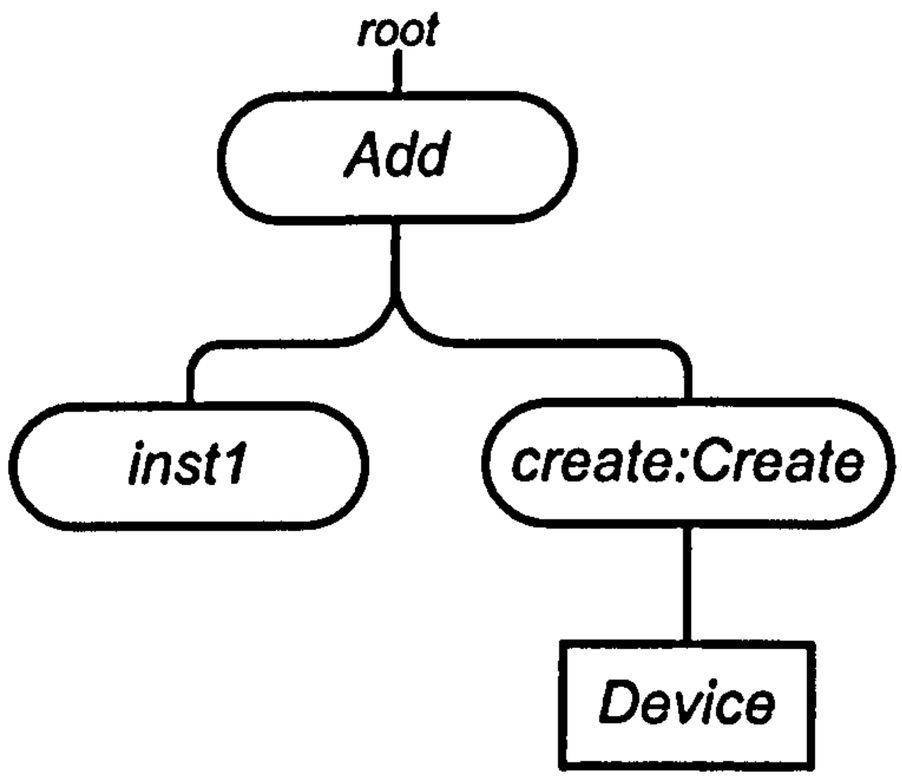


Figure 7.17: The result of UIIE translation for the case of instantiation of NMC (Domain-specific Component)

Translation Step	DS-PCTL Expression
(1) Create an instance `create` of the expression node for the DSO `Create`	$create = Create()$
(2) Set the value `"Device"` for the operand `deviceName` with the operation `setDeviceName`	$setDeviceName(create, "Device")$
(3) Create an instance `root` of the expression node for the operation `Add` and set the first operand to `inst1` and the second one to `create`	$root = Add(inst1, create)$

Table 7.2: Translation of the UIIE formed during the instantiation of NMC (Domain-specific Component)

7.6.2 Instantiation of an Operation

In case of instantiation of an operation the designer takes the NMC that represents a domain-specific operation type and then chooses NMCs that will be operands (parameters) for the chosen operation. These actions can be expresses with the following words: `"(1) instantiate a chosen operation and sequentially request parameters that meet requirements defined by the operation (2) execute the operation"`. Here is the exact UIIE expression for the case of `Delete` operation type and the operand, called `targetElement`, for which an instance named `inst1` was chosen :

```
INSTANTIATE OPERATION TYPE=Delete PARAM=targetElement|
INSTANCE inst1
```

The expression precisely describes a sequence of actions and elements these actions are applied to. This expression means the following:

1. Instantiation of a `Delete` operation.
2. Set the value `inst1` for the parameter `targetElement`.

The translation of UIIE results in the DS-PCTL expression. Table 7.3 shows more detailed description of the translation process and Figure 7.18 shows the resulted in domain-specific expression.

Translation Step	DS-PCTL Expression
(1) Create an instance `delete` of the expression node for the DSO `Delete`	$delete = Delete()$
(2) Set the value `"inst1"` for the operand `targetElement` with the operation `setTargetElement`	$setTargetElement(delete, inst1)$

Table 7.3: Translation of the UIIE formed during the instantiation of NMC (Domain-Specific Operation)

For the general case, when several operands are possible the step (2) is repeated for each operand of the operation.

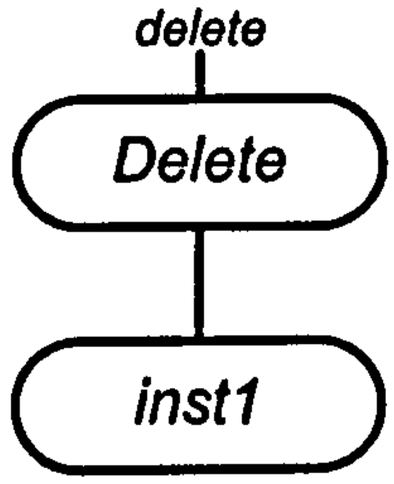


Figure 7.18: The result of UIIE translation for the case of instantiation of NMC (Domain-specific Operation)

7.6.3 Interaction Rules

7.6.3.1 Actions to UIIE

Domain experts interact with a DSVI via actions. Elements, the DSVI consists of, interpret these actions into an UIIE. The UIIE Parser checks the expression for correctness, translates the expression into a domain-specific expression (see Section 6.11) and forwards the produced domain-specific expression further. Figure 7.19 depicts a flowchart of an algorithm for the interpretation of actions into UIIE.

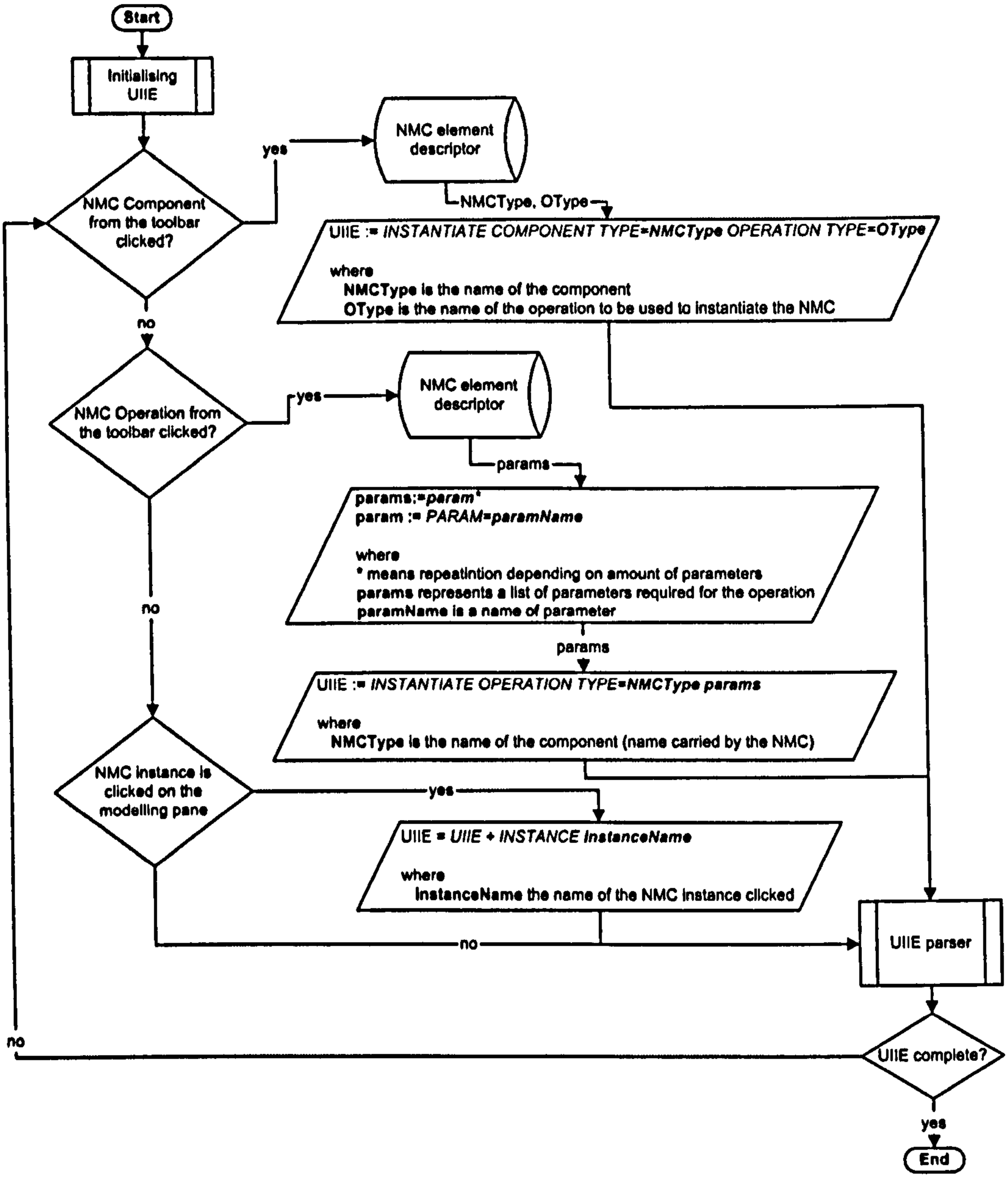


Figure 7.19: A flowchart of an algorithm for the interpretation of actions into UIIE

7.6.3.2 UIIE Parser

A UIIE that is formed during interactions with a DSVI is checked for correctness by the UIIE Parser. In Listing 7.2 the grammar according to which the UIIE is checked has been

already presented. Further, referring to this grammar, we explain the functionality of the UIIE Parser.

Stroke 1 in Listing 7.2 denotes the start point for the parser. Following the grammar rules parser checks an input UIIE and performs certain actions on the way to translate the UIIE into a DS-PCTL expression. These actions are defined further. Table 7.4 shows specifications of the actions (rules) for the strokes specified in the grammar specification. Compared to notation used in the JavaCC environment [56], the rules are written in a more simplified form. The complete specification of the syntax and semantics used by the UIIE Parser can be found in Listing E.1 in Appendix E. Table 7.5 explains rules from Table 7.4.

Stroke	Rules
0	(1)AbstractPctlExpressionNode expressionRoot=null; (2)boolean completedExpression = false; (3)Stack stack=new Stack(); (4)Stack paramStack=new Stack() (5)instanceStack=new Stack(); (6)HashMap globalValue=new HashMap();
11	(1)value=<ID>; (2)paramStack.push(value);
12	(1)value=<ID>; (2)stack.push(value);
14	(1)instanceName=<ID>; (2)String instOp=stack.pop(); (3)type=stack.pop(); (4)AbstractPctlExpressionNode instOper=null; (5)Class theClss=Class.forName(instOp); (6)instOper=theClass.newInstance(); (7)instOper.setLeaf(new StringPctlExpressionNode(type)); (8)SkwContext skw=getSkwContext(); (9)SkwNode node=skw.searchForNodeById(instanceName); (10)AbstractPctlExpressionNode parentNode=node.getDspct(); (11)Merge_DSO mergeOperation=new Merge_DSO(); (12)mergeOperation.setParent(parentNode); (13)mergeOperation.setChild(instOper); (14)expressionRoot=mergeOperation;

Stroke	Rules
	(15)completedExpression=true;

15	(1)instanceName=<ID>
	(2)instanceStack.push(instanceName);
	(3)int paramsAmount = instanceStack.size();
	(4)if (instanceStack.size()==paramStack.size()){
	(5)AbstractPctlExpressionNode operationObject=null;
	(6)String operation=stack.pop();
	(7)Class theClass=Class.forName(operation);
	(8)operationObject=theClass.newInstance();
	(9)for (int i=0; i<paramsAmount;i++){
	(10)SkwContext skwContext=getSkwContext();
	(11)String instString=instanceStack.pop();
	(12)SkwNode node=skwContext.searchForNodeById(instString);
	(13)AbstractPctlExpressionNode paramValue=node.getDspct();
	(14)HashMap setters=globalValue.get("PARAMS SETTERS");
	(15)String param=paramStack.pop();
	(16)String paramSetter=setters.get(param);
	(17)setParameterOfOperation(operationObject, paramSetter, paramValue);
	(18)}
	(19)expressionRoot=operationObject;
	(20)completedExpression=true;
	(21)}

Table 7.4: Rules of the UIIE Parser

Rule	Explanation
0 (1)	Declaration of a root node of an output DS-PCTL expression.
0 (2)	Declaration of a flag denoting the completeness of the output DS-PCTL expression.
0 (3)	Declaration of a stack.
0 (4)	Declaration of a stack where names of parameters, which are required by an operation, are stored.
0 (5)	Declaration of a stack where names of instances of NMCs are stored.
0 (6)	Declaration of a storage of variables and their values taking part in translation process.

Rule	Explanation
11 (1)	Obtaining a value carried by <ID>. This value denotes a parameter which is required for an operation.
11 (2)	Pushing the value into the stack that holds parameter names.
12 (1)	Obtaining a value of carried by <ID>.
12 (2)	Pushing the value into the common stack.
14 (1)	Obtaining a name of an instance.
14 (2)	Obtain the name of the instantiation operation previously pushed into the stack.
14 (3)	Obtain the name of the component previously pushed into the stack.
14 (4-6)	Creating an object for a specified instantiation operation. This object is a node in a DS-PCTL expression tree.
14 (7)	Setting the leaf for just created expression node. The leaf holds a name of a DSC.
14 (8)	Obtaining the SkwContext.
14 (9)	Look in the SkwContext for a SkwNode of an instance denoted by the `instanceName`.
14 (10)	Obtaining an DSC object related to the SkwNode `node`.
14 (11)	Creating an instance of the Merge_DSO.
14 (12)	Setting a `parent` parameter with the value carried by the `parentNode` variable.
14 (13)	Setting a `child` parameter with the value carried by the `instOper`.
14 (14)	Setting an object `mergeOperation` as a root of the DS-PCTL expression.
14 (15)	Setting the flag of expression completeness.
15 (1)	Obtaining an instance name.
15 (2)	Pushing the instance name into the stack.
15 (3-4)	Checking if amount of already collected instance names is equal to expected amount. If true further rules (5-21) are executed.
15 (5)	Initialisation of an object `operationObject`.
15 (6)	Obtaining previously saved operation.
15 (7-8)	Creating an object denoting the obtained operation. This object represents a node in the DS-PCTL expression tree.
15 (9-18)	Steps are repeated for each instance collected in the stack.
15 (10)	Obtaining the SkwContext.
15 (11)	Obtaining the next name of an instance.
15 (12)	Look in the SkwContext for a SkwNode of an instance denoted by the `instanceString`.
15 (13)	Obtaining a DSC object related to the SkwNode `node`. This object is held `paramValue` variable.

Rule	Explanation
15 (14)	Requesting all names of setter methods. These methods are for assigning a parameter of an operation with a value.
15 (15)	Obtaining next parameter name for an operation.
15 (16)	Requesting a setter method for the obtained parameter name.
15 (17)	Setting a value to a parameter of an operation.
15 (18)	If it is not the last parameter then go to stroke 10.
15 (19)	An object that denotes an operation, parameters of which were just assigned, is set as root of the DS-PCTL expression.
15 (20)	Setting the flag of expression completeness.

Table 7.5: Explanation of rules according to which UIIE Parser works

7.7 Views

Views interpret states of a designed software system. A state of a designed system is described with the domain ontology carried by SkwContext concept, defined at the Target Domain Level of composition and described in Section 6.4. The interpretation of states by a Views results in a modification of DSVI presented by NMCs. These NMCs are interfaces to the variables and values that describe the state. Figure 7.20 shows the workflow at design phase with respect to functionality of a View.

Before explaining the architecture of Views let's look how the View is notified about any change in system's state.

7.7.1 A Change of State

A View begins to work with a change of a designed system's state. Since this state is described with a graph-like structure a message describing a change of the state can be seen as a sequence of the following:

1. A relation between two nodes (terms) is added (or deleted).

2. A node added (or deleted).

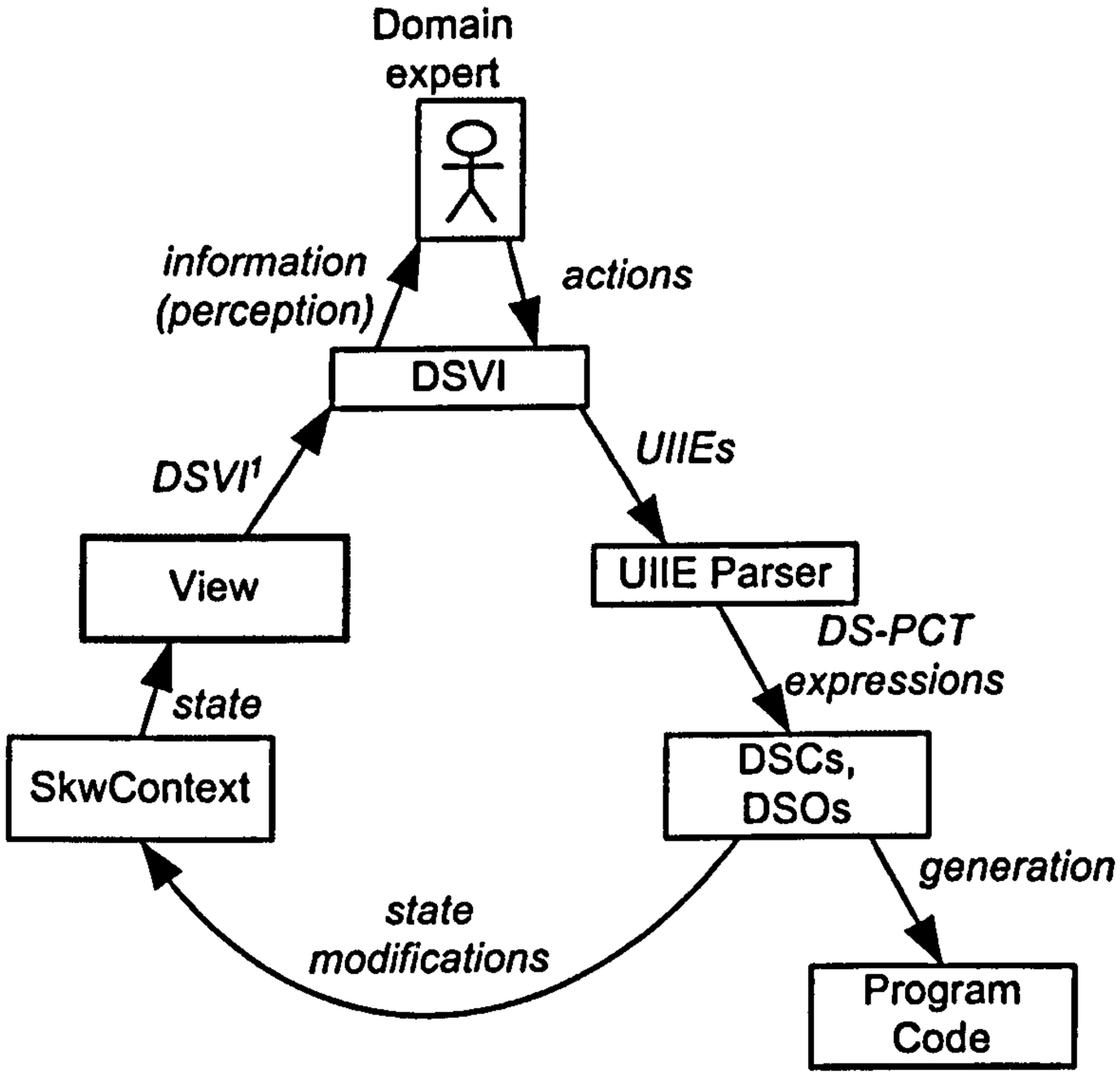


Figure 7.20: Role of Views in the common workflow for design phase

Lets consider the following example. There is a domain defined with the following terms and possible relationships:

- The term `Sensor` is defined that represents a virtual sensor device. It may be instantiated with the operation `CreateSensor`.
- The term `Actuator` is defined that represents a virtual actuator device. It may be instantiated with the operation `CreateActuator`.
- One sensor can be connected to several actuators with the operation `Connect`.

Initially the state is characterised by no elements. We are going to create a sensor, an actuator and connected them. Figure 7.21 shows the domain-specific expression (a) and the changed state of the designed system (b) after the expression is processed.

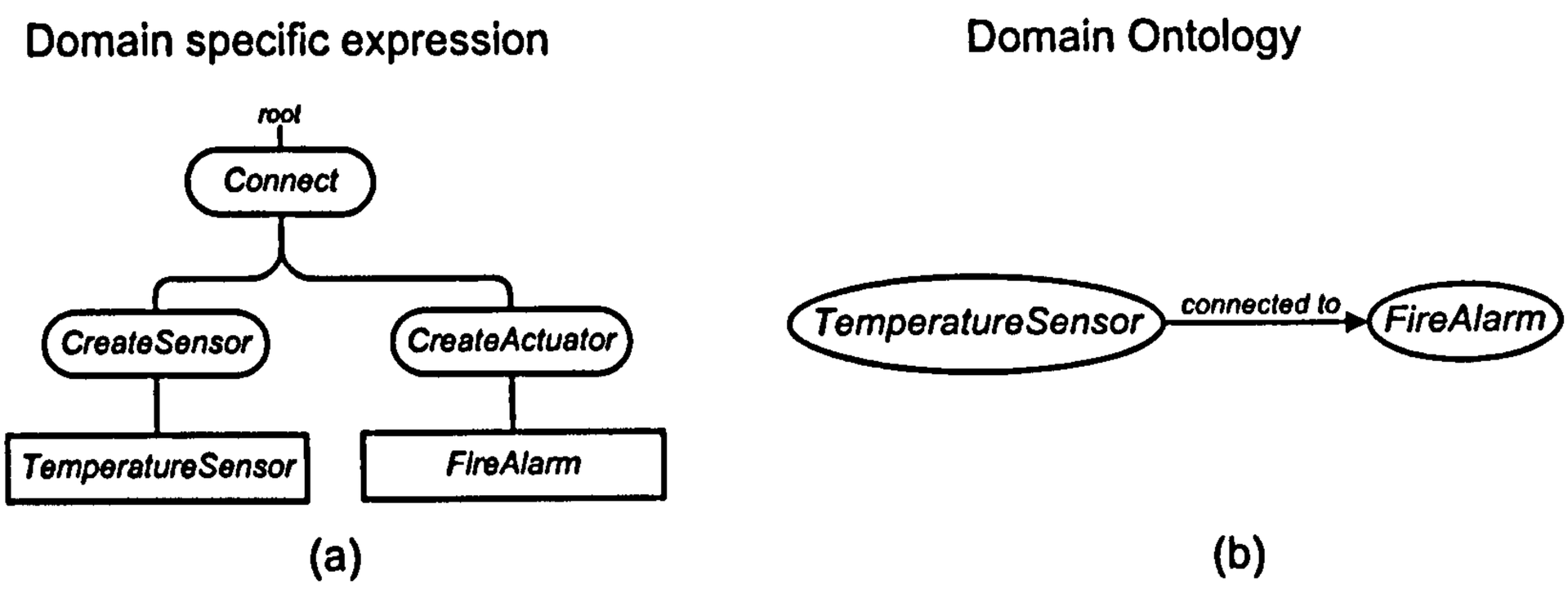


Figure 7.21: (a) The domain-specific expression for the Sensor-Actuator example (b) Resulted in state of the designed system

The notifications about changes made in the state of the designed system are sent to the View. For the example, these messages are the following:

- The node `Temperature Sensor` is created.
- The node `FireAlarm` is created.
- The relation `connected to` is created between the nodes `Temperature Sensor` and `FireAlarm`.

The messages initiate DSVI reconfiguration routines within the View.

7.7.2 Generation of Domain-specific Visual Interface

The View consists of the state interpretation mechanisms. Figure 7.22 illustrates an architecture of a View.

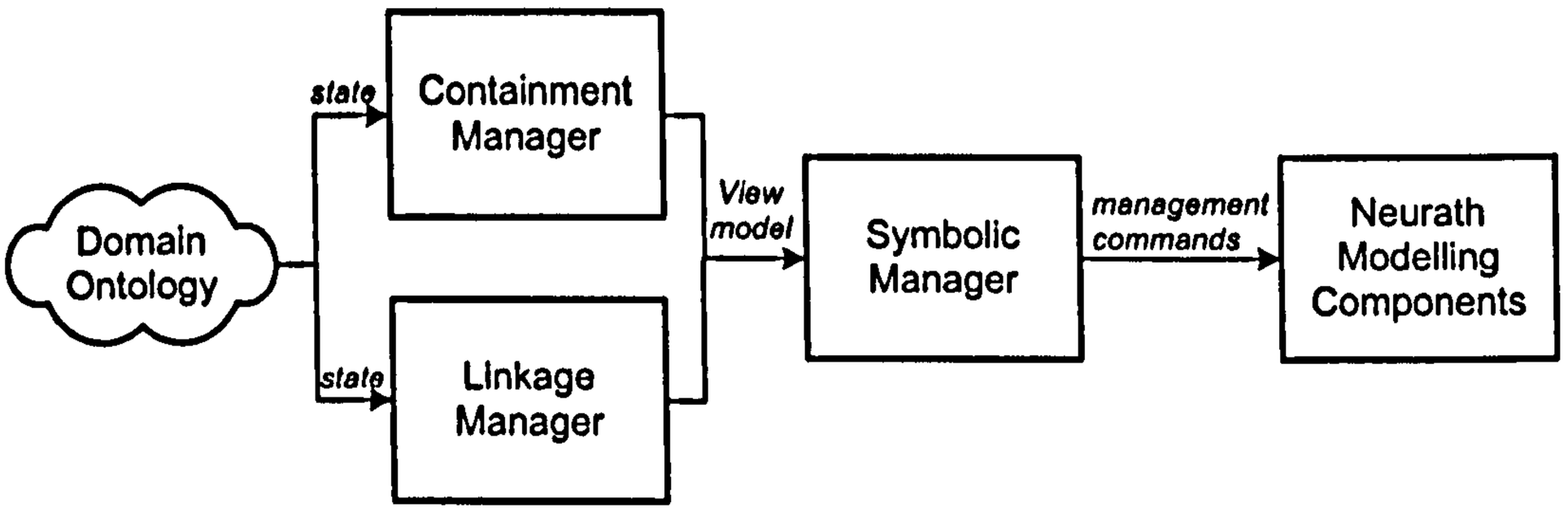


Figure 7.22: An architecture of a View

The following functional blocks are defined:

1. Containment Manager: interprets a state into a description of containment hierarchy of domain-specific terms that form the state. The containment hierarchy of terms shows what terms are nested.

2. Linkage Manager: interprets a state into a description of linkage defined between domain-specific terms that form the state. The linkage between terms shows what terms are related (no nesting).

3. Symbolic Manager: interprets domain-specific terms and relations into domain-specific visual symbols.

Basically, these three managers answer the following questions:

1. What visual symbol is defined for each of the domain-specific terms and relations defined between these terms?

2. What visual symbols are nested?

3. What visual symbols are graphically linked?

Figure 7.22 shows a state described by a domain ontology as an input to the Containment Manager and the Linkage Manager. Both produce a description called *View model.* A View model is a tree-like structure that describes the organisation of the visual interface abstracting it from visualisation library-specific details.

A View model is an input into the Symbolic Manager. This functional block interprets the View model and generates commands to visual components represented by Neurath Modelling Components. Practically the Symbolic Manager translated relatively abstract definition of containment hierarchy of components and linkage between components into a visualisation library specific terms. The commands generated by the Symbolic Manager can be, for example, an instantiation of visual component and placing one visual component inside the other one.

7.7.2.1 View Model

A View model describes how the domain ontology that represents a state of a designed system should be visualised. Figure 7.23 describes main concepts that form a View model.

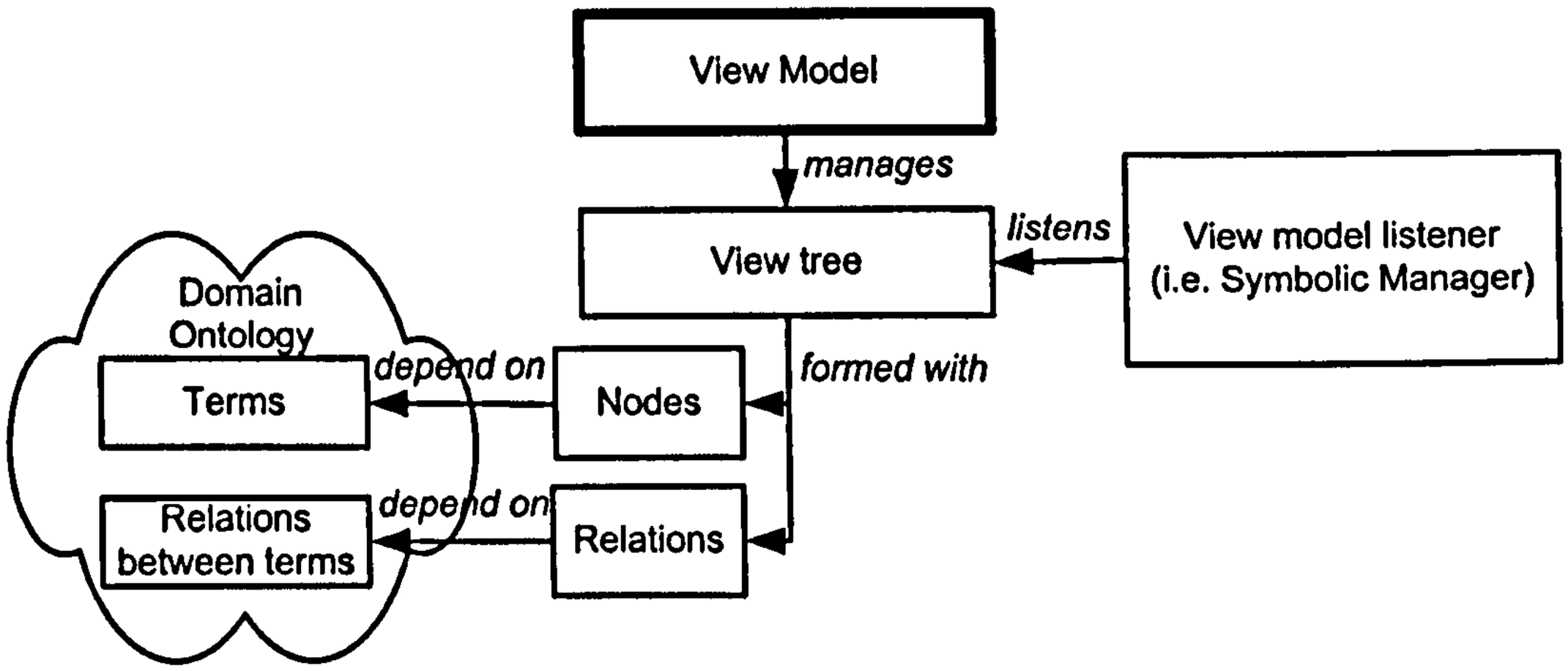


Figure 7.23: A View model and related concepts

The figure shows that the View model manages a View tree. It is a tree that has nodes and relations which represent the terms and relations between terms defined by the domain ontology. View model manages a View tree by changing a tree structure, i.e. adding new nodes, deleting existing ones and specification of new relationships between parent and child nodes. These changes are subjects of an observation. The observation means that every time a change happens, all registered listeners will be immediately informed. The main listener is a Symbolic Manager.

Two kinds of relationships between a parent node and a child node are possible:

1. The "contains" relationship states that a graphical symbol that represents the parent should contain inside a graphical symbol that represents the child.

2. The "links relationship states that a graphical symbol that represents the parent should be graphically linked to a graphical symbol that represents the child.

Figure 7.24 shows a UML class diagram describing View model and related concepts.

∧ View model is represented by the class `ViewModel`.

This class is needed for managing a tree structure of a View model. The tree structure is represented by its root node. The class `ViewModel` defines methods to access the root:

- The method `setRoot()` sets a new root instead of the previously defined one if any.

- The method `getRoot()` returns a reference to the defined root of a View model.

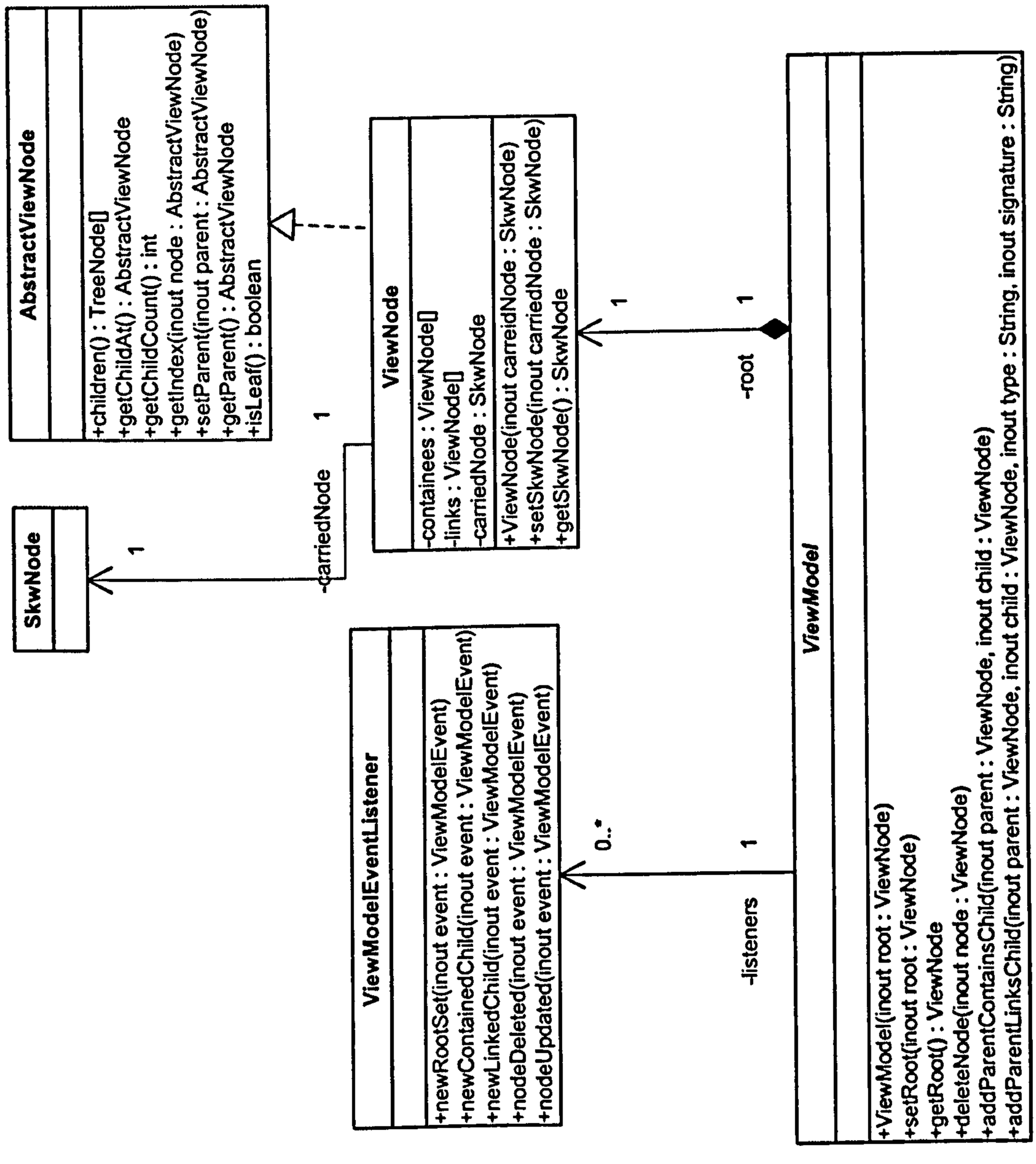


Figure 7.24: A UML class diagram describing a View model

Moreover, the class `ViewModel` defines management mechanisms to operate with nodes within a tree:

- Deletion of a node is defined with the method `deleteNode()`.
- Establishing of a "contains" relationship between the parent node and the child node is defined with the method `addParentContainsRelationship()`.
- Establishing of a "links" relationship between the parent node and the child node is defined with the method `addParentLinkChildRelationship()`.

The management of a tree results in change in the tree structure. Every time a change happens the event is fired to the registered listeners. Looking back at Figure 7.22, such a listener is a Symbolic Manager. The listeners are defined by the interface `ViewModelEventListener`. Each method in this interface is defined for a change kind in the tree structure:

- When the new root is set the method `newRootSet()` is called.
- When the "contains" relation between a parent and a child is established the method *newContainedChild()* is called.
- When the "links" relation between a parent and a child is established the method `newLinkedChild()` is called.
- When a tree node is deleted the method `nodeDeleted()` is called.
- When a domain-specific term carried by a tree node is changed the method `nodeUpdated()` is called.

Figure 7.24 describes the class `ViewNode`. It implements the behaviour of a tree defined by the class `AbstractViewNode`. Each node in a tree is connected to a domain-specific term defined by an instance of the class `SkwNode`. We have described this class in Section 6.4.1. This instance is later carried by visual components that reflect the designed system's state.

7.7.2.2 Containment Manager

The main role of the Containment Manager is to translate domain ontology that describes a state of a designed system into a View model (see Fig. 7.22). The Containment Manager build "contains" relationships between nodes within the View tree.

With that kind of relationship a containment hierarchy of domain-specific terms may be described. An example of a containment hierarchy is shown on Figure 7.25.

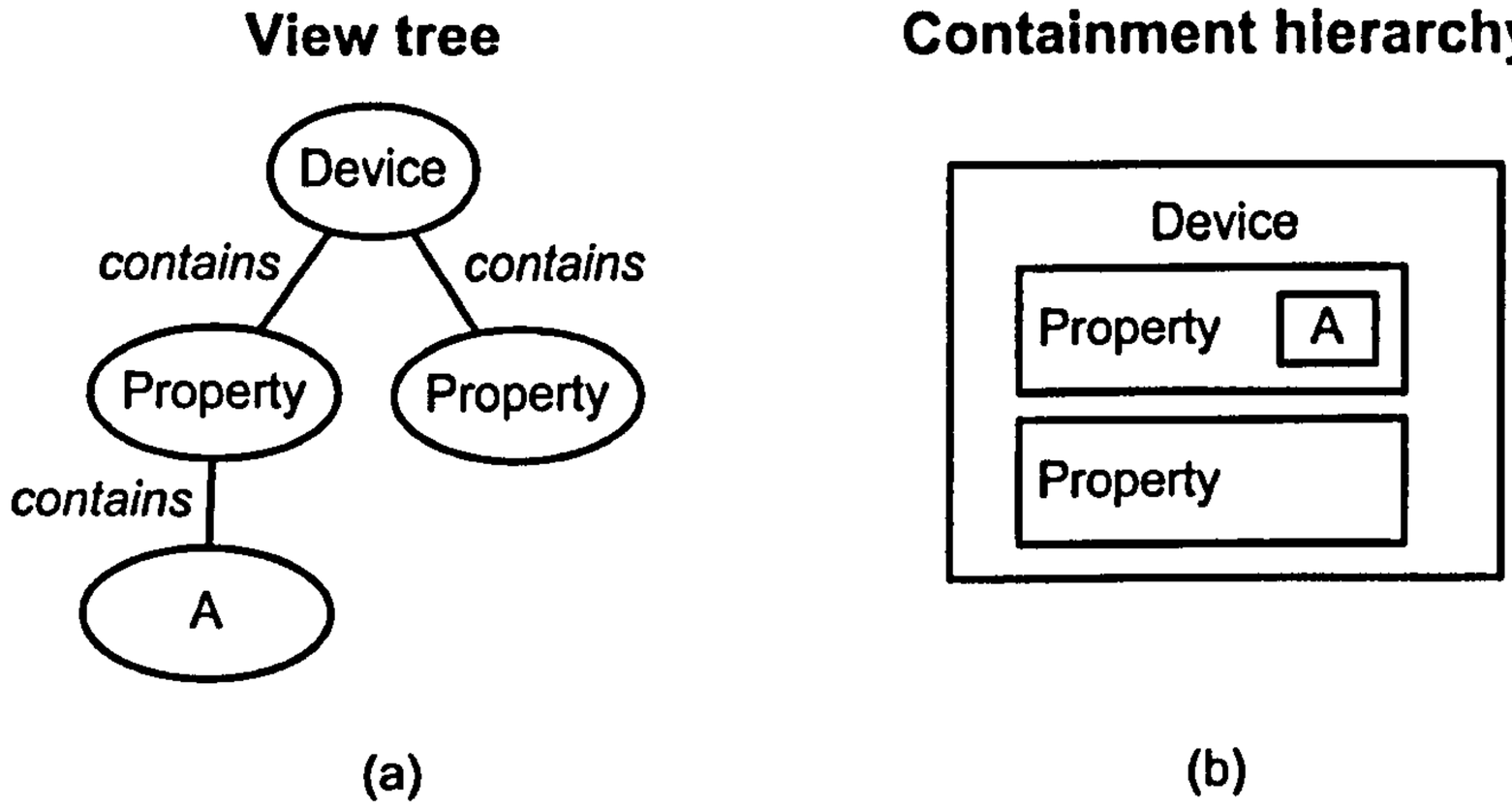


Figure 7.25: An example of a containment hierarchy (b) for a View tree (a)

It is up to the Domain Expert to decide how the relationship between domain-specific terms should be represented by means of the containment hierarchy. A Software Architect implements decisions made by the Domain Expert. Figure 7.26 shows a UML class diagram describing a Containment Manager.

