A Design Pattern Oriented Programming Environment

Diplomarbeit

Universität Rostock
Fachbereich Informatik

vorgestellt von Seemann, Normen
geboren am 31.05.1976 in Rostock
Matrikel-Nr.: 094200998
Betreuer: Prof. Dr.-Ing. habil. Peter Forbrig,
Dr.-Ing. Ralf Lämmel
Abgabedatum: 01.10.1999

Zusammenfassung

Ein Entwurfsmuster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß diese Lösung beliebig oft anwendbar ist, ohne daß man sie jemals ein zweites Mal gleich ausführen muß. Das Ziel dieser Arbeit besteht nun darin, diesen abstrakten Grundgedanken formal in einem Modell zu fassen, das Entwurfsmuster und damit vebundene Konzepte direkt unterstützt und als Erweiterung des objektorientierten Programmiermodells verstanden werden kann. Im Anschluß daran wird ein Sprachentwurf für eine sogenannte *design pattern orientierte* Programmiersprache vorgestellt, die es ermöglichen soll, Entwufsmuster zu implementieren und wiederzuverwenden.

Abstract

A design pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that one can use this solution many times over, without ever doing the same way twice. The aim of this master's thesis is to formalize this abstract basic idea in order to define a model which directly supports the notion of the design pattern and related concepts and which additionally can be conceived as extension of the object oriented model. Following that, a prototype of a *design pattern oriented* programming language is introduced allowing to implement and to reuse design patterns in an efficient manner.

CR-classification

D.1.5, D.2.1, D.2.2, D.2.3, D.2.13, D.3.1, D.3.2, D.3.3, F.3.2, F.3.3

Keywords

programming techniques, design patterns, software engineering, software development, reusability, programming languages, algebraic specifications, abstract data types, denotational semantics

Remarks

The results of the master's thesis by Stefan Bünnig ([6]) and this master's thesis have been contributed to paper [7]. For this purpose, the theoretical framework and the language *PP* have been revised and modified. Therefore, terminology and the notations used in this thesis may slightly differ from the terminology and the notations used in [7]. However, the meanings of notions and concepts have been preserved.

Contents

Chapter 1

Introduction

The development of large scale software systems in a systematic way is still a challenging task in software engineering. The use of structural, modular and object oriented programming techniques and environments have proven to be powerful and reliable for the creation of correct, reusable and maintainable software.

Specifically, the object oriented programming model has improved the quality of software by providing structures for better support of abstraction, encapsulation and reusability. In recent years, these properties have become more important since hardware and software systems have grown bigger and more complex. Also, customers have made higher requirements on the quality of software. Object oriented programming languages like Eiffel, Smalltalk or C++ provide basic facilities for network programming, database access, etc. in form of class libraries which exploit these concepts. This helps developers concentrate on the solving of the actual problems, without having to spend time on reinventing already implemented solutions to common problems.

The object-oriented paradigm primarily involve objects. These usually represent abstractions of real world entities. Objects are typically defined by classes in programming languages. Classes representing different entities may be related to each other in several different ways. These normally abstract the relationships between the real-word entities that they model. Relationships between classes can be categorized into two kinds - static and dynamic. Static relationships operate at the class level, and include inheritance and subtyping. Dynamic relationships such as association and aggregation operate on the object instance level. The relationships together form what is called the *class structure*.

Classes and relations between classes have to be identified by the designer. These can be achieved using object oriented analysis and design techniques. A specific implementation of the classes and their relations in an object oriented programming language represents the solution of the problem. It is the developer's responsibility to ensure that significant portions of the implementation can be reused in an appropriate way. The ability to do this usually requires deep understanding of object oriented programming techniques, as well as expertise in implementing software systems.

Anectdotal experience in [8] suggests that similar problems require similar designs of classes and their structures. These structures of classes and relationships lead to the notion of a design pattern. However, a design pattern captures more than just a class structure - it also contains algorithms for the so-called *higher behaviour* and paradigms which describe the purpose of the design pattern. In [8] there are listed 23 such design patterns for various purposes. Knowledge of these design pattern can help to create better software in terms of reusability, maintainability and extensibility.

It still remains the task of the developer to implement applications of a design pattern as part of a problem solution. The elements which describe such a design pattern may have to be forced into an ob-

ject oriented form. If the design pattern had been implemented for a very specific problem, and needs to be used in a second similar problem that's different in detail, it would have to be reimplemented from scratch.

Object oriented programming techniques allow the reuse and extension of classes by subtyping, inheritance and other refinement methods. However, since object oriented frameworks do not support any notion of a design pattern, it is impossible to reuse a design pattern itself.

In chapter 2, design pattern oriented notions are introduced informally. Then, a formal approach will be presented in the following chapters. A design pattern oriented model will be introduced as extension of the object oriented model based on algebraic specification techniques. It directly supports design patterns and design pattern oriented refinements in order to overcome the problems mentioned above. In chapter 6, this model will then be used to describe the denotational semantics of the design pattern oriented imperative programming language *PP*. *PP* is a kernel language that supports fundamental features of design pattern oriented programming. In this way, a formal basis is presented for subsequent considerations in this area.

Chapter 2

Design pattern oriented programming

The aim of this chapter is to introduce the way of design pattern oriented programming. To this end, it is necessary to define briefly some basic notions of the object oriented programming model which are crucial for the further understanding of this thesis. This is especially important for the creation of a common unique terminology since many of the following notions have different meanings in literature which are sometimes even contradicting to each other. In order to be independent from any programming language, the following summation of object oriented elements and concepts is based on the view of things from a rather theoretical perspective.

2.1 The object oriented programming model

Object, attributes — Objects in an object oriented program are abstract entities which can represent objects of the real world. From the perspective of the developer, objects contain data and methods which operate on this data. Since these methods represent the only interface for the access to the data of an object, the object is said to *encapsulate* its data. The data itself are usually structured by the use of typed *attributes*. The data, i.e. the state of all attributes, define the state of the object at particular time.
An object exists at the runtime of an object oriented program in its *life cycle*. The life cycle begins with the creation of the object and lasts until its deletion. Only in the time between, the so-called *lifetime* of an object, an object is called *valid*.
In order to distinguish a valid object from another valid object, it is required that each object must be uniquely identifiable at all times. This can be achieved by the use of a unique system-wide *object identity*. The object identity is pre-given by the system can not be changed during the whole life cycle of the object. Therefore, an object identity remains constant regardless the current state of the object.

Abstract data type (ADT) — ADTs are powerful means for the structured development of object oriented programs. They represent the theoretical basis for the typing of objects in object oriented programming languages, i.e. they are the semantic counterpart of classes and class specifications. Conceptually an ADT is a set of models. In this approach, these models are Σ-algebras which correspond to a signature Σ of sorts and operation symbols. A Σ-algebra defines carrier sets for the sorts in Σ and functions on these carrier sets for the operation symbols in Σ. In this way, the signature Σ describes the structure of every Σ-algebra. Chapter 3 provides a detailed, formal introduction to abstract data types.

Class specifications — Algebraic specification techniques are used to restrict the models of an ADT to the subset of all possible models that meet certain requirements. A class specification can be considered as abstract data type that is described by a language construct.
In functional algebraic specification languages, constraints and axioms specify the valid models of an ADT. Constraints are used to restrict carrier sets to a particular term generated form whereas axioms

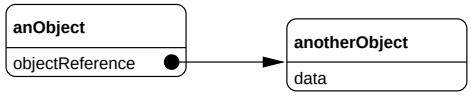

Figure 2.1: *An example for the reference of an object to another object.*

are predicates which can be interpreted in a particular algebra. They are used to ensure that the functions in that algebra have certain properties.

Classes and *interfaces* — A class specification can also be described by a *class* which is an imperative language construct that makes use of object oriented notions like *attributes*, *methods* and *method implementations*. Semantically, the signature of the underlying abstract data type defines sorts for object identities and attribute-based object states in order to provide a foundation for the representation of objects. The corresponding algebras are called *object algebras* (cf. [5]). In this approach, methods are always associated with a class. Therefore, they are also called *selfish methods* and represented by functions in the underlying ADT. They are implemented by an imperative command language specifying the functionality of the abstract data type. For this purpose, the implementation is executed in object algebras to ensure certain properties in an algebra of the ADT to hold. This process takes place analogously to the interpretation of terms provided in the functional approach. Roughly speaking, it can be said that the functional and the imperative approach for the description of class specifications are equivalent in their expressiveness.
From the perspective of the developer, a *class* can be conceived as a factory creating objects or as entirety of all objects of that class. On the other hand, it is often said that an object is associated with a certain class. Both ways are convenient to think of when speaking about the relationship between a class and its objects.
An *interface* of a class comprises the methods and attributes that are associated with that class.

Types — On the one hand, the term *type* is used to express that the value of an element of a programming language, i.e. a variable, a parameter, etc., is an an element of certain domain. In this approach, this set corresponds to a carrier set of a sort in a particular Σ-algebra. On the other hand, two objects are of the same *type* if they have the same interface, i.e. attributes can be accessed and methods can be invoked on both objects in the same way. This sort of interface compatibility can be taken for granted if both object are associated with one class. Therefore, objects of the same class are also of the same type. In the presented setting of this thesis, the notion of a type as domain of values implies interface compatibility, since the same functions can be called on two objects of the same domain in the underlying ADT.

Dynamic relations between objects, *reference semantics* — Dynamic relations between objects are usually modelled by the attributes of the participating objects. A typed attribute contains a value of a basic type (like *integer* or *boolean*) or the object identity of another object of a particular type. Objects can be linked to each other using this method which is also known as reference semantics. This type of relation between objects is called *dynamic*, because objects can participate in new relations or existing relations can be broken off dynamically during the runtime of the program.
However, this relatively simple way of associating one object to another one can cause a variety of problems in object oriented programs. The negligent handling of object identities can lead to what is called *dangling pointers* and to *memory leaks*. Another disadvantage is that certain requirements on the particular relation (e.g. the demand for symmetry of the relation) can not be guaranteed.
Figure 2.1 depicts the notation for a reference of an object to another object. The name of the attribute is denoted at the origin of the arrow.

Static relations between objects, *relations between classes* — Beside the mentioned dynamic relations between objects, there can also exist relations between objects which are static in nature and immutable during the life cycle of the participating objects. These relations are usually not directly associated with objects but with their corresonding classes.

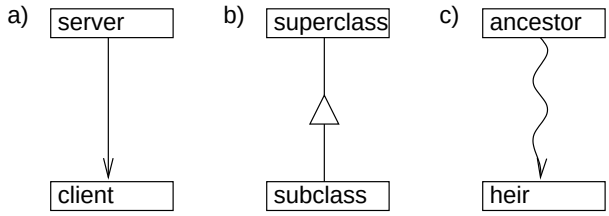

Figure 2.2: *Notations of relations between classes: a) the clientship relation b) subtyping c) inheritance*

In the following, the three main relations in object oriented programming are briefly described. They will be introduced formally in chapter 3.

Clientship relation — The client class uses the server class and adds its own functionality by new attributes and methods. This kind of relation is conceptually associated with the horizontal composition of classes.

Subtyping — A *subclass subtypes* a *superclass* if the subclass contains at least the attributes and the methods of the superclass[1]. Semantically, there is a strong interrelation between objects of the subclass and objects of the superclass. Objects of the subclass can also be treated as objects of the superclass. This conceptually provides the foundation for the *substitution principle*.
Usually programming languages provide an operator to create a subclass based on a specified superclass. Then it can be implied that if a class is created using such an operator, the class is also in a subtype relation to the class specified in the operator.
As a matter of fact, subtyping between classes, alike similar relations between classes, induces a subtype relation between the abstract data types corresponding to those classes. Later it will be become clear that it is important to make this distinction.

Inheritance — The heir class inherits the interface and the functionality from an ancestor class via renamings and model inclusion (simple inheritance) or model relations (generalized notion of inheritance).

Subtyping and inheritance are orthogonal concepts. While subtyping is used to describe relationships between carrier sets of superclass and subclass within a model of the ADT, the aim of inheritance is to refine an ADT consisting of many models. Conceptually, subtyping relates objects whereas inheritance relates models. Therefore it is said that subtyping is a fine-grain structuring mechanism in contrast to inheritance which can be considered as a large-grain structuring mechanism (cf. [5]).

However, a clear distinction between these two concepts can hardly be found in todays object oriented programming languages. It is of importance to notice that most object oriented programming languages implement subtyping but call it inheritance (like *C++*). *Eiffel* even provides a notion of inheritance that is rather a mixture of inheritance and subtyping.

Using these relations, the deseigner can arrange classes to form hierarchies respectively directed acyclic graphs (DAGs). The corresponding graphic notations are depicted in figure 2.2.