The figure shows a class `ContainmentManager` that represent a Containment Manager for the View. This class is an abstract one. Concrete Containment Managers are represented by the numbered classes `ContainmentManager 1`, `ContainmentManager 2` and so on. They extend the abstract class `ContainerManager`. The diagram specifies the following requirements to Containment Managers:

1. They are listeners of the events coming from the SkwContext that contains a description of a system's state. SkwContext was described in Section 6.4. This requirement is shown with the relationship from the class `ContainerManager` to the interface `SkwContextListener`.

2. They work with the View model represented by the class `ViewModel`.

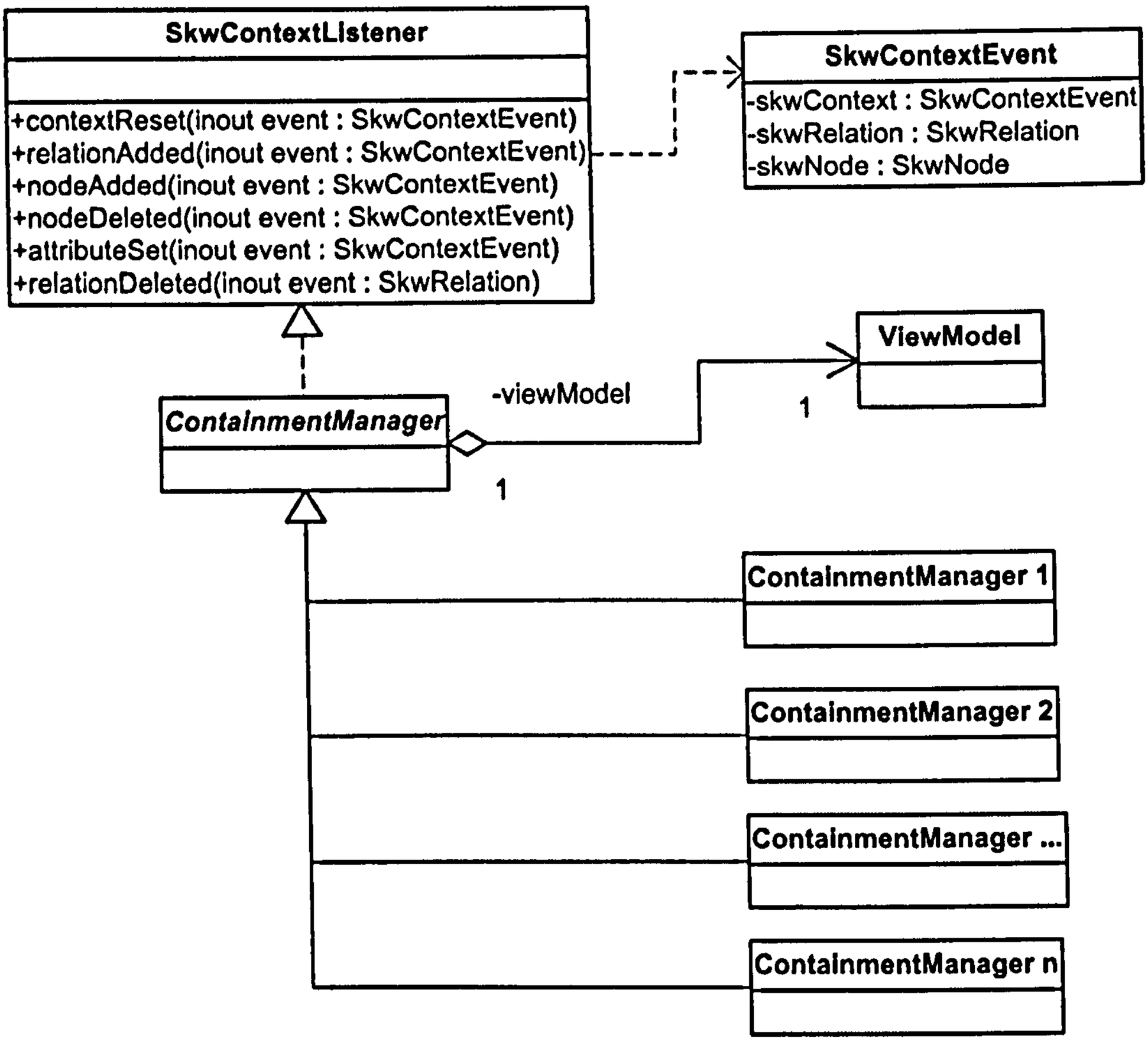


Figure 7.26: A UML class diagram describing a Containment Manager

The SkwContext generate the following events:

1. The SkwContext is reset, which means set to the initial state with no nodes and relationships. When this event occurs, the listener's method `contextReset()` is called.

2. A relation (SkwRelation) between two domain-specific terms (SkwNodes) is added. When this event occurs, the listener's method `relationAdded()` is called.

3. A domain-specific term (SkwNode) is added. When this event occurs, the listener's method `nodeAdded()` is called.

4. A domain-specific term (SkwNode) is deleted. When this event occurs, the listener's method `nodeDeleted()` is called.

5. A relation (SkwRelation) between two domain-specific terms (SkwNodes) is deleted. When this event occurs, the listener's method `relationDeleted()` is called.

The events coming from the SkwContext are described by the class `SkwContextEvent`. Depending on the event, instances of this class contain references to the SkwContext, SkwRelation or SkwNode.

A sample implementation of the Containment Manager can be found in Appendix C (Listing C.5).

7.7.2.3 Linkage Manager

The main role of the Linkage Manager is to translate a domain ontology that describes a state of a designed system into a View model (see Fig. 7.22). The Linkage Manager builds "links" relationships between nodes within the View tree.

With that kind of relationship a visual linkage between domain-specific terms may be described. An example of a visual linkage between two elements is shown on Figure 7.27.

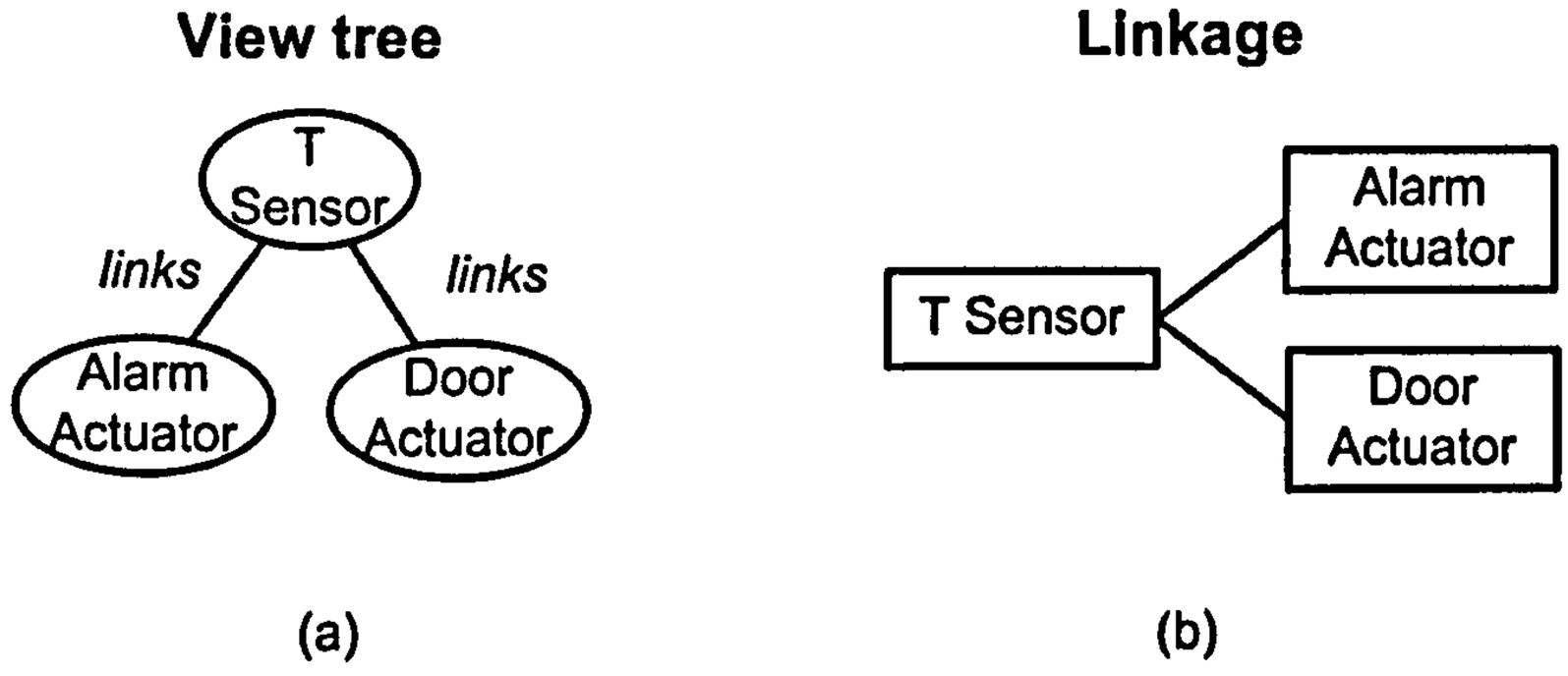


Figure 7.27: An example of a visual linkage between three elements (b) for a View tree (a)

Compared to the "contains" relationship, "links" is more complex one. A visual linkage is characterised by the following attributes:

- The *type* denotes a type of a "links" relationship. Different types will be visually reflected differently. For example, arrowed linkage and dashed linkage are different visualisations for different types.

- The *signature* denotes a text that is visually shown together with a linkage.

Figure 7.28 shows examples of possible "links" relationships with different types and signatures. For each linkage attributes *type* and *signature* are defined.

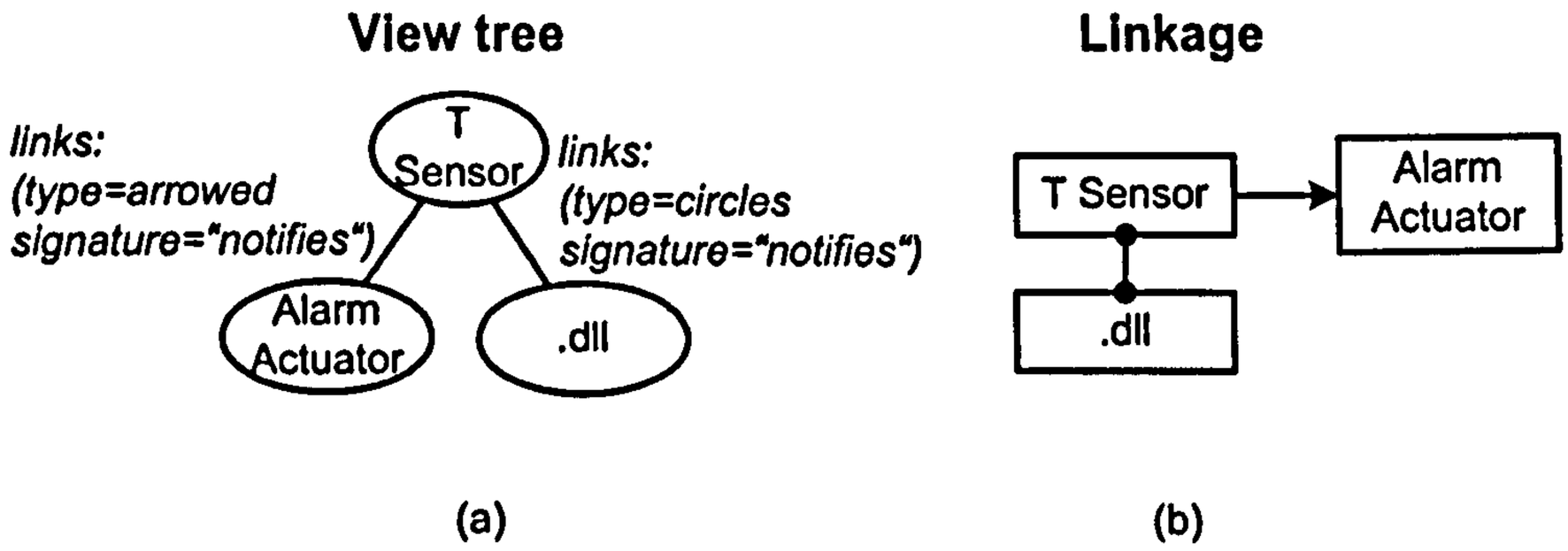


Figure 7.28: An example of a different types of visual linkages between elements (b) for a View tree (a)

It is up to the Domain Expert to decide how the relationship between domain-specific terms should be represented by means of the visual linkage. A Software Architect implements decisions made by the Domain Expert. Figure 7.29 shows a UML class diagram describing a Linkage Manager. The diagram is very similar to one for the Containment Manager shown in Figure 7.26.

Though the architecture of the Containment Manager and the Linkage Manager is the same, the main difference between them is that:

1. The Containment Manager controls "contains" relationships between nodes within a View tree and the Linkage Manager controls "links" relationships.

2. Compared to "contains" relationships, "links" relationships define two attributes that exact the identity of the visual linkage.

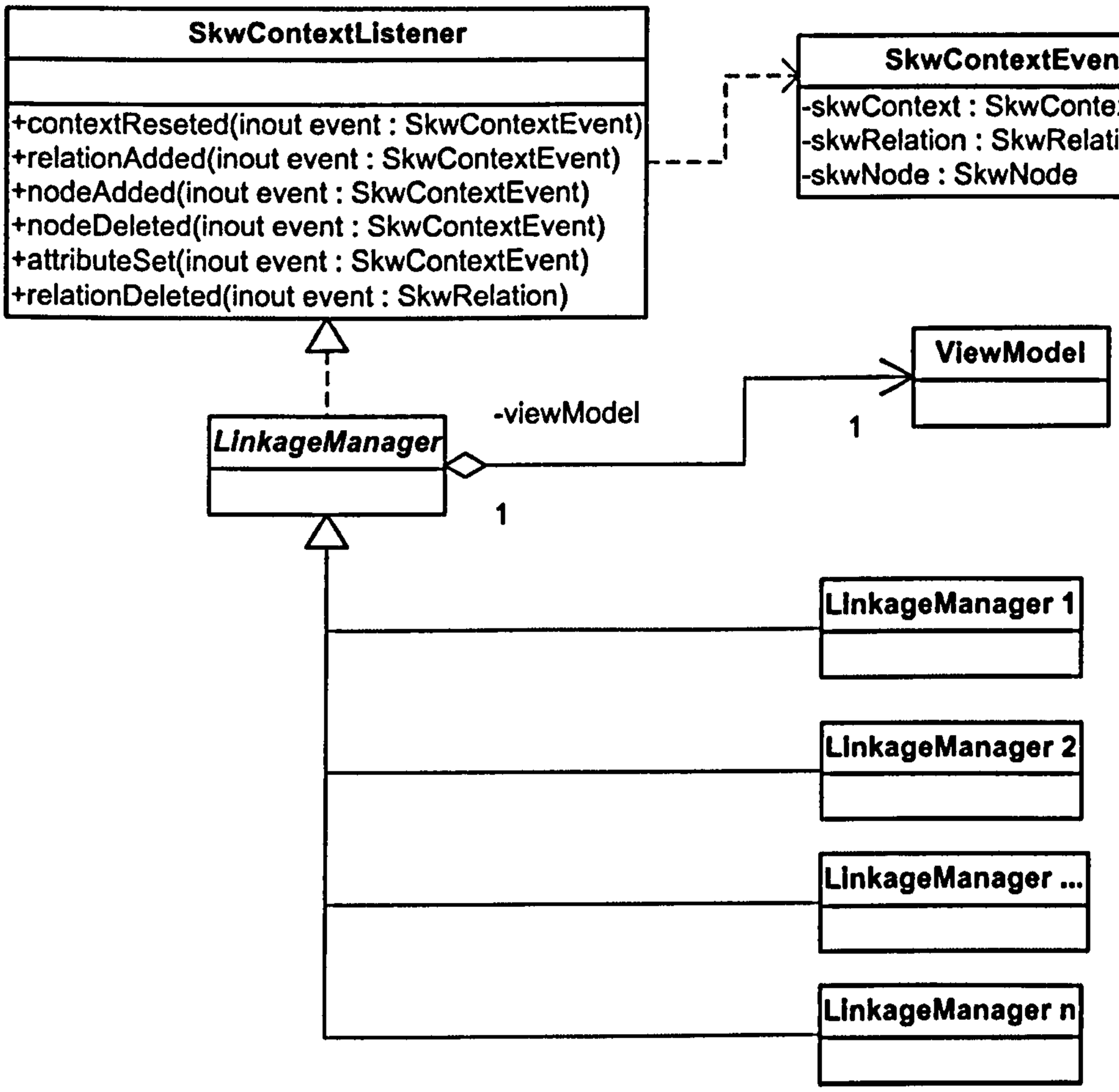


Figure 7.29: A UML class diagram describing a Linkage Manager

7.7.2.4 Symbolic Manager

The Symbolic Manager specifies how domain-specific terms and relations are visualised. These terms and relations are held by a View Model. Moreover, Symbolic Manager interprets "contains" and "links" relationships defined by a View model into terms of the Neurath Modelling Component (see Section 7.5).

When a View tree is changed, i.e. a node or relationship is added, the Symbolic Manager receives an event that carries information about the change. Figure 7.30 shows a UML class diagram that describes an architecture of a Symbolic Manager.

Symbolic Managers are presented by the class 7.30. This class is an abstract one. Concrete Symbolic Managers are represented by the numbered classes

`SymbolicManager 1`, `SymbolicManager 2` and so on. They extend the abstract class `SymbolicManager`. The diagram specifies the following requirements to Symbolic Managers:

1. They are listeners of the events coming from the View model that contains a description of a View. The View model was described in Section 7.7.2.1. This requirement is shown with the relationship from the class `SymbolicManager` to the interface `ViewModeEventListener`.

2. They work with the library of NMCs that are elements of the Neurath Modelling Language defined to design a software system for the particular application domain. The NMCs defined on top of some library of visual components which often a third-party library.

3. They define the visual component which is the root container in the modelling pane. The choice of this visual component depends on specific library of visual components taken. It is the method `setLibrarySpecific_MainContainer()` that have to be implemented by derived Symbolic Managers to set the visualisation library specific root container.

An example of the Symbolic Manager implementation can be found in Appendix C (Listing C.6). Implementations of Symbolic Managers are quite similar. Normally, the following aspects differ:

1. Library-specific container that depends on the GUI library taken to generate the domain-specific visual interface at design time. This is the container GUI component that plays role of modelling pane.

2. NMC components for different domain-specific terms. For different composition systems (or even for different Views within one composition system) different terms will be represented differently.

We defined a SM-Specification form to describe the functionality of the Symbolic Manager by means of two aspects we have just specified. Figure 7.31 shows an example of the SM-Specification.

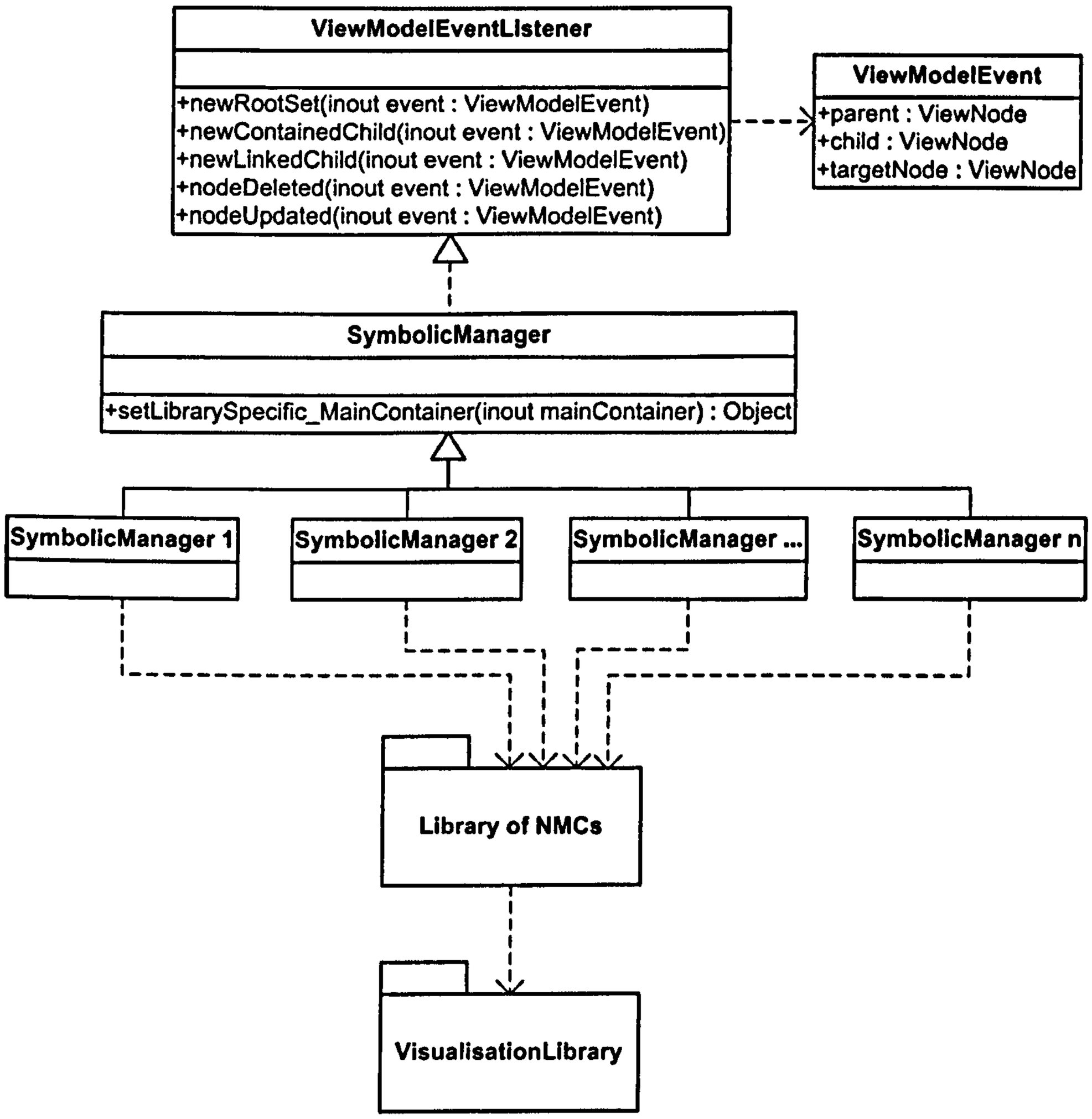


Figure 7.30: A UML class diagram describing a Symbolic Manager

SM-SPECIFICATION TABLE			
Number	***Name***	***Part***	***Visual Components***
1	HWSymbolicManager	(1) Library-specific container --- (2) Term [Main Container] --- (3) Term [Console Viewer Application]	javax.swing.JPanel --- MainContainerGUI --- ConsoleViewerGUI

Figure 7.31: An example of the SM-Specification

7.8 Deployment

The specification of the domain-specific visual interface together with correctly linked domain-specific composition system and the template composition system form the DSVCS. It is the deployment descriptor which collects all the information about the DSVCS in a predefined and well-structured form. The structure of the deployment descriptor is explained in Figure 7.32 that depicts so called DSVCS-Deployment Table.

DSVCS-Deployment Table		
Number	***Variable***	***Descrption***
1	`NML_ID_NAME`	The name of the Neurath Modelling Language (NML)
2	`DESCRIPTION`	The description of the NML
3	`INITIAL_EXPRESSION`	An initial expression to be executed before other design steps

4	`LANGUAGE_ELEMENTS`	The list of the paths to the XML files describing deployment of the DSCs and DSOs.
		
		Example:
		`DSC/MainContainerDSC.xml`
		`DSC/SensorDSC.xml`
		`DSC/ActionDSO.xml`
		`...`

5	`VIEW`	The characteristics of the View. These are `NAME`, `CM`, `LM` (optional) and `SM`.
		
		Example:
		`NAME = Device-types (statics)`
		`CM = viewSpec/HAContainerManager.cm`
		`SM = viewSpec/HASymbolicManager.sm`

Figure 7.32: Structure of the deplyoment descriptor for domain-specific visual composition system

The DSCs and DSOs are deployed according to their deployment specifications. Figure 7.33 depicts the table that explains the structure and the meaning of these deployment descriptors. An example of the XML file with the deplyoment desciptors for DSVCS can be found in Appendix C (Listing C.7).

DSC-Deployment Table	
Variable	***Description***
`LANGUAGE_ID`	The domain specific term which is represented by the DSC.
`DSC_ELEMENT`	The path to the .dsc file that holds the implementation of the DSC.
`TOOLBAR_ICON`	The path to the icon file in case if the DSC should be presented in the toolbar.

DSO-Deployment Table	
Variable	***Description***
`LANGUAGE_ID`	The domain specific operation which is represented by the DSO.
`DSO_ELEMENT`	The path to the .dso file that holds the implementation of the DSO.
`TOOLBAR_ICON`	The path to the icon file in case if the DSO should be presented in the toolbar.

`PARAM_SIGNATURES`	The description of parameters for this DSO. It includes the attributes `Name`, `Setter` and `Getter`. The `Name` is the name of the parameter, the `Setter` and `Getter` are methods names to access the parameter.
	
	Example :
	`Name = sensor`
	`Setter = setSensor`
	`Getter = getSensor`
	`-------------------------`
	`Name = controller`
	`Setter = setController`
	`Getter = getController`

Figure 7.33: Structure of the deplyoment descriptors for DSCs and DSOs

7.9 Implementation Environment

The concepts defined at the Visualisation and Interaction Level of composition are practically implemented in the *Visualisation and Interaction Library*. The library was implemented with the Java programming language. We defined the package `neurath.vailevel` as the main package of the library. The main packages and their interrelationships are shown on Figure 7.34.

The following main sub-packages are specified:

1. The sub-package `viewmodel` defines classes that describe the common features for Views. This includes classes `ContainmentManager`, `LinkageManager`,

SymbolicManager and ViewModel. The classes of this sub-package are used by the classes that specify Views for the target application domain (see dependency relation to the sub-package Package for a target domain. Besides, the classes of the sub-package viewmodel work with the classes defined by the Domain-specific Composition Library (see dependency to the package neurath.tdlevel).

2. The sub-package nmcmodel defines an abstract class NMCVisualElement that specifies main requirements for all Neurath Modelling Components (see dependency relation from the classes NMC).

3. The sub-package Package for a target domain defines classes that describe Neurath Modelling Language and Views for a target domain. Different NMCs are defined by the classes with the name NMC. Of course, this name is a only a temporary name as for the target application domain the concrete set of names is specified. Views are defined by instances of the class ViewV. NMCs and Views are implemented according to the specifications of the sub-package viewmodel (see dependency to this sub-package). Views manage NMCs that are located at the modelling pane (dependency from the ViewV to the ModellingPane).

4. The sub-package gui contains classes Toolbar and ModellingPane. The class Toolbar represents a toolbar graphical user interface component where NMCs are deployed (dependency between classes NMC and Toolbar). The class ModellingPane represents a window of the modelling pane. At design time it contains instances of NMCs (dependency the classes NMC).

5. The sub-package uiiparser defines a parser for User Interface Interaction Expressions (UIIEs) generated by NMCs during the interactions with the designer (dependency between NMC and the sub-package uiiparset).

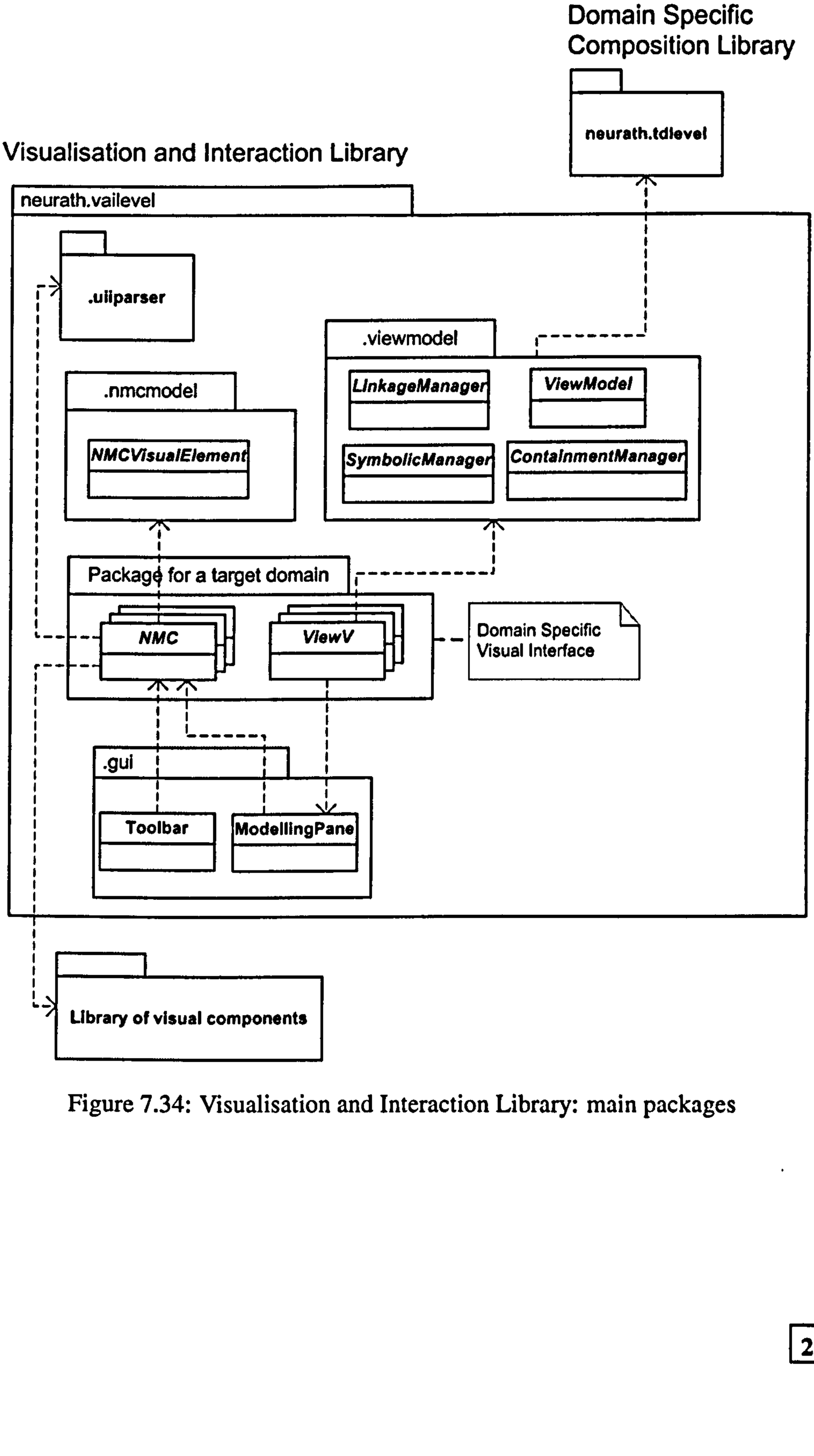


Figure 7.34: Visualisation and Interaction Library: main packages

7.10 Summary

This chapter has introduced concepts defined at the Visualisation and Interaction Level of composition. These concepts represent a domain-specific visual composition system. With this system it is possible to define domain-specific visual interfaces (DSVIs) on top of domain-specific composition languages, as they defined at Target Domain Level of composition, and apply these DSVIs to design a software system.

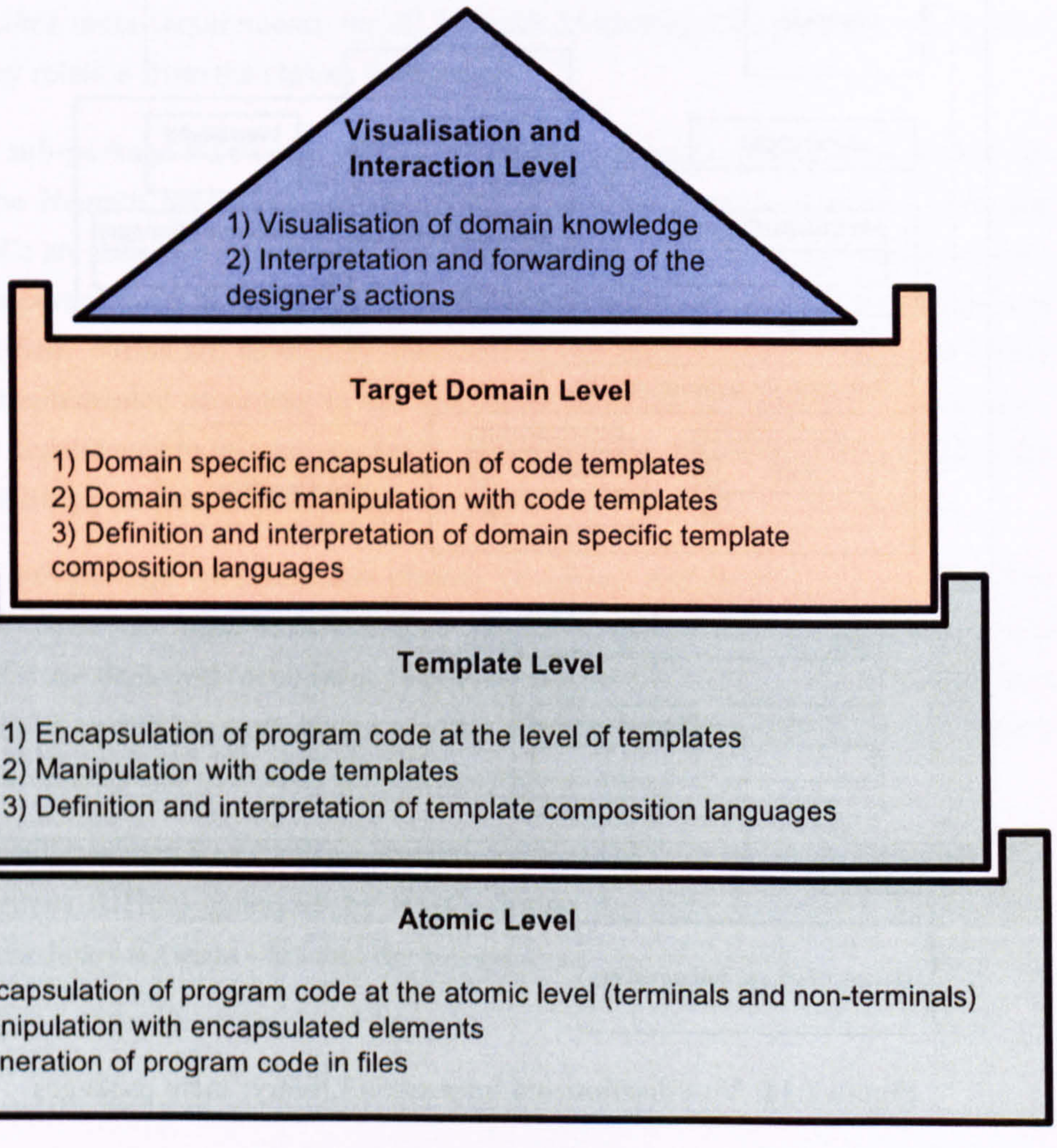


Figure 7.35: Levels of composition within the Neurath Composition Framework

With DSVI the Domain Experts is able to produce program code through DSVI without having any technical background in the programming language that is used to describe

a software system. We have shown how interaction between the DSVI and the Domain Expert works. We have described the Neurath Modelling Language and the View concepts that organise the process of interaction. Finally, we presented the organisation of the implementation environment for the concepts defined at the Visualisation and Interaction Level of composition. Figure 7.35 shows parts of the Neurath Composition Framework that we have described.

The next chapter explains an architecture of the Neurath Builder Tool and the Neurath Integration Platform. These tools represent a reference implementation of the Neurath Composition Framework.

Chapter 8

Tool Support and Evaluation

This chapter puts the concepts developed in the previous chapters into a practical level. It demonstrates the main features of the process of software design based on the Neurath Composition Framework. The demonstration is supported by the reference implementation for the framework. The architecture and design of the implementation is also covered in this chapter. The chapter ends with conclusions.

8.1 Overview

In this chapter, the NCF approach is evaluated with two use-cases:

1. The first use-case is called Console Viewer. It represents a simple application domain that consists of a DOS console application and a textual message. This use-case introduces the NCF life-cycle, showing the design phase and the composition language definition phase. The domain-specific visual composition system for Console Viewer represents minimal example, but can already show the externalisation of business logic.

2. The second use-case is called House Automation. It represents an application domain of some virtual instruments, such as actuators, sensors and controllers, that can be connected with each other. This use-case introduces the NCF life-cycle, showing the design phase and the composition language definition phase. The use-case shows the externalisation of business logic and introduces the related feature of the *descriptiveness* of components. The House Automation composition system introduces the power of the template composition with the Parametric Code Templates and Molecular Operations. Feature such as awareness and inheritance template components will be shown.

We have developed a reference implementation of the NCF to practically test the use-cases. The reference implementation is presented by the Neurath Builder Tool and the Neurath Integration Platform. The NBT is for the composition language definition phase. The NIP is for the design phase. Further, the architecture and design of the reference implementation is explained. After this, the Console Viewer and the House Automation use cases are presented.

8.2 Architecture

We have developed a reference implementation for the NCF. The architecture of the reference implementation is presented in Figure 8.1. As the architecture shows, there are two main tools defined. These include the NBT and the NIP. The productions of the NBT tool are specifications of a domain-specific visual composition system. This specification can be deployed into the NIP tool. Once deployed a domain-specific visual composition sys-

tem is used to visually model a software system for an application domain. Automatically, the program code for the modelled software system is generated.

The NBT is a tool to use during the composition system definition phase. At this phase, a domain-specific visual composition system is defined according to the requirements provided by the domain expert. The Software Architect works with the NBT tool. The Software Architect is an expert in programming languages. He/She uses the NBT in order to provide the following:

1. Composition system to operate with code templates (template composition system)
2. Externalise business logic of a template composition system by the specification of a domain-specific visual interface on top of it.

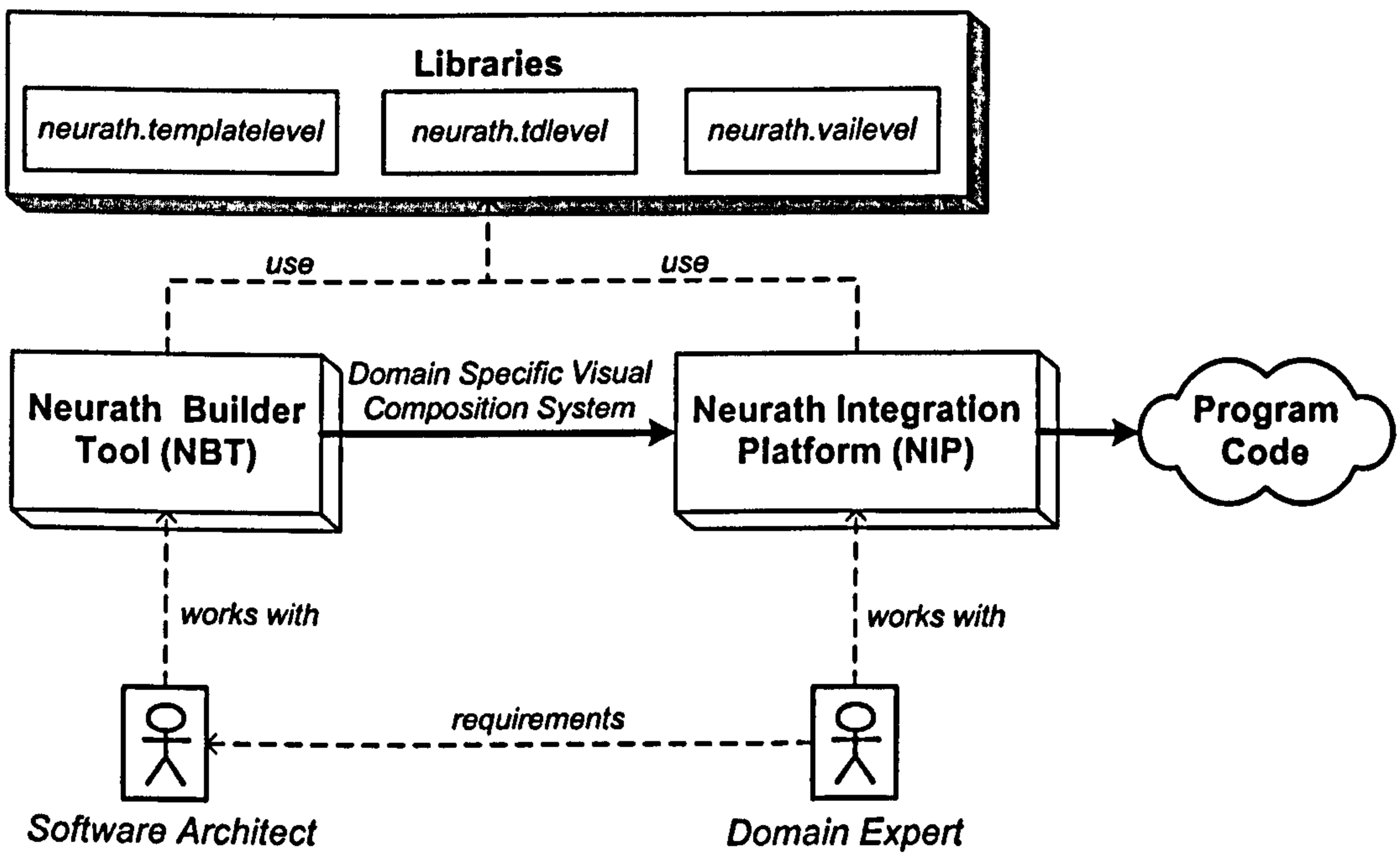


Figure 8.1: An architecture of the reference implementation developed for the Neurath Composition Framework

The NIP is a tool to be used during the design phase. It deploys the domain-specific visual composition systems produced by the NBT. The Domain Expert uses the deployed composition systems to design software systems within the required application domain. The Domain Expert *should* be an expert in that application domain and *should not* have a background in programming languages. During the work with the NIP, the following basic activities are performed:

1. Forming a domain-specific composition expression in terms of visual domain-specific symbols.

2. "Reading" and organisation of the designed system's state which is reflected with visual domain-specific symbols.

The architecture on Figure 8.1 also shows relationships to the libraries developed for composition levels that are defined in this thesis. Table 8.1 shows which library implements what composition level.

Level of Composition	**Library**
Template Level (Chapter 5)	`neurath.templatelevel` (Section 5.8)
Target Domain Level (Chapter 6)	`neurath.tdtelevel` (Section 6.13)
Visualisation and Interaction Level (Chapter 7)	`neurath.vailevel` (Section 7.9)

Table 8.1: Different libraries implementing different levels of composition

8.2.1 The Neurath Builder Tool

The architecture of the Neurath Integration Platform is presented in Figure 8.2. Further we describe each functional block, including relationships to other blocks.

The **Domain Requirements Specificator** is a functional block that is needed for fixing requirements for a composition system obtained from the Domain Expert. It works with the description specifications described in Section 3.6. For example, these specifications define domain-specific terms and relationships that may form the designed system. The output of the functional block is a well-formed description of requirements.

The **Template Language Creator** is a functional block to form the template composition language. It provides repositories of PCT components and molecular operations. The Software Architect chooses the required elements from the repositories or creates new elements and forms a new repository. The new repository contains templates and operations that are used to form the program code of domain-specific software systems

which will be designed later.

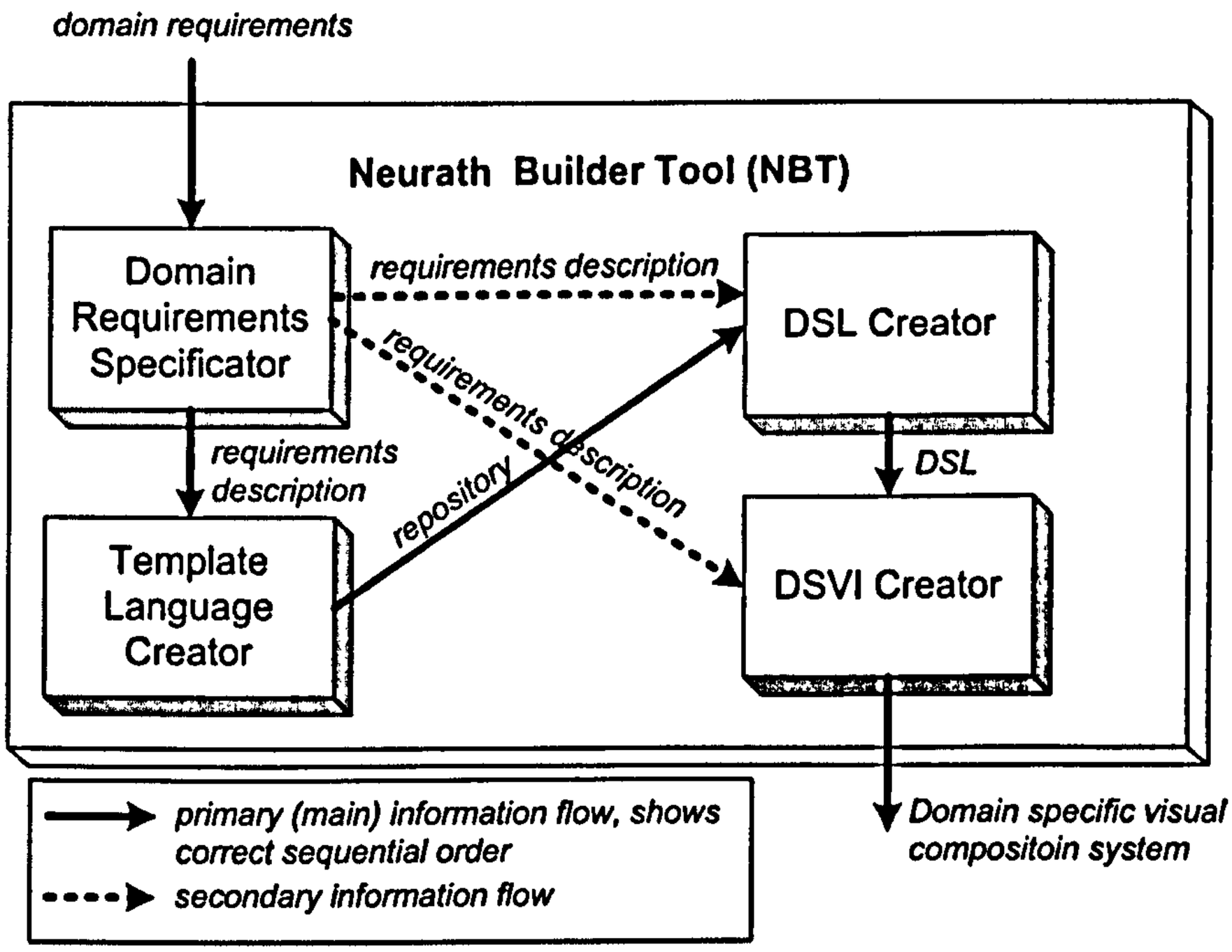


Figure 8.2: An architecture of the Neurath Builder Tool

The **DSL Creator** stands for "Domain-specific Language Creator". It is a functional block to form the domain-specific composition languages on top of the template composition language that is defined by the input repository. The DSL Creator provides semi-automatic code generation mechanisms to generate DSCs as well as DSOs for elements of the repository according to the specified requirements in the Domain Requirements Specificator. DSCs and DSOs form a DSL which is the main output of the DSL Creator block.

The **DSVI Creator** stands for "Domain-specific Visual Interface Creator". It is a functional block to define the Domain-specific Visual Interfaces (DSVIs) on top of the DSL, which is the main input for the DSVI Creator. The part of requirements' descriptions, as related to domain-specific appearance and defined in the Domain Requirements Specificator, are used by the block to semi-automatically generate specification of visual components (Neurath Modelling Components), which form the DSVI. The DSVIs de-

fined on top of a DSL according to the domain requirements represent a domain-specific visual composition system which is the main output of the DSVI Creator.

Currently, there is no visual interface that is provided for the NBT. The Software Architect uses API directly from the related libraries.

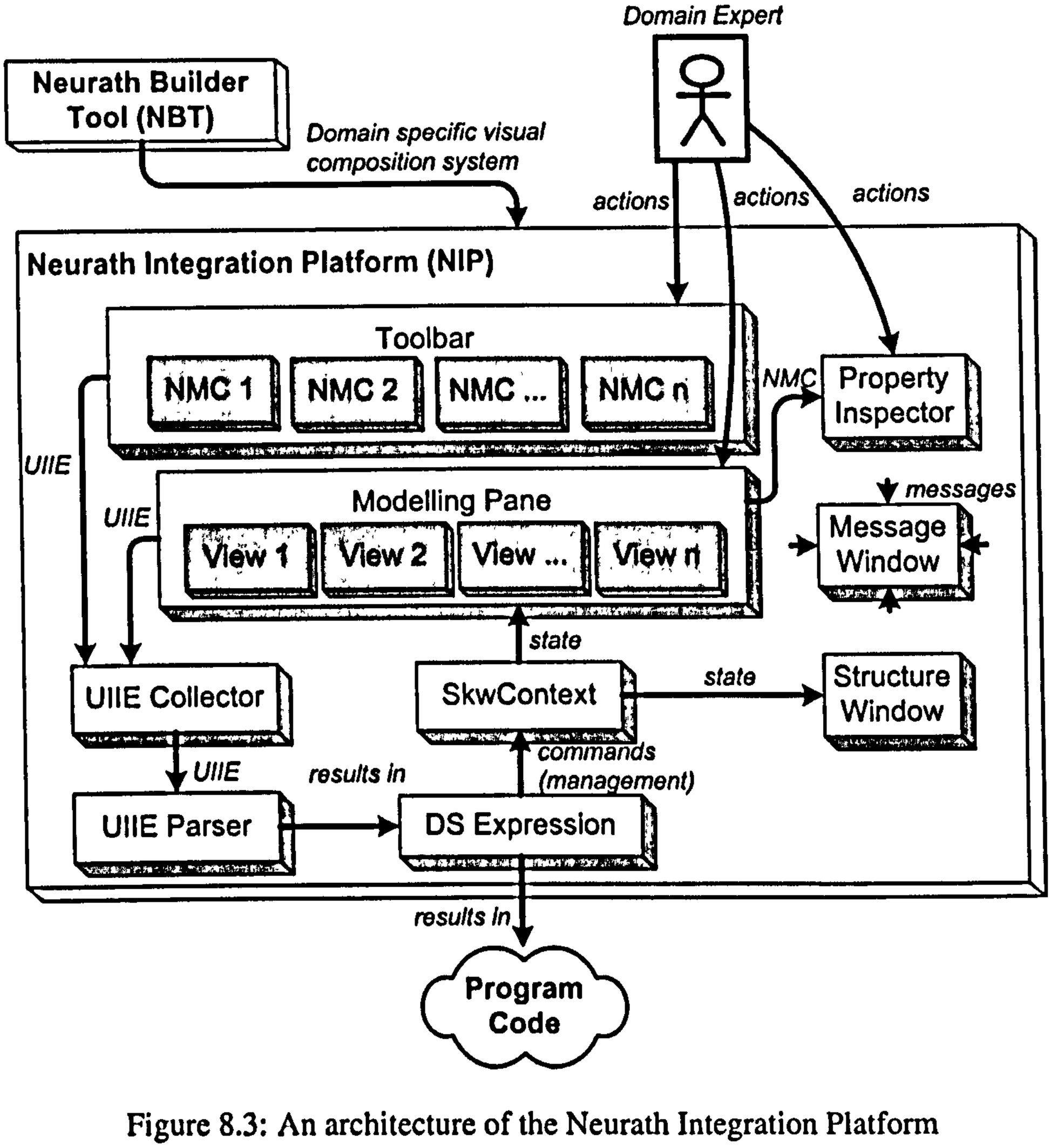


Figure 8.3: An architecture of the Neurath Integration Platform

8.2.2 The Neurath Integration Platform

The architecture of the NIP is presented in Figure 8.3. We have only shown the most important functional blocks that take part during the design process of a software system.

Furthermore, we describe each functional block including, relationships to other blocks.

The **Toolbar** is the GUI component of the NIP. This component contains Neurath Modelling Components that are language elements of the deployed domain-specific visual composition system. When interacted by the Domain Expert, these components generate User Interface Interaction Expressions (UIIEs) that are collected by other functional block called UIIE Collector. The NMCs in the Toolbar represent types.

The **Modelling Pane** is the GUI component of the NIP. It reflects the state of the designed software system in a visual domain-specific way. The state is reflected with Neurath Modelling Component instances. When interacted by the Domain Expert, these visual components generate User Interface Interaction Expressions (UIIEs) that are collected by the UIIE Collector functional block. The reflection of a system's state (defined by the SkwContext block) is performed by Views. They receive a state description of a designed system and generate new NMC instances that are placed within the Modelling Pane area.

The **UIIE Collector** stands for "User Interface Interaction Expression Collector". It collects expressions that are generated during actions applied by the Domain Expert to NMCs defined in the Toolbar and in the Modelling Pane. The UIIEs are collected in one expression which is forwarded to the UIIE Parser.

The **UIIE Parser** stands for "User Interface Interaction Expression Parser". It obtains the UIIE from the UIIE Collector and generates a functional block called DS Expression from it.

The **DS Expression** stands for "Domain-specific Expression". It is a functional block that contains an executable domain-specific expression that consists of DSCs and DSOs. The DS Expression executes the expression resulting in a (1) generation or transformation of Program Code; (2) changed description of the designed system's state, which is held by the functional block called SkwContext.

The **SkwContext** stands for "Simple Knowledge Web Context". It is an environment to describe the state of the designed system in the form of a Domain Ontology. It operates

with domain-specific terms, their attributes and relationships between them.

Property Inspector is a GUI component of the NIP. It contains a parameter-value pair for the NMC chosen in the Modelling Pane. The Property Inspector can be used to observe defined values for the parameters and to assign the new ones.

Message Window is a GUI component of the NIP. It receives and shows messages generated by different components of NIP. The Message Window reflects the information about the design process.

Structure Window is a GUI component of the NIP. It is a special kind of View. It reflects a structure of the designed system on the required level of details.

8.3 Design

A deeper view on how the NBT and the NIP are organised is presented by parts of their design. Figures 8.4 and 8.5 depict UML class diagrams showing interrelationships of main classes. Coloured and labelled areas extend the diagrams in order to group classes by their role. Additionally, the relations that could be seen as the most meaningful are drawn in bold.

Figure 8.4 shows classes that implement the idea of the NBT. Currently, the NBT has no support of a graphical user interface. Therefore, the Software Architect uses the API directly. The following areas of classes are presented in the figure:

1. **Green area (A)**: groups the definitions of PCTs and MOs, which form a common repository of templates for the Java programming language. The classes that extend the class `AbstractPctLeaf` are PCTs defined in the common repository. The classes derived from the class `MolecularOperation` are MOs defined in the common repository. Moreover, this group contains tools needed to create new PCTs and MOs. The green area represents a Template Level of composition. The connection to the Atomic Level of composition is shown with relationships between the green area and `aslt.java` package.

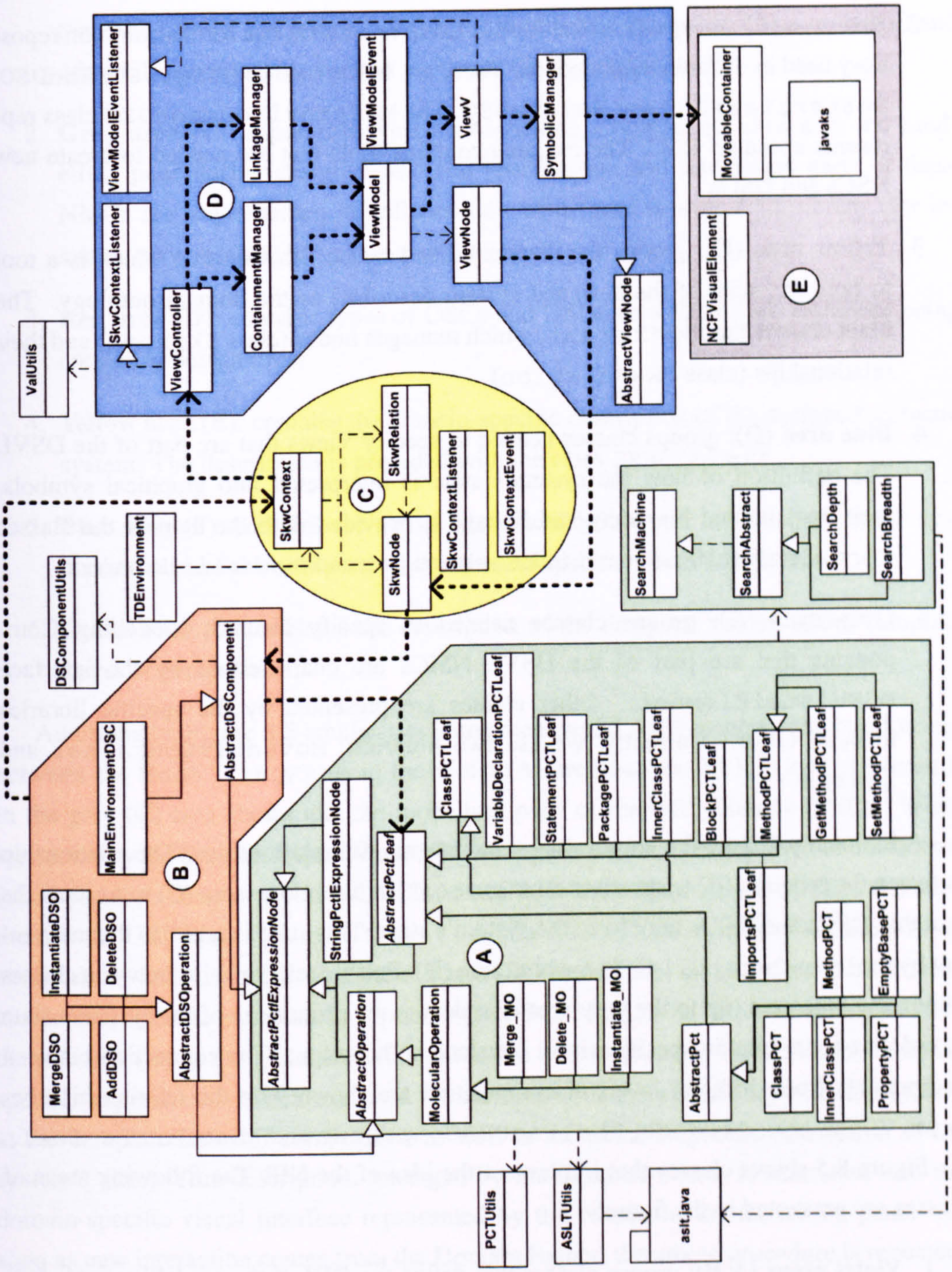


Figure 8.4: The design presenting the NBT. A - the Template Level of composition, B and C - the Target Domain Level of composition, D and E - the Visualisation and Interaction Level of composition

2. **Red area (B)**: groups the definitions of DSCs and DSOs that form a common repository used in different applications. These are DSOs such as `MergeDSO`, `AddDSO`, `InstantiateDSO` and `DeleteDSO`. The `MainEnvironmentDSC` class represents common DSC. The red area contains tools that are needed to create new DSCs and DSOs.

3. **Yellow area (C)**: groups the classes related to the SkwContext, which is a tool to hold the state of the designed system described as the domain ontology. The main class is the `SkwContext`, which manages nodes (class `SkwNode`) and their relationships (class `SkwRelation`).

4. **Blue area (D)**: groups classes needed to specify Views that are part of the DSVI. The definition of how the system's state is interpreted into graphical symbols, their containment hierarchy and linkage is provided with the help of the classes `ContainmentManager`, `LinkageManager` and `SymbolicManager`.

5. **Grey area** (E): groups classes needed to specify Neurath Modelling Components that are part of the DSVI. NMCs are characterised by the interface `NCFVisualElement`. Other classes are presented by the specific libraries to build GUIs. Currently, we use two libraries: `MoveableContainer` and `javaks`.

Additionally, Figure 8.4 emphasises some meaningful relationships. The relationship between the red area (B) to the class `SkwContext` in the yellow area (C) represents the fact that DSCs and DSOs may form the system's state. This state is taken as the main input by the `ViewController` in the blue area (D). Further relationships between classes within the blue area (up to the grey area (E)) denote the processing of the system's state in order to get a domain-specific visual interface at the output. The connection between components from different levels of composition are shown with the relationships between `ViewNode`, `SkwNode`, `AbstractDSComponent` and `AbstractPctLeaf`.

Figure 8.5 shows classes that implement the idea of the NIP. The following areas of classes are presented in the figure:

1. **Violet areas (A,B)**: groups classes that define the GUI of the modelling tool. It is implemented according to the Model-View-Controller architecture pattern. The part A is the "View" part and part B is the "Model" and "Controller" parts. The GUI of the NIP consists of the modelling pane (class `ModellingPane`), the toolbar (class

NmlToolBar), the property inspector pane (class PropertyInspector) and other GUI components such as windows, dialogs and menus.

2. **Grey area (C)**: represents Neurath Modelling Components shown in the modelling pane during design time. The visualisation and interaction parts of these NMCs are implemented in collaboration with some specific GUI library (for instance MoveableContainer or javaks).

3. **Red area (D)**: contains types of DSCs and DSOs as well as their instances being created at design time.

4. **Yellow area (E)**: contains the domain-specific description of the designed software system. The description is provided with the class SkwContext.

5. **Blue area (F)**: contains the active Views which interpret any changes in the system's state into the new parts of the domain-specific visual interface.

6. **Green area (G)**: contains the description of the designed system at the Template Level of composition.

Additionally, Figure 8.5 emphasises some meaningful relationships. The relationship between the ModellingPane in the violet (A) area and the NCFVisualElement in the grey (C) area shows that the modelling pane of the NIP contains NMCs. When interacted by the Domain Expert, these NMCs - together with NMCs defined in the toolbar (class NmlToolbar) - produce User Interface Interaction Expressions (UIIEs) and send them to the UIIE Parser (class UIIE_Parser) in order to be translated. The relationship between the UIIE_Parser and the red area (D) denotes that the products of translation are domain-specific expressions formed with DSCs and DSOs. The relationships from the red area (D) to the green area (G) and to the yellow area (E) show that the domain-specific expressions - when executed - transform both the underlying software system at the Template Level of composition as well as the domain-specific description of the system's state. This description, managed by the SkwContext, is translated into the domain-specific visual interface represented by the NMCs in the modelling pane. As soon as new interaction comes from the Domain Expert, the whole procedure is repeated.

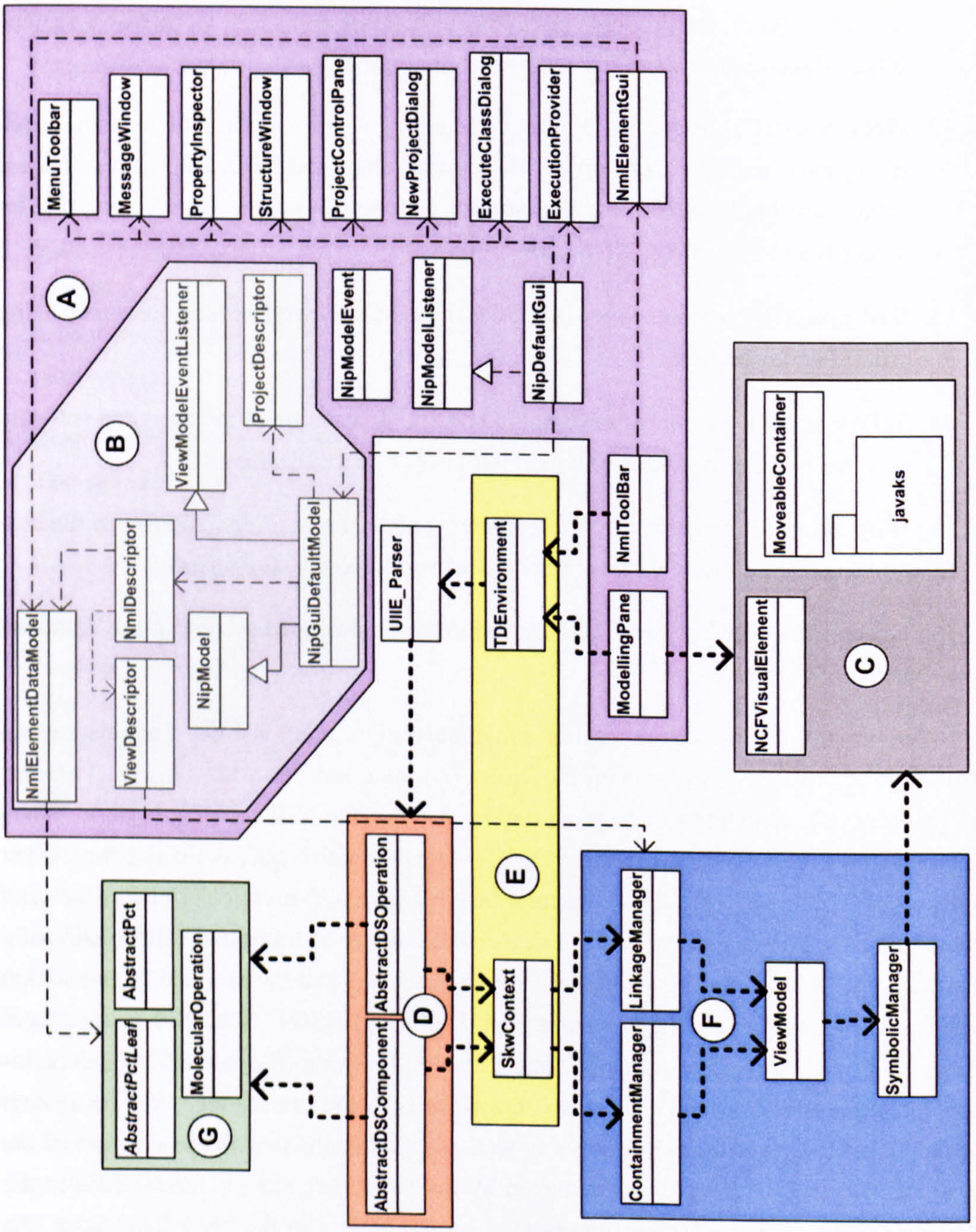


Figure 8.5: A and B - the GUI of the NIP, C and F - Visualisation and Interaction Level of composition, D and E - Target Domain Level of composition, G - Template Level of composition

A short description of each class of the reference implementation can be found in Appendix B. Libraries and documented program code can be taken from [96]. The NBT and NIP tools can be loaded from [97, 98]. Further, we introduce two use-cases implemented and shown with the reference implementation for the Neurath Composition Framework approach.

8.4 Evaluation: Console Viewer

The Console Viewer is a simple console application that prints out some text and then terminates. Being developed by a Software Architect, such a console system may be defined with a program code of ten to twenty lines of Java program code. Imagine that the Domain Expert, which has no background in programming languages, would like to produce different instances of a Console Viewer application by only operating with business logic, with no knowledge of programming involved. In this section, we show how - with help of the NCF - the business logic of the Console Viewer application domain can be externalised. Basically, the Console Viewer is a quite primitive example of applying the NCF. However, it is needed to serve the purpose of introducing the following:

1. Feature of externalisation of business logic. It is shown that the Domain Expert gets an ownership to design a software system within the defined application domain.

2. Steps of the life-cycle defined by the NCF. Step by step, the phases of the software life-cycle are introduced. The same steps are repeated for the other use-case.

After looking through the Console Viewer, we assume it will be easier for the reader to work with descriptions for the next use-case. The second use-case reveals the practical applicability of the NCF approach by introducing more features.

8.4.1 Specification of Domain Requirements

According to the software life-cycle model described in Section 3.5 (see Figure 3.6) the first step during the composition system definition phase is the domain requirements analysis. At this step, the requirements specified by the Domain Expert are put into the defined form. According to the description strategy proposed in Section 3.6 during the domain requirements analysis, there should be several specifications provided. We provide each of them.

The first table is the **Requirements as English text.** It is provided by the Domain Expert in the English language (see Figure 8.6).

Requirements as English text	
Number	***Requirement***
1	The domain is called Console Viewer
2	The software system is a console application
3	When the application is executed, it prints out the text and then terminates
4	The text can be defined at design time
5	Initially the text is „Hello World"
6	It should be possible to define multiple console applications

Figure 8.6: Requirements as English text for the Console Viewer application domain

After the requirements are described in English, the Software Architect and the Domain Expert formulate the **Domain Ontology**, trying to reveal the terms, which are going to be operated during design time as well as possible relationships between these terms. Figure 8.7 shows the domain ontology for the Console Viewer.

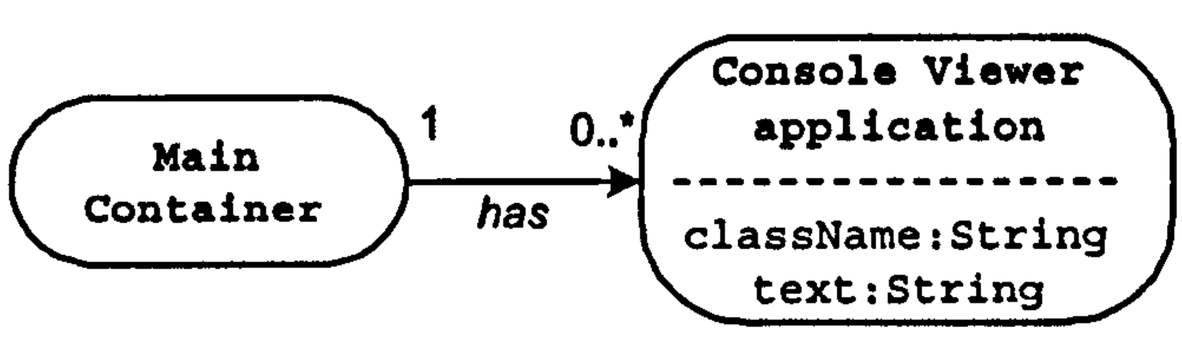


Figure 8.7: Domain Ontology for the Console Viewer application domain

Further, the **Term-English table** depicted in Figure 8.8 states what each term, defined in the domain ontology, means. The domain ontology describes the application domain statically. To describe the dynamic changes in domain ontology, the Domain Expert and the Software Architect work out the **Actions-State table.** This is a table that shows dependencies between sequences of actions that are done when designing a system at the design phase and the desired changes in the system's state.

TERM-ENGLISH TABLE		
Number	***Term***	***Description***
1	`Main Container`	Represents the main container that is a placement area for other elements.
2	`Console Viewer Application`	Represents a console application, characterized by the attributes `ClassName` and `Text`. The `ClassName` represent the name of an application instance. The attribute `Text` represents the Text to be printed out by the application.

Figure 8.8: Term-English table for the Console Viewer application domain

Actions are defined according to the definition of what the composition process with externalised business logic stated in Section 3.2 is. Actions are defined as follows:

1. *click component type* - means that the Domain Expert clicks the "component type" which is represented by an icon in the toolbar.

2. *click operation type* - means that the Domain Expert clicks the "operation type" which is represented by an icon in the toolbar.

3. *click component instance* - means that the Domain Expert clicks the "component instance", located in the modelling pane.

Figure 8.9 depicts the Actions-State table.

ACTIONS-STATE TABLE			
Number	***Actions***	***State 1***	***State 2***

1	`click component type [Console Viewer Application]`
	`----------`
	`click instance [Main Container]`
		`[Main Container]`
		
		`[Console Viewer Application]`
			`[Main Container]`
			`-has->`
			`[Console Viewer Application]`

2	`click operation type [DeleteDSO]`
	`----------`
	`click instance [Console Viewer Application]`
		`[Console Viewer Application]`	

Figure 8.9: Actions-State table for the Console Viewer application domain

Up to now, the Domain Expert and the Software Architect specified the semantics of a domain-specific composition system. The next step is to describe the visual interface for this system.

The appearance specification for so called "component types" and "components operations" located in the toolbar as well as for "component instances" located in the modelling pane is provided by the **Types/Instances-appearance table**. This table shows the domain-specific components and operations and specifies dependencies between them and visual appearance. Figure 8.10 depicts the table.

TYPES/INSTANCES-APPEARANCE TABLE				
Number	***Element***	***Kind***	***Appearance***	***Description***
1	[Main Container]	component instance		It is a container which has black borders and white background.
2	[Console Viewer Application]	component type	<message>	It is an icon
3	[Console Viewer Application]	component instance	<text> C:\	It is a container which has black borders and white background. It contains an icon at the bottom and shows the value of the attribute `text` above the icon. The icon shows a console window.
4	[DeleteDSO]	operation type		It is an icon

Figure 8.10: Types/Instances-appearance table for the Console Viewer application domain

From this table, the Software Architect will later create visual components which constitute the domain-specific visual interface, reflecting a designed system's state at the design phase. The Types/Instances-appearance table shows statics rather than dynamics of the component's appearance. The dynamics are described with the **ORel-GRel table**, which basically holds information about how the change in the system's state influences domain-specific visual interface. This shows the dependency between relationships defined by the domain ontology and relationships between components of the domain-specific visual interface. The graphical relations were described in Section 7.7.2.1. Currently, there are two types of relationships: "contains" and "links". Figure

8.11 depicts the table.

OREL-GREL TABLE		
Number	***Ontology Relation***	***Graphic Relation***
1	[Main Container] **-has->** [Console Viewer Application]	[Main Container] **-contains->** [Console Viewer Application]

Figure 8.11: ORel-GRel table for the Console Viewer application domain

The provided specifications are concluded with the overall schematic view (or simulated by tools) on the domain-specific visual composition process. Further, we give this overall conclusion demonstrating the screenshots of NIP made during designing applications in the Console Viewer application domain.

Figure 8.12 depicts the NIP, which has a toolbar on the top, a modelling pane in the middle and a property inspector on the right side. The View that defines how the state is visually represented is called "View 1". The View shows a container with a black line border. This is the `MainC ontainer` component that is initially present for all composition systems. This container may contain instances of the Console Viewer applications. The toolbar contains one component type called `Console Viewer application`, and one operation type called `Delete`.

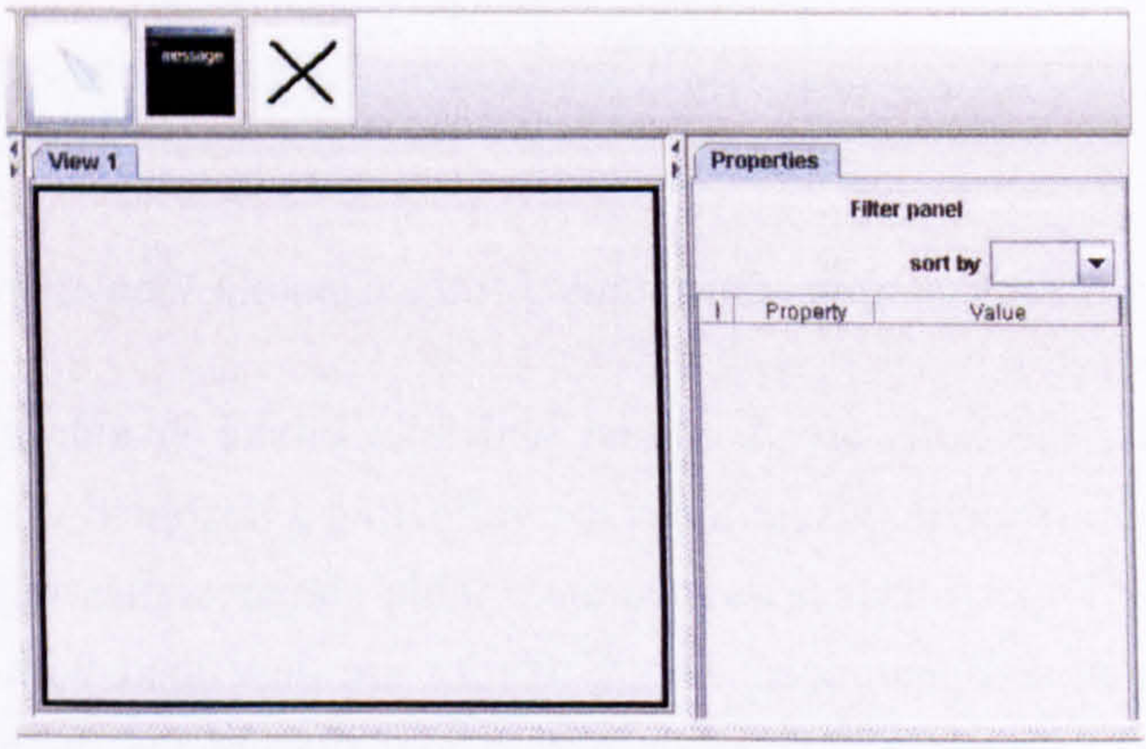


Figure 8.12: Screenshot 1 of the NIP: design phase for the Console Viewer use case

The `Console Viewer application` components type is shown with the black icon (represents a console). The `Delete` operation type is depicted as white icon with

cross on it (represents termination). When the `Console Viewer application` and then the `Main Container` in the modelling pane are clicked sequentially, the new visual component instance appears, as depicted in Figure 8.13. This instance represents the generated program code of the Console Viewer. It shows the name of the application as well as the textual message to be printed after the application is started.