Substitution principle, *subtype polymorphism* and *dynamic binding* — The substitution principle is closely related to the subtyping of classes. It means that an object of a certain class can always be substituted by an object of any subclass of this class. This requires several properties of the interface to be satisfied. The methods and attributes of the superclass must also be defined for each subclass. This is ensured by the subtype relation between classes. However, on the level of the programming language, the developer certainly does not want to reimplement all methods or redefine the attributes of a superclass. In fact,

[1] This is a very informal statement. Subtyping will be introduced formally in chapter 3.

instead of doing this, a developer usually tries to reuse existing implementations. Hence, methods can be *overridden*. Overriding means, that a particular method can get a new implementation in a subclass. In most cases, this new implementation of the method calls the old implementation in order to reuse code and to avoid an implementation overhead. If a method is not overridden by a new implementation it remains unchanged in the subclass. Semantically, all implementations of a method, i.e. the original implementation in the class where the method is defined first and the reimplementations when this method is overridden, are used to define the corresponding function in the underlying abstract data type. However, each such implementation, considered separately, only determines what actions have to be performed when this method is invoked for objects of the type this particular implementation is defined for. This means that according to the *dynamic type*[2] of the object, an appropriate implementation is selected. This procedure is known as *dynamic binding* or *dynamic dispatch* on *selfish methods*.

2.2 Implementing design patterns

2.2.1 The notion of the design pattern

The first step in the development of a large scale object oriented program is the analysis of the problem in the real world. Objects and their relations have to be identified and abstracted in order to find a starting point for the object oriented design. Appropriate methods for this process can be found in [2].

The object oriented design as second step follows this analysis phase. Common properties of objects are identified in order to find potential candidates for classes. Additionally, roles and responsibilities of objects as well as interactions between them are investigated. By doing this, the developer tries to determine dynamic and static relations between objects and classes. However, this whole process is not algorithmic in nature. It requires the developer to be experienced and creative, since he already has to focus on reusability and maintenance beside the actual solution of the problem.

Reuse of object oriented programs means that existing classes or class structures are reused. This is not always a simple process, since these classes can be tangled with other, more specific classes which are not subject to reuse. The developer is supported in his endeavor by object oriented programming languages providing features like subtyping, composition, etc. But, as a matter of fact, it is the task of the developer to use these means properly in order to find the necessary abstraction level for a successful reuse.

After the design and the implementation of several object oriented applications one can recognize that similar problems often require similar class structures and similar implementations. The abstraction of these structures leads to the notion of the *design pattern*. The (informal) definition of a design pattern can be found in [8].

Quotation 2.1 design pattern (informal)
'Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same way twice' (Christopher Alexander)[3]

□

As a matter of fact, a design pattern in its abstract sense has nothing special to do with computer science in general or with object oriented programming in particular. However, the nature of a design pattern as

[2]The dynamic type of an object is the type of the object whereas the static type is the declared type of the variable, attribute, etc. Sometimes, the dynamic type of an object will also be referred to as *minimal type*.

[3]This definition corresponds to patterns in the context of buildings and towns. However, this definition stands for the essence of every kind of design pattern, especially for the design patterns in the context of computer science.

an abstract concept is to serve as a kind of template which is used in applications of this design pattern. Applied to the object oriented world, one can think of a design pattern as an object oriented construct. Its application represents the actual object oriented program.

The following uniform structure is given by [8] as fundamental way to describe design patterns.

Representation 2.2 *Structure for the description of a design pattern (informal)*[4]

Design pattern name and classification — The name of the design pattern conveys its essence succinctly. The classification reflects the nature of the design pattern. It helps to find related design pattern. Possible categories include *creational*, *structural* and *behavioral design patterns*.

Participants and their responsibilities — The components (classes) and/or objects participating in the design pattern and their responsibilities.

Component (class) structure — The components in the design pattern and their relations to each other define the component structure which is graphically represented using a notation based on the *Object Modeling Technique (OMT)*. Additionally, methods and attributes for the components are listed here.

Interactions and Collaborations — Interactions an collaborations are crucial for the understanding of the dynamic behavior of the design pattern. Graphic interaction diagrams are helpful to visualize the information flow in the design pattern.

Higher behavior — The design pattern can also be conceived as an entity. The higher behavior describes the behavior of a design pattern from a perspective above the components. It can be used to express *global integrity conditions* or to define how a design pattern should react as a unit within the context of other design patterns or classes.

□

As stated in the head of the above definition, such a structure can be used to describe a design pattern in an informal way. This means that such a description can be interpreted in different ways. These ambiguities are only avoidable if there is a *design pattern oriented* formalism for the description of a design pattern. This formalism will then be called a *design pattern oriented model*. If the description of a design pattern is written in a programming language which bases on such a design pattern oriented model then this programming language is called *design pattern oriented language* and the description of the design pattern is called *implementation*. The entirety of all concepts notions and ideas including the means for the implementation of design patterns is called the *design pattern oriented paradigm*.

In the sequel, it will strictly be distinguished between the implementation and the application of a design pattern. In the design pattern oriented paradigm it is possible to do both. This is an advantage over the object oriented programming model.

The next section will explain why the facilities of object oriented programming languages are not sufficient for implementations of design patterns. To this end, problems occurring when design patterns are implemented in object oriented programming languages will be explained and underpinned using the example of the *composite design pattern*.

The definition for a design pattern and the possibilities for its description given in this section are informal for the time being. The notion of a design pattern will be formally introduced in chapter 4.

[4] In [8] there are several other items listed. However, they are not needed for subsequent considerations within this thesis. These items include *Intent*, *Also known as*, *Known uses*, etc. On the other hand, the category *Higher behavior* has been added. Later, this will be relevant for the conception of a design pattern as entity.

2.2.2 Problems when design patterns are implemented

This section is opened with a thesis that is substantial for further considerations.

Thesis 2.3
Using object oriented programming techniques, it is possible to apply a design pattern to a special problem, however, it is not possible to implement the design pattern itself. For this purpose, it is necessary to use more advanced, design pattern oriented programming techiques.

□

The concepts provided by object oriented progamming languages, are sufficient even for the implementation of huge software systems. In certain situations the developer can even choose from several ways to design a particular feature. Then, why are there problems implementing design patterns using the object oriented programming model?

An adequate approach for the implementation of a design pattern in an object oriented programming language could be to represent the components by actual classes of the language. Static relations between components are modelled by object oriented relations. Responsibilities, interactions of the components of the design pattern could be implemented by methods of the corresponding classes whereas the higher behaviour could be implemented by global functions or by an additional class. The refinement process could then be modelled by conservative object oriented mechanisms like subtyping, etc. However, a developer doing so will soon encounter almost invincible problems.

For the following summation of problems it is assumed that an arbitrary design pattern is implemented in an object oriented programming language supporting *normal* object oriented concepts (e.g. *Eiffel* or *C++*). In this case, the following problems occur:

Encapsulation problems — A design pattern is a closed system which encapsulates its component structure from the outside. However, an object oriented program usually consists of many homogeneous classes. Some of these classes may be the result of an application of a design pattern. Other classes may have been added by the developer. The consequence is a set of classes which are all treated in the same way. Therefore, a separation of design pattern internals from external classes and other design patterns is not possible. This problem will be called *static encapsulation problem*. The concept of *nested classes* has been added in recent versions of the programming language *Java*. It overcomes the static encapsulation problem easily. However, it is not part of the actual object oriented paradigm and therefore not part of subsequent considerations.
Furthermore, classes are instantiated obtaining objects. However, the objects of an application of a design pattern should be considered separately from all other objects. They can be grouped into dynamic units which should be conceived as separate systems of objects. These units have to be encapsulated and treated as entities which correspond to an instance on a higher level. An application of a design pattern can have many such instances. Each instance is associated with an encapsulated group of component instances. Hence, component instances should exist in a dependent, separate space. Object oriented programming languages do not provide facilities for this extended concept of instances, therefore they have to be simulated. However, analogously to the first mentioned problem, this can hardly be achieved in an adequate way using object oriented programming languages without violating principles of good and clean programming. This problem will be called *dynamic encapsulation problem*.

Reusability problems — The component structure of a design pattern is usually very abstract in nature. A well designed design pattern should contain a general component structure which is not committed to any application in the first place. When a design pattern is applied, the class structure of the application is defined by the component structure of the design pattern. This process requires refinement mechanisms like renamings and/or reimplementations to take place. In an object oriented language, components and their interrelations can be represented by classes and object oriented relations. The

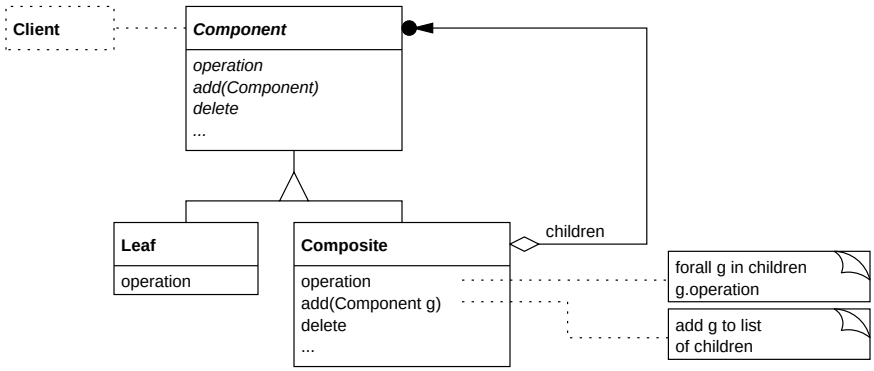

Figure 2.3: *The design pattern* Composite *(only structure).*

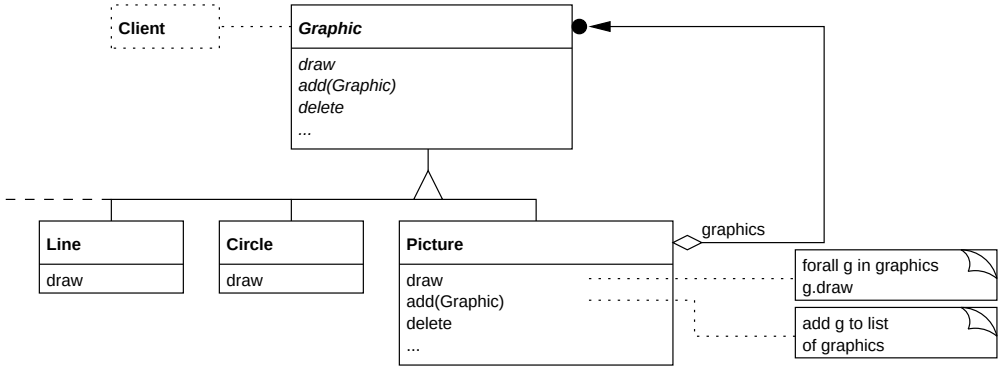

Figure 2.4: *An application of the design pattern* Composite — *The* graphic Composite.

components are refined by application-specific methods and method implementations leading to the classes of the application. However, using object oriented programming techniques, it is impossible to refine a component of the source design pattern in an adequate way without breaking its object oriented relations to other components. Therefore, the component structure of the design pattern can not be transferred to its application. However, this property is essential for a proper application of a design pattern. Thus, the developer is forced to reimplement the class structure of the design pattern in the application without actually reusing the abstract component structure in the implementation of the design pattern. This is a contradiction to the intuitive definition of a design pattern (cf. quotation 2.1) since parts of the original design pattern are lost during the reuse process. A detailed description of reusability problems the programmer can encounter can be found in [6]. Example 2.4 demonstrates this problem.

Example 2.4 Figure 2.3 depicts the component structure of the design pattern *Composite* as proposed in [8] whereas figure 2.4 shows the structure of an application of this design pattern which could be used for graphic programs, CAD software, etc. Graphic elements can either be elementar or groupings of graphic elements. It can easily be recognized, that the component structure of design pattern is very similar to the class structure of its application. Especially, the component *Component* and *Composite* in the design pattern are obviously related to the classes *Graphic* and *Picture* in the application. If the design pattern is implemented in an object oriented prorgamming language, it could be said that *Graphic* and *Picture* have to be refinements of *Component* and *Composite*. Refinements in object oriented programming languages correspond either to subtyping or to delegation.
Therefore, *Graphic* could be designed as subclass of *Component* in order to extend *Component* by a mecha-

nism to draw itself on the screen. Analogously, *Picture* could be designed as subclass of *Composite* in order to implement a routine for the drawing of all contained graphic elements. In this case, however, *Graphic* and *Picture* are not automatically in a subtyping relation as demanded by the *graphic Composite*. Both classes do not know about their mutual refinements. Therefore, they can not be used in their intentional way, since e.g. the draw method in *Picture* can not invoke the draw routine in *Graphic* since *Picture* uses objects of type *Component* instead of using objects of type *Graphic*. One could argue that this problem could be overcome by the use of multiple subtyping[5]. But in this case, the subtyping relation is specified redundantly in the design pattern and in the application. Besides, readability, the subsequent reusability and the maintainability of the code would suffer to a large extent.