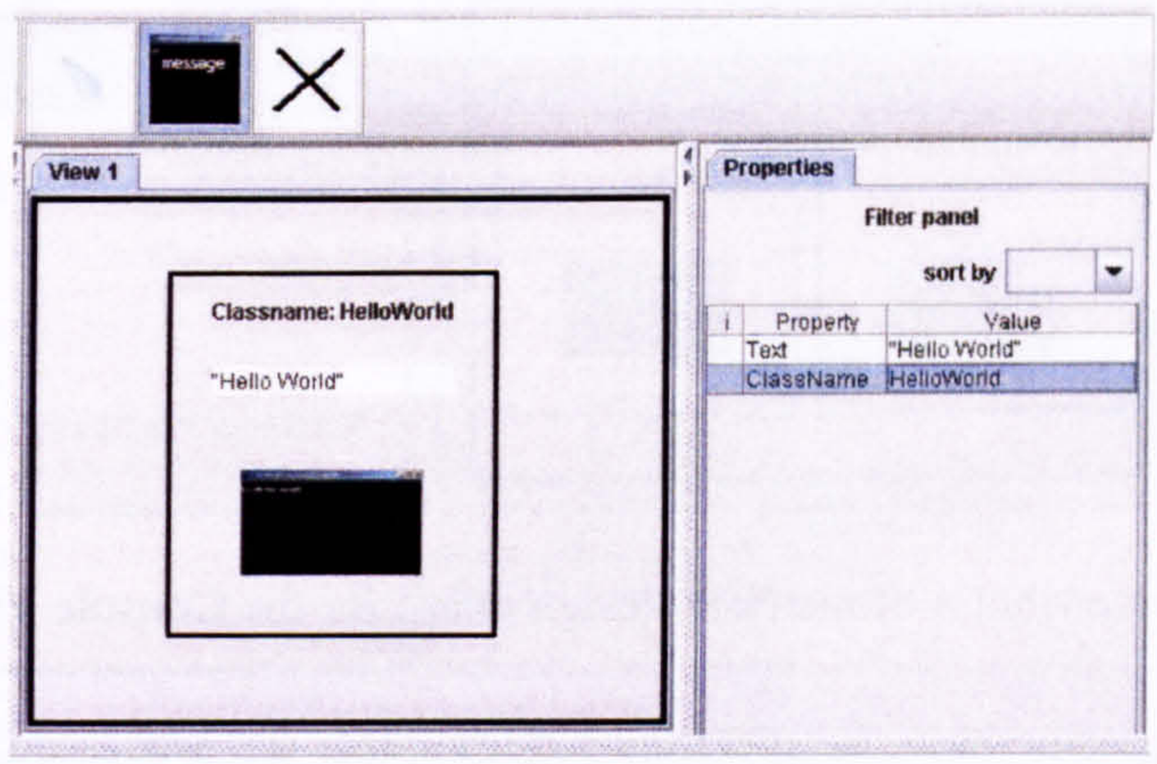


Figure 8.13: Screenshot 2 of the NIP: design phase for the Console Viewer use case

The Property Inspector of the Screenshot 2 shows the attributes for the chosen instance of the Console View. These attributes can be changed by the Domain Experts. For instance, instead of the message "Hello World!", we put the message "Hello UK!". As soon as this is done, the program code is transformed accordingly and the change in state is visually reflected with the new text in the modelling pane (see Figure 8.14).

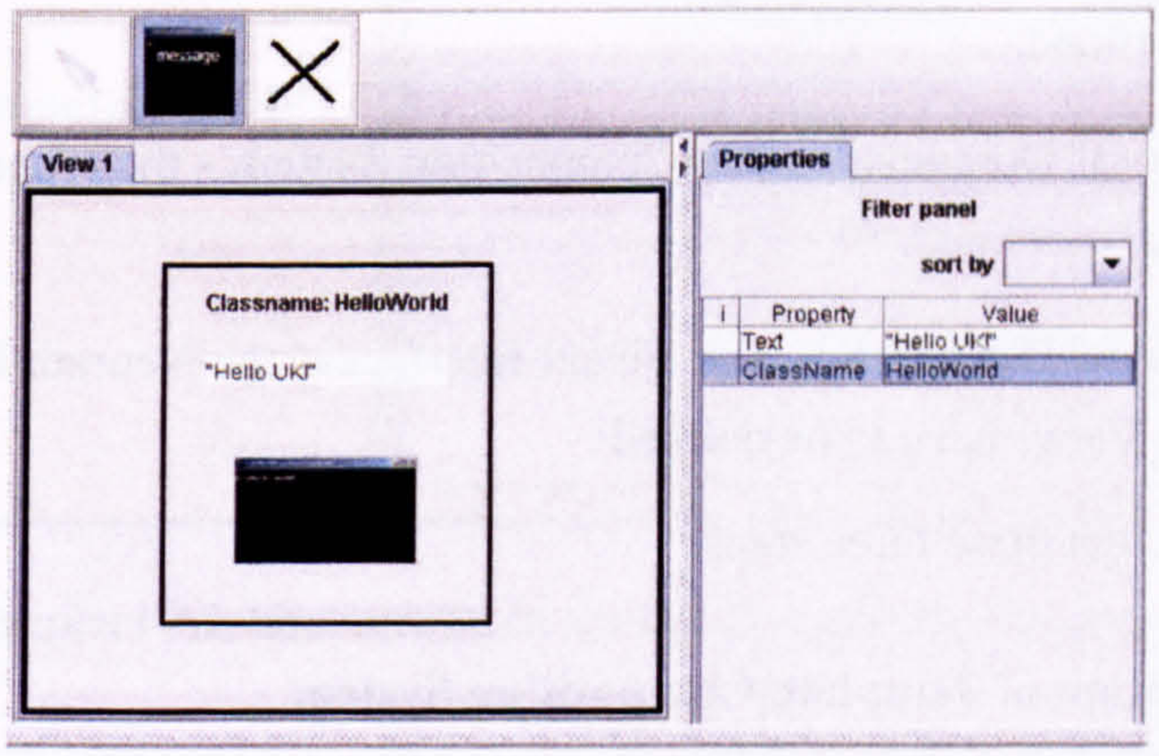


Figure 8.14: Screenshot 3 of the NIP: design phase for the Console Viewer use case

In the same manner, another instance of the `Console Viewer application` can be created and configured. Figure 8.15 shows the resulting domain-specific visual interface.

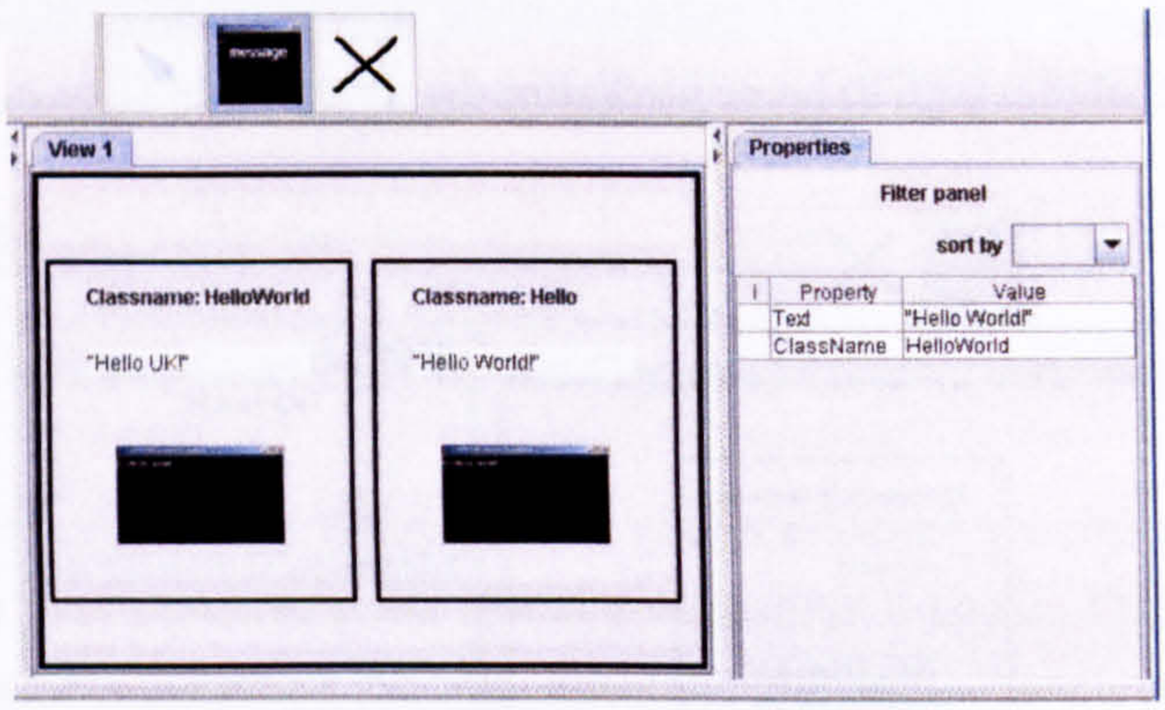


Figure 8.15: Screenshot 4 of the NIP: design phase for the Console Viewer use case

Further, we describe how the specified requirements are processed by the Software Architect in order to create the domain-specific visual composition system for the Console Viewer application domain.

8.4.2 Processing the Domain Requirements

The Software Architect processes the domain requirements in order to create a domain-specific visual composition system. The following steps are carried out:

1. Specification of Template Composition System - the repository of PCTs and MOs have to be defined.

2. Specification of Domain-Specific Composition System - the repository DSCs and DSOs has to be defined.

3. Specification of Domain-Specific Visual Interfaces: the Neurath Modelling Components and Views have to be defined

Further we go through these three steps.

8.4.2.1 Specification of Template Composition System

Being based on the requirements defined by the Domain Expert, the Software Architect may choose the repository of templates presented in Figure 8.16. It is shown how each

template from the new repository is composed with the existing templates and molecular operations from the common repository. The elements from the common repository are marked with the light red colour. The composition steps can be seen as actions done by a Software Architect working with the Neurath Builder Tool. In the case of the visual interface being provided, this process becomes significantly more automatic and easier.

TEMPLATE-REPOSITORY TABLE

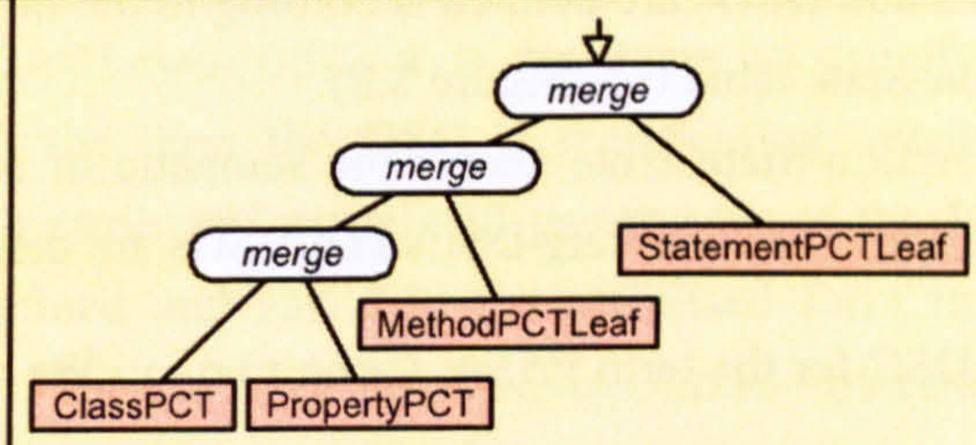
Number	*Tempate*	*Composition Expression*
1	`ConsoleViewerPCT`	

Figure 8.16: Template-Repository table for the Console Viewer application domain

TEMPLATE-ARCHITECTURE TABLE

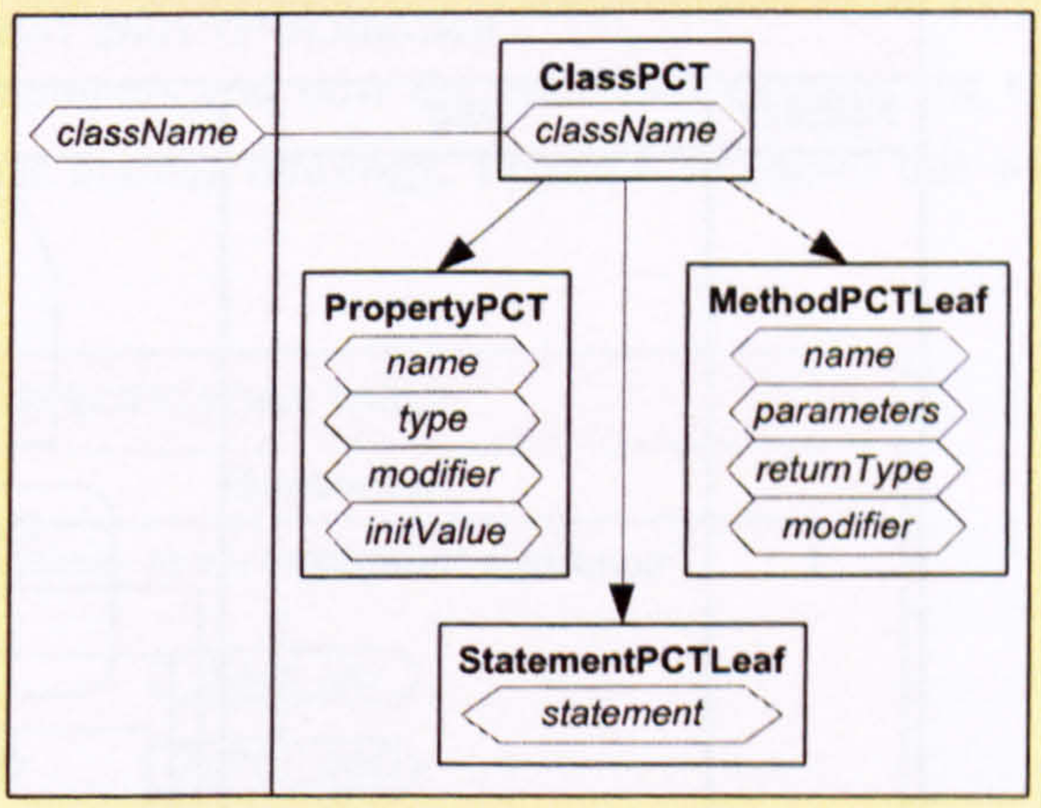
Number	*Tempate*	*Architecture*
1	`ConsoleViewerPCT`	

Figure 8.17: Template-Architecture table for the Console Viewer application domain

The architecture of the template that is resulted during the composition steps is shown in Figure 8.17. The program that creates specified templates in the form of PCTs is shown

in Listing C.1 from Appendix C. The program is written in Java programming language using the Template Composition System Library (see Section 5.8).

Newly defined elements of the repository can be tested in the NBT. This would include activities such as applying the tested element together with other elements, setting parameters and generating a program code.

8.4.2.2 Specification the of Domain-specific Composition System

The DSCs and DSOs are defined according to the information that is mainly provided in the Action-State table (see Figure 8.9).

The Action-State table shows the semantic of actions done by the designer at the design phase. The following DSCs and DSOs are defined:

1. A DSC for the term `Main Container`. We call it `MainContainerDSC`.
2. A DSC for the term `Console Viewer Application`. We call it `ConsoleViewerDSC`.
3. A DSO for the operation `Delete`. We call it `DeleteDSO`.

Figure 8.18 shows the DSC-Architecture table. It shows the configuration of DSCs.

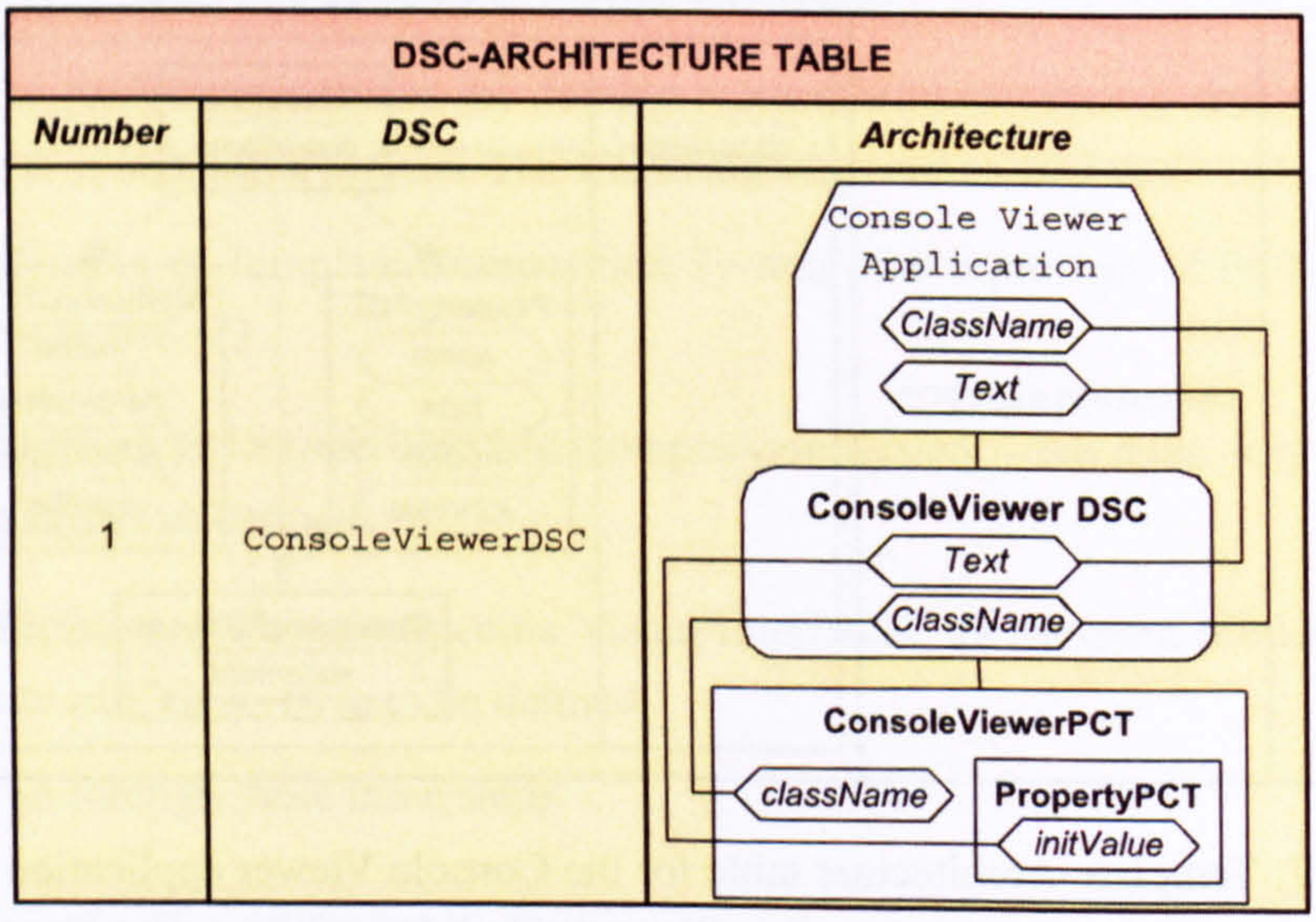


Figure 8.18: DSC-Architecture table for the Console Viewer application domain

The DSC `MainContainerDSC` can be potentially generated by the NBT tool (in the case of having a GUI). All that is needed for specification is the related PCT, the name of the DSC and the term the DSC is representing. The related PCT is `EmptyBasePCT`. The name of the DSC is `MainContainerDSC`. The term that the DSC is representing is `Main Container`.

Listing C.2 from Appendix C shows the specification of the `MainContainerDSC` as a Java class generated according to the architecture shown in the DSC-Architecture table.

To generate the DSC `ConsoleViewerDSC`, it is necessary to specify the related PCT, the name of the DSC, the term the DSC is representing, attributes for the related ontology node and each attributes' correlated parameters of the PCT. The related PCT have already been defined and saved in the serialised form in the file `PCT_ConsoleViewerPCT.pct`. The name of the DSC is `ConsoleViewerDSC`. The term that the DSC is representing is `Console Viewer Application`. The attributes are `Text` and `ClassName`. The correlated parameters are the `initValue` of the composite `PropertyPCT` and the `className`.

Listing C.3 from Appendix C shows the specification of the `ConsoleViewerDSC` as a generated Java class according to the architecture shown in the DSC-Architecture Table.

To generate the specification of the `DeleteDSO`, it is necessary to know its parameters, the expression the DSO implements and how the operation changes the state of the designed system described by the domain ontology. Figure 8.19 shows this with the DSO-Specification table.

DSO-SPECIFICATION TABLE

DSO	*Expression*

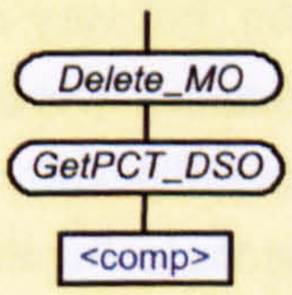
`DeleteDSO`	`DeleteDSO(comp:AbstractDSComponent) =`
	`(1)`
	 
	
	`(2) node=SkwContext.searchForNodeById(comp);`
	`SkwContext.deleteNode(node);`

Figure 8.19: DSO-Architecture table for the Console Viewer application domain

Listing C.4 from Appendix C shows the generated specification of the `DeleteDSO` as a Java class.

8.4.2.3 Specification of Domain-specific Visual Interfaces

The DSVIs are formed with Neurath Modelling Components and Views. According to the requirements defined by the Domain Expert for the Containment Manager, Symbolic Manager and NMCs are defined as follows.

The ORel-GRel table that resulted during the domain requirements analysis contains enough information which to generate the Containment Manager. However we additionally we have to provide a name for the Container Manager, which is `ConsoleViewerContainmentManager`, and the name of DSC for the main container, which is `MainContainerDSC`. Listing C.5 from Appendix C contains the specification of the Containment Manager written in Java programming language.

To create a Symbolic Manager, we require the SM-specification table shown in Figure 8.20. It is sufficient to have the values specified in the table to generate the Symbolic Manager (see Listing C.6 from Appendix C).

SM-SPECIFICATION TABLE			
Number	***Name***	***Part***	***Visual Components***

1	`HWSymbolicManager`	(1) Library-specific container
		--- 	
		(2) Term [Main Container]
		--- 	
		(3) Term [Console Viewer Application]
			`javax.swing.JPanel`
			--- 	
			`MainContainerGUI`
			--- 	
			`ConsoleViewerGUI`

Figure 8.20: SM-Specification table for the Console Viewer application domain

The next specification which is needed is the specification of NMCs. These visual components have the same skeleton, but vary depending on the graphic library used to perform the visualisation and interaction. The basic information about how the skeletons of NMCs are filled in with the library specific code is shown in the NMC-Specification table depicted in Figure 8.21. The icons for the Toolbar are defined in the deployment descriptors (XML files) during the specification of the Domain-specific Composition System.

NMC-SPECIFICATION OF `ConsoleViewerGUI`

```
public class  ConsoleViewerGUI  extends MoveableContainer
implements NCFVisualElement   {
  protected ViewNode carriedNode = null;
    protected JLabel isotype = null;
   protected JLabel className = null;
   protected JTextArea text = null;
   ...
   public ConsoleViewerGUI(  ViewNode node ) {
       super();
       setViewNode(node);
       addMouseListener(this);
       ...
   }
   public void mouseReleased(MouseEvent e){
       ...
   }
   public void updateUIAccSkwNode(){
    text.setText(getViewNode().getSkwNode().
              getAttributeValue("Text").toString());
   }
   ...
   public void addContainee(NCFVisualElement element){   }
   public void removeContainee(NCFVisualElement element){   }
   public NCFVisualElement getParentContainer(){
      return (NCFVisualElement)super.getParent();
   }
}
```

GUI-Library-specific code for appearance
GUI-Library-specific code for interaction
NCF Skeleton
Rest of code

Figure 8.21: NMC-Specification table for the Console Viewer application domain

8.4.2.4 Deployment Descriptors

The specification of the Domain-specific Composition System is generated in the deployment description in the form of XML files. The XML files are generated according to the information provided in the DSVCS-Deployment table depicted in Figure 8.22. Listing C.7 from Appendix C shows the contents of the generated XML file. Each DSC and DSO has its own deployment descriptor. Figure 8.23 depicts deployment descriptors for all components. Listings C.8, C.9, C.10 and C.11 from Appendix C show the contents of the

generated XML file. When the specification is deployed into the NIP the design phase starts.

DSVCS-Deployment Table		
Number	***Variable***	***Value***
1	NML_ID_NAME	ConsoleViewer
2	DESCRIPTION	Domain specific visual composition system to create Console Viewer applications
3	INITIAL_EXPRESSION	InstantiateDSO(MainContainerDSC)
4	LANGUAGE_ELEMENTS	DSC/ConsoleViewerDSC.xml DSC/MainContainerDSC.xml DSO/DeleteDSO.xml
5	VIEW	NAME = View 1 CM = viewSpec/HWContainerManager.cm SM = viewSpec/HWSymbolicManager.sm

Figure 8.22: DSVCS-Deployment table for the Console Viewer application domain

DSC-Deployment Table		
Number	***Variable***	***Value***
1	XML_FILE	MainContainerDSC.xml
2	LANGUAGE_ID	MainContainerDSC
5	DSC_ELEMENT	/dspct/MainContainerDSC.dsc

DSC-Deployment Table		
Number	***Variable***	***Value***
1	XML_FILE	ConsoleViewerDSC.xml
2	LANGUAGE_ID	ConsoleViewer
3	DSC_ELEMENT	/dspct/ConsoleViewer.dsc
4	TOOLBAR_ICON	/isotypes/ConsoleIcon.jpg

DSO-Deployment Table		
Number	***Variable***	***Value***
1	XML_FILE	DeleteDSO.xml
2	LANGUAGE_ID	DeleteDSO
3	DSO_ELEMENT	/dspct/DeleteDSO.dso
4	TOOLBAR_ICON	/isotypes/DeleteIcon.jpg
5	PARAM_SIGNATURES	Name = targetElement Setter = setTargetElement Getter = getTargetElement

Figure 8.23: DSC/DSO-Deployment tables for the Console Viewer application domain

Design Phase			
Number	***Action Sequences***	***State***	***Appearance***
1	Execution of the initialisation exp-ression	Main Container	View 1
2	(1) Clicking the `ConsoleViewer` icon in the toolbar (2) Clicking the `MainContainerGUI`	Main Container — has → Console Viewer Application: `Text="Hello World"`	<message> View 1 Hello World! C:\
3	(1) Clicking the instance of `ConsoleViewerGUI` (2) Changing the value for field `Text` in Property inspector to „Hello UK!“	Main Container — has → Console Viewer Application: `Text="Hello UK!"`	Property Insepctor: Text „Hello UK!“ View 1 Hello UK! C:\
4	(1) Clicking the `ConsoleViewerGUI` in the toolbar (2) Clicking the `MainContainerGUI`	Main Container — has → Console Viewer Application: `Text="Hello UK!"`; has → Console Viewer Application: `Text="Hello World"`	<message> View 1 Hello UK! C:\ Hello World! C:\
5	(1) Clicking the Delete icon in the toolbar (2) Clicking the instance of the `ConsoleViewerGUI` with the text „Hello World!“	Main Container — has → Console Viewer Application: `Text="Hello UK!"`	<message> View 1 Hello UK! C:\

Figure 8.24: Design phase table for the Console Viewer application domain

8.4.3 Design Phase

Figure 8.24 depicts design steps done by the Domain Expert to create a Console Viewer application software system in the NIP. The figure shows how, depending on designer's actions, the state of the designed system and the visual model in the modelling pane are changed. For each step, the program code of the designed software system is automatically generated. Section C.2 in Appendix C collects listings for each design step.

Figure 8.25 shows the screenshot of the application, which corresponds to the 3 step from the Design-steps table in Figure 8.24.

Figure 8.25: Screenshot of the Console Viewer application instance after being executed

8.5 Evaluation: House Automation

In this section, we are going to introduce a use-case for the Neurath Composition Framework. The use-case is for the domain-specific visual composition system to have House Automation software systems designed by the experts in this domain; they do not need a technical background in programming languages.

When we speak about a set of hardware devices installed in the house in order to perform certain actions as a response to certain events, we speak about a control system for the House Automation. Typically, such control systems manage certain sub-systems in the house, i.e. security, climate, "look and feel" etc. Three types of devices can be distinguished: sensors, controllers and actuators. Sensors recognise any change in the environmental state, i.e. the temperature has changed, something moved etc. Controllers are connected to sensors and collect all the incoming signals in order to process them and make a decision. The decision, in form of a signal, is sent to the connected Actuators, which perform the decision made by the connected controller(s).

Let's consider an example presented in Figure 8.26. There is a house with sensors, a controller and actuators installed.

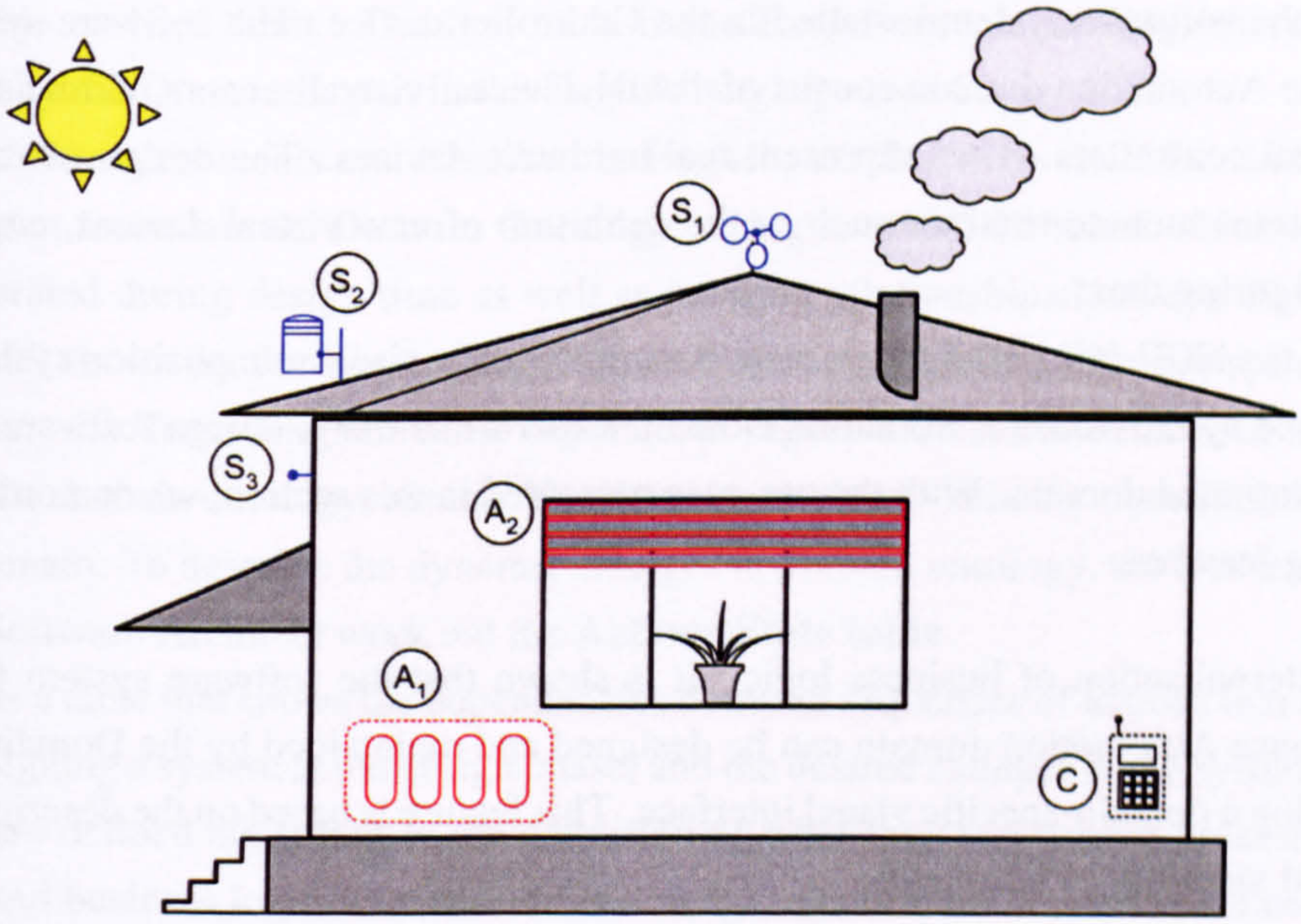


Figure 8.26: A house with a house automation control system installed

The sensors are marked with S: the Wind Sensor S_1, the Light Sensor S_2 and the Temperature Sensor S_3. The controller is marked with C. The devices controlled by the actuators are marked with A: the Heater A_1 and the outside Window Shutter A_2.

The controller could be programmed so that, for instance, the following communication schemes are in operation:

1. The Temperature Sensor sends the value of the current outside temperature to the Controller. The Controller checks if it is too cold and then it activates the Heater. Otherwise, the Heater is switched off.

2. The Light Sensor and the Temperature sensor send the value of the current lightness and temperature to the Controller. If it is too light and the temperature outside is too high, the controller tells to the Window Shutter to shut the window. Otherwise, it is opened.

3. The Wind Sensor sends the current wind speed to the Controller. If the wind speed is too high, the Controller commands the Window Shutter to shut the window.

Often, the connection and configuration of sensors, actuators and controllers is managed by the software system installed in the Controller device. The software systems in the House Automation domain consist of virtual devices: virtual sensors, virtual actuators and virtual controllers. They represent real hardware devices. The design of such software systems include routines such as the definition of new virtual devices, connecting and configuring them.

With the NCF, it is possible to create domain-specific visual composition systems that can be used by the House Automation Domain Experts in order to design software systems in the mentioned domain. With the use-case presented in this section, we demonstrate the following features:

1. Externalisation of business logic. It is shown that the software system from the House Automation domain can be designed and maintained by the Domain Expert using a domain-specific visual interface. This feature is based on the descriptiveness and visibility of templates.

2. Derivation of templates. The development of templates through inheritance is demonstrated. The inheritance of hierarchies introduces a good form of categorisation and implementation reuse. This eases the development of templates.

3. Awareness of templates. We show that the templates implemented according to NCF may react on the environmental constraints and adapt to the new conditions.

4. Reuse of templates. We demonstrate reuse of same templates for different domain-specific components.

5. Automation of design patterns. We also show how the design patterns used by Software Architects can be automated, externalised and used by Domain Experts with no technical background in both programming and software architectures.

6. Parameterisation "by value" and "by structure" of templates.

8.5.1 Specification of Domain Requirements

According to the software life-cycle model described in Section 3.5 (see Figure 3.6), the first step during the composition system definition phase is the domain requirements' analysis. At this step, the requirements specified by the Domain Expert are put into the defined form. According to the description strategy proposed in Section 3.6 during the domain requirements analysis, there should be several specifications provided. Further, we provide each of them. The first table is the **Requirements as English text.** It is provided by the Domain Expert in the English language (see Figure 8.27).

After the requirements are described in English, the Software Architect and the Domain Expert formulate the **Domain Ontology,** trying to reveal the terms which are going to be operated during design time as well as possible relationships between these terms. Figure 8.29 shows the domain ontology for the House Automation software system.

Further, the **Term-English table** depicted in Figure 8.28 states what each term, defined in the domain ontology, means. The domain ontology statically describes the application domain. To describe the dynamic changes in domain ontology, the Domain Expert and the Software Architect work out the **Actions-State table.**

This is a table that shows the dependencies between sequences of actions that are done when designing a system at the design phase, and the desired changes in the system's state. Actions are defined according to the definition of what the composition process is with externalised business logic, as stated in Section 3.2. Actions are defined as follows:

1. *click component type* - means that the Domain Expert clicks the "component type" which is represented by an icon in the toolbar.

2. *click operation type* - means that the Domain Expert clicks the "operation type" which is represented by an icon in the toolbar.

3. *click component instance* - means that the Domain Expert clicks the "component instance", located in the modelling pane.

Figure 8.30 depicts the Actions-State table.

REQUIREMENTS AS ENGLISH TEXT	
Number	***Requirement(s)***
1	The domain is called „House Automation"
2	The software system consists of components called **Sensors** , **Controllers** and **Actuators** . These are so called virtual instruments.
3	**Sensors** are running virtual instruments that receive signals and forward them to the connected Controllers. Sensors can be of different types. Each signal carries a value which is a number (int). Sensors may emulate the signal.
4	**Controllers** are running functional blocks that receive signals from the connected Sensors. Controllers can be of different types. When a Controller receives signals, it processes them. The act of processing means giving a command to the connected Actuators according to the specified condition. The condition rule may operate with values of incoming signals from the connected Sensors.
5	**Actuators** are running virtual instruments that may receive commands from the connected Controllers. When an Actuator receives a command from the Controller, it starts the corresponding action. The action does nothing, as it is a place-holder to be filled when additional requirements are specified.
6	The domain specific visual language has to be suited for the following tasks: 1) Definition of new types of Sensors, Actuators and Controllers 2) Marking new types of Sensors and Actuators with icons that can be specified by the designer 3) Specification of an emulation mechanism for Sensors (by demand). The minimum and maximum for the generated values can be set. 4) Connecting Sensors and Controllers 5) Connecting Controllers and Actuators 6) Specifying conditions to call a command of connected Actuators for Controllers 7) Specifying a command of a connected Actuator to be called for Controllers 8) Starting a designed system to see a working system as a dos console application

Figure 8.27: Requirements as English text for the House Automation application domain

TERM-ENGLISH TABLE		
Number	***Term***	***Description***
1	`Main Container`	The main container for Sensors, Controllers and Actors
2	`Sensor`	The Sensor virtual instrument. It is characterized by the attributes `Type` and `Icon`. The `Type` attribute denotes the type of the Sensor. The `Icon` attribute denotes the icon of the Sensor.
3	`Controller`	The Controller functional block. It is characterized by the attributes `Type` and `condition`. The `Type` attribute denotes the type of the Controller. The `condition` attribute denotes the condition rule of the Controller.
4	`Actuator`	The Actuator virtual instrument. It is characterized by the attributes `Type` and `Icon`. The `Type` attribute denotes the type of the Actuator. The `Icon` attribute denotes the icon of the Actuator.
5	`Emulator`	The Emulator of input signals for the Sensor. It is characterized by the attributes `Max` and `Min`. The `Max` attribute denotes the maximal value that can be generated. The `Min` attribute denotes the minimal value that can be generated.
6	`Action`	A possible reaction of an Actuator to the commands sent by connected Controllers. The attribute name denotes the name of an action.
7	`Signal`	Each `Signal` represents a connection channel between a connected Sensor and Controller.

Figure 8.28: Term-English table for the House Automation application domain

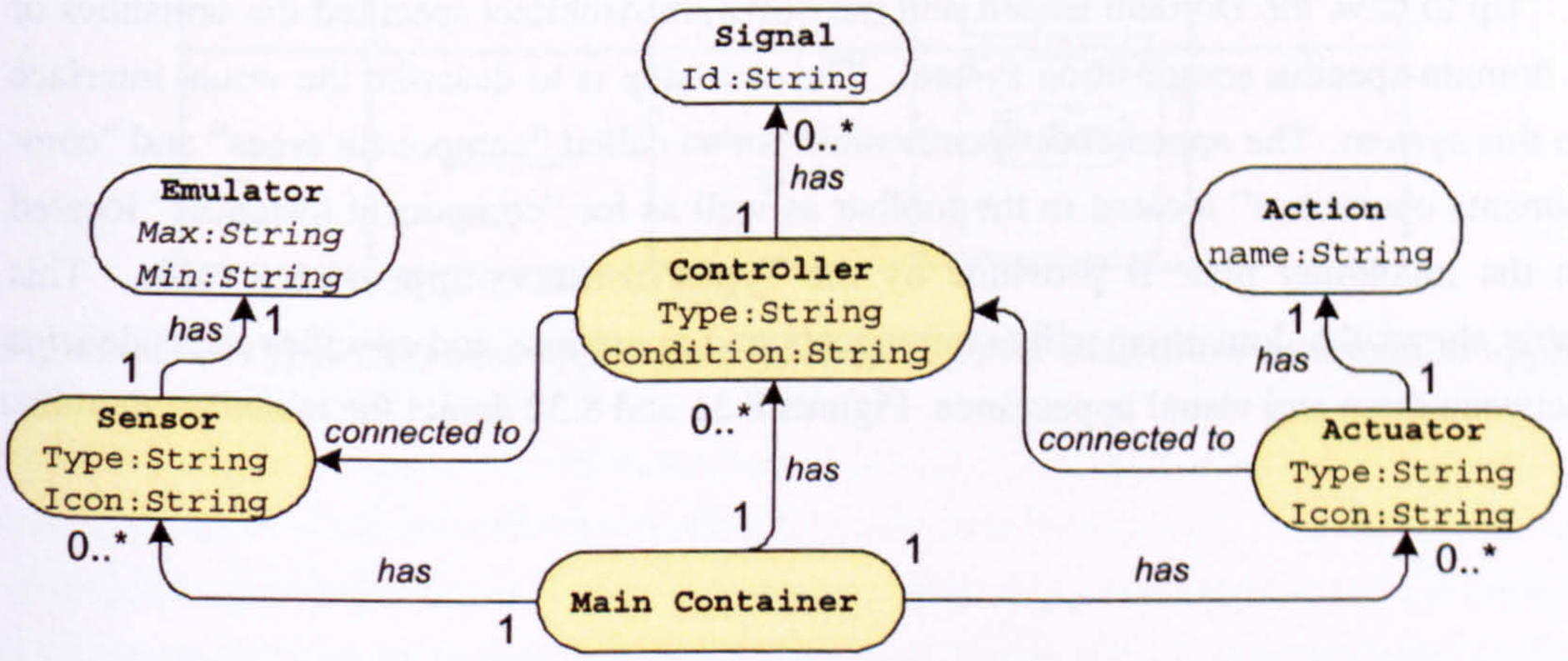


Figure 8.29: Domain Ontology for the House Automation application domain

ACTIONS-STATE TABLE			
Number	***Action***	***State 1***	***State 2***
1	(1) click component type [Sensor] (2) click instance [Main Container]	[Main Container] [Sensor]	[Main Container] **-has->** [Sensor]
2	(1) click component type [Controller] (2) click instance [Main Container]	[Main Container] [Controller]	[Main Container] **-has->** [Controller]
3	(1) click component type [Actuator] (2) click instance [Main Container]	[Main Container] [Actuator]	[Main Container] **-has->** [Actuator]
4	(1) click component type [Emulator] (2) click instance [Sensor]	[Sensor] [Emulator]	[Sensor] **-has->** [Emulator]
5	(1) click component type [Action] (2) click instance [Actuator]	[Actuator] [Action]	[Actuator] **-has->** [Action]
6	(1) click operation type [ConnectSC] (2) click instance[Sensor] (3) click instance[Controller]	[Sensor] [Controller]	[Controller] **-connected to->** [Sensor], [Controller] **-has->** [Signal]
7	(1) click operation type [ConnectCA] (2) click instance[Controller] (3) click instance[Actuator]	[Controller] [Actuator]	[Actuator] **-connected to->** [Controller]

Figure 8.30: Actions-State table for the House Automation application domain

Up to now, the Domain Expert and the Software Architect specified the semantics of a domain-specific composition system. The next step is to describe the visual interface to this system. The appearance specification for so called "component types" and "components operations" located in the toolbar as well as for "component instances" located in the modelling pane is provided by the **Types/Instances-appearance table**. This table shows the domain-specific components and operations and specifies dependencies between them and visual appearance. Figures 8.31 and 8.32 depict the table.

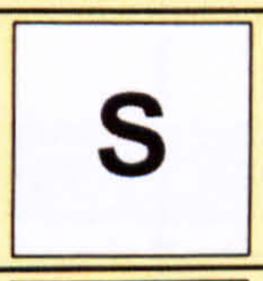
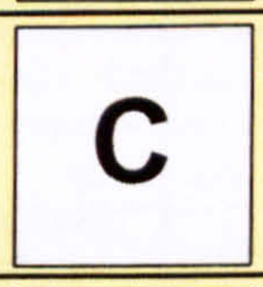
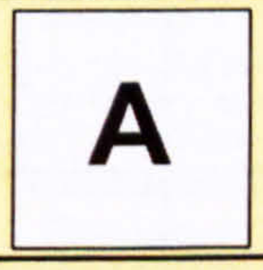
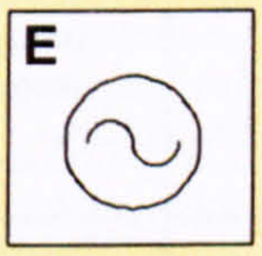
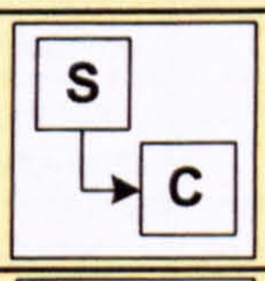
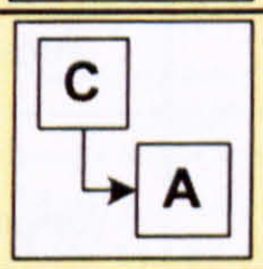
TYPES/INSTANCES-APPEARANCE TABLE				
Number	***Element***	***Kind***	***Appearance***	***Description***
1	[Sensor]	component type		It is an icon
2	[Controller]	component type		It is an icon
3	[Actuator]	component type		It is an icon
4	[Emulator]	component type		It is an icon
5	[ConnectSC]	operation type		It is an icon
6	[ConnectCA]	operation type		It is an icon
7	[Action]	component type		It is an icon

Figure 8.31: Types/Instances-appearance table (part 1) for the House Automation application domain

TYPES/INSTANCES-APPEARANCE TABLE				
Number	***Element***	***Kind***	***Appearance***	***Description***
8	[Sensor]	component instance	<Type> → S → 1 <Icon>	This is a container. It shows values of the Sensor's attributes <Type> and <Icon> (as loaded picture). The placement area 1 is for Emulator.
9	[Controller]	component instance	<Type> → C → 2 1	This is a container. It shows values of the Controller's attributes <Type> and <condition>. The 1 denotes the placement area for <condition>. The placement area 2 marks the approximate location of [Signal] components
10	[Actuator]	component instance	<Type> → <Icon> A 1	This is a container. It shows a value of the Controller's attribute <Type> and an <Icon> (as loaded picture). The placement area 1 is for [Action] components.
11	[Emulator]	component instance	~	The icon of Emulator.
12	[Action]	component instance	<Text> →	This text field contains the value of the attribute <Text>.
13	[Signal]	component instance	<Name>	This text field contains the value of the attribute <Name>
14	[Main Container]	component instance		It is a container which has black borders and white background.

Figure 8.32: Types/Instances-appearance table (part 2) for the House Automation application domain

From this table, the Software Architect will later create visual components which constitute the domain-specific visual interface, reflecting a designed system's state at the design phase. The Types/Instances-appearance table shows statics rather than the dynamics of a component's appearance. The dynamics are described with the **ORel-GRel table**, which basically holds information about how the change in the system's state influences the domain-specific visual interface. This shows the dependency between relationships defined by the domain ontology and relationships between components of the domain-specific visual interface. The graphical relations were described in Section

7.7.2.1. Currently, there are two types of relationships: "contains" and "links". Figure 8.33 depicts the table.

OREL-GREL TABLE

Number	*Ontology Relation*	*Graphic Relation*
1	[Main Container] **-has->** [Sensor]	[Main Container] **-contains->** [Sensor]
2	[Main Container] **-has->** [Controller]	[Main Container] **-contains->** [Controller]
3	[Main Container] **-has->** [Actuator]	[Main Container] **-contains->** [Actuator]
4	[Sensor] **-has->** [Emulator]	[Sensor] **-contains->** [Emulator]
5	[Controller] **-connected to->** [Sensor]	[Sensor] **-links->** [Controller]
6	[Actuator] **-connected to->** [Controller]	[Controller] **-links->** [Actuator]
7	[Actuator] **-has->** [Action]	[Actuator] **-contains->** [Action]
8	[Controller] **-has->** [Signal]	[Controller] **-contains->** [Signal]

Figure 8.33: ORel-GRel table for the House Automation application domain

The provided specifications are concluded with the overall schematic view (or simulated by tools) of the domain-specific visual composition process. Further, we give this an overall conclusion demonstrating the screenshots of NIP made during the designing of applications in the Console Viewer application domain.

Figures 8.34 and 8.35 depict the schematic representation of a domain-specific visual design that is being requested by the Domain Expert. Each window shows the design step and consists of the toolbar with domain-specific components (and operations), the modelling pane with the visual model of the designed system and the property inspector window to assign values to the attributes of the visual components in the modelling pane.

The View that defines how the state of the designed system is visually represented is called "Device types (statics)".

Initially, the View shows a container with a black line border. This container may contain instances of the domain-specific components. The toolbar contains five component types and two operation types. The component types are (according to sequence in the toolbar): Sensor, Controller, Actuator, Emulator and Action. The two last icons in the toolbar are operation types ConnectSC and ConnectCA.

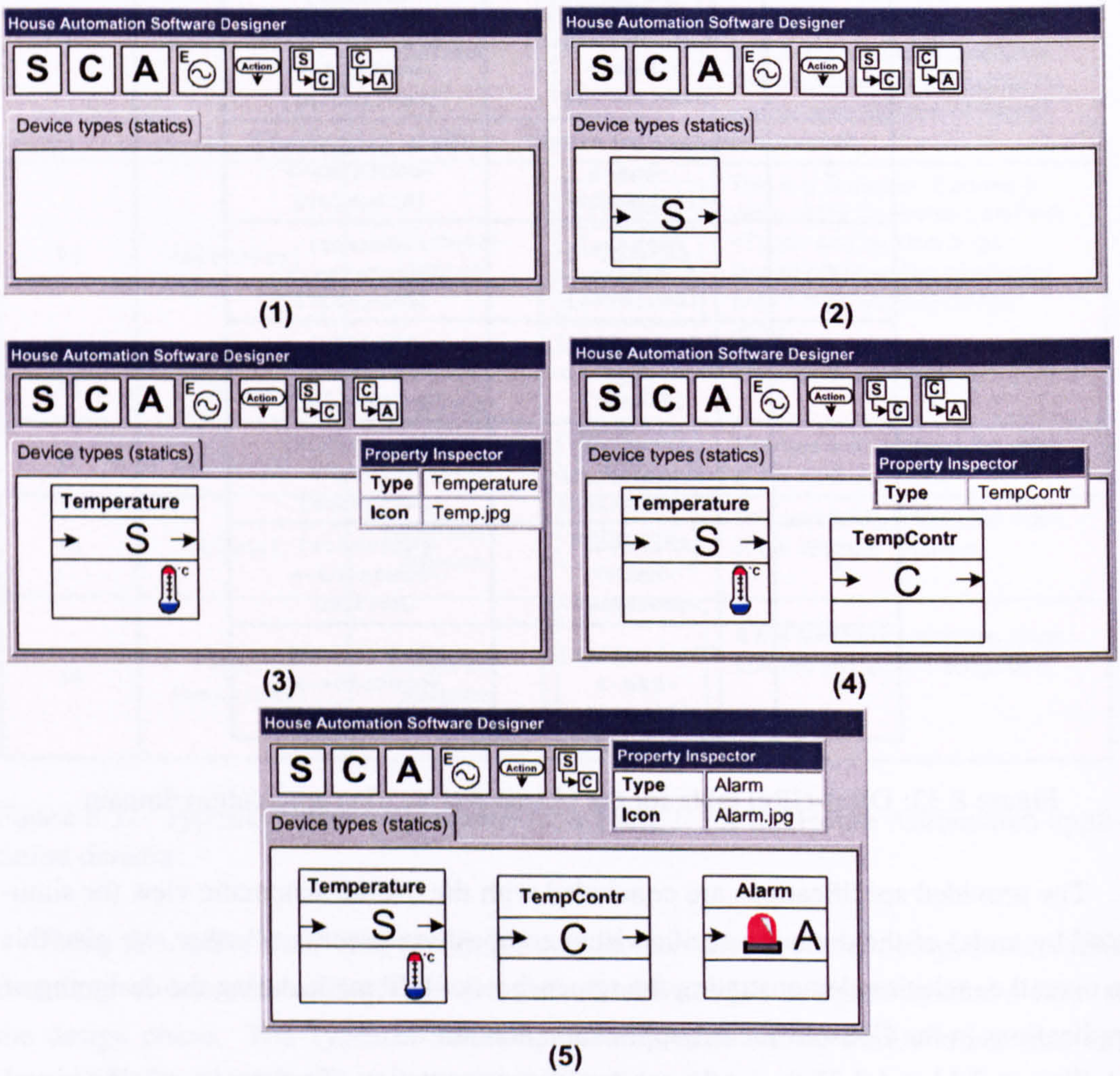


Figure 8.34: Steps (1-5) of designing a House Automation software system

Further, the design steps are briefly described:

1. The modelling pane is empty.

2. The Sensor instance is dropped onto the modelling pane.

3. The attributes of the created instance are edited in the property inspector. The attributes `Type` and `Icon` are assigned with values. According to new values the appearance of the instance is changed automatically. Now, the instance represents a `Temperature` Sensor.

4. The Controller instance is dropped onto the modelling pane. The attribute `Type` is edited in the property inspector. The Controller is now called `TempContr`.

5. The Actuator instance is dropped onto the modelling pane. The attributes `Type` and `Icon` are assigned with values. Now, the Actuator represents the `Alarm` Actuator.

6. Two Action instances are dropped onto the Alarm Actuator. The attribute `name` for both actions is assigned with the value.

7. The operation ConnectSC is applied to the `Temperature` Sensor and `TemprCont` Controller. This operation results in the establishment of a communication channel between them, denoted as `T1`.

8. The attribute condition for the `TempContr` is assigned with the value "T1>100", which means that the Controller will generate a signal to the connected Actuators if the signal coming through the channel `T1` is higher than `100`. Additionally, the operation ConnectCA is applied to the `TemprContr` and `Alarm`. This operation results in the establishment of a communication channel between the `TempContr` and `Alarm`.

9. The Emulator instance is dropped onto the `Temperature` Sensor. This results in the installation of the signal emulation mechanisms.

The result of the design step will be the program code of the House Automation software system. Once started, the system will work as specified by the domain expert. At the design time, the Domain Expert can create any new types of Sensors, Actuators and Controllers and combine them flexibly in order to function with different kinds of hardware devices.

Further, we describe how the domain-specific visual composition system to design described software systems is created by the Software Architect. Afterwards, we demonstrate this system in operation.

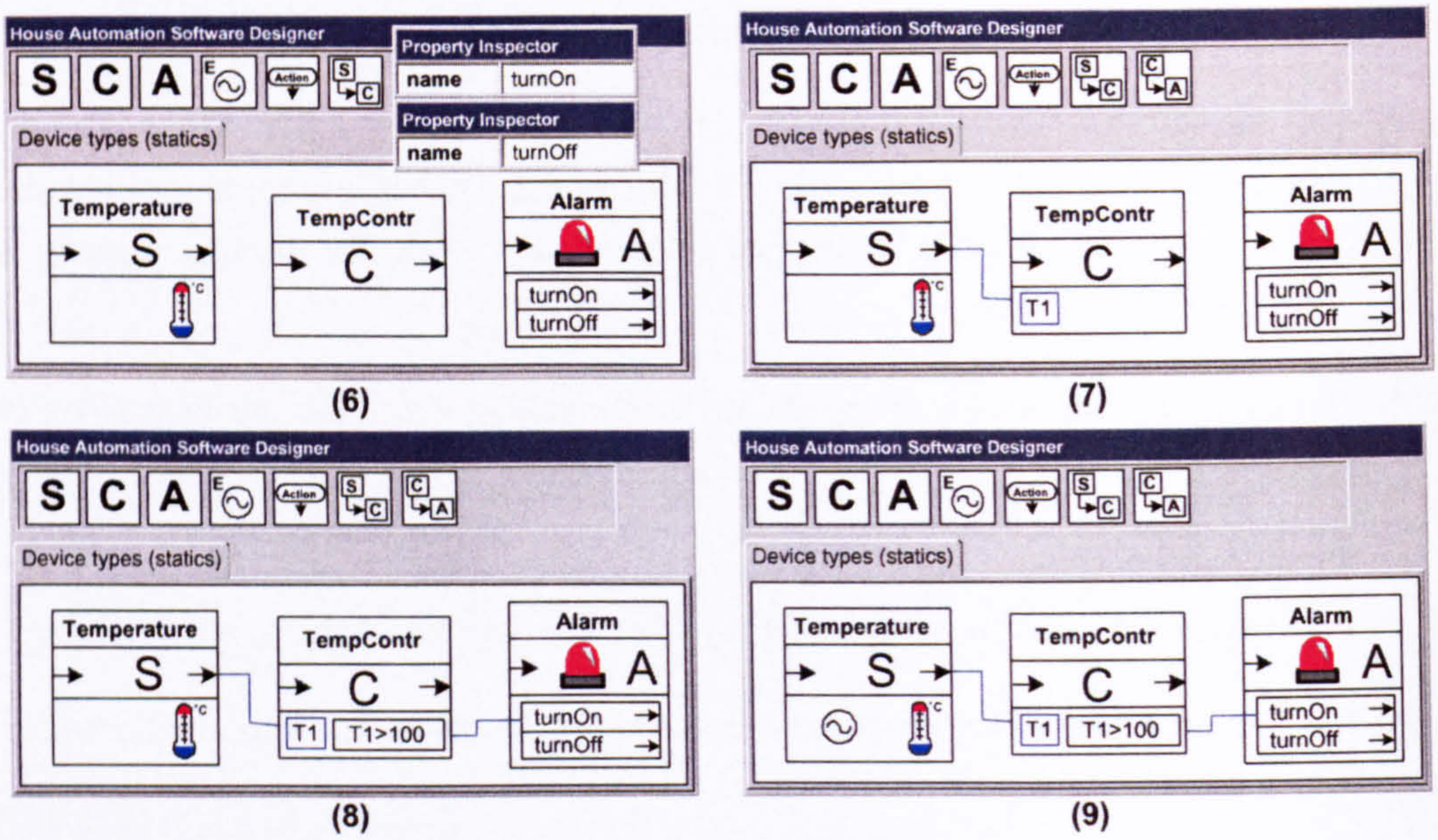


Figure 8.35: Steps (6-9) of designing a House Automation software system

8.5.2 Processing the Domain Requirements

The Software Architect processes the domain requirements in order to create a domain-specific visual composition system. The following steps are done:

1. Specification of Template Composition System - the repository of PCTs and MOs has to be defined.

2. Specification of Domain-specific Composition System - the repository DSCs and DSOs have to be defined.

3. Specification of Domain-specific Visual Interfaces - the Neurath Modelling Components and Views have to be defined.

   Furthermore, we go through these three steps.

8.5.2.1 Specification of Template Composition System

Being based on the requirements defined by the Domain Expert, the Software Architect may choose the repository of templates presented in Figures 8.36 and 8.37. The templates

are implemented according to the Parametric Code Template component model. The values and the structure of program code carried by PCTs are parameterised with parameters of PCTs and molecular operations.

Further, it is shown how each template from the new repository is composed with the existing templates and molecular operations from the common repository. The elements from the common repository are marked with the light red colour. We can see that they are quite often ***reused*** in the context of the House Automation domain. Moreover, non-common templates, which are marked with orange colour, are ***reused*** within the context of the House Automation domain.

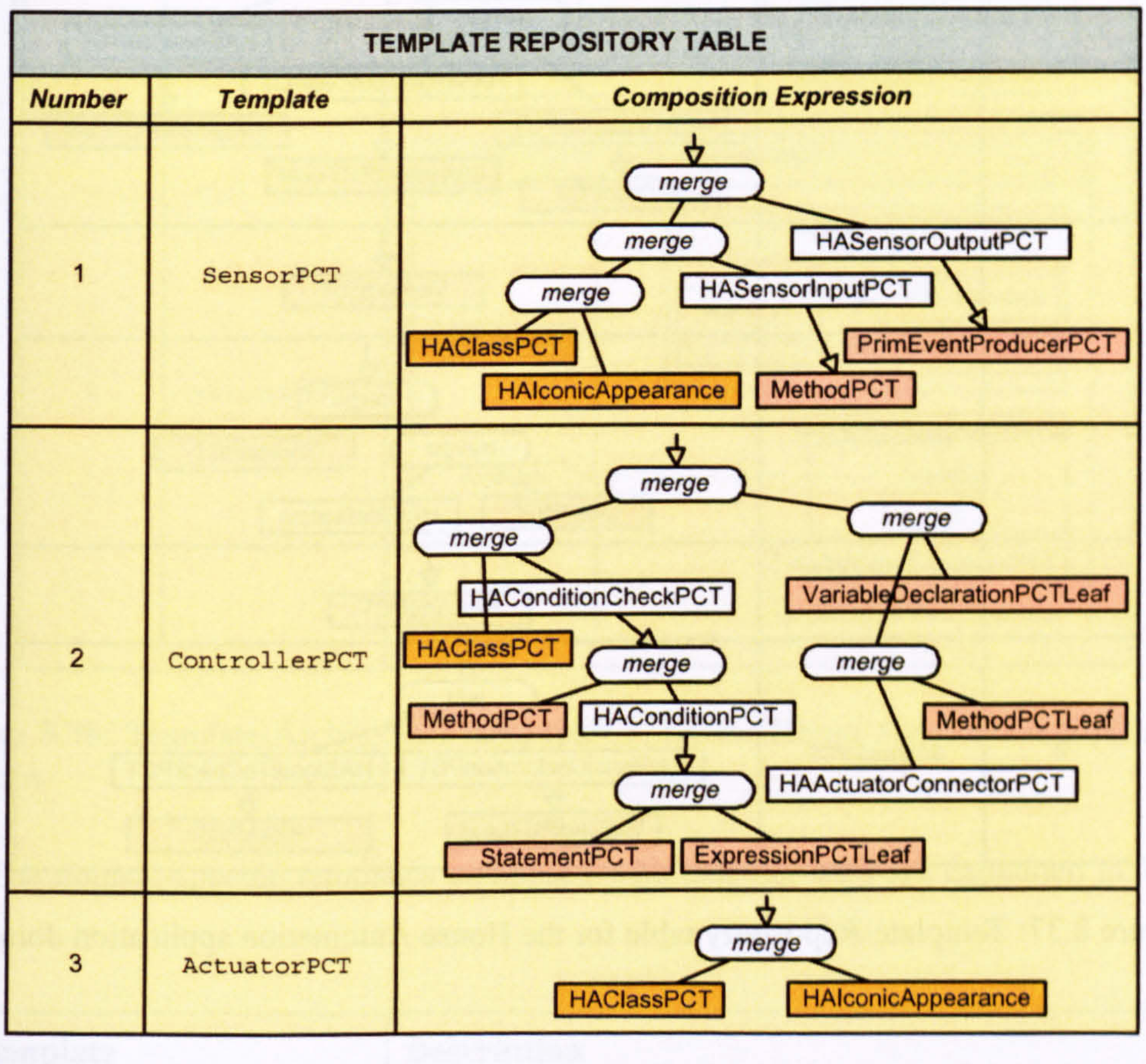


Figure 8.36: Template-Repository table for the House Automation application domain

The arrow between templates (but not between the operation and template) denotes an ***inheritance*** relationship. This relationship provides the feature of an extension of already

defined components. The derived component specify only has to specify the additional logic, and the common logic will be shared.

The composition steps presented in Figures 8.36 and 8.37 can be seen as actions done by a Software Architect working with the Neurath Builder Tool. In the case of the visual interface provided, this process becomes significantly more automatic and easier.

TEMPLATE REPOSITORY TABLE

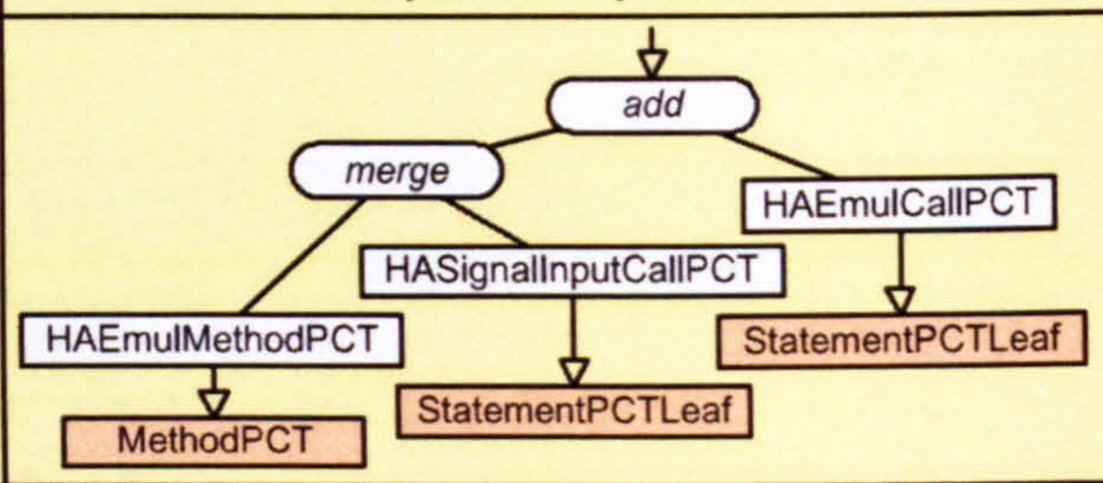
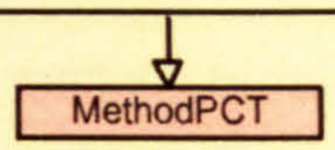
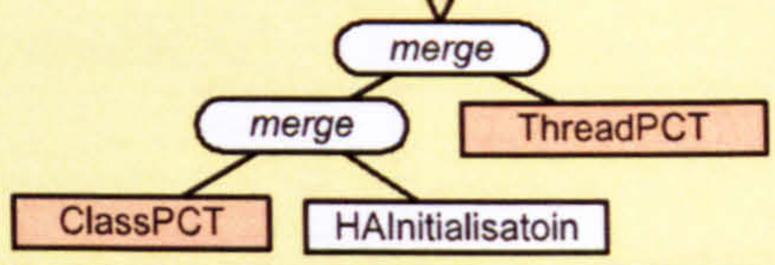
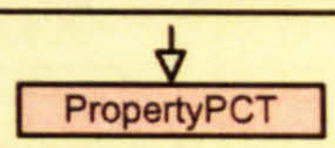
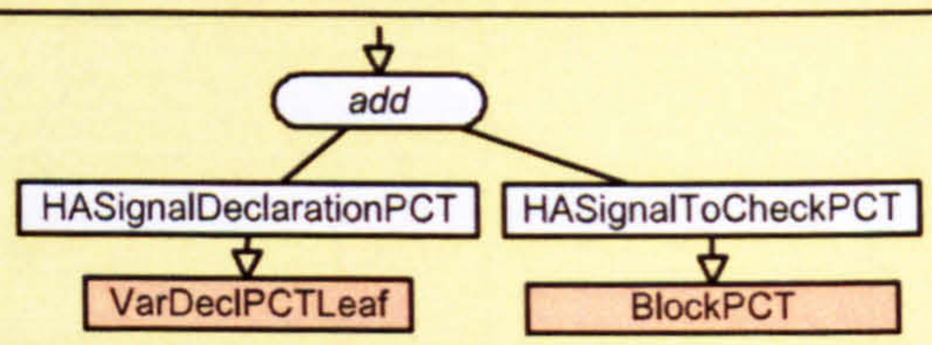
Number	*Template*	*Composition Expression*
4	`EmulatorPCT`	
5	`ActionPCT`	
6	`HAClassPCT`	
7	`HAIconicAp-pearance`	
8	`SignalPCT`	

Figure 8.37: Template-Repository table for the House Automation application domain

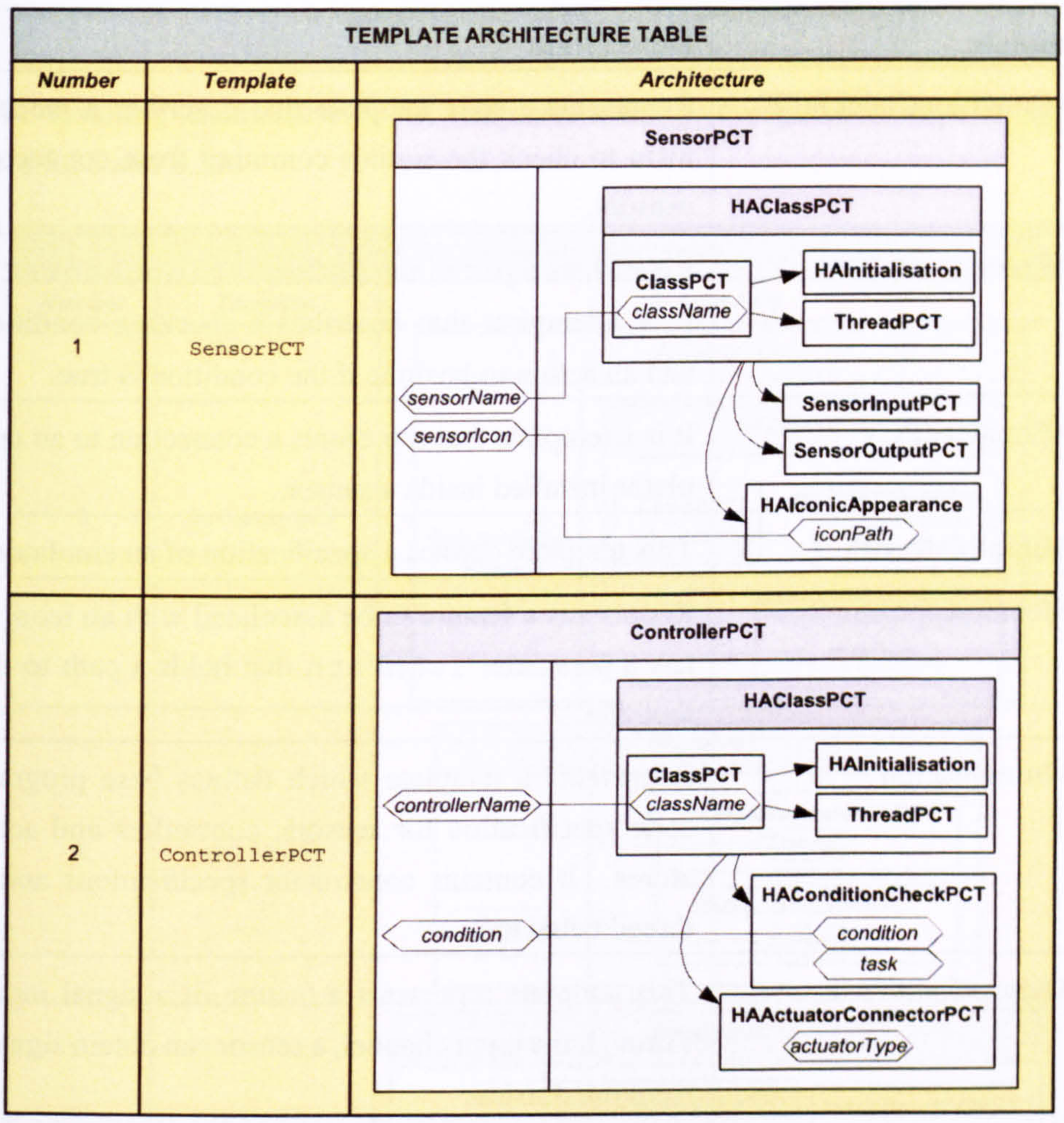


Figure 8.38: Template-Architecture table (part 1) for the House Automation application domain

The domain-specific templates composed with existing ones are explained in Table 8.2.

Template	Description
HAActuatorConnectorPCT	Represents a code template that describes a connection between a controller and an actuator. It has a parameter `actuatorType` that contains the name of the connected actuator.

Template	Description
HAConditionCheckPCT	Represents a code template that describes a mechanism to check the signals comming from connected sensors.
HAConditionPCT	Represents a part of the `HAConditionCheckPCT`. It is a template that describes a checking condition and an action to be done if the condition is true.
HAEmulCallPCT	It is a template that represents a connection to an emulator installed inside a sensor.
HAEmulMethodPCT	This template defines a specification of an emulator.
HAIconicAppearance	Represents a feature to be associated with an icon. It has a parameter `iconPath` that holds a path to the icon file.
HAInitialisation	Represents a template which defines base program code specification for sensors, controllers and actuatores. It contains constructor specifications and a thread behaviour.
HASensorInputPCT	This template represents a feature of a signal input. Through the input channel, a sensor can obtain signals from the outside.
HASensorOutputPCT	This template represents an observer design pattern [36] for sensors and components that may listen for signals of sensors. It defines main features such as a registry service for listeners and a notification mechanism. The template has parameters such as `sensorName` and `interfaceMethodName`
HASignalInputCallPCT	Represents a connection between emulator and a sensor's signal input mechanism.

Table 8.2: The repository of domain-specific templates defined for the case of the House Automation application domain

The resulting architecture of the templates after the composition steps is shown in Figures 8.38 and 8.39.

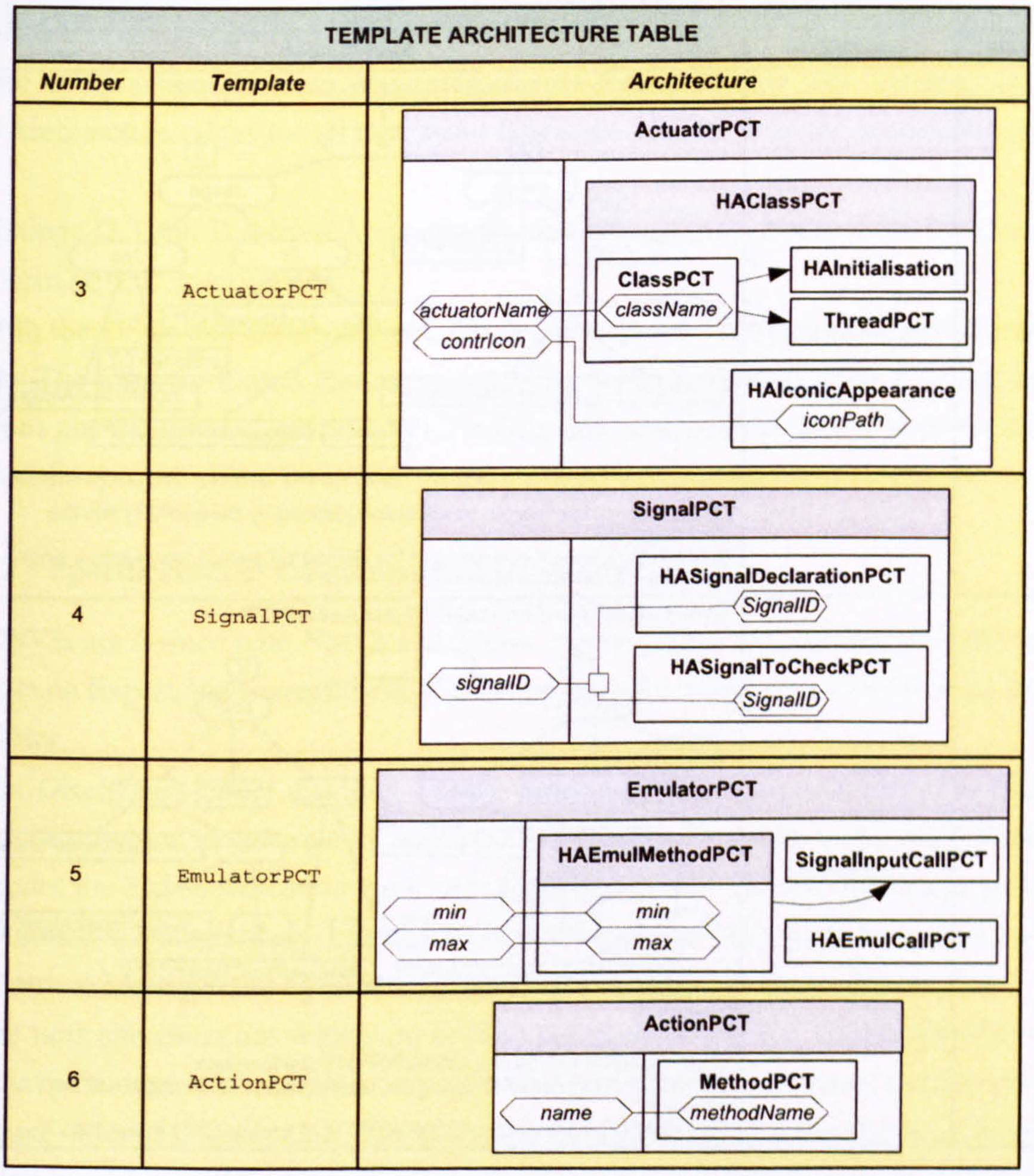


Figure 8.39: Template-Architecture table (part 2) for the House Automation application domain

We presented a program code specification of one PCT. Listing D.1 from Appendix D contains a program code of a class SensorPCT.

The repository of a Template Composition System for the House Automation application domain has two operations. Figure 8.40 depicts specifications of the operations contained in the Operation-Specification table.

OPERATION-SPECIFICATION TABLE

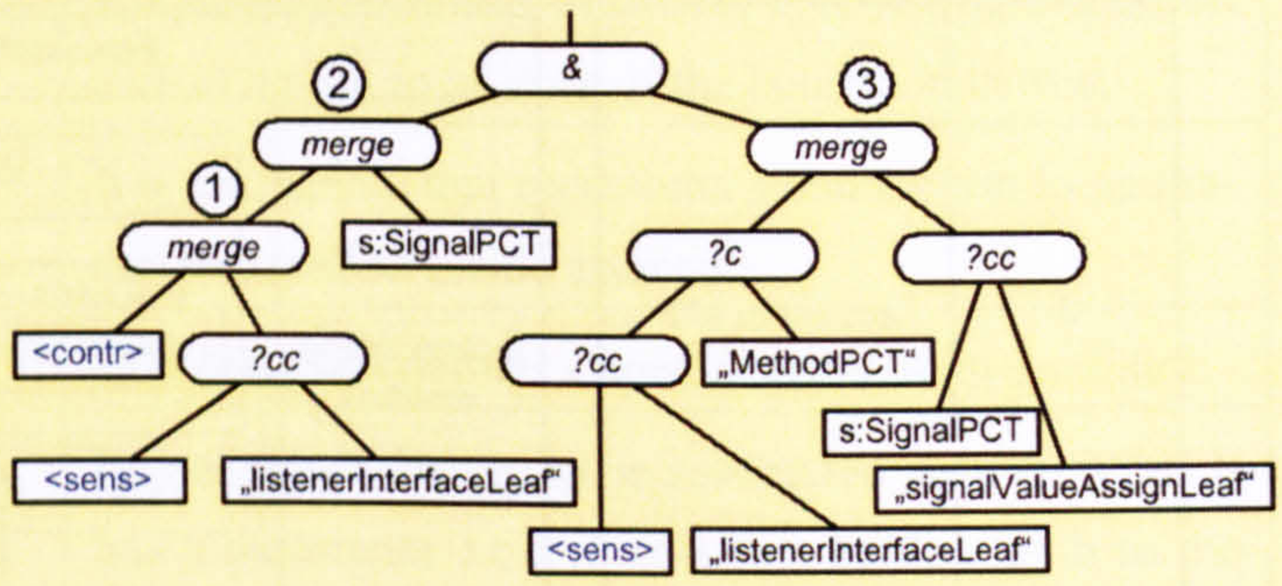
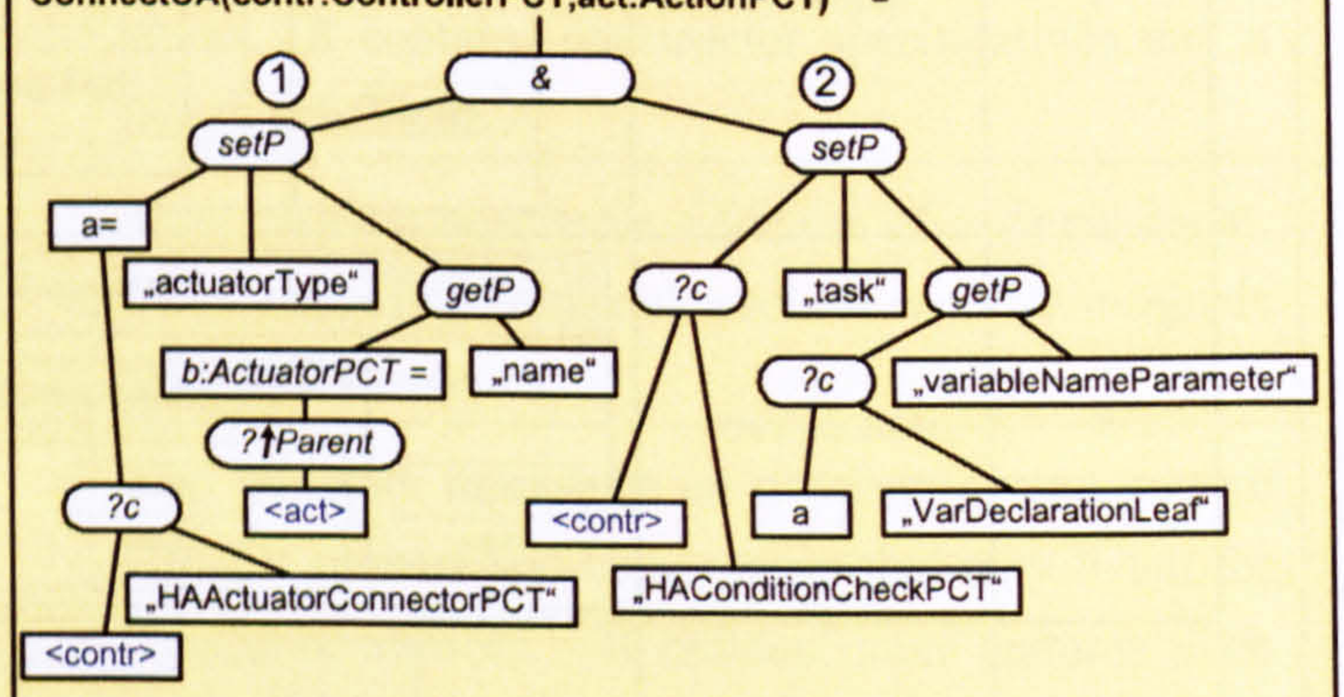
Number	*Operation*	*Specification*
1	`ConnectSC`	`ConnectSC(sens:SensorPCT,contr:ControllerPCT)` =  (1) Injection of the listener specification (defined by SensorPCT) into the ControllerPCT (2) and (3) represent a merging of the SignalPCT (which represents a singal channel) into the ControllerPCT.
2	`ConnectCA`	**ConnectCA(contr:ControllerPCT,act:ActionPCT)** =  (1) Initialisation (inside of a ControllerPCT) of a template (HAActuatorConnectorPCT) which is responsible for connection with an actuators (2) Initialisation (inside of a ControllerPCT) of an Actuator's action call

Figure 8.40: Operation-Specification table for the House Automation application domain

The `ConnectCS` operation demonstrates the feature ***automation of the design pattern*** known as Observer [36]. Listing D.2 in Appendix D has program code specifications of the `ConnectCS` operation.