□

A design pattern oriented programming model must provide solutions for these problems. Besides, it also has to fulfill the following requirements.

Configurablility — A design pattern should support the development of software systems. Beside its actual implementation which should be as abstract as possible and as concrete as needed, it is also important that the refinement process itself is not rigid. It should rather be possible to configure the refinement, e.g. it should be specified by the user.

Identifiability — This property is related to the *encapsulation problems*. A design pattern should be identifiable in the source code, i.e. classes of applications of a design pattern can be associated with that design pattern. In analogy to that, the design pattern oriented programming model should also provide mechanisms to identify application instances (cf. encapsulation problems).

2.3 Design pattern oriented software engineering

The preceeding sections proved the necessity for a design pattern oriented model, henceforth called *PatternModel*, in order to support the desired features mentioned above. It will become intelligible in the sequel that the introduced *PatternModel* can be conceived as an extension of the object oriented model.

This approach for a design pattern oriented model will eventually be used for the definition of the design pattern oriented language *PP*. *PP* belongs to the class of imperative programming languages. Its syntax bases on the syntax of **OP** as introduced in [5]. The meaning of the individual object oriented statements and constructs should be obvious without further explanations.

This section is structured as follows. The first subsection introduces the design pattern in the context of *PP*, its instantiation and its role in a design pattern oriented program. Several parts of a design pattern are implemented using an imperative *command language*. In the second subsection, restrictions are imposed on this language to ensure that an *PP* program is type safe. The subsection 2.3.3 addresses refinements of design patterns in *PP*. A refinement represents a fundamental mechanism to reuse and combine design patterns. The concluding section discusses the usage of design patterns in *PP*.

2.3.1 The design pattern in *PP*, Instantiation

The notion of the design pattern represents the fundamental element in *PP*. It can be pictured as static unit which can be instantiated at runtime. In this process, an instance is associated with an *identity* and created with an initial state. The state is then modified by functions, the identity, however, is immutable. Finally, the instance is deleted. These three stages altogether form the life cycle of the instance. Up to this point, a

[5] *Picture* subtypes both *Composite* and *Graphic*.

design pattern instance behaves identically like a normal object of a class.

A design pattern is specified by a language construct like e.g. the class construct in object oriented programming languages. This design pattern construct must be powerful enough to implement a design pattern according to representation 2.2, i.e. a design pattern implementation written in *PP* must reflect the schematic structure of a design pattern given in representation 2.2. A design pattern in *PP* is defined as follows.

Representation 2.5 *design patterns in* PP
A design pattern in *PP* constists of the following elements:

Components — Components implement the participants of the design pattern. They are described by embedded component constructs. One can think of a component as a class in the object oriented sense within the boundaries of the design pattern. This is also emphasized by the syntax of the component construct which is very similar to the syntax in which classes are described in object oriented languages. A component consists of attributes and methods implementing its responsibilites on a local level. Moreover, it can be in any object oriented relation to other components of the design pattern in order to reflect the component structure of the design pattern. Components are not visible from outside the design pattern, although, a component is aware of other design patterns. This means, that attributes or methods of a component can use any component type of this design pattern and any other design pattern for their definition. As can be seen later, this strict encapsulation is crucial for every kind of refinement.
Components are instantiated at runtime. Alike classes, a component instance is associated with an immutable identity and a state which is modelled by attributes of the component. Component instances behave like instances of design patterns except for one property. These instances are associated with one particular design pattern instance. In contrast to object oriented classes, the instantiation of a design pattern conceptually opens a new space which is exclusively reserved for component instances whose components are defined inside that design pattern. Considered from this point of view, there is a strong dependency relationship between the design pattern instance and the contained component instances. It can even be said that a design pattern instance contains component instances.

Attributes — Design patterns also have attributes. These attributes are classified into *internal* and *external* attributes. Internal attributes are of a component type whereas external attributes refer to any basic or design pattern type. The reason for this distinction lies in the fact that it is not allowed to access internal attributes from outside the design pattern, since the components of the design pattern are encapsulated by the design pattern.

Methods — Methods of design patterns are used to implement its higher behavior. These methods are also selfish, i.e. they are invoked using a design pattern instance. Alike attributes, methods of a design pattern can also be classified into *internal* and *external* methods depending on the parameter and return types.

□

Example 2.6 Figure 2.5 depicts the design pattern *List* implementing a usual linked list. The design pattern itself is surrounded by a frame symbolizing the encapsulation of its components. The design pattern contains two components: *Item* and *ListComp*. *ListComp* implements a linked list of *Item* elements. It contains an attributes *first* and *current* in order for being able to navigate in the list. Additionally, it provides methods like *add*, *delete* or *rewind* which serve an obvious purpose. *Item* contains an attribute *next* which points to the next element in the list.

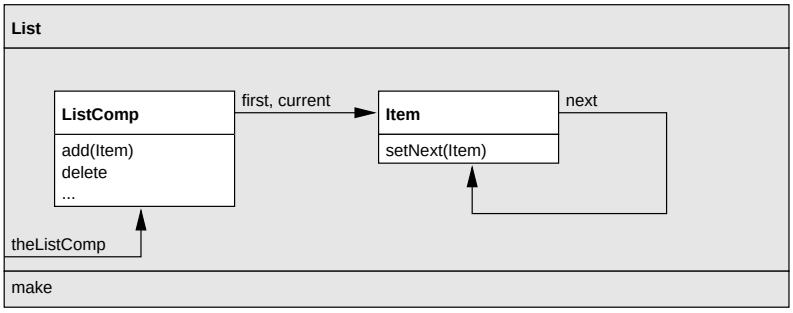

Figure 2.5: *The design pattern* List *in OMT-like notation.*

The design pattern *List* itself contains an (internal) design pattern attribute $theListComp$ to hold an instance of the component *ListComp*. The external function *make* is used as a constructor of the design pattern. □

The definition of a set of *attributes* is normally used to model the state of a class instance in the object oriented world. An attribute of a class has a type and for an instance of that class it is associated with the identity or a value of that type. The current state of a class instance is determined by the current state of all attributes. The current state of a design pattern instance, however, is determined by the current state of the attributes together with the state of all current component instances contained by that design pattern instance.

The associations of identites with states for a system of design patterns and components are described by an environment. One special environment represents the state of the system, the set of all possible environments represents every possible state of the system. The state of the system can be compared to the state of a single design pattern instance. The system contains instances of design patterns, whereas the design pattern in turn contains instances of components.

A design pattern oriented program is a system of design patterns of the above form. It can be executed in the *PatternModel* by the instantiation of a specified top design pattern and the following execution of a designated main method of that design pattern.

design patterns can be considered as generalization of a class. In particular, a class can be considered as design pattern without components. Even the refinement relation between two design patterns without components behaves equal to the subtype relation between classes in the object oriented programming model.

As one can see, there are many similarities to object-oriented concepts. However, they have to be extended and adapted in order to describe the semantics of *PP*.

2.3.2 Implementations, locations, levels and visible elements

Beside its data definition part, *PP* also integrates a *command language* for the implementation of design pattern- and component methods. This follows the idea of commands in imperative object oriented languages applied to the design pattern oriented world. A command is executed in an environment transforming the current state of the system to a new one. In this way, a command can implement a method which is then called *method implementation*.

Due to the encapsulation of components by design patterns, it is necessary to define and to execute commands dependent on the location where this command is used. A location can be a design pattern or a component inside a design pattern or the *global location* or *global level*. It is said that a component method implementation of a method contained by component C is executed in C, analogously a design pattern method implementation of a method contained by design pattern P is executed in P.

The location of a method implementation respectively a command induces *visible* components, design patterns and methods that can be accessed by the command. All other elements are invisible and can therefore not be accessed. The definition of a visible elements is of importance for type safety. The rules for the definition of elements that are visible from a particular level can be summarized as follows.

Definition 2.7 *visible elements*
The following elements of PP are visible from the global level:

- all design patterns
- all attributes and methods of design pattern which only use design pattern types or basic type in their definition

The following elements of PP are visible from a design pattern P:

- all elements that are visible from the global level,
- all components of P
- all attributes and methods of all components of P,
- all attributes and methods of P

The following elements of PP are visible from a component C inside a design pattern P:

- all elements that are visible from the design pattern P,

□

The above definition of visible elements considers three levels of nesting: the global level (the same as the global location), the level of all design patterns and the level of all components C inside design patterns. However, the theoretical framework presented in the following chapters can deal with design patterns that have an arbitrary depth of nesting. In this context, design patterns and components are treated in an orthogonal way, i.e. design patterns can contain components, components in turn can contain other components and so on. Then, there is no need for a distinction between a design pattern and a component. Related concepts (including the definition of the visibilty of elements) that are briefly introduced in this section, have to be generalized in order to support this orthogonal nesting of components. Since this chapter is intended to address programming related issues in PP which only supports three levels of nesting, only these three levels are considered at this point.

The *external part* of a design pattern is defined as that part of the design pattern which is visible from the global level. The *internal part* of a design pattern is defined as that part of the design pattern which is not visible from the global level.

Considered from *outside* a design pattern, i.e. from the global level or from a different design pattern, its internal part and especially its components are not visible. Therefore, a design pattern behaves like a normal class with attributes (its external design pattern attributes) and methods (its external design pattern methods).

The design pattern construct in *PP* serves the purpose of static encapsulation of the components. Beside this, a design pattern in *PP* also represents a meta level for its components. For instance, static attributes[6] in components can be modelled by design pattern attributes. A static attribute does not depend on any particular component instance, but on the component itself. Since component instances are considered with respect to a corresponding design pattern, the design pattern instance itself can also store this static attribute of a component in a *normal* attribute of the design pattern. In this way, there is no need for static attributes in this setting. A design pattern instance could additionally even hold type information about its components.

A design pattern can be pictured as meta level for its components. But, are there any higher meta levels? It is imaginable to implemenent a design pattern as a component of another design pattern. In this way, a higher meta level, i.e. a meta level for the meta level, can be found. It can be continued doing so which yields to even higher levels. The *PatternModel* supports this feature, however, due to complexity reasons, *PP* does not make use of it.

Example 2.8 The excerpt shown in figure 2.6 implements the design pattern *List* in *PP*. The whole implementation is presented in appendix C.1.
The meaning of the constructs should be obvious. A design pattern *List* is defined. Inside *List*, there are two components *Item* and *ListComp* as well as the design pattern attribute *theListComp* and the design pattern method *make* defined. The component *Item* in turn defines an attribute *next* together with some auxiliary methods on component level. The component *ListComp* declares and implements the usual methods and attributes for handling the List.
The design pattern *List* itself contains an (internal) attribute *theList* to hold an instance of the component *ListComp*. The external function *make* is used as a constructor of the design pattern. It creates an instance of *ListComp* with an associated initial state and assigns its identity to *theList*. For this purpose, the command create List is executed with the implicitly passed parameter *self*. The newly created component instance is then created inside the design pattern instance.

□

2.3.3 Refinements

The design pattern *List* in example 2.8 is not designed for instantiation but for reuse. Altough, it can be instantiated by an appropriate command like create List, an instance of *List* would not be able to do anything since the components are not accessable from the outside and there is only the visible *make* constructor.

However, it is not intended that *List* is used in this way. Applying the notion of refinements intoduced in this section, it will be possible to reuse and specialize this implementation of a linked list. The result of this refinement process is then bound to a specific application, e.g. a list dealing with *string* items. A hypothetic design pattern *StringList* could then provide all necessary design pattern methods to access the strings of the list. In contrast to the design pattern *List* which is very abstract, the instantiation of *StringList* is very useful in a situation when a e.g. temporary *StringList* is needed.

Design patterns can be implemented in *PP* from scratch. In most cases, however, the developer wants to reuse already implemented design pattern. In *PP*, this aim can be achieved using the mechanism of *refinement*.

[6] The concept of *static attributes* (also called *class attributes*)is well-known in the object oriented world. A static attribute is an attribute that is instantiated once per class. In this way, all instances of that class share this attribute. The notion of static attributes can easily be applied to components.

```
design pattern List
 components
  component Item
   attributes
    next : Item

   methods
    setNext(anItem : Item) returns Item

   method implementations
    setNext(anItem : Item) returns Item is
     ...
    end
  end component

  component ListComp
   uses components
    Item

   attributes
    first : Item,
    current : Item

   methods
    add(anItem : Item) returns ListComp,
    delete returns ListComp,

     ...

   method implementations
    add(anItem : Item) returns ListComp is
    local
     tempItem: Item

    do
     if self.isEmpty
     then
      self.first := anItem;
      self.rewind
     else
      tempItem := self.current;
      self.current := anItem;
      anItem.setNext(tempItem.next);
      tempItem.setNext(anItem);
      self
     end
    end,

    ...

  end component

 attributes
  theListComp : ListComp

 methods
  make returns List

 method implementations
  make returns List
  do
   self.theListComp := create List::ListComp
  end
end design pattern
```

Figure 2.6: *Implementation of the design pattern* List *in* PP.

In *PP*, a refinement is supported as construct in order to describe an operator that refines a (source) design pattern into a (refined) design pattern. Then, the source design pattern is in a *refinement relation* to the refined design pattern. The refinement relation is defined as follows.

Definition 2.9 *refinement relation between design pattern*
A (source) design pattern is in a refinement relation to a (refined) design pattern iff

1. each component of the source design pattern is injectively refined into a component of the refined design pattern using the notion of generalized inheritance (cf. object oriented relations in section 2.1 and [5]),
2. the object oriented component structure of the source design pattern is preserved in the refined design pattern,
3. the internal part of the source design pattern is refined into the internal part of the refined design pattern using the notion of generalized inheritance,
4. the external part of the refined design pattern subtypes the external part of the source design pattern.

□

The above definition describes the essence of a refinement from a model-theoretical perspective. The refinement operator in *PP* which is based on *program transformations* comprising *renamings* of components, methods, etc. induces a specialized refinement relation.

In *PP*, the refine construct allows to specify the refinement on the design pattern level, whereas the recast construct (cf. appendix A) which as part of the component definition is used to specify the refinement on the component level. As implicated in definition 2.9, a refinement operator must be able to transform the components, design pattern methods and attributes of the source design pattern into the refined design pattern. This includes the possibility to rename the components, their methods and attributes as well as the internal design pattern methods and attributes[7]. The refinement used by *PP* translates implementations of methods of the source design pattern into the context of the refined design pattern with respect to the renamings specified by the refine and recast constructs.

In the refined design pattern, the components of the source design pattern play their original role in the refined environment. They can comprise additional functionality in form of new and reimplemented methods, but their basic behaviour is inherited from the source design pattern. For this purpose, it is important that all components of the source design pattern participate in the refinement and that existing relations between components still hold between their refined counterparts in the refined design pattern.

Beside the mentioned renamings, a refinement in *PP* also allows to reimplement component methods and design pattern methods of the source design pattern. The whole refinement takes place on the level of the language. This implies that e.g. the translation of an implementation of a method in the source design pattern which calls a method that is reimplemented in the refinement process will then call the reimplemented method. By this approach and due to the fact that a refinement preserves the component structure (cf. definition 2.9) of the source design pattern, components can be extended and modified without breaking the component structure which is essential for the reuse of whole component structures. Object oriented programming languages do not support this kind of reuse as mentioned in section 2.2.2.

It is possible to refine from more than one source design pattern. In a complex case, components can be band together in the refined design pattern. Methods of these components can have the same name and signature. The same conflict can happen on the design pattern level. Therefore, a select statement is provided by *PP* which can be used to select the implementation of a method from one of the source design patterns. Since the selection of method implementations of components takes place in the component definitions, it is necessary to label the refinement constructs in order to distinguish between them outside the refinement. Example 2.10 shows the usage of the refine construct.

In the context of other design patterns or their components, the refined design pattern behaves like a subclass of the source design pattern. This also means that design pattern instances behave polymorphic and that the substitution principle can be applied to instances of design patterns. In this way, the design pattern instances are treated like class instances in object oriented systems. This fact is of importance, since the *PatternModel* is designed to extend the object oriented programming model.

The substitution principle in the *PatternModel* relies on the fact, that components of design patterns are invisible from the outside. Otherwise, methods of components of a design pattern could be accessed from other design patterns which would lead to problems of type-safety since these methods do not necessarily have the same name nor take compatible parameters in refinements of that design pattern.

[7]External methods and attributes can not be renamed due to the fourth item in definition 2.9

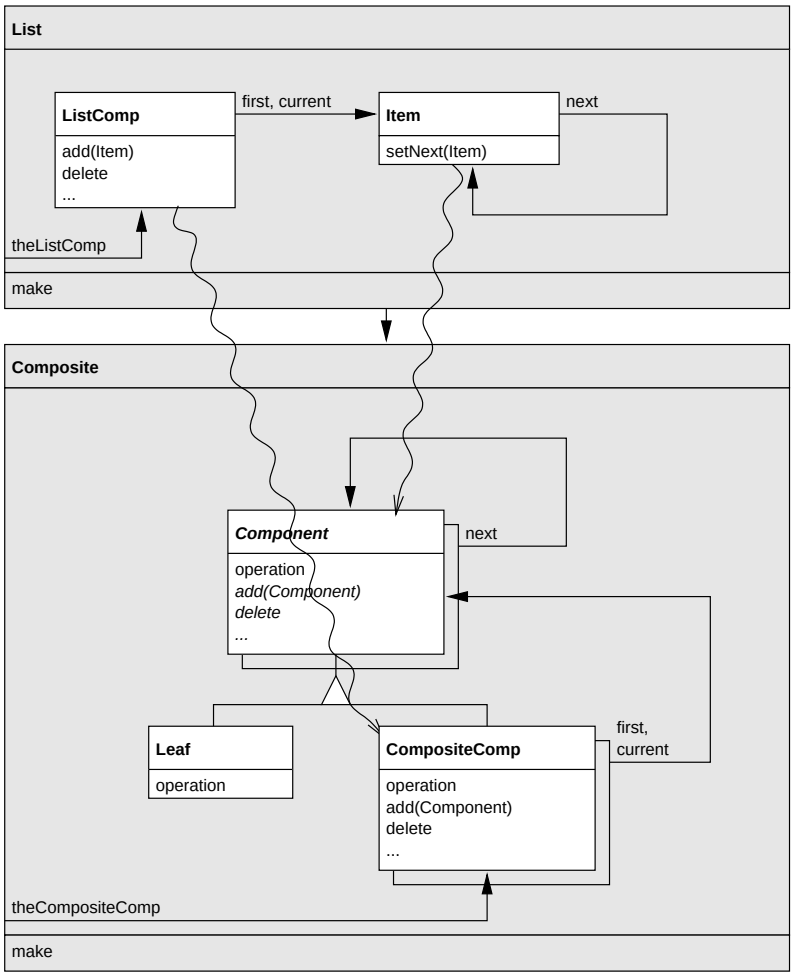

Figure 2.7: *The refinement of design patterns:* ListRef *refining from* List *to* Composite

Example 2.10 Figure 2.8 shows the implemention of the design pattern *Composite* in *PP*. For this purpose, the design pattern *List* (cf. example 2.8 and appendix C.1) and the design pattern *Subtyping* (cf. appendix C.2) are reused by two refinements. Figure 2.3 depicts the refinement *ListRef* refining from *List* to *Composite* graphically. The inheritance relation between the components of the design patterns is pictured by wavy lines.

The design pattern *Composite* basically consists of three components: *Component*[8] together with the subcomponents *Leaf* and *CompositeComp*. *Component* solely serves the purpose to provide a common interface. The components *Leaf* and *CompositeComp* implement their desired behaviour by component methods. *Leaf* represents a sample element whereas the *CompositeComp* is a container of components. All methods declared for *Component* can also be invoked on *Leaf* and *CompositeComp*. *Leaf* provides a certain functionality, *CompositeComp*, however, delegates the method calls to all contained elements.

It is easy to see that the design pattern *Composite* can perfectly reuse the functionality of the design pattern *List*. For that purpose, the components *Item* and *ListComp* in *List* are refined into *Component* and *Composite* in *Composite* using the refinement labelled *ListRef*. Besides, the design pattern attribute *theListComp* is renamed into *theCompositeComp*.

In order to make the examples a little more complex, the subtype relation between e.g. *Component* and

[8] Unfortunately, this name clash could not be avoided.

```
design pattern Composite
 refinements
  SubtypingLeafRef refines Subtyping
   refine
    Parent into Component,
    Child into Leaf
  end refinement,

  SubtypingCompositeRef refines Subtyping
   refine
    Parent into Component,
    Child into CompositeComp
  end refinement,

  ListRef refines List
   refine
    Item into Component,
    ListComp into CompositeComp

   rename by
    theListComp → theCompositeComp
  end refinement

 components
  component Component
   methods
    operation returns Component,
    add(anItem : Component)
     returns Component,

    delete returns Component,
    ...

   method implementations
    operation returns Component is do self end,
    ...

  end component

  component Leaf

  ...

  end component

  component CompositeComp
   methods
    operation returns CompositeComp

   method implementations
    operation returns CompositeCom is
    do
     from
      self.rewind;
      self.current.operation
     until
      (self.next).isVoid
     loop
      self.current.operation
     end
    end
  end component
end design pattern
```

Figure 2.8: *Implementation of the design pattern* List *in* PP.

Leaf are refined from the design pattern *Subtyping* in the refinements *SubtypingLeafRef* respectively *SubtypingCompositeRef*. However, this could also be achieved *on the fly* without such a refinement.
The design pattern *GraphicComposite* represents one special application of the design pattern *Composite*. It is implemented in appendix C.4.

□

2.3.4 Reuse vs. Instantiation of design patterns

According to quotation 2.1, a design pattern is designated for reuse purposes only. This corresponds to refinements in *PP*. A design pattern in *PP* can also be instantiated which seems to contradict this principle in part since instantiation stands for the active use of a design pattern itself. It should be used only in later applications whose objects can be instantiated then. In order to unify these partially contrary ideas, the notion of the design pattern is extended as follows. Roughly speaking, everything in *PP* is a design pattern. A design pattern in the sense of [8] can still be represented by a design pattern in *PP*, but even an application of such a design pattern is now a design pattern in *PP*. Therefore, a design pattern in *PP*

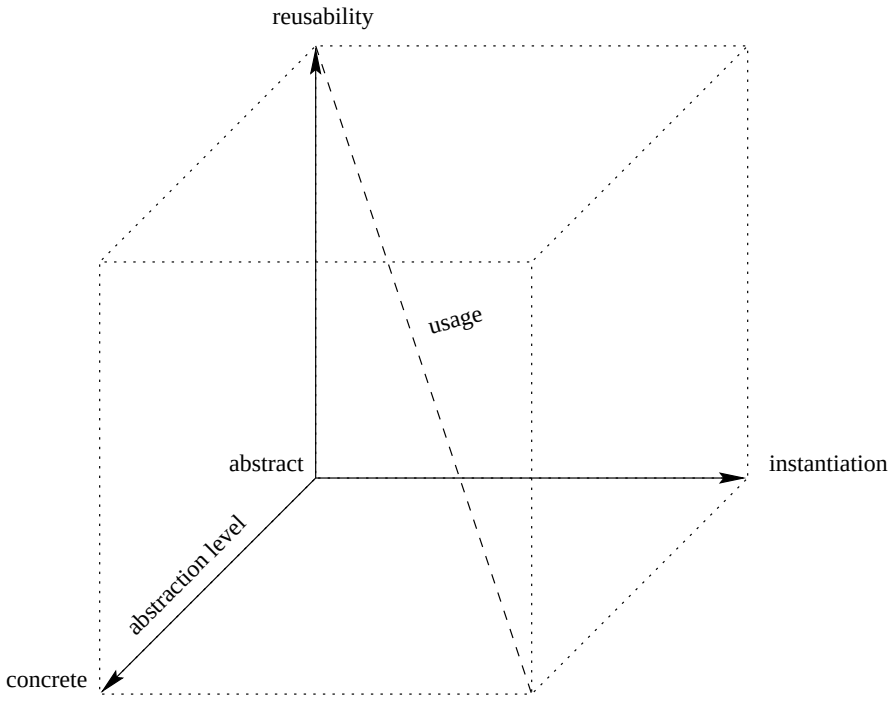

Figure 2.9: The usage of a design pattern: relations between reusability, instantiation and the level of abstraction of design patterns

integrates both ideas: refinement and instantiation. There is no separation between design patterns and applications. Abstract design patterns are refined and combined as needed, eventually leading to a design pattern which actually represents the application in the sense of [8].

Figure 2.9 shows the relations between the reusability, instantiation and the level of abstraction of design patterns. Design patterns that are abstract (e.g. *GoF* design patterns) are usually not suited for instantiation but ideal for reuse purposes. The more specialized a design pattern gets the more useful an instantiation of this design pattern can be. However, at the same time, this design pattern is more difficult to reuse. At the opposite extrem, i.e. a very specialized design pattern, the design pattern can hardly be reused. At this stage, an instantiation makes the most sense just because the design pattern is so specialized.