8.5.2.2 Specification of Domain-specific Composition System

The DSCs and DSOs are defined according to the information that is mainly provided in the Action-State table (see Figure 8.30).

The Action-State table shows the semantic of actions done by the designer at the design phase. It is also observes which components and operations take part in the domain-specific composition process. Figures 8.18 and 8.42 show the DSC-Architecture and DSO-Architecture tables for all DSCs and DSOs recognised from the requirements analysis.

Listings D.3 and D.4 from Appendix D contain a program code of the `SensorDSC` and `ConnectSC_DSO` classes.

With the DSCs and DSOs defined, the domain-specific composition system is established. The Domain Expert can already design software systems with it, using textual notations and the domain-specific API. Further, this composition system is extended with the domain-specific visual interface.

8.5.2.3 Specification of Domain-specific Visual Interfaces

The DSVIs are formed with NMCs and Views. According to the requirements defined by the Domain Expert, the Containment Manager, Symbolic Manager and NMCs are defined as follows.

The ORel-GRel table, resulting during the domain requirements analysis, contains enough information to generate a Containment Manager. To create a Symbolic Manager, we require the SM-specification table shown in Figure 8.43. Having the values specified in the table it is enough to generate the Symbolic Manager. Listings D.5 and D.6 for both Containment Manager and Symbolic Manager can be found in Appendix D.

The next specifications which are needed are those of NMCs. These visual components have the same skeleton, but vary depending on the graphic library used to perform the visualisation and interaction. The basic information about how the skeletons of NMCs are filled in with the library specific code is shown in the NMC-Specification tables depicted in Figures 8.44, 8.45, 8.46, 8.47, 8.48 and 8.49.

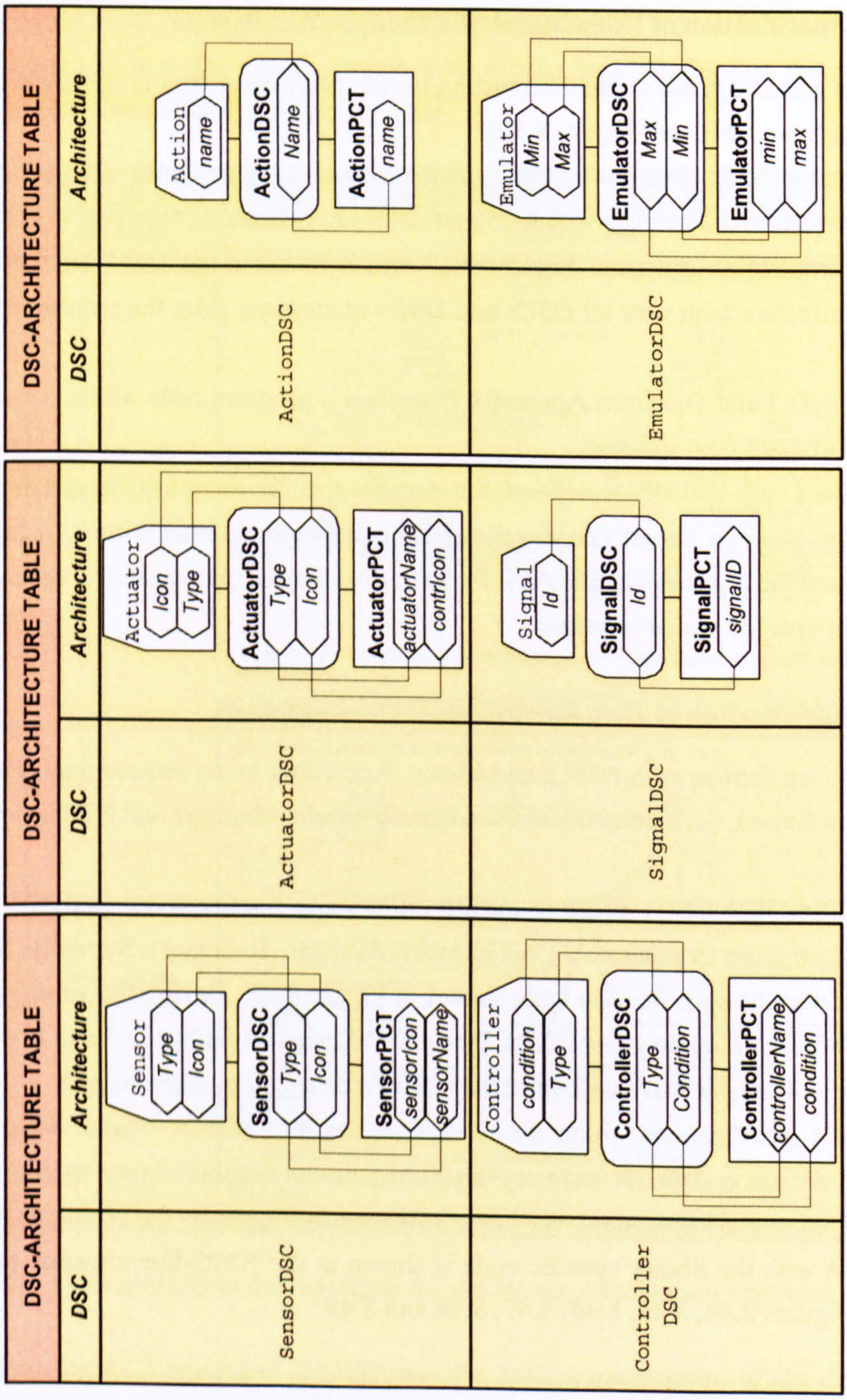


Figure 8.41: DSC-Architecture table for the House Automation application domain

DSO-SPECIFICATION TABLE

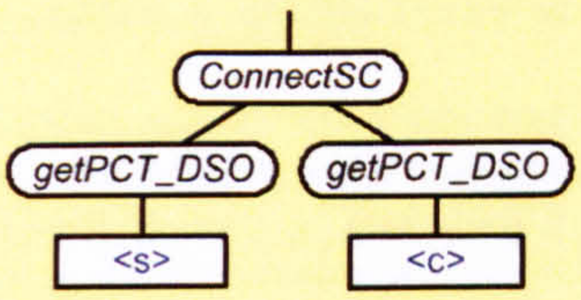
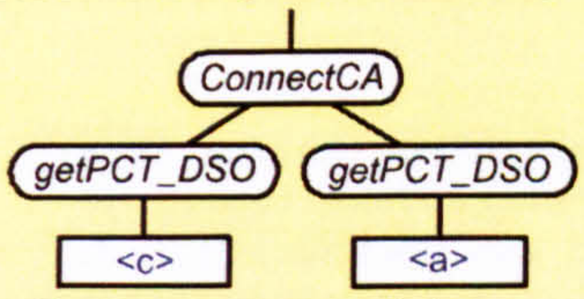
DSO	*Specification*
ConnectSC_DSO	ConnectSC_DSO(s:SensorDSC,c:ControllerDSC) = (1)  (2) SkwNode sNode = SkwContext.searchForNodeById(s); SkwNode cNode = SkwContext.searchForNodeById(c); SkwContext.createRelation(„connected to", cNode, sNode);
ConnectCA_DSO	**ConnectCA_DSO(c:ControllerDSC, a:ActionDSC) =** (1)  (2) SkwNode cNode = SkwContext.search(c); SkwNode aNode = SkwContext.search(a); SkwContext.createRelation(„connected to", aNode, cNode);

Figure 8.42: DSO-Architecture table for the House Automation application domain

SM-Specification Table

Name	*Part*	*Visual Components*
HASymbolicManager	(1) Library-specific container	javax.swing.JPanel
	(2) Term [Main Container]	MainContainerGUI
	(3) Term [Sensor]	SensorGUI
	(2) Term [Controller]	ControllerGUI
	(3) Term [Actuator]	ActuatorGUI
	(3) Term [Property]	PropertyGUI
	(2) Term [Emulator]	EmulatorGUI
	(3) Term [Action]	ActionGUI

Figure 8.43: SM-Specification table for the House Automation application domain

NMC-Specification of SensorGUI

```
public class SensorGUI extends MoveableContainer implements
NCFVisualElement {
  protected ViewNode carriedNode = null;
    ...
    private JPanel bottomContainer = new JPanel();
    private JLabel typeLabel = new JLabel();
    private JLabel iconLabel = new JLabel();
    private BorderLayout borderLayout2 = new BorderLayout();
    ...
    public SensorGUI( ViewNode node ) {
        super();
        setViewNode(node);
        addMouseListener(this);
        ...
    }
    public void setType(String type){
        typeLabel.setText(type); }
    public void setIcon(String icon){
        iconLabel.setIcon(new ImageIcon(icon));
        iconLabel.setText(""); }
    public void setEmulator(EmulatorGUI emulator){
        bottomContainer.add(emulator, BorderLayout.CENTER);}
    public void removeEmulator(EmulatorGUI emulator){
        bottomContainer.remove(emulator); }
    public void mouseReleased(MouseEvent e){
        ...     }
    public void updateUIAccSkwNode(){
        SkwNode node = getViewNode().getSkwNode();
        setType((String)node.getAttributeValue("Type"));
        setIcon((String)node.getAttributeValue("Icon"));    }
    ...
    public void addContainee(NCFVisualElement element){
     if (element.getClass().getName().equals("EmulatorGUI"))
                        setEmulator((EmulatorGUI)element);
     ... }
    public void removeContainee(NCFVisualElement element){
     if (element.getClass().getName().equals("EmulatorGUI"))
                    removeEmulator((EmulatorGUI)element);
     ... }
    public NCFVisualElement getParentContainer(){
        return (NCFVisualElement)super.getParent();
    }}
```

GUI-Library-specific code for appearance
NCF Skeleton
GIU-Library-specific code for interaction
Rest of code

Figure 8.44: NMC-Specification for the SensorGUI component

NMC-Specification of `ControllerGUI`

```
public class ControllerGUI extends MoveableContainer implements
NCFVisualElement {
  protected ViewNode carriedNode = null;
    ...
    private JLabel typeLabel = new JLabel();
    private JTextArea conditionArea = new JTextArea();
    private JPanel signalContainer = new JPanel();
    private int signalCounter=0;
    ...
    public ControllerGUI( ViewNode node ) {
        super();
        setViewNode(node);
        addMouseListener(this);
        ...}
public void setType(String type){
        typeLabel.setText(type); }
public void setCondition(String condition){
    conditionArea.setText(condition); }
public void addSignal(SignalGUI signal){
    ...
    signalContainer.add(signal);
    ...  }
public void removeSignal(SignalGUI signal){
    ...
    signalContainer.remove(signal);
    ...   }
    public void mouseReleased(MouseEvent e){
       ... }
    public void updateUIAccSkwNode(){
    SkwNode node = getViewNode().getSkwNode();
    setType((String)node.getAttributeValue("Type"));
      setCondition((String)node.getAttributeValue("condition"));    }
    ...
    public void addContainee(NCFVisualElement element){
      if (element.getClass().getName().equals("SignalGUI"))
                        addSignal((SignalGUI)element);
      ... }
    public void removeContainee(NCFVisualElement element){
      if (element.getClass().getName().equals("SignalGUI"))
                  removeSignal();
    ...   }
public NCFVisualElement getParentContainer(){
    return (NCFVisualElement)super.getParent();
}}
```

GUI-Library-specific code for appearance
GUI-Library-specific code for interaction
NCF Skeleton
Rest of code

Figure 8.45: NMC-Specification for the ControllerGUI component

NMC-Specification of `ActuatorGUI`

```
public class ActuatorGUI extends MoveableContainer implements
NCFVisualElement {
  protected ViewNode carriedNode = null;
    ...
    private JPanel actionContainer = new JPanel();
    private JLabel typeLabel = new JLabel();
    private JLabel iconLabel = new JLabel();
    private int actionsCounter=0;
    ...
    public ActuatorGUI( ViewNode node ) {
        super();
        setViewNode(node);
        addMouseListener(this);
        ... }
    public void setType(String type){
        typeLabel.setText(type);  }
    public void setIcon(String icon){
        iconLabel.setIcon(new ImageIcon(icon));
        iconLabel.setText(""); }
    public void addAction(ActionGUI action){
        ...
        actionContainer.add(action);
        ... }
    public void removeAction(ActionGUI action){
        ...
        actionContainer.remove(action);
        ... }
    public void mouseReleased(MouseEvent e){
        ... }
    public void updateUIAccSkwNode(){
        SkwNode node = getViewNode().getSkwNode();
        setType((String)node.getAttributeValue("Type"));
        setIcon((String)node.getAttributeValue("Icon"));   }
    ...
    public void addContainee(NCFVisualElement element){
      if (element.getClass().getName().equals("ActionGUI"))
                        addAction((ActionGUI)element);
      ...   }
    public void removeContainee(NCFVisualElement element){
      if (element.getClass().getName().equals("SignalGUI"))
                    removeEmulator();
        ... }
    public NCFVisualElement getParentContainer(){
        return (NCFVisualElement)super.getParent();
    }}
```

GUI-Library-specific code for appearance
NCF Skeleton
GUI-Library-specific code for interaction
Rest of code

Figure 8.46: NMC-Specification for the ActuatorGUI component

NMC-Specification of `ActionGUI`

```
public class  ActionGUI  implements NCFVisualElement   {
  protected ViewNode carriedNode = null;
    private JLabel actionName = new JLabel();
    ...
    public ActionGUI( ViewNode node ) {
        super();
        setViewNode(node);
        addMouseListener(this);
        ...}
   public void setActionName(String name){
        actionName.setText(name); }
   public void mouseReleased(MouseEvent e){
        ... }
   public void updateUIAccSkwNode(){
        SkwNode node = getViewNode().getSkwNode();
        setActionName((String)node.getAttributeValue("name"));    }
   ...
   public void addContainee(NCFVisualElement element){   }
   public void removeContainee(NCFVisualElement element){   }
   public NCFVisualElement getParentContainer(){
        return (NCFVisualElement)super.getParent();
}}
```

GUI-Library-specific code for appearance

GUI-Library-specific code for interaction

NCF Skeleton

Rest of code

Figure 8.47: NMC-Specification for the ActionGUI component

NMC-Specification of `EmulatorGUI`

```
public class  EmulatorGUI  implements NCFVisualElement   {
  protected ViewNode carriedNode = null;

    ...
    public EmulatorGUI( ViewNode node ) {
        super();
        setViewNode(node);
        addMouseListener(this);
        ...
    }
    public void setActionName(String name){
        actionName.setText(name);
    }
    public void mouseReleased(MouseEvent e){
        ...
    }
    public void updateUIAccSkwNode(){   }
    ...
    public void addContainee(NCFVisualElement element){   }
    public void removeContainee(NCFVisualElement element){   }
    public NCFVisualElement getParentContainer(){
        return (NCFVisualElement)super.getParent();
    }
}
```

GUI-Library-specific code for appearance
NCF Skeleton
GUI-Library-specific code for interaction
Rest of code

Figure 8.48: NMC-Specification for the EmulatorGUI component

NMC-Specification of `SignalGUI`

```
public class  SignalGUI  implements NCFVisualElement   {
  protected ViewNode carriedNode = null;

    private JLabel signalName = new JLabel();
    ...
    public SignalGUI() {
        super();
        setViewNode(node);
        addMouseListener(this);
        ...
}
public void setSignalName(String name){
    signalName.setText(name);
}
public void mouseReleased(MouseEvent e){
    ...
}
public void updateUIAccSkwNode(){
    SkwNode node = getViewNode().getSkwNode();
    setSignalName(node.getAttributeValue("Id"));
}
...
public void addContainee(NCFVisualElement element){   }
public void removeContainee(NCFVisualElement element){   }
public NCFVisualElement getParentContainer(){
    return (NCFVisualElement)super.getParent();
}
}
```

GUI-Library-specific code for appearance
GUI-Library-specific code for interaction
NCF Skeleton
Rest of code

Figure 8.49: NMC-Specification for the SignalGUI component

8.5.2.4 Deployment Descriptors

The specification of the Domain-specific Composition System is generated in the deployment description specification in the form of XML files (Listings are presented in Section D.1.4, from Appendix D). The XML files are generated according to the information provided in the DSVCS-Deployment table depicted in Figure 8.50.

DSVCS-Deployment Table		
Number	***Variable***	***Value***
1	NML_ID_NAME	House Automation
2	DESCRIPTION	The software system may consist of virtual Sensors, Controllers and Actuators. With help of controllers outputs of Sensors and inputs of Actuators can be connected. Controllers specify conditions under which the signal can be sent to connected Actuators.
3	INITIAL_EXPRESSION	Instantiate_DSO(MainContainer)
4	LANGUAGE_ELEMENTS	DSC/MainContainerDSC.xml DSC/SensorDSC.xml DSC/ControllerDSC.xml DSC/ActuatorDSC.xml DSC/EmulatorDSC.xml DSC/ActionDSO.xml DSO/ConnectSC.xml DSO/ConnectCA.xml
5	VIEW	NAME = Device-types (statics) CM = viewSpec/HAContainerManager.cm SM = viewSpec/HASymbolicManager.sm

Figure 8.50: DSVCS-Deployment table for the House Automation application domain

Each DSC and DSO has its own deployment descriptor. Figures 8.51 and 8.52 depict deployment descriptors for all components.

DSC-Deployment Table	
Variable	***Value***
XML_FILE	MainContainerDSC.xml
LANGUAGE_ID	MainContainer
DSC_ELEMENT	/dspct/MainContainerDSC.dsc

DSC-Deployment Table	
Variable	***Value***
XML_FILE	SensorDSC.xml
LANGUAGE_ID	Sensor
DSC_ELEMENT	/dspct/SensorDSC.dsc
TOOLBAR_ICON	/isotypes/SensorIcon.gif

DSC-Deployment Table	
Variable	***Value***
XML_FILE	ControllerDSC.xml
LANGUAGE_ID	Controller
DSC_ELEMENT	/dspct/ControllerDSC.dsc
TOOLBAR_ICON	/isotypes/ControllerIcon.gif

DSC-Deployment Table	
Variable	***Value***
XML_FILE	EmulatorDSC.xml
LANGUAGE_ID	Emulator
DSC_ELEMENT	/dspct/EmulatorDSC.dsc
TOOLBAR_ICON	/isotypes/EmulatorIcon.gif

DSC-Deployment Table	
Variable	***Value***
XML_FILE	ActionDSC.xml
LANGUAGE_ID	Action
DSC_ELEMENT	/dspct/ActionDSC.dsc
TOOLBAR_ICON	/isotypes/ActionIcon.gif

DSC-Deployment Table	
Variable	***Value***
XML_FILE	ActuatorDSC.xml
LANGUAGE_ID	Actuator
DSC_ELEMENT	/dspct/ActuatorDSC.dsc
TOOLBAR_ICON	/isotypes/ActuatorIcon.gif

Figure 8.51: DSC-Deployment tables (part 1) for the House Automation application domain

DSO-Deployment Table	
Variable	***Value***
XML_FILE	ConnectSC.xml
LANGUAGE_ID	ConnectSC_DSO
DSO_ELEMENT	/dspct/ ConnectSC_DSO.dso
TOOLBAR_ICON	/isotypes/ ConnectSCIcon.gif
PARAM_SIGNATURES	Name = s Setter = setS Getter = getS ---------- Name = c Setter = setC Getter = getC

DSO-Deployment Table	
Variable	***Value***
XML_FILE	ConnectCA.xml
LANGUAGE_ID	ConnectCA_DSO
DSO_ELEMENT	/dspct/ ConnectCA_DSO.dso
TOOLBAR_ICON	/isotypes/ ConnectCAIcon.gif
PARAM_SIGNATURES	Name = c Setter = setC Getter = getC ---------- Name = a Setter = setA Getter = getA

Figure 8.52: DSO-Deployment tables (part 2) for the House Automation application domain

When the specification is deployed into the NIP, the design phase starts.

8.5.3 Design Phase

Figures 8.53, 8.54, 8.55 and 8.56 depict the design steps of the Domain Expert in order to create a House Automation software system in the NIP. The figure shows how depending on designer's actions, the state of the designed system and the visual model in the modelling pane are changed.

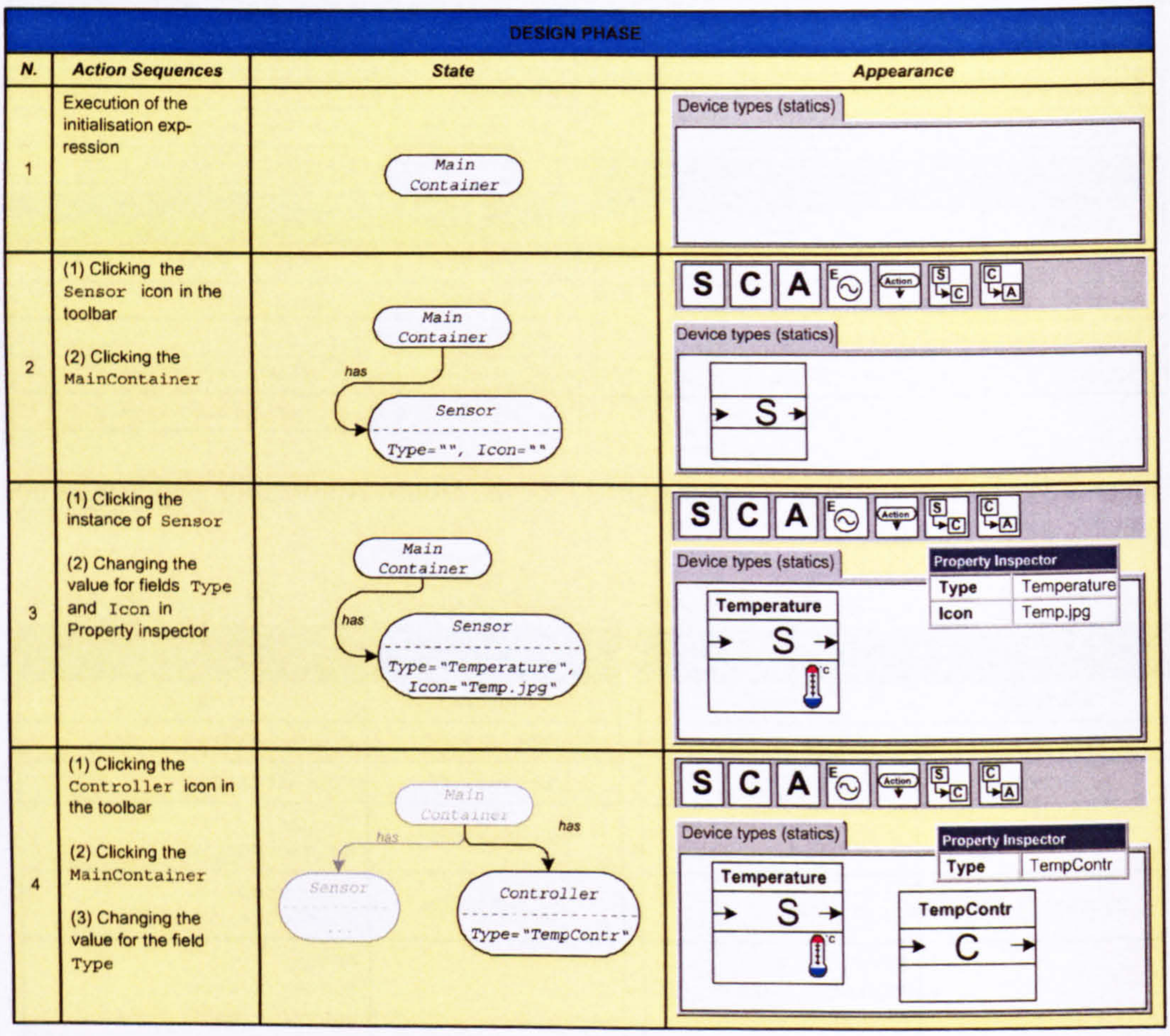


Figure 8.53: Design phase table for the House Automation software system (part 1)

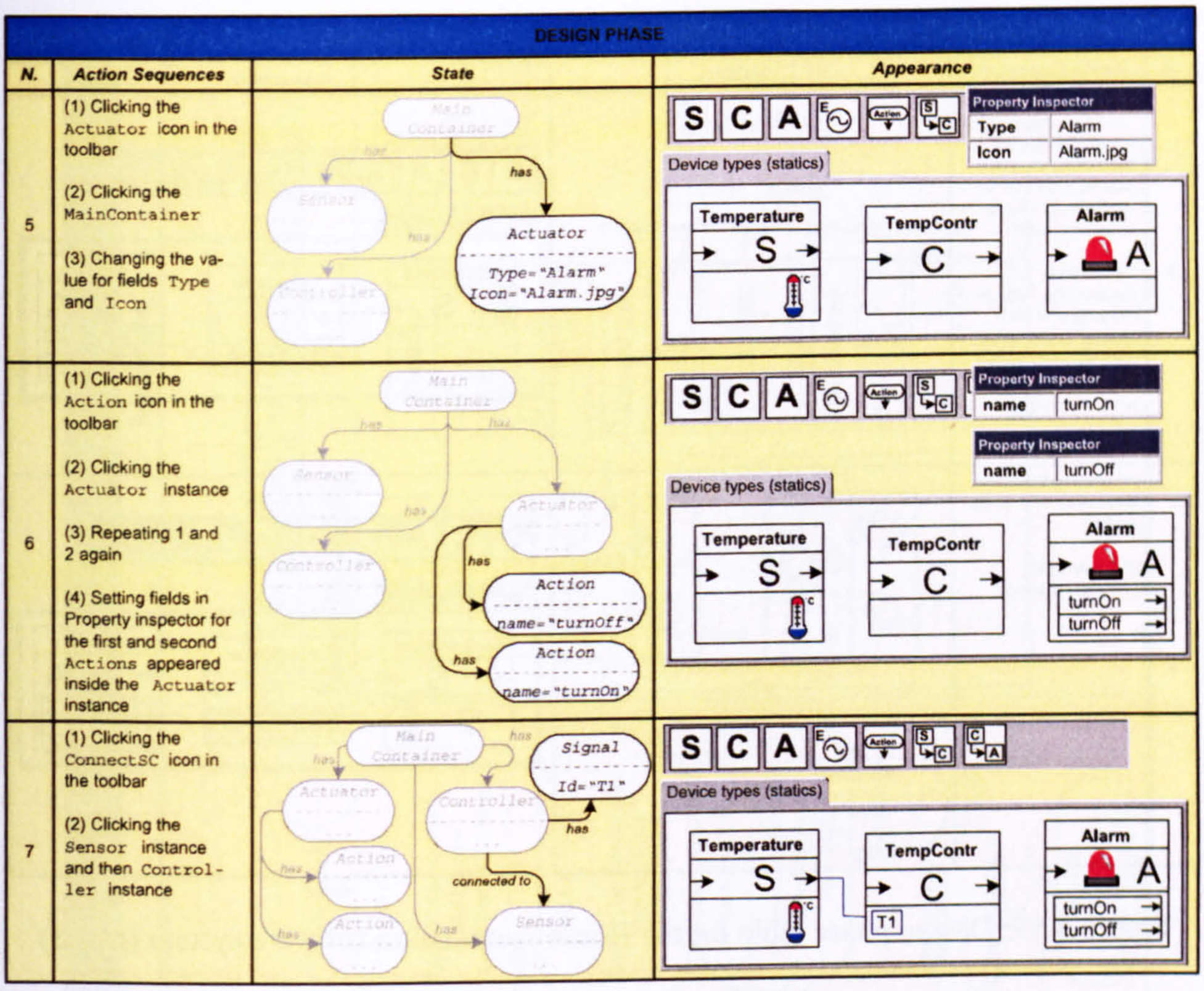
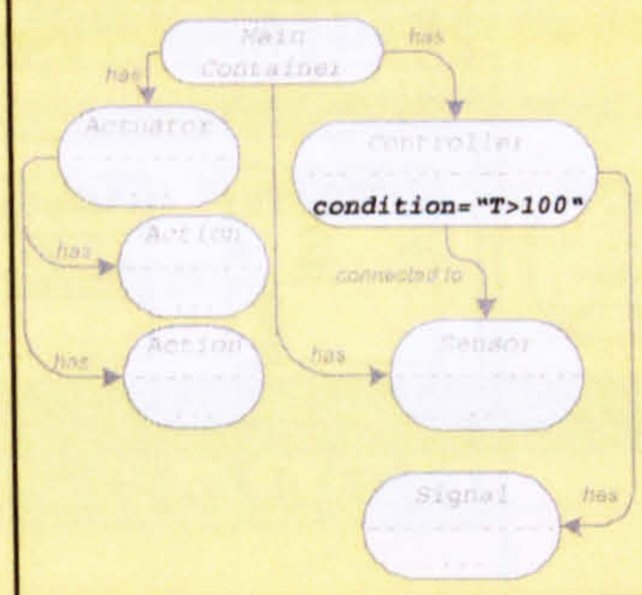
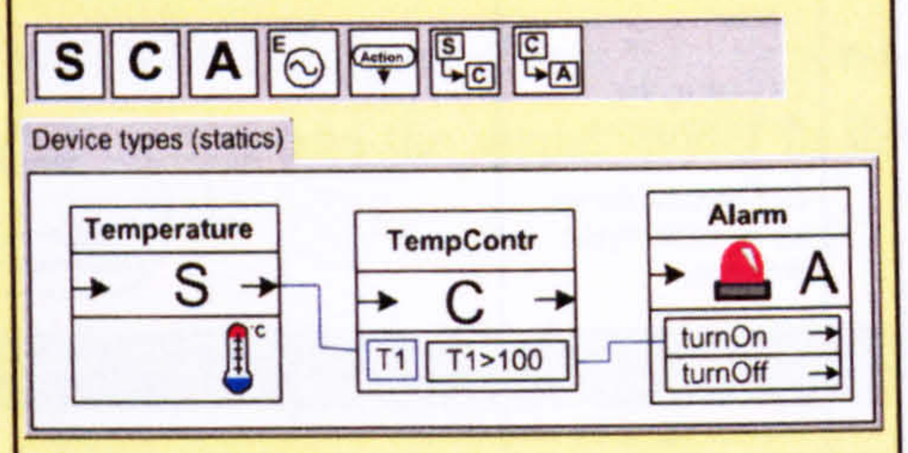


Figure 8.54: Design phase table for the House Automation software system (part 2)

<table>
<tr><th colspan="4">DESIGN PHASE</th></tr>
<tr><th>N.</th><th>Action Sequences</th><th>State</th><th>Appearance</th></tr>
<tr><td>8</td><td>(1) Clicking the <code>Controller</code> instance in the modelling pane and setting the attribute <code>condition</code> in the Property inspector to „<code>T1>100</code>"

(2) Clicking <code>ConnectCA</code> icon in the toolbar, then clicking <code>Controller</code> instance and needed action <code>Action</code> instance inside the <code>Actuator</code></td><td> 
</td><td> 
</td></tr>
<tr><td>9</td><td>(1) Clicking the <code>Emulator</code> icon in the toolbar

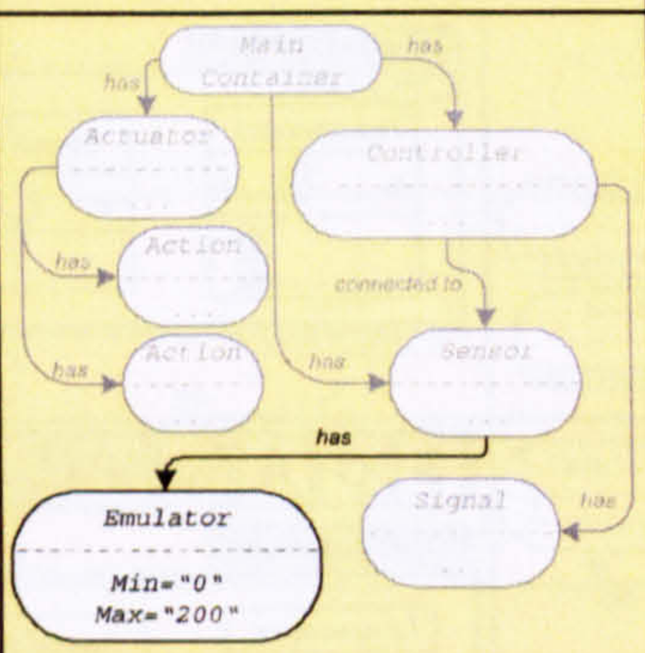
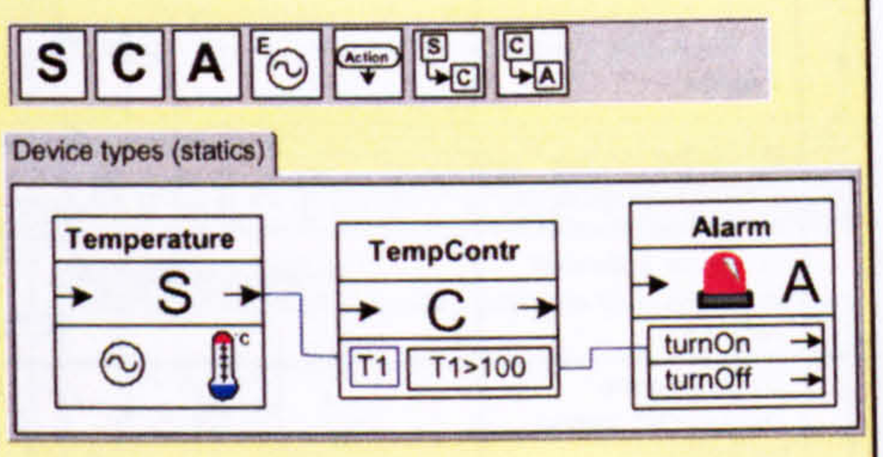
(2) Clicking the <code>Sensor</code> instance</td><td> 
</td><td> 
</td></tr>
</table>

Figure 8.55: Design phase table for the House Automation software system (part 3)

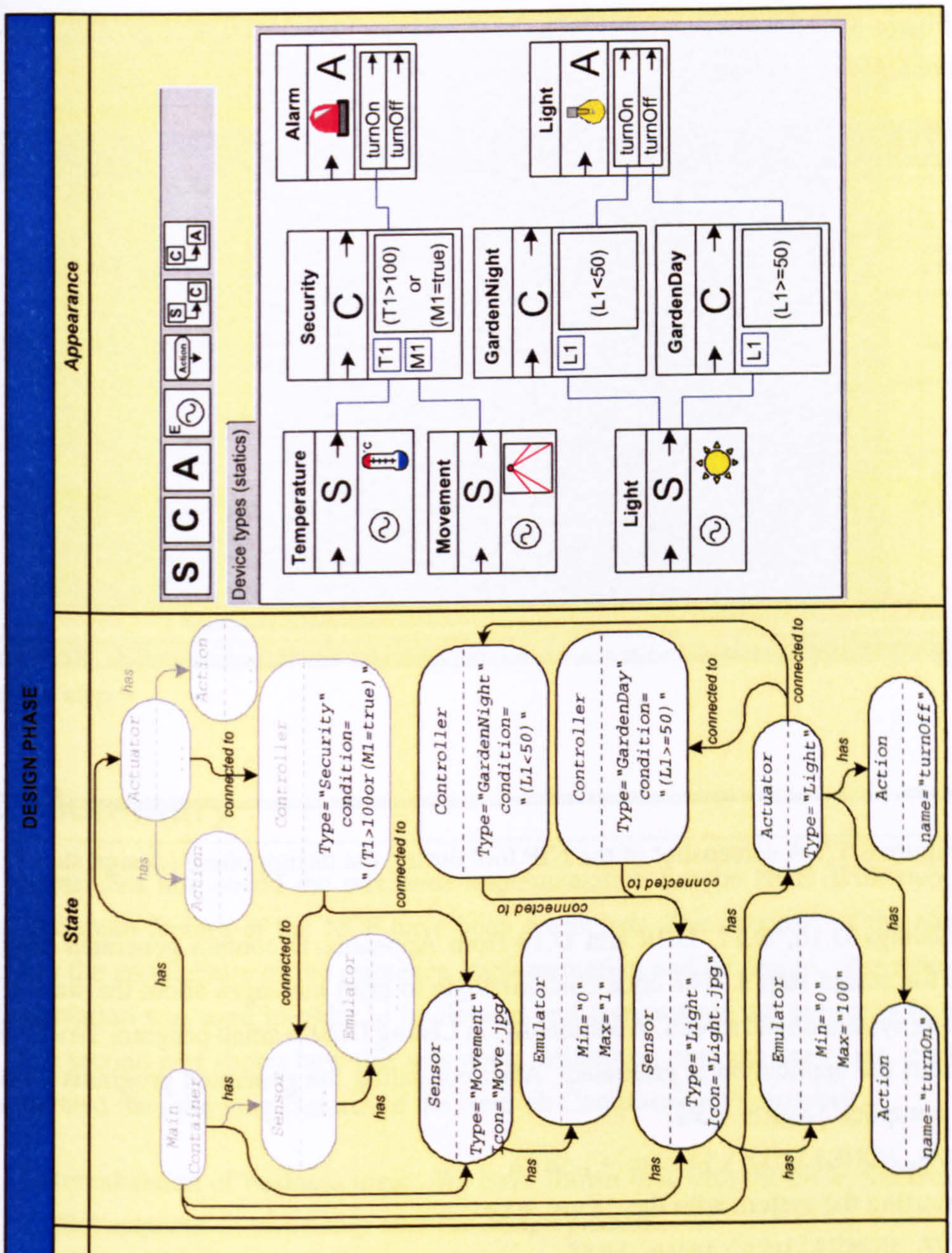


Figure 8.56: Design phase table for the House Automation software system (part 4)

Figure 8.57 illustrates a screenshot of the NIP. It shows design step 9 explained in Figure 8.55.