2.4 Related work

The programming language *Java* uses the concept of *nested classes*. A class can be defined within an embracing class. The nesting mechanism is not limited to three levels, so that there can be further classes nested in this already nested class and so on. When a class in *Java* is subtyped by a subclass, all nested classes of the superclass are automatically nested classes of the subclass. This implies that it is impossible to refine the nested classes themselves. Therefore, the refinement operator on design patterns as introduced in this thesis, works in a more general way since it allows to extend the components when a design pattern is refined.

Since the refinement process itself is very complex, *PP* as a prototype of a design pattern oriented programming only uses the mentioned three levels (global level, design pattern level, component level). From this point of view, the concept of nested classes is more orthogonal than the introduced approach in *PP*. However, a orthogonal nesting[9]in *PP* would make refinements and their consequences on command transla-

tions, etc. difficult to handle. Because of this reason, only the basic theoretical framework regarding abstract datatypes in chapter 3 and regarding object algebras in chapter 5 includes the facility to nest design pattern and components to an arbitrary depth.

From the designer's perspective, it is questionable if it is a limitation to have only the global level, design pattern level and the component level. In most cases, a deeper nesting would affect important properties of programming (e.g. traceablility and maintainability) in a negative way. Besides, refinements would become programming constructs which would be almost impossible to deal with.

2.5 Remarks and outlook

In this chapter, basic notions of the design pattern oriented programming model and *PP* have been introduced. This will help understanding the ideas and concepts in the subsequent chapters, which are more theoretical in nature. However, this introduction does not cover design pattern oriented strategies in software engineering. For this purpose, the reader is referred to [6] which captures a more general introduction into design pattern oriented programming techniques. Additionally, the design pattern oriented imperative programming language *PAL* is introduced using the same concepts as *PP*.

Design patterns will be introduced formally in chapter 4. For this purpose, in chapter 3, a framework of algebraic specification techniques will be presented. The obtained results will then be applied in chapter 5 and 6 in order to define the syntax and semantics of *PP*.

[9] In this case, *PP* would have to provide a unified construct for the definition of both design patterns and components. Besides, then it would have to be possible to nest these statements as needed.

Chapter 3

Basic notions of algebraic specifications and abstract data types

This chapter defines basic notions of algebraic specifications and abstract data types based on [5]. However, they have been adapted in order to support design pattern oriented features which are crucial for the description of the *PatternModel* and of the design pattern oriented language *PP*. The *PatternModel* itself has to be conceived as entirety of all concepts and notions presented in this and the subsequent chapters.

3.1 Partially ordered sets, dependency sets

A *partially ordered set* is used to model a subclass-relationship between sorts in signatures. It will be used later to impose a subset relation on carrier sets of these sorts.

Definition 3.1 *partially ordered set*
A pair $(S, \leq)$ is called *partiallay ordered set* iff S is a set and $\leq \subseteq S \times S$ is a partial ordering, i.e. $\leq$ is reflexive, transitive and antisymmetric.

The ordering $\leq$ is extended to strings of elements with equal length, i.e. $s_1 \ldots s_n \leq t_1 \ldots t_n$ iff $s_i \leq t_i$ for all $i = 1, \ldots, n$.

□

A *dependency set* is used to model a notion of dependency relationship between elements. It will be used in the sequel, when pattern specifications are introduced. A sort representing a component of a design pattern will then depend on a sort representing the corresponding design pattern. In this way, the complex interrelations between design patterns and components are handled at the lowest level. As pointed out in section 2.3.2, this notion of a dependency set is more general than it is actually needed in *PP* since it is not restricted to three levels of nesting. It is introduced in this way, since the presented framework should be as general as possible in order to provide a fundament for later considerations.

An element $s \in S$ is in relation $s' \Vdash s$ with some other element $s' \in S$ if s depends on s'. The relation $\Vdash$ has to satisfy the following properties:

- an element in S is never in relation with itself,
- an element in S can only depend on one other element in S,
- directed circles in the dependency relation are excluded.

Definition 3.2 *dependency set*
A pair $(S, \Vdash)$ is called *dependency set* iff S is a set and $\Vdash \subseteq S \times S$ satisfies the following properties

1. $\forall s_1, s_2$ holds that $s_1 \Vdash s_2 \Longrightarrow s_1 \neq s_2$,
2. $\forall s_1, s_2, s \in S$ holds that $s_1 \Vdash s \wedge s_2 \Vdash s \Longrightarrow s_1 = s_2$.
3. the transitive and reflexive closure of $\Vdash$ is antisymmetric.

The transitive closure of $\Vdash$ is denoted by $\Vdash^+$, the transitive and reflexive closure is denoted by $\Vdash^*$.

A dependency set is called *complete* if there is an $\bot \in S$ with $\bot \Vdash^* s$ for all $s \in S$. This element $\bot$ is then called the *global location* or the *global level*. The *complete closure* of a dependency set $(S, \Vdash)$ is defined by $(S^\circ, \Vdash^\circ) =_{def} (S \cup \bot, \Vdash \cup \{(\bot, s') : \nexists s \in S \text{ with } s \Vdash s'\})$.

□

The term *depend* is often used in conjunction with *direct* or *indirect*. Given a dependency set S, an element $s' \in S$ depends *directly* on an element $s \in S$ iff $s \Vdash s'$. s' depends *indirectly* on s iff $s \Vdash^+ s'$ and $s \not\Vdash s'$.

For a given subset of S, the following function returns the sort s, if all elements in this subset depend directly on s. Otherwise the function is undefined. If the elements in S do not depend on any element, the function returns $\bot$. It is also said, that the element s *encapsulates* the elements in S.

Definition 3.3 *encapsulating element of a set of elements*
Let $(S, \Vdash)$ be a dependency set. For a finite set $\mathbb{S} \subseteq S$, a function $\Uparrow$ is defined by:

$$\Uparrow_{(S,\Vdash)} : \wp_{fin}(S) \to S^\circ$$

$$\Uparrow_{(S,\Vdash)}(\mathbb{S}) =_{def} \begin{cases} s, \text{ if } s \in S \text{ and } \forall s' \in \mathbb{S} \quad s \Vdash^\circ s' \\ \text{undefined otherwise.} \end{cases}$$

The index $(S, \Vdash)$ is dropped if the context is clear. □

A dependency set can be visualized as a set of trees from the perspective of graph theory. The following definition of *visibility of elements* allows the selection of a *visible* subgraph based on a certain element in a dependence set. Using the dependency relation $\Vdash$, it is possible to define sets based on S which represent those elements in S which are *visible* from a particular element in the following way.
Every element c can encapsulate other elements. These dependent elements are only considered to be meaningful together with c. Therefore, e.g. directly dependent elements of an element $c \in S$ should only be visible from other dependent elements of c. The following definition introduces *visibility* of elements in a formal way. Later, it will be used to define the visibility of sorts in signatures.

Definition 3.4 *set of visible elements*
Let $(S, \Vdash)$ be a dependency set. The *set of visible elements* $\mathcal{S}^c$ *from a* $c \in S^\circ$ is inductively defined by

1. $s' \in \mathcal{S}^c$ if $s' \in S$ and $c \Vdash^\circ s'$,
2. $\mathcal{S}^{s'} \subseteq \mathcal{S}^c$ if $s' \in S^\circ$ and $s' \Vdash^\circ c$.

An element $s' \in S$ is visible from an element $c \in S^\circ$ iff $s' \in \mathcal{S}^c$. Note that the element $\bot$ is not contained in any set of visible elements.
Furthermore, $\Re(\mathcal{S}^c)$ will denote the set of elements which are visible form c but not from c. It is formally defined by

$$\Re(\mathcal{S}^c) =_{def} \begin{cases} \mathcal{S}^c \backslash \mathcal{S}^{\Uparrow(\{c\})}, \text{ if } c \neq \bot \\ \text{undefined otherwise.} \end{cases}$$

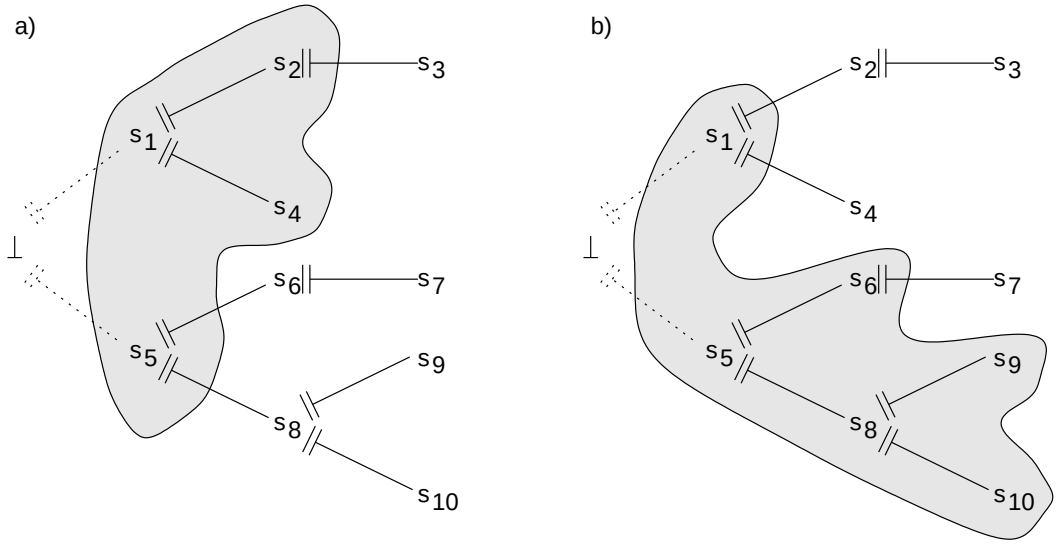

Figure 3.1: *Visualization of visible elements considered from s_1 (a) and s_{10} (b).*

□

Example 3.5 Let $(S, \Vdash)$ be a dependency set where the components are defined as follows.

$$\begin{array}{lll} S & =_{def} & \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}\} \\ \Vdash & =_{def} & \{(s_1, s_2), (s_2, s_3), (s_1, s_4), (s_5, s_6), (s_6, s_7), (s_5, s_8), (s_8, s_9), (s_8, s_{10})\} \end{array}$$

Figure 3.1 visualizes the set of visible elements in a dependency set considered from the perspective of s_1 (a) and s_{10} (b). Note that the element $\perp$ is not part of the dependency set itself. However, often the complete closure of a dependency set is used in order to obtain a more homogeneous environment for a particular consideration.

□

In order to avoid problems when both a partially ordered set and a dependency set are defined on the same basic set, the following notion of compatibility between these sets is introduced. Compatibility requires that whenever two elements in S are comparable by $\leq$, they also have to be in the same encapsulating element in $\Vdash$. This implies that if two elements in S are comparable by $\Vdash$ then they can not be compared by $\leq$.

Definition 3.6 *compatible sets (partially ordered set and dependency set)*
A partially ordered set $(S, \leq)$ is *compatible* with a dependency set $(S, \Vdash)$ iff $\forall s_1, s_2 \in S$ holds that if $s_1 \leq s_2$ then $\Uparrow (\{s_1, s_2\})$ is defined.

□

Partially ordered sets as well as dependency sets can have subsets[1]. However, in most cases, a simple subset notion is not sufficient. Therefore, the notion of a *closed component* also considers the relations $\leq$ respectively $\Vdash$ in such a set.

[1]In fact, partially ordered sets as well as dependency sets are pairs or tuples in the mathematical sense. In most cases, however, it is more convenient to think of these pairs as sets. Then, subrelations etc. can be defined on the components of the tuple

Definition 3.7 *closed component*
A pair (S', R') is called *closed component* in a pair (S, R) iff $S' \subseteq S, R' \subseteq R$ and $r \ R \ s$ implies either $r \ R' \ s$ or $r, s \notin S'$ for all $r, s \in S$.

□

3.2 Signatures, Σ-algebras

In this thesis, Σ-*algebras* will be used for the description of the denotational semantics of *PP*. A Σ-algebras can be conceived as a set of what is called *carrier sets* together with *operations* defined on them. Besides, they have to satisfy a certain form which can be described by *(order-sorted) signatures*.

Definition 3.8 *(order-sorted) signature* (cf. [5])
An *(order-sorted) signature* $\Sigma = (S, \leq, \Vdash, F, class)$ consists of

1. a partially ordered set of sorts $(S, \leq)$,
2. a dependency set of sorts $(S, \Vdash)$, such that $\leq$ is compatible with $\Vdash$,
3. an $S^* \times S$ indexed family $F = (F_{w,s})_{w \in S^*, s \in S}$ of operation identifiers f satisfying the following conditions
   - $\forall f \in F_{s_1 \ldots s_n, s}$ holds that there is a $c \in S^\circ$ such that $\{s_1, \ldots, s_n, s\} \in \mathcal{S}^c$ based on $(S, \Vdash)$.
   - if $f \in F_{w_1,s_1} \cap F_{w_2,s_2}$ and $w_1 \leq w_2$ then $s_1 \leq s_2$ or there is no $c \in S^\circ$ such that $\{s_1, s_2\} \in \mathcal{S}^c$ based on $(S, \Vdash)$.
4. a sort $class \in S$.