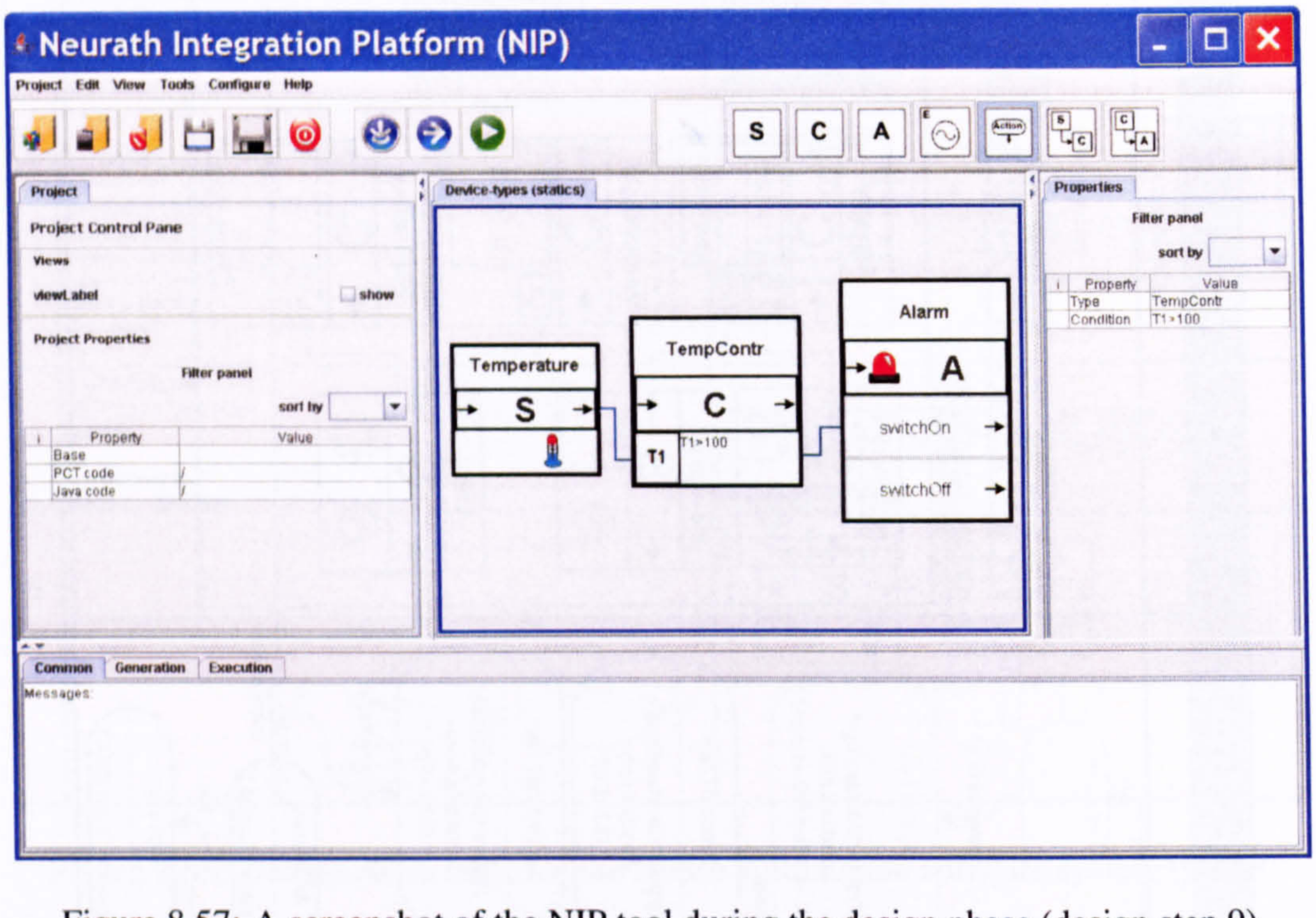


Figure 8.57: A screenshot of the NIP tool during the design phase (design step 9)

Listings D.16, D.17, D.18 and D.19 from Appendix D contain generated program code for design step 9. The code contains logic to print messages about the state of the running system on the screen. Additionally, in Listing D.20, a small program `Test` which can start the application is presented. After compiling the generated programs with the Java compiler `javac.exe`:

```
%JAVA_HOME%\bin\javac *.java
```

and starting the system with the `java.exe`:

```
%JAVA_HOME%\bin\java Test
```

the running system looks as depicted in Figure 8.58.

Figure 8.58 shows a screenshot of the running House Automation software system generated at design step 9.

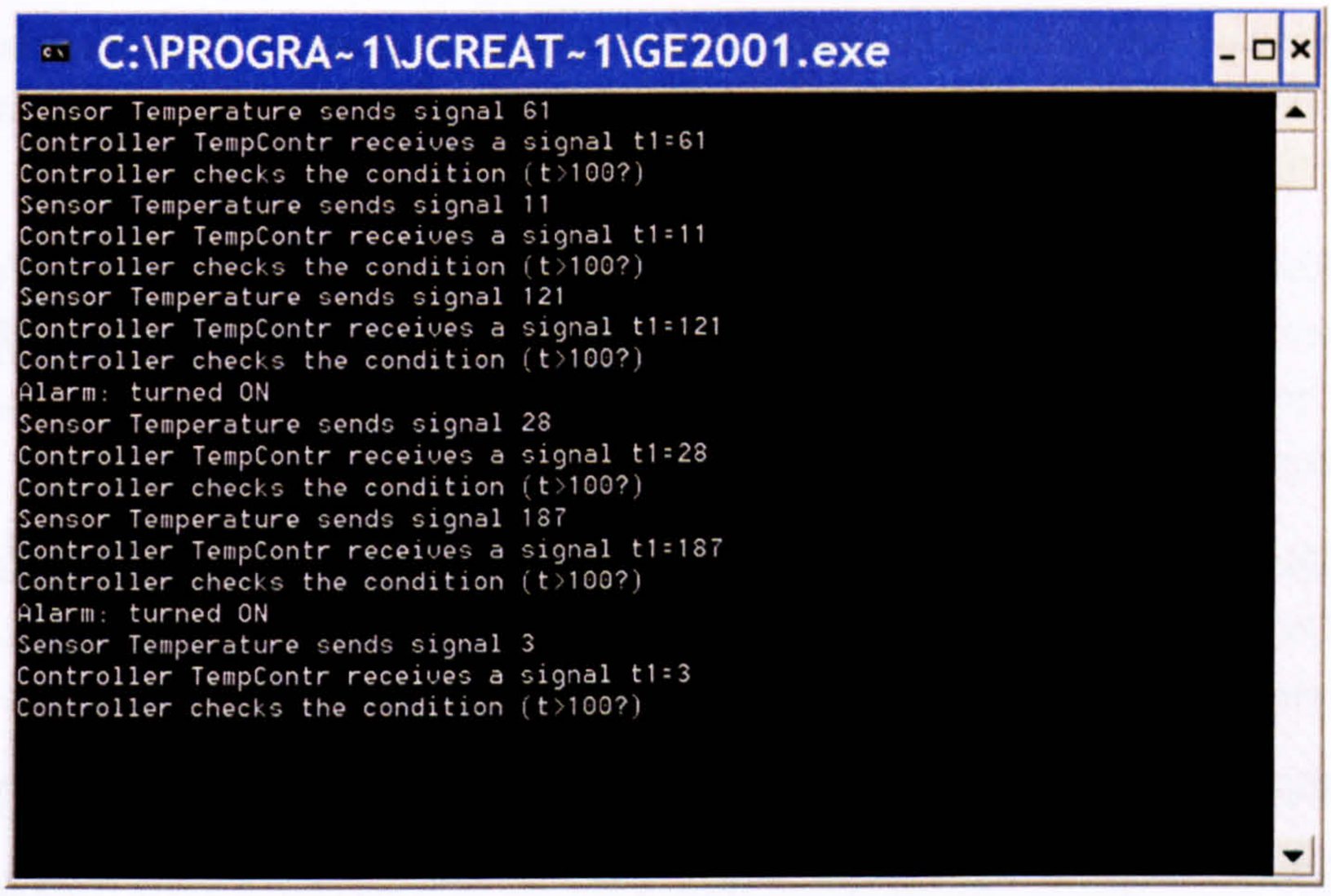


Figure 8.58: A screenshot of the running House Automation software system generated at design step 9

8.6 Summary

The chapter has introduced the reference implementation for the NCF. With two use-cases, the main feature of the NCF have been presented. The first part of the chapter describes the architecture of the reference implementation and its design. The reference implementation was used in order to implement and demonstrate the use-cases in practice. The second part shows how this was done. Moreover, with the use-cases, we have demonstrated the following features of the Neurath Composition Framework:

1. Externalisation of business logic. We have shown that with the NCF, the Domain Expert acquires an ownership over the design process.

2. Comprehensive development of templates. We have shown how the templates can be developed using the inheritance which results in the categorisation of templates and implementation sharing. Moreover, we presented how the templates can be combined with already existing ones.

3. We have demonstrated the feature of awareness of components, when the templates may adapt themselves to the changed environment. This increases adaptability and survivability of the design.

4. Reuse of templates. We have shown how the same templates can be reused for different domain-specific components within one application domain and for different components for two different application domains (which can be generalised for more domains).

5. The grey-box nature of templates was successfully implemented with Parametric Code Templates and Molecular Operations, thereby introducing a high level of parameterisation.

6. We have demonstrated the automation of some design patterns typically used by the Software Architect.

The next chapter provides main conclusions and work to be done in the future.

Chapter 9

Conclusion and Future Work

This chapter gives a summary of the work done in the thesis, highlights the contributions and outlines the future work to extend the contributions and to address some of the criticism.

9.1 Conclusions

Grey-box software composition approaches have been quite popular in academia during the past decade. Compared to black-box approaches, grey-box approaches increase parameterisation of the design and potentially result in the higher reusability, extensibility and adaptability of components (and therefore a system). However, they could not be successfully applied to real tasks in business and industry domains. One of the main reasons for this was that with the parameterisation of the design, the complexity of the development process - and the resulting cost of development - has significantly increased. With respect to this, we have stated the following main research question:
"**How does one create a template-based composition systems that is practically applicable for the end-user, so that even domain experts with no technical background in programming languages could efficiently design software systems for his/her specific domain?**".

To overcome such a gap between grey-box composition systems and the business domain, we have defined a new requirement for composition systems:

Composition systems should externalise business logic

Externalisation can encapsulate the complexity of the template-based design providing a domain-specific interface for domain experts to work with a template composition system.

In this thesis, we have proposed a layered approach called Neurath Composition Framework in order to compose software systems according to well-defined requirements which have been externalised. The NCF describes a strategy to give the ownership over the design to the end-user. The NCF answers the following questions that have been stated in Chapter 1:

1. *What is the strategy to bridge template-based composition systems up to the domain expert level?* The strategy has been defined by the conceptual Neurath Composition Framework in Chapter 3. An architecture of the framework, a terminology and basic requirements to parts of the framework have been defined. The chapter stated the scene for the research.

2. *What is the component model, the composition technique and the composition language of a template-based composition system?* Chapter5 described a component model, a composition technique and a composition language for a template-based composition system. The component model and a composition technique was presented with PCTs and MOs. PCT and MO component models have been defined according to object-oriented technology principles. Together with the introduced wide parameterisation opportunity, this resulted in such useful features as the reuse of templates, event-based awareness of templates as well as categorisation and inheritance of templates. The simple composition language to formulate composition expressions has been presented.

3. *How does one bridge a template-based composition system up to the level of a domain expert?* In Chapter 6, we presented a way to externalise business logic for a template composition system with the help of ontology based descriptions, a domain-specific component model and a domain-specific composition technique. We have defined DSCs and DSOs on top of PCTs and MOs. With DSCs and DSOs, domain experts can form and execute domain-specific expressions that result in template-based transformations of the underlying design.

4. *How has the perception and interaction with a template-based composition system increased?* In Chapter 7, we have specified how the interaction between the Domain Expert and the domain-specific composition systems can be increased with the help of Domain-specific Visual Interface (DSVI). DSVIs enhanced interactivity with the designed system and perception of the design's state.

The main levels of composition for the NCF which also reflect the core architecture of the framework are illustrated in Figure 9.1.

In Chapter 8, we have shown all the concepts which constitute the NCF in practice. We developed and described the reference implementation for the NCF, which we have used to directly demonstrate the main features of the framework. The evaluation has practically shown that the framework meets the defined requirements and answers the main research questions.

Two main general conclusions can be made:

- Template Composition systems are potentially powerful and with the proper tools, they can be usefully and efficiently applied in practice - even by Domain Experts with no technical background in programming languages.

- The Domain Experts can extend domain-specific visual composition systems by introducing new domain-specific components and operations that consist of previously defined ones.

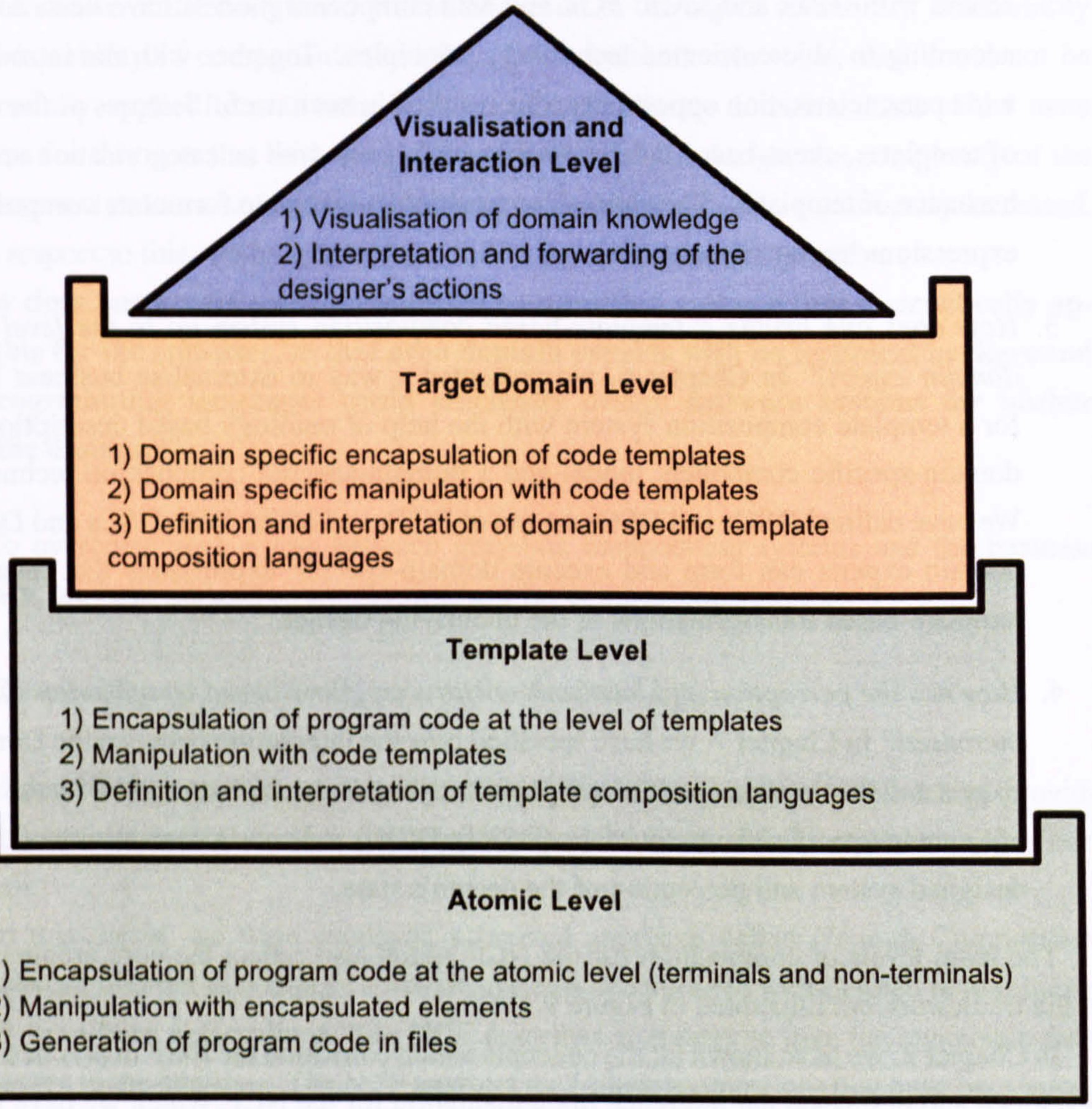


Figure 9.1: Levels of composition within the Neurath Composition Framework

9.2 Practical Realisation

We have developed tools that implement the idea of the Neurath Composition Framework. We have tried to demonstrate the most relevant features of the approach and its practical applicability to real life problems. The following features were demonstrated:

1. Externalisation of business logic. We have shown that with the NCF, the Domain Expert gets an ownership over the design process.

2. Comprehensive development of templates. We have shown how the templates can be developed using the inheritance, which results in the categorisation of templates and implementation sharing. Moreover, we presented how the templates can be combined with already existing ones.

3. We have demonstrated the feature of awareness of components, when the templates may adapt themselves to the changed environment. This increases the adaptability and survivability of the design.

4. Reuse of templates. We have shown how the same templates can be reused for different domain-specific components within one application domain and for different components for two different application domains (which can be generalised for more domains).

5. The grey-box nature of templates was successfully implemented with Parametric Code Templates and Molecular Operations, which introduced a high level of parameterisation.

6. We have demonstrated the automation of some design patterns typically used by the Software Architect.

9.3 Future Work

We consider that this makes the approach very attractive for future investigations. We recognise that the following work is to be done in the future:

1. **Formal underpinning**. The Parametric Code Templates, Molecular Operations and other components defined at different levels of composition require a formal underpinning. Being formalised, they become subjects of deeper analysis and investigation. Existing features of the NCF approach could be soundly proven and more features could be discovered.

2. **Extension of Domain-specific Visual Interface (DSVI)**. We have defined quite simple DSVI that operates with basic concepts regarding appearance and interaction. The part of appearance could be generically defined with the help of specifi-

cations for graphical linkage relationships between visual components and for the dynamic building of containment hierarchy of visual components. More powerful descriptions are required so that the appearance and interaction schemes could be more flexibly described.

3. **Design by request.** Descriptiveness and visibility of templates provided with NCF results in the potential feature of "design by request". This means that domain-specific operations could be formulated on-demand by the Domain Expert during the design phase. This makes applicability of domain-specific visual composition systems even wider and more flexible.

4. **Practical realisation of the NCF approach for other languages.** The programming language we were experimenting with was Java. However, the NCF approach can be potentially applied to any textual composition language. The NCF approach can be improved by verifying other language.

5. **Wider repository of PCTs and Molecular Operations.** A common repository is used by the Software Architect during the development of the domain-specific visual composition system. It is important to have a wide and complete repository with categorisation and search mechanisms provided.

6. **Support of advanced information visualisation and interaction.** Future development of software systems can be seen as 3D virtual reality, where the design can be effectively reflected using advanced information visualisation and interaction techniques. This can make the process of development not only more intuitive and effective, but also more pleasant.

7. **Automation of development of templates.** The NBT tool can be extended with the graphical user interface as well as with the engine to automatically generate parts of specifications for the domain-specific visual composition system.

Appendix A

Repositories

We have collected together the main components and operations that are defined at different levels of compositions. Further we describe the repositories defined at Atomic, Template and Target Domain levels of composition.

A.1 Atomic Level of Composition

Name	Notation	Description
Instantiate	$new^a(t)$	Creates and returns an instance of the specified node type denoted as t
Remove	$\oslash^a(N)$	Removes a sub-tree represented by a node N and returns its parent if any
Initialise value	$ival^a(N, p, v)$	Initialises a property p of a node N with a value v and returns that node
Request value	$rval^a(N, p)$	Returns a value held by the property p of the node N
Attach	$\oplus^a(N, M)$	Verifies the types of N and M nodes and, if compatible, sets the node M as a child of the node N
Detach	$\ominus^a(N, M)$	Removes parent-child relationship, if present, between nodes N and M
Walk Down	$walk\downarrow^a(N, t)$	Resturns a child specified by the type t of the parent N

Name	Notation	Description
Walk Up	$walk \uparrow^a (N)$	Resturns a parent of a child N
Clone	$clone^a(N)$	Returns a copy of a node N
Search	$find^a(N, M)$	Starting from the node N it searches a match for a node described by the node M. Returns found nodes.

Table A.1: Atomic operations

A.2 Template Level of Composition

A.2.1 Parametric Code Templates

Table A.2 shortly explains PCTs that have been practically used in the thesis. The explanation consist of a state about the role of the PCT and relevant parameters defined by the PCT. Talble does not describe management behaviour specified for each PCT. After this a more precise description, based on development steps described in Section 5.7, of each PCT follows.

Template	Description
BlockPCTLeaf	Manages a block code fragment exchengeable through the `blockCodeFragment` parameter.
ClassPCT	Manages templates inside a class. By default contains the `ClassPCTLeaf` composite. The name of the class is set via the parameter `classNameParam`.
ClassPCTLeaf	Manages a class code fragment. The name of the class is set via the parameter `classNameParam`.
ConstructorPCT	Manages templates inside a class constructor. Initially contains `ConstructorPCTLeaf` instance.
ConstructorPCTLeaf	Represents a constructor code fragment.
EmptyBasePCT	This is a common PCT container which is initially empty. Often is used as a root PCT container for new template definitions.

Template	Description
ExpressionPCTLeaf	Holds an expression code fragment.
GetMethodPCTLeaf	Holds a getter method code fragment. A variable is set through the parameter `variableToGetParameter`. A type of a variable is set through the parameter `typeOfVariableParameter`.
IFStatementPCTLeaf	Holds an IF statement code fragment.
ImportsPCTLeaf	Holds a code fragment that specifies imported packages.
InnerClassPCT	Manages templates inside an inner class. By default contains the `InnerClassPCTLeaf` composite. The name of the class is set via the parameter `classNameParam`.
InnerClassPCTLeaf	Hold a code fragment of an inner class. The name of the inner class is set via the parameter `classNameParam`.
InterfacePCTLeaf	Manages an interface program code fragment. The name of the interface is set via the parameter `interfaceNameParameter`.
MethodPCT	This PCT manages templates within a method program code fragment. Method's return type, name, modifiers and parameters are set via the parameters `returnTypeParameter`, `methodNameParameter`, `modifierParameter` and `parametersParameter`.
MethodPCTLeaf	Holds a code fragment for a method. Defines the same parameters as the PCT container `MethodPCT`.
PackagePCTLeaf	Holds a code fragment for a package declaration of a class. The name of the package is defined through a parameter `packageName`

Template	Description
PrimEventProducerPCT	This PCT implements the observer design pattern [36] for the case of primitive events. A name of the event producer may be set via parameter `eventProducerName`. A primitive type of data transferred with events is set via parameter `setPrimType`.
PropertyPCT	This PCT represents a template of a variable declaration together with related setter and getter methods. Through the parameters `propertyNameParam`, `propertyTypeParam`, `propertyInitValueParameter` and `modifierParameter` a name, a type, an intial value and a modifier of a variable can be set.
SetMethodPCTLeaf	Holds a setter method code fragment. A return type is defined through the parameter `typeOfVariableParameter`. A variable name canbe be set through the parameter `typeOfVariableParameter`.
StatementPCT	Manages templates that form a statement. By default, initially containes a `StatementPCTLeaf`.
StatementPCTLeaf	Holds a code fragment of a statement.
ThreadPCT	A PCT that represents a behaviour of a thread.
VarDeclPCTLeaf	A code fragment of a variable declaration. A name, a type, a modifier and an initial value of a variable are set via the parameters `variableNameParameter`, `variableTypeParameter`, `modifierParameter` and `variableValueParameter`.

Table A.2: Short explanation of the Parametric Code Templates from the common repository

A.2.2 Molecular Operations

Table A.3 shortly explains molecular operations that have been practically used in the thesis.

Identifier	Notation	Description
Add_MO	$add(P, C)$	Adds composite C into the PCT denoted by P.
And_MO	$\&(P_1, P_2, ..., P_n)$	Defines a seuqence P_1, P_2,...,P_n of molecular operations to be executed.
Delete_MO	$delete(P)$	Deletes a specified PCT P.
GetCByType_MO	$?c(P, T)$	Looks for a composite denoted by type T inside a PCT denoted by P.
GetCByInst_MO	$?cc(P, iN)$	Looks for a composite denoted by an instance name iN inside a PCT denoted by P.
GetP_MO	$getP(P, pN)$	Returns a value held by a parameter denoted by a parameter name pN in a PCT P.
GetSParent_MO	$?\uparrow Parent(C)$	Returns the very top parent for a composite C.
Instantiate_MO	$instantiate(t)$	Creates an instance of a PCT of a type denoted by t.
Merge_MO	$merge(C, L)$	Merges a PCT container C and a PCT leaf L.
SetP_MO	$setP(P, pN, pV)$	Assigns a value pV to a parameter denoted by the parameter name pN in a PCT denoted by P.

Table A.3: Short explanation of the molecular operations

from the common repository

A.3 Target Domain Level of Composition

There is one DSC and several DSOs which are commonly reused in different application domains. The `MainContainerDSC` represents the very root container. It is a super template that contains all other templates during the design phase. Table A.4 explains common domain specific operations.

Identifier	Description
Delete_DSO(A)	Deletes a DSC denoted by A.
GetPCT_DSO(A)	Returns a PCT carried by a DSC A.
Instantiate_DSO(A)	Instantiates a DSC denoted by A.
Merge_DSO (A,B)	Merges DSCs A and B.

Table A.4: Domain Specific Operations which are commonly reused in different application domains

Appendix B

Tools Implementation: Description of Main Classes

Class	Description
`AbstractDSComponent`	A super class for all DSCs.
`AbstractDSOperation`	A super class for all DSOs.
`AbstractOperation`	A super class for all operations.
`AbstractPct`	A super class for all PCT containers.
`AbstractPctLeaf`	A super class for all PCT leafs.
`AbstractPctlExpressionNode`	Represents a expression node for all the expressions defined at the Template Level of composition and at the Target Domain Level of composition.
`AbstractViewNode`	Represents nodes of the tree-like data model of the View.
`AddDSO`	A operation type often used by different domains. This operation puts a specified composite into a specified PCT container.
`ASLTUtils`	Contains methods that simplify working with the Asbstract Syntax Language Tree (ASLT).

APPENDIX B. TOOLS IMPLEMENTATION: DESCRIPTION OF MAIN CLASSES

Class	Description
`BlockPCTLeaf`	Represents a PCT leaf that manages the structure of the block programming language element.
`ClassPCT`	Defines a PCT to manage the templates inside the class flie.
`ClassPCTLeaf`	Defines a PCT leaf to manage the class language element.
`ContainmentManager`	It is a super class that every Containment Manager should extend. It specifies the connection to the model of the View.
`Delete_MO`	Defines the molecular operation of deletion.
`DeleteDSO`	Specifies the domain specifici operation of deletion.
`DSComponentUtils`	Methods defined by this class simplify saving and loading of DSCs and DSOs. DSCs and DSOs are stored as files in a serialised form.
`EmptyBasePCT`	Represents a templates which is a super container and often represents the whole system being composed.
`ExecuteClassDialog`	GUI component of the NIP that is a dialog window to request which generated class files have to be executed.
`ExecutionProvider`	The class contains logic to perform the execution process.
`GetMethodPCTLeaf`	The PCT leaf to manage getter method.
`ImportPCTLeaf`	Represents the PCT leaf that manages a block of import declarations.
`InnerClassPCT`	Defines a PCT to manage the templates inside the innner class flie.

Class	Description
InnerClassPCTLeaf	Defines a PCT leaf to manage the inner class language element.
Instantiate_MO	Defines the molecular operation of instantiation.
InstantiateDSO	Defines the domain specific operation of deletion.
LinkageManager	Describes how the relations between terms in the domain ontology are translated into the linkage relations between tree nodes of the ViewModel.
MainEnvironmentDSC	It is a DSC that represents the very super container for other DSCs.
MenuToolbar	The menu of the GUI of NIP.
Merge_MO	This is the specification of the molecular operation of merging.
MergeDSO	This is the specification of the domain specific operation of merging.
MessageWindow	The message window GUI component. Listens for data models of other components of NIP and prints out the messages received.
MethodPCT	The PCT which manages templates inside a method.
MethoPCTLeaf	The PCT leaf to manage the method.
ModellingPane	Represents a modelling pane of the NIP.
MolecularOperation	This is a super class for all molecular operations.
MoveableContainer	This is simple GUI components which can be dragged with mouse.

APPENDIX B. TOOLS IMPLEMENTATION: DESCRIPTION OF MAIN CLASSES

Class	Description
`NCFVisualElement`	This interface that all Neurath Modelling Components have to implement.
`NewProjectDialog`	The dialog window GUI for specification of the project properties in the NIP.
`NipDefaultGui`	The main GUI component of NIP.
`NipModel`	This is the inteface implemented by the main NIP data model.
`NipModelEvent`	The event generated by the NIP data model.
`NipModelListener`	Listeners of events generated by the NIP data model have to implement this interface.
`NmlDescriptor`	Each object of this class desribes a Neurath Modelling Language that has benn deployed.
`NmlElementDataModel`	Represents the data model of a NMC located in the toolbar.
`NmlelementGui`	Represents GUI of a NMC located in the toolbar.
`NmlToolBar`	A toolbar GUI.
`PackagePCTLeaf`	A PCT leaf to manage package declaration.
`PCTUtils`	Contains methods that simplify the saving and loading of serialised persisted PCTs.
`ProjectControlPane`	The project control pane GUI component of the NIP tool.
`ProjectDescriptor`	The data model of the project.
`PropertyInspector`	The property inspector GUI component.
`PropertyPCT`	Defines a PCT to manage the property code templates.
`SearchAbstract`	Implementation of pattern search routines. It a base class of search library.

Class	Description
SearchBredth	Implementation of bredth search routines in the ASLT.
SearchDepth	Implementation of depth search routines in the ASLT.
SearchMachine	Base Interface of the search library. All search classes have to implement these routines.
SetMethodPCTLeaf	The PCT leaf to manage a setter method.
SkwContext	Represents a domain ontology at design time. The class implements management of nodes and relationships between them.
SkwContextEvent	This event is fired is domain ontology is changed.
SkwContextListener	Each listener of events from the domain ontology must implement this interface.
SkwNode	Represents a node in the domain ontology.
SkwRelation	Represents a relation in the domain ontology.
StatementPCTLeaf	The PCT leaf that is able to manage a statement.
StringPctlExpressionNode	Special case of the node in the PCT-L expression tree. It represents a operand that is a string.
StructureWindow	A window GUI component in the NIP, to hold a structure of the designed system.
SymbolicManager	This is a super class for Symbolic Managers.
TDEnvironment	This class is used by the NIP to direclty access the domain ontology.
UIIE_Parser	This is a parser for User Interface Interaction Expressions.

Class	Description
`VaiUtils`	Some utilities to be used at the Visualisation and Interaction level of composition.
`VariableDeclarationPCTLeaf`	Represents a PCT leaf to manage a variable declaration.
`ViewController`	A controller of a View.
`ViewDescriptor`	A descriptor of a View.
`ViewModel`	A model of a View.
`ViewModelEvenListener`	An interface that should be implemented by all listeners of the events coming from View model.
`ViewModelEvent`	This is an event generated by a View model.
`ViewNode`	A node in the tree-like datastructure describing a View model.
`ViewV`	Represents a view part of a View.

Table B.1: Description of classes for the tool implementation

Appendix C

Console Viewer: Generated Code

C.1 Requirements Analysis and Processing

C.1.1 Template Composition System

```
...
// Instantiation and initialisation of ClassPCT
ClassPCT c = new ClassPCT( );
c.setClassNameParam("HelloWorld");

// Instantiation and initialisation of PropertyPCT
PropertyPCT p = new PropertyPCT();
p.setPropertyNameParam("text");
p.setPropertyTypeParam("String");
p.setModifierParameter("static" );
p.setPropertyInitValueParameter(" \" Hello World!\" " );

// Merging ClassPCT instance and PropertyPCT instance
c .addCompositeAndMerge(p);

// Instantiation and initialisation of MethodPCTLeaf
MethodPCTLeaf m = new MethodPCTLeaf();
m.setMethodNameParameter("main" );
m.setParametersParameter("(String[] args)");
m.setReturnTypeParameter("void") ;
```

```
21 m.setModifierParameter(MethodPCTLeaf.PUBLIC);
22 m.setModifierParameter(MethodPCTLeaf.STATIC);
23
24 // Merging ClassPCT instance and MethodPCTLeaf instance
25 c.addCompositeAndMerge(m);
26
27 // Instantiation and initialisation of StatementPCTLeaf
28 StatementPCTLeaf s = new StatementPCTLeaf ( );
29 s.setStatement("System.out.println(gettext());");
30
31 // Merging ClassPCT instance and StatementPCTLeaf instance
32 c.addCompositeAndMerge(s);
33
34 // Definition of signing the ClassPCT instance as
      ConsoleViewerPCT
35 c.defineNewPctSignature("ConsoleViewerPCT");
36
37 // Saving the ConsoleViewerPCT in a serialised form
38 c.saveAsSerialized("HW/repository/pct1/");
39
40 // Generating a program code of the template
41 c.generateCode("HW/repository/code/" ) ;
42 ...
```

Listing C.1: Program that generates Parametric Code Templates for the Console Viewer domain

C.1.2 Domain Specific Composition System

```
1 ...
2 public class MainContainerDSC extends AbstractDSComponent{
3   public MainContainerDSC(){
4     super(new EmptyBasePCT()); }
5   public String toStringAsTreeNode(){
6     return "MainContainerDSC"; }
7   public String toTypeOfSKWNodeString(){
8     return "Main Container";}
```

```
}
```

Listing C.2: Specification of the `MainContainerDSC` for the Console Viewer domain

```
...
public class ConsoleViewerApplicationDSC extends
    AbstractDSComponent{
 public ConsoleViewerApplicationDSC(){
  // loading and relating the PCT
  this("HW/repository/pct1/" , "PCT_HellWorldPCT.pct");

  // specifying attributes for the related ontology node
  SkwNode node = getSkwNode();
  node.addAttribute("Text","String");
  node.addAttribute("ClassName", "String") ;
 }
 // set method to access the attribute Text
 public void setText(String text){
  ClassPCT classpct = (ClassPCT)getPctCarried();
  PropertyPCT leaf =
   (PropertyPCT)classpct.compositeSearch("PropertyPCT");
  leaf.setInitValue(text) ;
 }
 // get method to access the attribute Text
 public String getText(){
  ClassPCT classpct = (ClassPCT)getPctCarried() ;
  PropertyPCT leaf =
   (PropertyPCT)classpct.compositeSearch("PropertyPCT") ;

  return leaf.getInitValue();
 }
 // set method to access the attribute ClassName
 public void setClassName(String name){
  ClassPCT classpct = (ClassPCT) getPctCarried();
  classpct . setClassName(name);
 }
 // get method to access the attribute ClassName
 public String getClassName(){
```

```
34   ClassPCT classpct = (ClassPCT getPctCarried();
35   return classpct.getClassName();
36  }
37  // Specification of term that is represented by the DSC
38  public String toTypeOfSKWNodeString(){
39    return "Console Viewer Application";
40  }}
```

Listing C.3: Specification of the `ConsoleViewerApplicationDSC` for the Console Viewer domain

```
1
2  public class DeleteDS extends AbstractDSOperation{
3    protected AbstractPctlExpressionNode targetElement = null;
4    protected AbstractPctlExpressionNode theParent = null;
5    public Delete_DSO(){ super(); }
6    public Delete_DSO(AbstractPctlExpressionNode targetElement){
7        super();
8        setTargetElement(targetElement);
9    }
10   public void setTargetElement(AbstractPctlExpressionNode
       targetElement){
11    this.targetElement = targetElement;
12    targetElement.setParent(this);
13   }
14   public AbstractPctlExpressionNode getTargetElement(){
15    return targetElement; }
16   public AbstractPctlExpressionNode operate(){
17    if (processed == true) return getResult();
18    AbstractPctlExpressionNode calculatedTarget =
         getTargetElement().operate();
19   DeleteMO dlMolOp = new DeleteMO(calculatedTarget.getResult())
       ;
20   // SKW
21     SkwNode node = ((AbstractDSComponent)(getTargetElement().
         getDSResult()).getSkwNode();
```

```
22      neurath.tdframework.TDEnvironment.getSkwContext().deleteNode
            (node);
23    // SKW
24    theParent = dlMolOp.operate();
25    processed =true;
26    return theParent;}
27  public String toStringAsTreeNode(){
28    return "Delete("+getTargetElement()+")";
29  }
30  public AbstractPctlExpressionNode getResult(){
31    if (isProcessed()) return theParent;
32     else return null; } }
```

Listing C.4: Specification of the `DeleteDSO` for the Console Viewer domain

C.1.3 Domain Specific Visual Interface

```
1
2  public class HWContainmentManager extends ContainmentManager{
3    private LinkedList unbindedNodes = new LinkedList();
4    public void contextReseted(SkwContextEvent event){}
5    public void relationAdded(SkwContextEvent event){
6      SkwRelation relation = event.getSkwRelation();
7      ViewNode subjectNode = null;
8      ViewNode objectNode = null;
9      for (int i=0;i<unbindedNodes.size();i++){
10       ViewNode tnode = (ViewNode)unbindedNodes.get(i);
11       //object
12       if (tnode.getSkwNode().equals(relation.getObject())){
13         objectNode = (ViewNode)unbindedNodes.remove(
14             unbindedNodes.indexOf(tnode));
15       }
16       //subject
17       if (tnode.getSkwNode().equals(relation.getSubject())){
18         subjectNode = (ViewNode)unbindedNodes.remove(
19             unbindedNodes.indexOf(tnode));
20       }
```

```
21    } //for
22    if (subjectNode==null){
23      subjectNode =
24       findNodeInTheModel(model.getRoot(), relation.getSubject())
            ;
25    }
26    if (objectNode==null){
27      objectNode =
28        findNodeInTheModel(model.getRoot(), relation.getObject());
29    }
30    getViewModel().addParentChildRelationship(objectNode,
         subjectNode);
31  }
32  public void nodeAdded(SkwContextEvent event){
33    SkwNode skwNode = event.getSkwNode();
34    ViewNode node = new ViewNode(skwNode);
35    if (skwNode.getType().equals("MainContainerDSC"))
36      model.setRoot(node);
37    else
38      unbindedNodes.add(node);   }
39  public ViewNode findNodeInTheModel(ViewNode viewNode, SkwNode
       skwnode)
40  { if (viewNode.getSkwNode().equals(skwnode)) return viewNode;
41    for (Enumeration e = viewNode.children(); e.hasMoreElements()
         ;) {
42      ViewNode nextNode = (ViewNode)e.nextElement();
43      ViewNode result = findNodeInTheModel(nextNode, skwnode);
44      if (result!=null) return result; }//for
45    return null; }
46  public void attributeSet(SkwContextEvent event){
47    ViewNode node = model.findBySkwNode(event.getSkwNode());
48    model.nodeUpdated(node); }
49  public void nodeDeleted(SkwContextEvent event){
50    SkwNode skwNode = event.getSkwNode();
51    ViewNode node = model.findBySkwNode(skwNode);
52    NipGuiDefaultModel.setChosenNMC(((ViewNode)node.getParent()).
         getSkwNode());
```

```
53    model.deleteNode(node); }
54  public static void main(String[] args){
55    HWContainmentManager hwcm = new HWContainmentManager();
56    VaIFUtils.saveCMAsSerialized("HW/nml/viewSpec", hwcm);
57  }}
```

Listing C.5: Specification of the Containment Manager for the Console Viewer domain

```
1  public class HWSymbolicManager extends SymbolicManager {
2   protected JPanel mainLibraryGUIContainer = null;
3   private MainContainerGUI mainContainer;
4   public void newRootSet(ViewModelEvent event){
5    if (is_viewNode_visualElement_Relationship_Present(event.
        getTargetNode())){
6     MainContainerGUI oldMainContainer =
7     (MainContainerGUI) get_visualElement_by_viewNode(event.
        getTargetNode());
8     oldMainContainer = null; }
9    mainContainer = new MainContainerGUI(event.getTargetNode());
10   add_viewNode_visualElement_Relationship(event.getTargetNode()
       ,mainContainer);
11   mainLibraryGUIContainer.add(mainContainer);
12   mainLibraryGUIContainer.validate();
13   mainLibraryGUIContainer.repaint(); }
14  public void newParentChildRelationship(ViewModelEvent event){
15   ViewNode parentNode = event.getParent(); //main Container
16   ViewNode childNode = event.getChild();
17   NCFVisualElement guiParentElement =
18    (NCFVisualElement) get_visualElement_by_viewNode(parentNode)
       ;
19   NCFVisualElement guiChildElement = null;
20   if (childNode.getSkwNode().getType().equals("ConsoleViewerDSC
       ")){
21    guiChildElement = new ConsoleViewerGUI(childNode);
22   }
23   if(guiChildElement != null){
24    guiParentElement.addContainee(guiChildElement);
```

```
25      add_viewNode_visualElement_Relationship(childNode,
            guiChildElement);
26    }}
27  public void nodeUpdated(ViewModelEvent event){
28    ViewNode targetNode = event.getTargetNode();
29    NCFVisualElement guiElement =
30      (NCFVisualElement) get_visualElement_by_viewNode(targetNode)
            ;
31    if (guiElement!=null)
32      guiElement.updateUIAccSkwNode();
33  }
34  public static void main(String[] args){
35    HWSymbolicManager hwsm = new HWSymbolicManager();
36    VaIFUtils.saveSymbolicManagerAsSerialized("HW/nml/viewSpec",
          hwsm);
37  }
38  public void nodeDeleted(ViewModelEvent event){
39    ViewNode targetNode = event.getTargetNode();
40    NCFVisualElement guiElement =
41      (NCFVisualElement) get_visualElement_by_viewNode(targetNode)
            ;
42    NCFVisualElement guiParentElement = guiElement.
          getParentContainer();
43    guiParentElement.removeContainee(guiElement);
44    remove_viewNode_visualElement_Relationship(targetNode);
45  }
46  public void setLibrarySpecific_MainContainer(Object
      mainContainer){
47    mainLibraryGUIContainer = (JPanel)mainContainer;
48  }}
```

Listing C.6: Specification of the Symbolic Manager for the Console Viewer domain

C.1.4 Deployment Descriptors

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
```

```
<NmlIdName> ConsoleViewer</NmlIdName>
<Description> Console Viewer console application </Description>
<InitialExpression>
  <DSComponentElement>MainContainerDSC</DSComponentElement>
  <DSOperationElement>InstantiateDSO</DSOperationElement>
</InitialExpression>
<LanguageElements>
  <Element>DSC/ConsoleViewerDSC.xml</Element>
  <Element>DSC/DSC/MainContainerDSC.xml</Element>
  <Element>DSOp/InstantiateDSO.xml</Element>
  <Element>DSOp/DeleteDSO.xml</Element>
</LanguageElements>
 <Views>
   <View>
    <Id>View 1</Id>
    <ContainerManager>viewSpec/HWContainmentManager.cm</
       ContainerManager>
    <SymbolicManager>viewSpec/HWSymbolicManagerJKS.sm</
       SymbolicManager>
   </View>
</root>
```

Listing C.7: Main deployment descritor for the Console Viewer domain

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
  <LanguageId> MainContainerDSC </LanguageId>
  <DSPCTLElement>/dspctl/MainContainerDSC.dsc </DSPCTLElement>
</root>
```

Listing C.8: Deployment descritor MainContainer.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
  <ToolBarIcon> /isotypes/ConsoleIcon.jpg </ToolBarIcon>
  <LanguageId> ConsoleViewerDSC </LanguageId>
  <DSPCTLElement> dspctl/ConsoleViewerDSC.dsc </DSPCTLElement>
</root>
```

Listing C.9: Deployment descritor ConsoleViewerDSC.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
  <LanguageId>InstantiateDSO</LanguageId>
  <DSPCTLElement>/dspctl/InstantiateDSO.dsmop</DSPCTLElement>
</root>
```

Listing C.10: Deployment descritor InstantiateDSO.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
  <ToolBarIcon> /isotypes/DeleteIcon.jpg </ToolBarIcon>
  <DSPCTLElement> /dspctl/DeleteDSO.dsmop </DSPCTLElement>
  <LanguageId>DeleteDSO</LanguageId>
  <ParamSignatures>
    <Parameter>
      <Name>targetElement</Name>
      <Setter>setTargetElement</Setter>
      <Getter>getTargetElement</Getter>
    </Parameter>
  </ParamSignatures>
</root>
```

Listing C.11: Deployment descritor DeleteDSO.xml

C.2 Design Phase

```
// ##Filename:HelloWorld.java
public class HelloWorld {
  static String text= "Hello World";
   static void settext(String text1) {
    text = text1;
   }
  static String gettext() {
    return text;
```

```
  }
  public static void main(String[] args) {
    System.out.println(gettext());
  }
}
```

Listing C.12: Generated Console Viewer software system (Step 1)

```
// ##Filename:HelloWorld.java
public class HelloWorld {
  static String text= "Hello UK!";
   static void settext(String text1) {
    text = text1;
   }
  static String gettext() {
    return text;
  }
  public static void main(String[] args) {
    System.out.println(gettext());
  }
}
```

Listing C.13: Generated Console Viewer software system (Step 2)

Appendix D

House Automation: Generated Code

D.1 Requirements Analysis and Processing

D.1.1 Template Composition System

We present a program code of one Paramteric Code Template that realises the `SensorPCT` template.

```
1  ...
2  public class SensorPCT extends AbstractPct{
3    // A reference to the main composite of this PCT container
4    private ClassPCT haClassPCT;
5    // Two declared parameters of the PCT
6    private String sensorName;
7    private String sensorIcon;
8    // A reference to one of internal composites supposed to be
       used later
9    private InterfacePCTLeaf listenerInterfaceLeaf;
10
11   // Constructor
12   public SensorPCT(){
13     super();
14     // Load HAClassPCT
15     AbstractPctLeaf haClassPCT = loadAsSerialized(
16       "repository/ha", "PCT_HAClassPCT.pct");
17     // Load HAIconicAppearance
```

```
18    AbstractPctLeaf haIconicAppearance = loadAsSerialized(
19     "repository/ha", "PCT_neurath.pctframework.pctl.
          HAIconicAppearance.pct");
20    // Load HASnesorInputPCT
21    AbstractPctLeaf haSensorInputPCT = loadAsSerialized(
22     "repository/ha", "PCT_neurath.pctframework.pctl.
          HASensorInputPCT.pct");
23    // Load HASensorOutputPCT
24    AbstractPctLeaf haSensorOutputPCT = loadAsSerialized(
25     "repository/ha", "PCT_neurath.pctframework.pctl.
          HASensorOutputPCT.pct");
26
27    // Spefication of the merging operation mo1 for
28    // the HAClassPCT and HAIconicAppearance instances
29    Merge_MolOp mo1 = new Merge_MolOp(haClassPCT,
          haIconicAppearance);
30
31    // Spefication of the merging operation mo2 for
32    // the result of mo1 and and the HASensorInput instance
33    Merge_MolOp mo2 = new Merge_MolOp(mo1, haSensorInputPCT);
34
35    // Spefication of the merging operation mo3 for
36    // the result of mo2 and and the HASensorOutput instance
37    Merge_MolOp mo3 = new Merge_MolOp(mo2, haSensorOutputPCT);
38
39    // Starting the operation mo3
40    this.haClassPCT = (ClassPCT)mo3.operate();
41
42    // The result of the mo3 operation add as a composite into
43    // this PCT container
44    addComposite(haClassPCT);
45
46    // Save reference to the internal composite in order to
47    // simplify search of this composite later.
48    listenerInterfaceLeaf = ((HASensorOutputPCT)haSensorOutputPCT
          ).getListenerInterfaceLeaf();
49  }
```

```
50  // A method to assign a value to the parameter "sensorIcon"
51  public void setSensorIcon(String icon){
52    sensorIcon = icon;
53     AbstractPctLeaf pctLeaf = haClassPCT.
          nonDeepSearchCompositeOfType(
54        "neurath.pctframework.pctl.HAIconicAppearance");
55     if (pctLeaf!=null){
56      HAIconicAppearance pct1 = (HAIconicAppearance)pctLeaf;
57      pct1.setIconPath(icon);
58     }
59  }
60  // A method to request a value of the parameter "sensorIcon"
61  public String getSensorIcon(){
62    return sensorIcon; }
63  // A method to assign a value to the parameter "sensorName"
64  public void setSensorName(String name){
65    sensorName = name;
66     AbstractPctLeaf pctLeaf = haClassPCT.
          nonDeepSearchCompositeOfType(
67                          "neurath.pctframework.pctl.ClassPCTLeaf");
68     if (pctLeaf!=null){
69      ClassPCTLeaf pct1 = (ClassPCTLeaf)pctLeaf;
70      pct1.setClassNameParameter(name);
71     } }
72  // A method to request a value of the parameter "sensorName"
73  public String getSensorName(){
74    return sensorName; }
75  public InterfacePCTLeaf getListenerInterfaceLeaf(){
76    return listenerInterfaceLeaf; }
77  public String toStringAsTreeNode(){
78    return "iSensorPCT"; }
79   public ClassPCT getClassPCT(){
80     return haClassPCT; } }
```

Listing D.1: Program code of the SensorPCT template

```
1  ...
2  public class ConnectSC extends MolecularOperation{
```

```
3   // Declaration of the operands for this operation
4   protected AbstractPctlExpressionNode sens = null;
5   protected AbstractPctlExpressionNode contr = null;
6   // Constructor
7   public ConnectSC(){  super();  }
8   // Constructor
9   public ConnectSC(AbstractPctlExpressionNode sens,
        AbstractPctlExpressionNode contr){
10   super();
11   setSens(sens);
12   setContr(contr); }
13  // Setter method for the operand "sens"
14   public void setSens(AbstractPctlExpressionNode sens){
15   this.sens = sens;
16   sens.setParent(this); }
17  // Getter method for the operand "sens"
18  public AbstractPctlExpressionNode getSens(){
19   return sens; }
20  // Setter method for the operand "contr"
21   public void setContr(AbstractPctlExpressionNode contr){
22   this.contr = contr;
23   contr.setParent(this); }
24  // Getter method for the operand "contr"
25  public AbstractPctlExpressionNode getContr(){
26   return contr; }
27  // The operation body
28  public AbstractPctlExpressionNode operate(){
29   // process the first operand (in case if it is nested
        operation)
30   sens.operate();
31   AbstractPct calculatedSens = (AbstractPct)sens.getResult();
32   // process the second operand (in case if it is nested
        operation)
33   contr.operate();
34   AbstractPct calculatedContr = (AbstractPct)contr.getResult();
35   try{
```

```
36      // Look for a event listener specification held by the
           processed
37      // "sens" operand which is expected to be SensorPCT
38      InterfacePCTLeaf listenerInterface =
39        ((SensorPCT)calculatedSens).getListenerInterfaceLeaf();
40
41      // Merge the event listener interface template with the
           processed
42      // "contr" operand which is expected to be ControllerPCT
43      (((ControllerPCT)calculatedContr).getClassPCT()).
44         addCompositeAndMerge(listenerInterface);
45
46      // Load the SignalPCT
47      SignalPCT signalPCT = (SignalPCT)AbstractPctLeaf.
           loadAsSerialized(
48         "repository/ha", "PCT_neurath.pctframework.pctl.SignalPCT
              .pct");
49
50        // Merge the SignalPCT and the "contr" operand
51        (((ControllerPCT)calculatedContr).getClassPCT()).
             addCompositeAndMerge(signalPCT);
52
53        // Request a MethodPCT contained in the event listener
             interface template
54      MethodPCT methodPct = (MethodPCT)listenerInterface.
           getMethods().get(0);
55
56      // Merge a MethodPCT instance with a leaf contained in
           SignalPCT instance
57      methodPct.addCompositeAndMerge(signalPCT.
           getSignalValueAssignLeaf());
58    } catch (Exception e){ e.printStackTrace();}
59      processed = true;
60      return calculatedContr; }
61    // Returns string interpretation of the operation
62    public String toStringAsTreeNode(){
63      return "ConnectSC("+
```

```
    getSens().toStringAsTreeNode()+
    ","+
    getContr().toStringAsTreeNode()+")"; }
public AbstractPctlExpressionNode getResult(){
  if (isProcessed()) return contr.getResult();
   else return null; }}
```

Listing D.2: Program code of the ConnectSC operation

D.1.2 Domain Specific Composition System

```
...
public class SensorDSC extends AbstractDSComponent{
  // Constructor
  public SensorDSC(){
    // Load SensorPCT
    this("repository/ha", "PCT_neurath.pctframework.pctl.
       SensorPCT.pct");
   // Request the SkwNode that have been already automatically
       generated for this component
   SkwNode node = getSkwNode();
   // Add an attribute "Type" to the SkwNode representing this
       component
   node.addAttribute("Type","String");
   // Add an attribute "Icon" to the SkwNode representing this
       component
   node.addAttribute("Icon","String"); }
 // Constructor
 public SensorDSC(String repositoryDir, String
    pctInstanceFileName){
   super(repositoryDir, pctInstanceFileName); }
 // Returns a string representation of this DSC component
 public String toStringAsTreeNode(){
   return "Sensor"; }
 // Returns a term (from the domain ontology carried by
    SkwContext) which this DSC represent
 public String toTypeOfSKWNodeString(){
```

```
21    return "Sensor"; }
22  // Setter method for the attribute "Type"
23  public void setType(String type){
24    SensorPCT sensorPct = (SensorPCT)getPctCarried();
25    sensorPct.setSensorName(type); }
26  // Getter method for the attribute "Type"
27  public String getType(){
28    SensorPCT sensorPct = (SensorPCT)getPctCarried();
29    return sensorPct.getSensorName(); }
30  // Setter method for the attribute "Icon"
31  public void setIcon(String icon){
32    SensorPCT sensorPct = (SensorPCT)getPctCarried();
33    sensorPct.setSensorIcon(icon); }
34  // Getter method for the attribute "Icon"
35  public String getIcon(){
36    SensorPCT sensorPct = (SensorPCT)getPctCarried();
37    return sensorPct.getSensorIcon(); } }
```

Listing D.3: Program code of the SensorDSC component

```
1  ...
2  public class ConnectSC_DSO extends AbstractDSOperation{
3    // Declaration of an operand "s"
4    protected AbstractPctlExpressionNode s;
5    // Declaration of an operand "c"
6    protected AbstractPctlExpressionNode c;
7    // Constructor
8    public ConnectSC_DSO(){ super();}
9    // Constructor
10   public ConnectSC_DSO(AbstractPctlExpressionNode s,
         AbstractPctlExpressionNode c){
11       super();
12       setS(s);
13       setC(c); }
14   // Setter method for the operand "s"
15   public void setS(AbstractPctlExpressionNode s){
16     this.s = s;
17     s.setParent(this); }
```

```
18  // Getter method for the operand "s"
19  public AbstractPctlExpressionNode getS(){
20    return s; }
21  // Setter method for the operand "c"
22  public void setC(AbstractPctlExpressionNode c){
23    this.c = c;
24    c.setParent(this); }
25  // Setter method for the operand "c"
26  public AbstractPctlExpressionNode getC(){
27    return c; }
28  // Body of an operation
29  public AbstractPctlExpressionNode operate(){
30    // if this operation have been already processed then return
        the operand "s"
31    if (processed == true) return s;
32    // Create a molecular operation ConnectSC; specify operands
        for this operation.
33    // These operands are PCTs carrieds by the operands "s" and "
        c"
34    ConnectSC connect = new ConnectSC(s.operate(),c.operate());
35    // Process the molecular operation
36    connect.operate();
37    // Change the SkwContext
38    // (1) Request the SkwNode which represents an operand "s"
39    SkwNode sNode = ((AbstractDSComponent)s.getDSResult()).
        getSkwNode();
40    // (2) Request the SkwNode which represents an operand "c"
41    SkwNode cNode = ((AbstractDSComponent)c.getDSResult()).
        getSkwNode();
42    SkwContext context = neurath.tdframework.TDEnvironment.
        getSkwContext();
43    // (3) Create a relation (SkwRelation), called "connected to
        ", between nodes (SkwNodes) that represent
44    // operands "s" and "c"
45    SkwRelation relation =
46       context.createRelation("connected to", cNode, sNode);
47    processed =true;
```

```
48    return s; }
49   // Return a string interpretation of this DSO
50   public String toStringAsTreeNode(){
51    return "ConnectSC_DSO("+getS().toStringAsTreeNode()+
52        ","+
53        getC().toStringAsTreeNode()+")";
54   }
55   // Return result of this operation applied to specified
       operands
56   public AbstractPctlExpressionNode getResult(){
57    if (isProcessed()) return getS();
58     else return null;
59   } }
```

Listing D.4: Program code of the ConnectSC_DSO Domain Specific Operation

D.1.3 Domain Specific Visual Interface

```
1  public class HAContainerManager extends ContainmentManager{
2   // To store ViewNodes for created, but not related SkwNodes
3   private LinkedList unbindedNodes = new LinkedList();
4   // This method is automatically called when the SkwContext is
       reseted
5   public void contextReseted(SkwContextEvent event){
6     ... }
7   // This method is automatically called when the SkwRelation
       between
8   // two SkwNodes (so called object and subject) is created
9   public void relationAdded(SkwContextEvent event){
10    SkwRelation relation = event.getSkwRelation();
11    ViewNode subjectNode = null;
12    ViewNode objectNode = null;
13    // Looking for ViewNodes that hold subject and object
        SkwNodes
14    for (int i=0;i<unbindedNodes.size();i++){
15     ViewNode tnode = (ViewNode)unbindedNodes.get(i);
16     if (tnode.getSkwNode().equals(relation.getObject())){
```

```
17        objectNode = (ViewNode)unbindedNodes.remove(
18            unbindedNodes.indexOf(tnode));
19        }
20        if (tnode.getSkwNode().equals(relation.getSubject())){
21          subjectNode = (ViewNode)unbindedNodes.remove(
22            unbindedNodes.indexOf(tnode));
23        } }
24      if (subjectNode==null){
25        subjectNode = findNodeInTheModel(model.getRoot(), relation.
              getSubject());
26      }
27      if (objectNode==null){
28        objectNode = findNodeInTheModel(model.getRoot(), relation.
              getObject());
29      }
30      // Analysing of a relation
31      String id = relation.getId();
32      if (id.equals("has")){
33        getViewModel().addParentContainsChild(objectNode,
              subjectNode);
34      }
35      if (id.equals("connected to"))   {
36        getViewModel().addParentLinksChild(objectNode, subjectNode,
              relation.getId());
37      } }
38    // This method is automatically called when the SkwNode is
        created
39    public void nodeAdded(SkwContextEvent event){
40      SkwNode skwNode = event.getSkwNode();
41      // Creating the ViewNode for the SkwNode
42      ViewNode node = new ViewNode(skwNode);
43      // Analysis of the SkwNode
44      if (skwNode.getType().equals("MainContainer")){
45        model.setRoot(node);
46      }
47      else
48        unbindedNodes.add(node);   }
```

```
49  // This method is automatically called when the SkwNode is
        deleted
50  public void nodeDeleted(SkwContextEvent event){
51    SkwNode skwNode = event.getSkwNode();
52    // Look for related VieWNode
53    ViewNode node = model.findBySkwNode(skwNode);
54    NipGuiDefaultModel.setChosenNMC(((ViewNode)node.getParent()).
          getSkwNode());
55    // Give a command to the ViewModel to delete related ViewNode
56    model.deleteNode(node); }
57
58  // This method is automatically called when the attribute of a
        SkwNode is changed
59  public void attributeSet(SkwContextEvent event){
60  // Look for related VieWNode
61    ViewNode node = model.findBySkwNode(event.getSkwNode());
62    // Give a command to the ViewModel to update related ViewNode
63    model.nodeUpdated(node); }
64  // Method which helps searching ViewNodes in the ViewModel
65  public ViewNode findNodeInTheModel(ViewNode lroot, SkwNode
        skwnode){
66    // Going through the tree of the View Model
67    if (lroot.getSkwNode().equals(skwnode)) return lroot;
68    for (Enumeration e = lroot.children(); e.hasMoreElements() ;)
          {
69      System.out.println("Requesting children!");
70      ViewNode nextNode = (ViewNode)e.nextElement();
71      ViewNode result = findNodeInTheModel(nextNode, skwnode);
72      if (result!=null) return result;
73    }
74    return null;
75  }}
```

Listing D.5: Specification of the HAContainerManager

```
1  public class HASymbolicManager extends SymbolicManager{
2    // Reference to the main container GUI
3    protected JPanel mainLibraryGUIContainer = null;
```

```
4
5   // This method is automatically called when the new root (
        ViewNode) for the
6   // ViewModel is set
7   public void newRootSet(ViewModelEvent event){
8     // If the old GUI for the root node exist, reset it.
9     if (is_viewNode_visualElement_Relationship_Present(event.
        getTargetNode())){
10      MainContainerGUI oldMainContainer =
11      (MainContainerGUI) get_visualElement_by_viewNode(event.
          getTargetNode());
12      oldMainContainer = null;
13    }
14
15    // Create a new root container GUI
16    MainContainerGUI newMainContainer = new MainContainerGUI(
        event.getTargetNode());
17
18    // Save the relationship between the root ViewNode and the
        created GUI component
19    add_viewNode_visualElement_Relationship(event.getTargetNode()
        , newMainContainer);
20
21    // Add created GUI to the parent library-specific GUI
        component which
22    // represents a start pane
23    mainLibraryGUIContainer.add(newMainContainer);
24
25    // Refresh newly generated GUI
26    mainLibraryGUIContainer.repaint();
27    mainLibraryGUIContainer.validate(); }
28
29  // This method is called automatically when parent-child
      relationship is created
30  // between two ViewNodes in the ViewModel
31  public void newParentChildRelationship(ViewModelEvent event){
32    ViewNode parentNode = event.getParent();
```

```
33    ViewNode childNode = event.getChild();
34
35    // Request already created GUI element of the parent
36    NCFVisualElement guiParentElement = (NCFVisualElement)
37       get_visualElement_by_viewNode(parentNode);
38
39    // Initialising a GUI for the child
40    NCFVisualElement guiChildElement = null;
41
42    // Analysis of the child ViewNode.
43    // Creation of GUI components according to the child ViewNode
          characteristics
44    if (childNode.getSkwNode().getType().equals("Sensor"))
45     guiChildElement = new SensorGUI(childNode);
46
47    if (childNode.getSkwNode().getType().equals("Controller"))
48     guiChildElement = new ControllerGUI(childNode);
49
50    if (childNode.getSkwNode().getType().equals("Actuator"))
51     guiChildElement = new ActuatorGUI(childNode);
52
53    if (childNode.getSkwNode().getType().equals("Action"))
54     guiChildElement = new ActionGUI(childNode);
55
56    if (childNode.getSkwNode().getType().equals("Emulator"))
57     guiChildElement = new EmulatorGUI(childNode);
58
59    // If the relation is "contains" then add child GUI
60    // into the parent GUI container
61    guiParentElement.addContainee(guiChildElement);
62
63    // If the relation is "links" then add graphical linkage
64    // between GUIs of the parent ViewNode and the child ViewNode
65    // into the parent GUI container
66    ...
67    // Saving the relationship between ViewNode and the created
         GUI
```

```
68      add_viewNode_visualElement_Relationship(childNode,
            guiChildElement);
69    }
70
71    // This method is automatically called when the ViewNode is
          deleted
72    public void nodeDeleted(ViewModelEvent event){
73      ViewNode targetNode = event.getTargetNode();
74
75      // Request existing GUI of the ViewNode which is deleted
76      NCFVisualElement guiElement =
77        (NCFVisualElement) get_visualElement_by_viewNode(targetNode)
              ;
78
79      // Remove GUI childs from its GUI container
80      NCFVisualElement guiParentElement = guiElement.
            getParentContainer();
81      guiParentElement.removeContainee(guiElement);
82
83      // Remove relationship between GUI element and ViewNode
84      remove_viewNode_visualElement_Relationship(targetNode);
85    }
86
87    // This method is called when the ViewNode is updated
88    public void nodeUpdated(ViewModelEvent event){
89      ViewNode targetNode = event.getTargetNode();
90
91      // Give a command to the related GUI component to update its
            appearance
92      // according to the ViewNode
93      NCFVisualElement guiElement =
94        (NCFVisualElement) get_visualElement_by_viewNode(targetNode)
              ;
95      if (guiElement!=null)
96        guiElement.updateUIAccSkwNode();
97    }
98
```

```
99  // Method to set a library-specific very parent container
100 public void setLibrarySpecific_MainContainer(Object
        mainContainer){
101   mainLibraryGUIContainer = (JPanel)mainContainer;
102 }}
```

Listing D.6: Specification of the HASymbolicManager

D.1.4 Deployment Descriptors

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3   <NmlIdName> House Automation </NmlIdName>
4   <Description>
5    The software system may consist of virtual Sensors,
        Controllers and Actuators. With help
6    of controllers outputs of Sensors and inputs of Actuators can
         be connected. Controllers
7    specify conditions under which the signal can be sent to
        connected Actuators.
8   </Description>
9   <InitialExpression>
10   <DSComponentElement> MainContainer </DSComponentElement>
11   <DSOperationElement> Instantiate_DSO </DSOperationElement>
12  </InitialExpression>
13  <LanguageElements>
14   <Element>DSC/MainContainerDSC.xml</Element>
15   <Element>DSC/SensorDSC.xml</Element>
16   <Element>DSC/ControllerDSC.xml</Element>
17   <Element>DSC/ActuatorDSC.xml</Element>
18   <Element>DSC/EmulatorDSC.xml</Element>
19   <Element>DSC/ActionDSC.xml</Element>
20   <Element>DSO/ConnectSC.xml</Element>
21   <Element>DSO/ConnectCA.xml</Element>
22  </LanguageElements>
23  <Views>
24   <View>
```

```
25      <Id>Device-types (statics)</Id>
26      <ContainerManager>viewSpec/HAContainerManager.cm</
           ContainerManager>
27      <SymbolicManager>viewSpec/HASymbolicManager.sm</
           SymbolicManager>
28    </View>
29   </Views>
30 </root>
```

Listing D.7: Main deployment descriptor for the composition system for the House Automation domain

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3    <LanguageId> MainContainer  </LanguageId>
4    <DSPCTLElement> MainContainerDSC.dsc </DSPCTLElement>
5  </root>
```

Listing D.8: Deployment descriptor MainContainer.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3    <ToolBarIcon> SensorIcon.gif </ToolBarIcon>
4    <LanguageId> Sensor </LanguageId>
5    <DSPCTLElement> SensorDSC.dsc </DSPCTLElement>
6    <InstantiationOperation>
7      neurath.tdframework.dspctl.houseautomation.Instantiate_DSO
8    </InstantiationOperation>
9  </root>
```

Listing D.9: Deployment descriptor SensorDSC.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3    <ToolBarIcon> ControllerIcon.gif </ToolBarIcon>
4    <LanguageId> Controller </LanguageId>
5    <DSPCTLElement> ControllerDSC.dsc </DSPCTLElement>
6    <InstantiationOperation>
7      neurath.tdframework.dspctl.houseautomation.Instantiate_DSO
```

```
8   </InstantiationOperation>
9  </root>
```

Listing D.10: Deployment descriptor ControllerDSC.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3   <ToolBarIcon> ActuatorIcon.gif </ToolBarIcon>
4   <LanguageId> Actuator </LanguageId>
5   <DSPCTLElement> ActuatorDSC.dsc </DSPCTLElement>
6   <InstantiationOperation>
7    neurath.tdframework.dspctl.houseautomation.Instantiate_DSO
8   </InstantiationOperation>
9  </root>
```

Listing D.11: Deployment descriptor ActuatorDSC.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3   <ToolBarIcon> EmulatorIcon.gif </ToolBarIcon>
4   <LanguageId> Emulator </LanguageId>
5   <DSPCTLElement> EmulatorDSC.dsc </DSPCTLElement>
6   <InstantiationOperation>
7    neurath.tdframework.dspctl.houseautomation.Instantiate_DSO
8   </InstantiationOperation>
9  </root>
```

Listing D.12: Deployment descriptor EmulatorDSC.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <root>
3   <ToolBarIcon> ActionIcon.gif </ToolBarIcon>
4   <LanguageId> Action </LanguageId>
5   <DSPCTLElement> ActionDSC.dsc </DSPCTLElement>
6   <InstantiationOperation>
7    neurath.tdframework.dspctl.houseautomation.Instantiate_DSO
8   </InstantiationOperation>
9   <MergingOperation>
```

```
    neurath.tdframework.dspctl.houseautomation.
        Merge_Actuator_Action_DSO
  </MergingOperation>
</root>
```

Listing D.13: Deployment descriptor ActionDSC.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
 <ToolBarIcon> ConnectSCIcon.gif </ToolBarIcon>
  <LanguageId> ConnectSC_DSO </LanguageId>
  <DSPCTLElement> ConnectSC_DSO.dso </DSPCTLElement>
  <ParamSignatures>
    <Parameter>
      <Name>s</Name> <Setter>setS</Setter>  <Getter>getS</Getter>
    </Parameter>
    <Parameter>
      <Name>c</Name> <Setter>setC</Setter>  <Getter>getC</Getter>
    </Parameter>
  </ParamSignatures>
</root>
```

Listing D.14: Deployment descriptor ConnectSC.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root>
  <ToolBarIcon> ConnectCAIcon.gif </ToolBarIcon>
  <LanguageId> ConnectCA_DSO </LanguageId>
  <DSPCTLElement> ConnectCA_DSO.dso </DSPCTLElement>
  <ParamSignatures>
    <Parameter>
      <Name>c</Name> <Setter>setC</Setter>  <Getter>getC</Getter>
    </Parameter>
    <Parameter>
      <Name>A</Name> <Setter>setA</Setter>  <Getter>getA</Getter>
    </Parameter>
  </ParamSignatures>
</root>
```

Listing D.15: Deployment descriptor ConnectCA.xml

D.2 Design Phase

```
1  // ##Filename:Temperature.java
2  public class Temperature extends Thread {
3   private String icon= "" ;
4   private java.util.LinkedList listeners= new java.util.
       LinkedList();
5
6   public Temperature() {}
7
8   public void run() {
9    while (true ) {
10     try {
11       Thread.currentThread().sleep(1000 );
12       emulateSignal();
13     }
14       catch (Exception e){e.printStackTrace();}
15   }
16  }
17
18  public void seticon(String icon1) {
19   icon = icon1; }
20
21  public String geticon() {
22   return icon; }
23
24  public void receivingSignal(int signal) {
25   fireTemperaturePrimEvent(signal); }
26
27  public void addTemperatureEventListener(
       TemperatureEventListener listener) {
28   listeners.add(listener); }
29
```

```
30  public void removeTemperatureEventListener(
       TemperatureEventListener listener) {
31    listeners.remove(listener); }
32
33  public void fireTemperaturePrimEvent(int value) {
34    System.out.println("Sensor Temperature sends a signal "+value
       );
35    for (int i= 0 ;i < listeners.size();i++) {
36    ((TemperatureEventListener) listeners.get(i)).
37        processTemperaturePrimEvent(value);
38    }
39  }
40
41  public void emulateSignal() {
42    double number= Math.random();
43    int min= 0 ;
44    int max= 200 ;
45    int signal= (int) (min + (max - min) * number);
46    receivingSignal(signal);
47    }
48  }
```

Listing D.16: Generated program code for the `Temperature` sensor

```
1  // ##Filename:TempContr.java
2  public class TempContr extends Thread implements
       TemperatureEventListener {
3    private Alarm actuator= null ;
4    public int t1 = 0 ;
5
6    public TempContr() {}
7
8
9    public void run() {
10     while (true) {
11       try {
12         Thread.currentThread().sleep(1000 );
13       }catch (Exception e){e.printStackTrace();}
```

```
14    }
15  }
16
17  private void check() {
18    System.out.println("Controller checks the condition (t>100?)
        ");
19    if (t1 > 100) actuator.turnOn(); }
20
21  public void setActuator(Alarm a) {
22    actuator = a;}
23
24  public void processTemperaturePrimEvent(int value) {
25    System.out.println("Controller TempContr receives a signal t1
        ="+value);
26    t1 = value;
27    check(); }}
```

Listing D.17: Generated program code for the `TempContr` controller

```
1  // ##Filename:Alarm.java
2  public class Alarm extends Thread {
3   private String icon= "" ;
4
5   public Alarm() {}
6
7
8   public void run() {
9     while (true ) {
10      try {
11        Thread.currentThread().sleep(1000 );
12      }catch (Exception e){e.printStackTrace();}
13    }
14  }
15
16  public void seticon(String icon1) {
17    icon = icon1;}
18
19
```

```
20  public String geticon() {
21    return icon;}
22
23
24  public void turnOn() {
25     System.out.println("Alarm: turned ON");
26   }
27
28   public void turnOff() {
29     System.out.println("Alarm: turned OFF");
30   }
31 }
```

Listing D.18: Generated program code for the `Alarm` actuator

```
1 // ##Filename:TemperatureEventListener.java
2 public interface TemperatureEventListener extends java.util.
     EventListener {
3   public void processTemperaturePrimEvent(int value) ;
4 }
```

Listing D.19: Generated program code for the `TemperatureEventListener` listener interface

```
1  // ##Filename:Test.java
2  public class Test{
3    public static void main(String[] args){
4      Temperature s = new Temperature();
5      TempContr c = new TempContr();
6      Alarm a = new Alarm();
7
8      s.addTemperatureEventListener(c);
9      c.setActuator(a);
10
11     s.start();
12   }
13 }
```

Listing D.20: A program to start the designed House Automation system

Appendix E

Specification of the UIIE Parser

Listing E.1 shows a UIIE parser specification written in the JavaCC [56] environment. This part defines how a UIIE expresison turns into DS-PCTL expression.

```
1  options{
2    STATIC=false;
3  }
4  PARSER_BEGIN(UIIE_Parser)
5   package neurath.vailevel.uiieparser;
6
7   import java.util.*;
8   import neurath.templatelevel.pctl.*;
9   import neurath.tdlevel.skw.*;
10  import neurath.tdlevel.dspctl.*;
11  import neurath.tdlevel.dspctl.dsos.*;
12  import java.lang.reflect.*;
13
14  public class UIIE_Parser{
15    public static HashMap globalValue = new HashMap();
16    private AbstractPctlExpressionNode expressionRoot = null;
17    private boolean completedExpression = false;
18    private Stack stack = new Stack();
19    private Stack paramStack = new Stack();
20    private Stack instanceStack = new Stack();
21
22    public boolean isCompletedExpression(){
```

```
23      return completedExpression; }
24    public AbstractPctlExpressionNode getExpressionRoot(){
25      return expressionRoot; }
26    public void reset(){
27      expressionRoot = null;
28      completedExpression = false;
29      stack.clear();
30      paramStack.clear();
31      instanceStack.clear();
32      globalValue.clear(); }
33    public void resetIfComplete(){
34      if (completedExpression)  reset();
35    }
36    public static void setGlobalValue(String key, Object value){
37      globalValue.put(key, value);
38    }
39    protected void setParameterOfOperation(Object operation,
          String setMethodName, AbstractPctlExpressionNode value){
40    try{
41      Class[] paramTypes = new Class[1];
42      paramTypes[0] = Class.forName("neurath.templatelevel.pctl.
          AbstractPctlExpressionNode");
43      Method setMethod =
44        operation.getClass().getMethod(setMethodName, paramTypes);
45
46      Object[] val = new Object[1];
47      val[0] = (AbstractPctlExpressionNode) value;
48      setMethod.invoke(operation, val);
49    } catch (Exception e){e.printStackTrace(); }
50  }}
51
52  PARSER_END(UIIE_Parser)
53  SKIP : {" "|"\t"|"\n"|"\r"|"\n\r"}
54  TOKEN:{< INSTANTIATE: "INSTANTIATE">}
55  TOKEN:{< COMPONENT: "COMPONENT" >}
56  TOKEN:{< TYPE: "TYPE">}
57  TOKEN:{< PARAM: "PARAM">}
```

```
58 TOKEN:{< EQUALS: "=">}
59 TOKEN:{< INSTANCE: "INSTANCE" >}
60 TOKEN:{< MERGING: "MERGING" >}
61 TOKEN:{< OPERATION: "OPERATION" >}
62 TOKEN:{< SEP: "|" >}
63 TOKEN:{< ID: ["a"-"z","A"-"Z","_","."](["a"-"z","A"-"Z","_","0"-
       "9","."])*>}
64
65 boolean Start():{}
66 { Original()
67   {return completedExpression;}
68 }
69 void Original():{}
70 { LOOKAHEAD(2)
71   clicked_component_type() <SEP> clicked_instance()
72   |
73   clicked_operation_type() (<SEP> clicked_object_for_parameter()
       )*
74 }
75 void clicked_component_type():{ Token t;}
76 {  component_type() }
77 void clicked_operation_type():{ Token t;}
78 {  operation_type() }
79 void component_type():{ Token comp; Token op;}
80 {  <INSTANTIATE> <COMPONENT> typeAssign() <OPERATION> typeAssign
       () <MERGING> mergeAssign()
81 }
82 void operation_type():{ Token op;}
83 {  <INSTANTIATE> <OPERATION> typeAssign() (<PARAM> <EQUALS>
       paramAssign())*
84 }
85 void paramAssign():{Token value;}
86 {  value=<ID> {paramStack.push(value.image);}
87 }
88 void typeAssign():{ Token value;}
89 {  <TYPE> <EQUALS> value=<ID>{stack.push(value.image);}
90 }
```

```
91  void mergeAssign():{ Token value;}
92  {   <EQUALS>value=<ID>{stack.push(value.image);} }
93  void clicked_instance():{ Token t;}
94  {   instance() }
95  void instance():{ Token instanceName; }
96  {   <INSTANCE> instanceName=<ID> {
97
98      // pop the saved name of the merging operation type
99      String mergeOp = (String) stack.pop();
100     // pop the saved name of the instantiation operation
101     String instOp = (String) stack.pop();
102     // pop the saved name of the type
103     String type = (String) stack.pop();
104     //instantiate the type with DOMAIN SPECIFIC instantiation
          molop
105     AbstractPctlExpressionNode instOperation = null;
106     try{
107       // operation
108       Class theClass = Class.forName(instOp);
109       instOperation = (AbstractPctlExpressionNode)theClass.
            newInstance();
110       instOperation.setLeaf(new StringPctlExpressionNode(type));
111     }catch(Exception e){e.printStackTrace();}
112     // Looking for the instance object in the context
113     SkwContext skwContext = neurath.tdlevel.TDEnvironment.
          getSkwContext();
114     SkwNode node = skwContext.searchForNodeById(instanceName.
          image);
115     AbstractPctlExpressionNode parentNode = node.getDspct();
116     //instantiate the type with DOMAIN SPECIFIC instantiation
          molop
117     Merge_DSO mergeOperation = null;
118     try{
119       // operation
120       Class theClass = Class.forName(mergeOp);
121       mergeOperation = (Merge_DSO)theClass.newInstance();
122       mergeOperation.setParent(parentNode);
```

```
123       mergeOperation.setChild(instOperation);
124       System.out.println("UIIE Parser: instance(), Merge_DSO is
              formed");
125     }catch(Exception e){e.printStackTrace();}
126     expressionRoot = mergeOperation;
127     completedExpression = true;
128   }}
129 void clicked_object_for_parameter():{Token instanceName;}
130 { <INSTANCE> instanceName=<ID>{
131     // Pushing the name of the instance
132     instanceStack.push(instanceName.image);
133     //check if all parameters are assigned with concrete
            variables
134     //only if all parameters are defined the operation will be
            applied
135     int paramsAmount = instanceStack.size();
136     if (instanceStack.size()==paramStack.size()){
137       AbstractPctlExpressionNode operationObject = null;
138       //pop the saved name of the instantiation operation
139       String operation = (String) stack.pop();
140
141       try{
142         //instantiate operation from the string name of its type
143         Class theClass = Class.forName(operation);
144         operationObject = (AbstractPctlExpressionNode)theClass.
              newInstance();
145       }catch(Exception e){e.printStackTrace();}
146
147       for (int i=0; i<paramsAmount;i++){
148         //Processing param number i;
149         //Request instance
150         //Looking for the instance object in the context
151         SkwContext skwContext = neurath.tdlevel.TDEnvironment.
              getSkwContext();
152         String instString = (String) instanceStack.pop();
153         SkwNode node = skwContext.searchForNodeById(instString);
154         AbstractPctlExpressionNode paramValue = node.getDspct();
```

```
155
156         //Request all setters
157         HashMap setters = (HashMap)globalValue.get("PARAMS
                SETTERS");
158         //Request all getters
159         HashMap getters = (HashMap)globalValue.get("PARAMS
                GETTERS");
160
161          //Get next parameter from the paramStack
162
163          String param = (String)paramStack.pop();
164         //Request setter
165         String paramSetter = (String)setters.get(param);
166         //Request getter
167         String paramGetter = (String)getters.get(param);
168
169         //Set parameter
170         setParameterOfOperation(operationObject, paramSetter,
                paramValue);
171         } // for
172
173         expressionRoot = operationObject;
174         completedExpression = true;
175       } // if
176   }}
```

Listing E.1: A specification of the UIIE Parser