We define $(S, \leq, \Vdash, F, class) \subseteq (S', \leq', \Vdash', F', class')$ iff $S \subseteq S', \leq \subseteq \leq', \Vdash \subseteq \Vdash'$ and $F \subseteq F'$.

The *set of visible sorts* of a sort c in a signature Σ is defined by the set of visible elements of c in the dependency set $(S, \Vdash)$. □

In the sequel, signatures will be used to describe the structure of ADTs. Each such ADT, corresponding to modules on the programming level, can then be associated with one characteristic sort in Σ. This sort is called *class sort* and an explicit part of Σ.

In contrast to signatures and Σ-algebras as defined in [5], signatures in this approach integrate a dependency set of sorts into the signature. Additionally, the dependency set of sorts requires operation symbols to be of a certain form. Depending on the visibility of sorts, several operations are ruled out, since they have an unintuitive combination of parameter- and result sorts and will therefore not be considered in the sequel. The first item requires that there is a sort for an operation symbol such that every parameter- and return sort is visible from that sort. The second item is a relaxation of the *monotonicity-condition* (cf. [5]). In this approach, the condition has only to be met if both operation symbols are visible from one sort at the same time. This resolves ambiguity problems in the interpretation of terms later in this chapter and of commands later in this thesis.

For convenience reasons the following notations for components of signatures are introduced.

Notation 3.9 *notations for components of signatures*

- $f \in F_{s_1 \ldots s_n, s}$ will also denoted by $f : (s_1, \ldots, s_n) \to s$ and called *operation symbol*.
- $f() \to s$ is also denoted by $f :\to s$ and called *constant*.

- given a signature $\Sigma = (S, \leq, \Vdash, F, class)$, $sorts(\Sigma), \leq_\Sigma, \Vdash_\Sigma, opns(\Sigma)$ and $ClassSort(\Sigma)$ will denote the components of Σ. Moreover,
  $name(f : (s_1, \ldots, s_n) \to s) =_{def} f$, $names(F) =_{def} \{name(f) : f \in F\}$ and
  $sorts(f : (s_1, \ldots, s_n) \to s) =_{def} \{s_1, \ldots, s_n, s\}$.

□

For most applications, it is necessary that a signature satisfies the additional properties of *coherence*. Coherence implies *regularity* which means that operation symbols can be associated with least sorts for their parameters. Furthermore, in a coherent signature every sort can be associated with a maximum sort.

Definition 3.10 *coherent, regular signature*
A signature $\Sigma = (S, \leq, \Vdash, F, class)$ is called *coherent* iff

1. it is regular, i.e. given $w_0, w_1 \in S^*$ with $w_0 \leq w_1$ and given $f \in F_{w_1,s_1}$ there is a least $w, s \in S^* \times S$ such that $f \in F_{w,s}$ and $w_0 \leq w$.
2. each sort $s \in S$ has a maximum in S, i.e. there is a sort $max(s) \in S$ such that $s \leq max(s)$ and $s \leq s'$ implies $s' \leq max(s)$ for all $s' \in S$.

□

Algebras, in general, consist of carrier sets and functions on these carrier sets. In the case of partial order-sorted Σ-algebras, their structure is described by partial order-sorted signatures. Besides, certain restrictions are imposed on the carrier sets and the functions itself. The notion partial order-sorted Σ-algebra is formalized in the following way.

Definition 3.11 *partial order-sorted Σ-algebra*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature.
A *(partial order-sorted) Σ-algebra* $A = \left((A_s)_{s \in S}, (f^A_{s_1 \ldots s_n s})_{f:(s_1,\ldots,s_n)\to s \in F}\right)$ consists of

1. carrier sets A_s for all $s \in S$ such that $s \leq t$ implies $A_s \subseteq A_t$,
2. partial functions $f^A_{s_1 \ldots s_n, s} : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for all $f : (s_1, \ldots, s_n) \to s \in F$ such that $f : (s_1, \ldots, s_n) \to s, f : (t_1, \ldots, t_n) \to t \in F$ and $t_1 \ldots t_n, t \leq s_1 \ldots s_n, s$ implies

   $$f^A_{s_1 \ldots s_n s} \mid_{A_{t_1} \times \ldots \times A_{t_n}} = f^A_{t_1 \ldots t_n t},$$

   i.e. $f^A_{s_1 \ldots s_n s}(a_1, \ldots, a_n) = f^A_{t_1 \ldots t_n t}(a_1, \ldots, a_n)$ or both sides are undefined for all $a_i \in A_{t_i}, i = 1, \ldots, n$.

The class of all Σ-algebras is denoted by $Alg(\Sigma)$. The sort index of functions is omitted if the context is clear.

□

3.3 Visibility of operations

Using the visibility of sorts, a notion of *visibility of operation symbols* can be defined. Both, the visibility of sorts and operation symbols, are essential for the context-dependent definition and the interpretation of terms. The subsequent properties and definitions provide a basic framework for encapsulation concepts in a design pattern oriented programming model.

An operation symbol is visible if all parameter sorts and the result sort are visible. This ensures that operation symbols are hidden if they use hidden sorts.

Definition 3.12 *set of visible operation symbols*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a coherent signature. The *set of visible operation sybols* $\mathcal{F}^c$ *of a sort* $c \in S^\circ$ is defined by

$$\mathcal{F}^c =_{def} \left(\mathcal{F}^c_{s_1 \ldots s_n, s}\right)_{s_1 \ldots s_n, s \in \mathcal{S}^c} \text{ where } \mathcal{F}^c_{s_1 \ldots s_n, s} =_{def} F_{s_1 \ldots s_n, s}.$$

Furthermore, $\Re(\mathcal{F}^c)$ will denote the set of operation symbols that are visible form c but not from the encapsulating sort of c. It is formally defined by

$$\Re(\mathcal{F}^c) =_{def} \begin{cases} \mathcal{F}^c \backslash \mathcal{F}^{\Uparrow(\{c\})} & \text{if } c \neq \perp \\ \mathcal{F}^\perp & \text{otherwise.} \end{cases}$$

□

The following lemma guaranties that all sets $\mathcal{F}^c_{s_1 \ldots s_n, s}$ are also contained in $\mathcal{F}^{c'}$, if $c \Vdash^\circ c'$.

Lemma 3.13 Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature, $c \Vdash^\circ c'$. Then the following holds.

1. $\forall s_1, \ldots, s_n, s \in \mathcal{S}^c$ holds that $\mathcal{F}^c_{s_1 \ldots s_n, s} = \mathcal{F}^{c'}_{s_1 \ldots s_n, s}$,
2. $\mathcal{F}^c \subseteq \mathcal{F}^{c'}$,
3. $\forall f_{s'_1 \ldots s'_n, s'} \in \Re\left(\mathcal{F}^{c'}\right)$ there is no $s_1, \ldots s_n, s$ with $s'_1 \ldots s'_n s' \leq s_1 \ldots s_n s$.

Proof

1. (a) $\subseteq$**:** Let $f(s_1, \ldots, s_n) \to s \in \mathcal{F}^c$. Then $s_1, \ldots, s_n, s \in \mathcal{S}^{c'}$ by the definition of visible sorts of c'. Therefore, $f : (s_1, \ldots, s_n) \to s \in \mathcal{F}^{c'}$. Hence, $\mathcal{F}^c_{s_1 \ldots s_n, s} \subseteq \mathcal{F}^{c'}_{s_1 \ldots s_n, s}$.
   (b) $\supseteq$**:** $s_1, \ldots, s_n, s \in \mathcal{S}^{c'}$ by the definition of visible sorts of c'. By the definition of visible operation symbols can be implied that an $f(s_1, \ldots, s_n) \to s \in \mathcal{F}^{c'}$ must also be in $\mathcal{F}^c$. Hence, $\mathcal{F}^c_{s_1 \ldots s_n, s} \supseteq \mathcal{F}^{c'}_{s_1 \ldots s_n, s}$.
2. Follows immediately by the definition of visible sorts.
3. Follows by the fact that $(S, \leq)$ and $(S, \Vdash)$ are compatible.

□

3.4 Terms and their interpretation

In the preceeding sections, the notion of the Σ-*algebra* was introduced. For most applications, a mechanism is needed that enables the developer to specify a class of algebras which satisfy required properties. In this approach, *terms* which base on a given signature Σ are *interpreted* in the environment of a particular Σ-algebra. The term interpretation itself is part of a calculus which is eventually used to determine if that Σ-algebra meets the requirements.

This section extends the notions *term* and *term interpretation* in order to handle dependency sets. This facility is essential for design pattern oriented considerations in the sequel. It is also important to point out that the adaptions of these well-known notions do not conflict with any concepts that base on terms and their interpretation in the usual sense, so that these concepts can still be applied.

A *term* is defined based on a given signature Σ and a given set X of disjoint typed variables. Unlike terms as defined in [5], terms in this thesis also depend on sort c which is also called location. This sort determines the set of (visible) sorts and (visible) operations that can be used in a term.

Furthermore, every term belongs to a certain sort which specifies its result type. This is of importance, since e.g. operations require their parameters to be of a certain type. In order to guarantee type-safety of the term interpretation, possible terms for a parameter are restricted to terms that are associated with the corresponding sort.

Definition 3.14 *term*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature, $c \in S^\circ$ and X a $\mathcal{S}^c$-indexed family of disjoint sets of variables. The $\mathcal{S}^c$-indexed family $T(\Sigma, X)^c$ of *terms* over Σ at a location c with variables X is inductively defined by

1. $f \in T(\Sigma, X)^c_s$ for all $f :\rightarrow s \in \mathcal{F}^c$,
2. $x \in T(\Sigma, X)^c_s$ for all $x \in X_s$,
3. $f(t_1, \ldots, t_n) \in T(\Sigma, X)^c_s$ for all $f : (s_1, \ldots, s_n) \rightarrow s \in \mathcal{F}^c, t_i \in T(\Sigma, X)^c_{s_i}, i = 1, \ldots, n$,
4. $T(\Sigma, X)^c_r \subseteq T(\Sigma, X)^c_s$ if $r \leq s$.

$T(\Sigma)^c =_{def} T(\Sigma, \emptyset)^c$ denotes the set of *ground terms*.

□

Although terms always have to be considered in conjunction with one single location, it is easy to see that there are interrelations between the sets of terms of different locations if these locations are correlated via the dependency set of sorts. Every term at a particular location is also valid in all dependent locations. This property can be formalized as follows.

Fact 3.15 Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature, $c \Vdash^\circ c'$, $s \in \mathcal{S}^c$ and X a $\mathcal{S}^c$-indexed family of disjoint sets of variables. Then the following holds.

1. $T(\Sigma, X)^c_s \subseteq T(\Sigma, X)^{c'}_s$,
2. $T(\Sigma, X)^c \subseteq T(\Sigma, X)^{c'}$.

Proof

1. by structural induction over $T(\Sigma, X)^c_s$. Let $s \in \mathcal{S}^c$, $t \in T(\Sigma, X)^c_s$. The following cases have to be considered:
   (a) $t = f$, then $f :\rightarrow r \in \mathcal{F}^c$ and $r \leq s$. By lemma 3.13 follows that $f :\rightarrow r \in \mathcal{F}^{c'}$. By the definition of terms follows that $t \in T(\Sigma, X)^{c'}_s$.
   (b) $t = x$, then $x \in X_s$. By the definition of visible sorts and the definition of terms can immediately implied that $t \in T(\Sigma, X)^{c'}_s$.
   (c) $t = f(t_1, \ldots, t_n)$, then $f : (s_1, \ldots, s_n) \rightarrow r \in \mathcal{F}^c$, $r \leq s$ and $t_i \in T(\Sigma, X)^c_{s_i}, i = 1, \ldots, n$. By lemma 3.13 follows that $f : (s_1, \ldots, s_n) \rightarrow r \in \mathcal{F}^{c'}$. By structural induction hypothesis holds that $t_i \in T(\Sigma, X)^{c'}_{s_i}$. By the definition of terms follows that $t \in T(\Sigma, X)^{c'}_s$.
2. follows by 1.

□

Terms of a signature Σ can be interpreted in a particular Σ-algebra A. To this end, each variable x of sort s contained by the variable set X is assigned to a element of the carrier set A_s. Beginning from this starting point, terms are interpreted according to their structure following a innermost-to-outermost strategy eventually leading to a result value of the corresponding type. E.g. a function call $f(t_1, \ldots, t_n)$ is interpreted by first interpreting the terms representing the parameters. The results are then applied to the function f^A in order to obtain a result value for this function call. In this way, the interpretation of terms can also be conceived as the *semantics* of the syntactic construct of a term in the semantic space of a Σ-algebra.

Definition 3.16 *assignment, interpretation*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a coherent signature, $c \in S^\circ$, X a $\mathcal{S}^c$-indexed family of disjoint sets of variables and $A \in Alg(\Sigma)$. An *assignment* from X to A is a family of functions $v = (v_s : X_s \to A_s)_{s \in \mathcal{S}^c}$. The *interpretation* of terms at a location c in A is described by a family of functions

$$(v^c)^* = \left((v^c)^*_s : T(\Sigma, X)^c_s \to A_s\right)_{s \in \mathcal{S}^c}.$$

These functions are defined in the following way.

$(v^c)^*_s(f) =_{def} f^A_r$ if $f :\to r \in \mathcal{F}^c, s \geq r$,
$(v^c)^*_s(x) =_{def} v_r(x)$ if $x \in X_r, s \geq r$
and if $f : (r_1, \ldots, r_n) \to r \in \mathcal{F}^c, s \geq r$ then
$(v^c)^*_s(f(t_1, \ldots, t_n)) =_{def}$
$\begin{cases} f^A_{r_1 \ldots r_n r}\left((v^c)^*_{r_1}(t_1), \ldots, (v^c)^*_{r_n}(t_n)\right) & \text{if } (v^c)^*_{r_i}(t_i) \text{ are defined for all } i = 1, \ldots, n \\ \text{undefined otherwise.} \end{cases}$

□

In the sequel, it will be assumed that these interpretation functions are well defined. Intuitively, a proof for the well-definedness of the interpretation functions of terms over a signature $\Sigma(S, \leq, \Vdash, F, class)$ at a location c could be founded on the proof in [5] for a signature $\Sigma' = \left(\mathcal{S}^c, (\leq)_{|\mathcal{S}^c}, \mathcal{F}^c\right)$[2] and on the fact that this signature is coherent[3] (by lemma 3.13) if Σ is coherent.

A term at a location is interpreted in all dependent locations of this particular location in the same way. The following fact formalizes this property.

Fact 3.17 Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a coherent signature, $c \Vdash^\circ c' \in S^\circ$, $s \in \mathcal{S}^c$, $A \in Alg(\Sigma)$. An assignment from X to A is given as a family of functions $v = (v_s : X_s \to A_s)_{s \in \mathcal{S}^c}$. Then for a $t_s \in T(\Sigma, X)^c_s$ the following holds.

$$\left(v^{c'}\right)^*_s(t) = (v^c)^*_s(t)$$

Proof Omitted. The idea behind the proof is to show by lemma 3.13 that the interpretation of terms is defined on the same functions in A at both locations.

□

As afore mentioned in this section, a calculus can be founded on the interpretation of terms in order to classify algebras according to their properties. One approach could be to use *predicative logic* in form of *well-formed formulas* as introduced in [5]. Another approach is to require that the carrier sets of the algebras to be *term-generated*. This means that each element of a carrier set can be *reached* by the interpretation of

[2] This signature is only a valid signature in [5]
[3] Again, this notion of coherence is defined in [5].

a term. *Constraints* can additionally be used to restrict the form a term, e.g. it can be required that only a certain subset of all operations [4] of the signature may be used in a term.

Definition 3.18 *term generation, constraint*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a coherent signature. Let $A \in Alg(\Sigma), S \subseteq sorts(\Sigma), F \subseteq (\Sigma)$. The algebra A is *reachable on S with F* iff for all $s \in S$ and $a \in A_s$ there is a location $c \in S$ with $s \in \mathcal{S}^c$ and a term $t \in T\left(\Sigma', (X_{s'})^c_{s' \in S'}\right)_s$ and an assignment v such that $(v^c)^*_s(t) = a$,
where $\Sigma' = (sorts(\Sigma), \leq_\Sigma, \Vdash_\Sigma, F, ClassSort(\Sigma))$ and $S' = \mathcal{S}^c - \{s' : s' \geq_\Sigma r \text{ for some } r \in S\}$.

If A is reachable on $sorts(\Sigma)$ with $opns(\Sigma)$ then A is called *term-generated*.

Let $s \in sorts(\Sigma)$. The algebra A is *generated on s by its subsorts* iff for all $a \in A_s$ there is some $r \in sorts(\Sigma), r <_\Sigma s$, such that $a \in A_r$.

We call a pair (S, F), or a sort s as above, a *constraint* with respect to Σ. The operations in F are called *constructors*. Given a set I of constraints with respect to Σ, a Σ-algebra A *satisfies* I, denoted by $A \models I$, iff A is reachable on S with F for all $(S, F) \in I$ and A is generated on s by its subsorts for all $s \in I$.

□

Notation 3.19 *notations for commonly used sets*

- $SORT$, $\wp_{fin}(OPN)$, $\wp_{fin}(CONSTRAINT)$, SIG, $TERM$ denote the sets of sorts, finite subsets of operation symbols and constraints, finite coherent signatures and terms.
- Moreover, it will be assumed in the sequel that there is an element $\Im$ in every $S \subseteq SORT$.
- $SPEC$ denotes the set of specifications. It is formally defined by

  $$SPEC =_{def} \{\langle \Sigma, \mathbb{C} \rangle : \Sigma \in SIG, \mathbb{C} \subseteq Alg(\Sigma)\}$$

  The functions Sig and Mod are defined for being able to access the components of $SPEC$. They are defined by

  $$Sig(\langle \Sigma, \mathbb{C} \rangle) =_{def} \Sigma,$$
  $$Mod(\langle \Sigma, \mathbb{C} \rangle) =_{def} \mathbb{C}.$$

□

3.5 Operators on signatures

The aim of this section is to introduce operators on signatures which will be used in subsequent chapters as an efficient way for their manipulation. Some of these operators are of a rather general use whereas e.g. the need for a hierarchical construction of signatures is strongly related to the definition of the semantics of PP.

A signature morphism is basically a pair of functions which renames the sorts and operation symbols of a signature. It is defined as follows.

Definition 3.20 *signature morphism*
A *signature morphism* $\sigma = (\sigma_S, \sigma_F)$ refining a sort $s^r \in S$ consists of an injective partial function σ_S on sorts and an injective function σ_F on operation symbols such that the following holds.

[4] These operations are called *constructors*.

1. $s^r \in dom(\sigma_F)$,
2. σ_S and σ_F are compatible in the following sense: if $\sigma_F(f : (s_1, \ldots, s_n) \to s_0)$ is defined then it is equal to $\sigma_F(f) : (s'_1, \ldots, s'_n) \to s'_0$ for some identifier $\sigma_F(f)$ where $s'_i =_{def} s_i$ or $s'_i =_{def} \sigma_S(s_i)$ for $s_i = s^r$ and $s'_i =_{def} \sigma_S(s_i)$ otherwise.
3. $name(\sigma_F(f : (s_1, \ldots, s_n) \to s_0)) = name(\sigma_F(f : (t_1, \ldots, t_n) \to t_0))$ for all operation symbols $f : (s_1, \ldots, s_n) \to s_0, f : (t_1, \ldots, t_n) \to t_0$.

The set of signature morphisms is denoted by $SIGMORPH$. If the context is clear, the indices S and F are omitted.

□

A signature morphism can refine a sort s^r which means that only dependent sorts of s^r may be changed by the morphism. Hence, terms in a non-dependent location of s^r in the source signature rely on the same sorts, since all dependent sorts of s^r are hidden.

Definition 3.21 *induced signature*
A signature morphism $\sigma = (\sigma_S, \sigma_F)$ refining a sort $s^r \in S$ and a signature $\Sigma = (S, \leq, \Vdash, F, class) \in SIG$ induce a signature $\sigma(\Sigma) = (S', \leq', \Vdash', \sigma_F(F), \sigma_S(class))$, where

$$S' =_{def} \{\sigma_S(s) : s^r \Vdash^* s\} \cup \{s^r\}$$

and $\sigma(s) \leq' \sigma(t)$ iff $s \leq t$ or $(s = \sigma(s^r)$ and $t = s^r)$ and $\sigma(s) (\Vdash')^\circ \sigma(t)$ iff $s \Vdash^\circ t$ or $(s = \Uparrow_{(S,\Vdash)} (\{s^r\})$ and $t = s^r)$.

Proof that $\sigma(\Sigma)$ is a coherent signature is omitted.

□

It is important to understand why the refining sort s^r is treated differently to all other sorts. Operation symbols that use s^r as a parameter sort or a result sort can be morphed in a hybrid way. This means that an induced signature Σ' of the signature morphism σ and a signature Σ can contain an operation symbol $\sigma_F(f)$ in which some occurences of s^r as parameter sort and/or result sort in f have been morphed into $\sigma(s^r)$ whereas some occurences of s^r in f remain unchanged in $\sigma_F(f)$. This property will be used in subsequent chapters to model a *current* type when design patterns are refined. A current type can be used to refine parameter sorts or result sorts in a dynamic way. However, in some cases, it is not desired or even not type-safe, to morph each single occurence (co-variance).

Notation 3.22 If $\Sigma \subseteq dom(\sigma)$ then we write $\sigma : \Sigma \to \Sigma'$ for any signature $\sigma(\Sigma) \subseteq \Sigma'$ and we write $\sigma^{-1}(\Sigma') = \Sigma$. Moreover, in the sequel it is assumed that in signature morphisms holds $\sigma_S(s) = s$ if $s^r \not\Vdash^+ s$.
□

Example 3.23 Figure 3.2 depicts the sorts of two signatures Σ and Σ' and the morphing of sorts σ_S which is part of the signature morphism $\sigma : \Sigma \to \Sigma'$. σ refines s_4 which causes $s'_4 \leq s_4$.

□

As a matter of fact, the *composition of signature morphisms* is not an operator on signatures but on signature morphisms themselves. It is defined as the consecutive application of both participating signature morphisms.

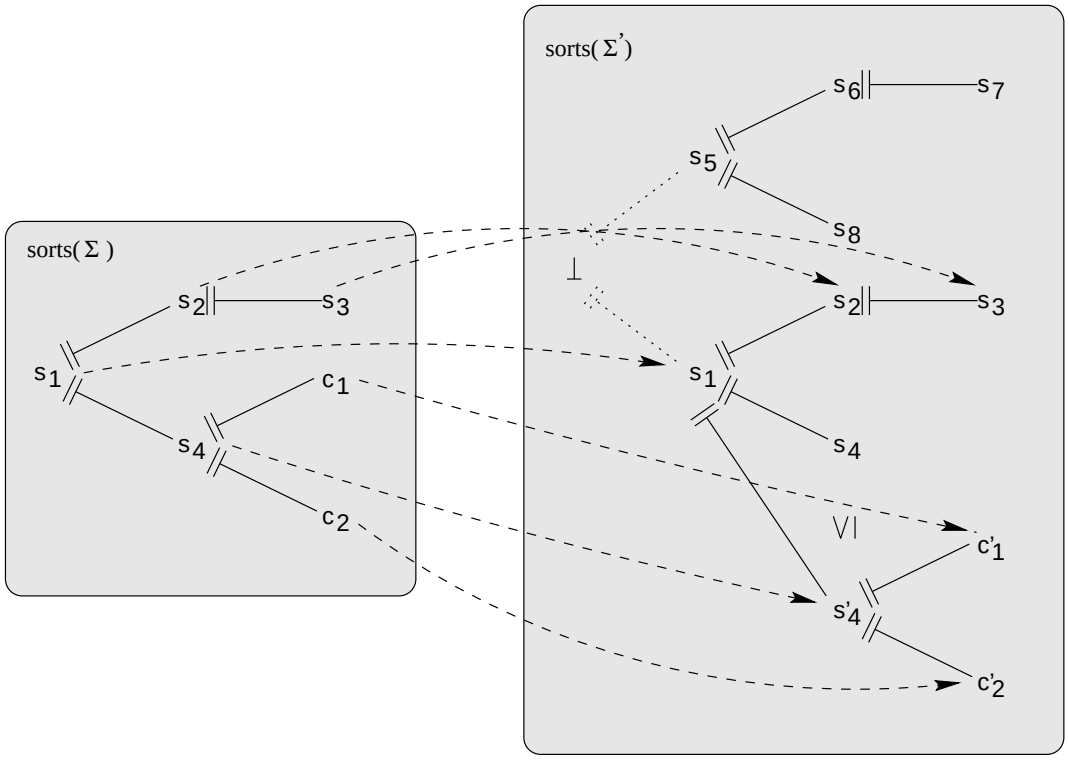

Figure 3.2: *Morphism of sorts from* $sorts(\Sigma)$ *to* $sorts(\Sigma')$ *refining* s_4.

Definition 3.24 *composition of signature morphisms*
The composition $\sigma_2 \circ \sigma_1 : \Sigma_1 \to \Sigma_3$ of two signature morphisms $\sigma_1 : \Sigma_1 \to \Sigma_2$ and $\sigma_2 : \Sigma_3 \to \Sigma_3$ is defined by

$$\sigma_2 \circ \sigma_1 \quad =_{def} \quad \begin{cases} \sigma_2(\sigma_1(x)) \text{ if } \sigma_1(x) \text{ and } \sigma_2(\sigma_1(x)) \text{ are defined} \\ \text{undefined otherwise} \end{cases}$$

for all sort and operation symbols x.

□

The notion of the signature morphism is a convenient way for the manipulation of signatures. The notion of the σ-reduct as introduced below can be conceived as its inverse operation. Given a signature Σ' and a signature morphism $\sigma : \Sigma \to \Sigma'$, the σ-reduct of Σ' results in Σ.

Definition 3.25 *σ-reduct, Σ-reduct*
Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism refining s^r, $A \in Alg(\Sigma')$. The Σ-algebra $A_{|\sigma}$ is called a *σ-reduct* of A. It is defined by

$$\begin{aligned} (A_{|\sigma})_s \quad &=_{def} \quad A_{\sigma(s)} \text{ for all } s \in sorts(\Sigma), \\ f^{A_{|\sigma}}_{s_1 \ldots s_n s_0} \quad &=_{def} \quad \sigma(f)^{A}_{s'_1 \ldots s'_n s'_0} \text{ for all } f : (s_1, \ldots s_n) \to s_0 \in opns(\Sigma) \end{aligned}$$

where $s'_i =_{def} \sigma_S(s_i)$ or $s'_i =_{def} s_i$ if $s_i = s^r$ or $s'_i =_{def} \sigma_S(s_i)$ otherwise.
If σ is the identity on Σ, hence $\Sigma \subseteq \Sigma'$, we denote $A_{|\sigma}$ also by $A_{|\Sigma}$ and call it *Σ-reduct* of A. For any class $\mathbb{C}$ of Σ'-algebras, $\mathbb{C}_{|\sigma}$ is defined by $\mathbb{C}_{|\sigma} =_{def} \{A_{|\sigma} : A \in \mathbb{C}\} \subseteq Alg(\Sigma)$.

□

The following definition provides operators for the *intersection* and the *sum* of two signatures as well as the *hierarchical construction* of a signature based on a source signature (cf. [5] for details).

Definition 3.26 *intersection ($\cap$), sum ($+$) of two signatures, hierarchical construction ($\oplus$)*
Let $\Sigma_1 = (S_1, \leq_1, \Vdash_1, F_1, class_1), \Sigma_2 = (S_2, \leq_2, \Vdash_2, F_2, class_2) \in SIG$.

$$\Sigma_1 \cap \Sigma_2 =_{def} \begin{cases} (S_1 \cap S_2, \leq_1 \cap \leq_2, \Vdash_1 \cap \Vdash_2, F_1 \cap F_2, \Im) \text{ if this signature is coherent} \\ \text{undefined otherwise} \end{cases}$$

In order to define the sum of two signatures, it is necessary to restrict the participating sets of operation symbols F_1, F_2 to be *closed* with respect to the ordering on sorts $\leq_1, \leq_2$. This notion guarantees that the resulting signature is still coherent.
Two sets of operation symbols F_1 and F_2 are *closed* with respect to the ordering on sorts $\leq_1, \leq_2$ iff whenever $f \in (F_1 \cup F_2)_{w,s}$ and $f \in (F_1 \cup F_2)_{w',s'}$ and there is some $w_o\, (\leq_1 \cup \leq_2)^+ w, w_o\, (\leq_1 \cup \leq_2)^+ w'$ then $w_0 \leq_i w, w'$ and $f \in (F_i)_{w,s}, f \in (F_i)_{w',s'}$ for either $i = 1$ or $i = 2$.

$$\Sigma_1 + \Sigma_2 =_{def} \begin{cases} (S', \leq', \Vdash', F', class') \text{ if } (S', \leq', \Vdash', F', class) \in SIG \text{ and } F_1, F_2 \text{ are closed wrt. } \leq_1, \leq_2 \\ \text{undefined otherwise,} \end{cases}$$

$$\text{where } (S', \leq', \Vdash', F', class') =_{def} \left(S_1 \cup S_2, (\leq_1 \cup \leq_2)^+, \Vdash_1 \cup \Vdash_2, F_1 \cup F_2, class_2\right).$$

Let $\Sigma = (S, \leq, \Vdash, F, class) \in SIG, s \in SORT, s_i \in sorts\,(\Sigma)$, with $s_i \neq s, i = 1, \ldots, n$, $p \in SORT \cup \bot, p \neq s$ and $c_j \in SORT, c_j \neq s, j = 1, \ldots, o$.

$$\Sigma \oplus (s, s < \{s_1, \ldots, s_n\}, p \Vdash s \Vdash \{c_1, \ldots, c_o\}, F') =_{def}$$
$$\begin{cases} (S', \leq', \Vdash', F \cup F', s), \text{ if this is a coherent signature and } \Uparrow(\{s_1, \ldots, s_n\}) \text{ is defined} \\ \text{undefined otherwise,} \end{cases}$$

where

$$S' =_{def} \begin{cases} S \cup \{s, p\} \cup \{c_1, \ldots, c_o\}, \text{ if } p \neq \bot \\ S \cup \{s\} \cup \{c_1, \ldots, c_o\}, \text{ otherwise,} \end{cases}$$
$$\leq' =_{def} (\leq \cup \{(s, s), (s, s_1), \ldots, (s, s_n)\}),$$
$$\Vdash' =_{def} \begin{cases} \Vdash \cup \{(p, s)\} \cup \{(s, c_1), \ldots, (s, c_o)\} \text{ if } p \neq \bot \\ \Vdash \cup \{(s, c_1), \ldots, (s, c_o)\}, \text{ otherwise.} \end{cases}$$

□

Analogously to the notion of closed components in partially ordered sets and dependency sets, a notion of a *closed component* is defined on signatures. Again, it ensures that a closed component in a signature Σ does not break off relations in Σ.

Definition 3.27 *closed components in SIG*
We call $\Sigma' \subseteq \Sigma$ a *closed component* in Σ iff $(sorts\,(\Sigma'), \leq')$ is a closed component in $(sorts\,(\Sigma), \leq)$, $(sorts\,(\Sigma'), \Vdash')$ is a closed component in $(sorts\,(\Sigma), \Vdash)$ and $\Sigma + (sorts\,(\Sigma), \leq, \Vdash, opns\,(\Sigma) - opns\,(\Sigma'), class)$ is defined.

□

Using the definition of visible sorts and visible operations on a given signature, a subsignature can be defined which only consists of visible components.

Definition 3.28 *visible signature*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature. The *visible signature* $\mathcal{V}^c(\Sigma)$ *of a sort* $c \in S^\circ$ is defined by

$$\mathcal{V}^c(\Sigma) =_{def} \begin{cases} \left(\mathcal{S}^c, (\leq)_{|\mathcal{S}^c}, (\Vdash)_{|\mathcal{S}^c}, \mathcal{F}^c, class\right), \text{ if } class \in \mathcal{S}^c \\ \Im, \text{ otherwise.} \end{cases}$$

□

3.6 Operators on specifications

In analogy to the operators on signatures as defined in the preceeding sections, this section will define operators on specifications that which will be used in later chapters for the efficient handling of specifications (cf. [5]).

Definition 3.29 *satisfiability of a specification*
A specification $sp \in SPEC$ is called *satisfiable* iff $Mod(sp) \neq \emptyset$.

□

Definition 3.30 *class sort*
Let $sp \in SPEC$.
A function $ClassSort$ is defined by:

$$ClassSort : SPEC \to SORT$$
$$ClassSort(psp) =_{def} ClassSort(Sig(sp))$$

□

Applying the notion of the visible signature, it is possible to define the visible part of a specification considered from a location c.

Definition 3.31 *visible part*
Let $sp \in SPEC$ and $c \in sorts(Sig(sp))^0$.
A function $visible^c$ is defined by:

$$visible^c : SPEC \to SPEC$$
$$visible^c(sp) =_{def} \left\langle \mathcal{V}^c(Sig(sp)), Mod(sp)_{|\mathcal{V}^c(Sig(sp))} \right\rangle.$$

□

Definition 3.32 *sum(+) of two specifications*

$$+ : \quad SPEC \times SPEC \to SPEC$$
$$\langle \Sigma_1, \mathbb{C}_1 \rangle + \langle \Sigma_2, \mathbb{C}_2 \rangle =_{def}$$
$$\begin{cases} \left\langle \Sigma_1 + \Sigma_2, \left\{ A \in Alg(\Sigma_1 + \Sigma_2) : A_{|\Sigma_1} \in \mathbb{C}_1 \wedge A_{|\Sigma_2} \in \mathbb{C}_2 \right\} \right\rangle \text{ if } \Sigma_1 + \Sigma_2 \text{ is defined} \\ \text{undefined otherwise.} \end{cases}$$

The function Sum is defined as follows.

$$\begin{aligned} Sum: \quad & \wp_{fin}(SPEC) \to SPEC \\ & Sum(\mathbb{C}) =_{def} \sum_{C \in \mathbb{C}} C. \end{aligned}$$

□

3.7 Relations in $SPEC$

3.7.1 Object oriented relations

The following object oriented relations are used to model object oriented relations that can exist between two specifications (cf. [5]).

Definition 3.33 *the clientship, subclass, inheritance relation in $SPEC$*
Let $sp, sp' \in SPEC$.

- *Clientship:*
  sp is a *client* of *server* sp' denoted by $sp' \longrightarrow sp$, iff
  1. $Sig(sp') \subseteq Sig(sp)$ and
  2. $Mod(sp)_{|Sig(sp')} \subseteq Mod(sp')$.
- *Inheritance:*
  sp is an *heir* of *ancestor* sp' via the signature morphism σ denoted by $sp' \overset{\sigma}{\rightsquigarrow} sp$, iff
  1. $\sigma : Sig(sp') \to Sig(sp)$ and
  2. $Mod(sp)_{|\sigma} \subseteq Mod(sp')$
- *Subtyping:*
  sp is a *subclass* of *superclass* sp' denoted by $sp \ll sp'$, iff
  1. $Sig(sp') \subseteq Sig(sp)$,
  2. $ClassSort(sp) \leq ClassSort(sp')$ in the ordering of $Sig(sp)$ and
  3. $Mod(sp)_{|Sig(sp')} \subseteq Mod(sp')$

□

Definition 3.34 *a generalized notion of inheritance*
Let $sp, sp' \in SPEC$. sp is a *generalized heir* of *ancestor* sp' via the signature morphism σ with respect to a relation Φ iff

1. $\sigma : \quad Sig(sp') \to Sig(sp)$ and
2. $\forall A \in Mod(sp)\ \exists B \in Mod(sp')$ with $\Phi\left(A_{|\sigma}, B\right)$.

In order to support the construction of generalized heirs, a function based on the newly introduced relation is defined:

$$\begin{aligned} simulate_{\Phi}(sp) =_{def} \langle Sig(sp), \quad \{ \quad & A \in Alg(Sig(sp)) : \\ & \Phi(A, B) \text{ for } B \in Mod(sp)\}\rangle \end{aligned}$$

□

All presented relations are reflexive and transitive. Detailed proofs can be found in [5].

3.7.2 The dependency relation

In addition to the afore mentioned object oriented relations another new relation is introduced to model a notion of dependency between two specifications. In chapter 4 the notion of a design pattern specification will be introduced which in turn uses separate specifications to represent the components and the higher behaviour of a design pattern. The *dependency relation* will then be used to describe the correlation between the component specifications and the specification representing the higher behaviour. Basically, the dependency relation requires the common part of both participating specifications to be identical and the class sort of the dependent specification to be dependent on the class sort of the other specification. Formally, the dependency relation is defined as follows.

Definition 3.35 *dependency relation*
A specification sp' depends on a specification sp, denoted by $sp \Vdash sp'$, iff

1. $ClassSort(sp'), ClassSort(sp') \in Sig(sp) \cap Sig(sp')$,
2. $ClassSort(sp') \Vdash ClassSort(sp)$ in the ordering of $Sig(sp) \cap Sig(sp')$,
3. $Mod(sp)_{|Sig(sp) \cap Sig(sp')} = Mod(sp')_{|Sig(sp) \cap Sig(sp')}$.

□

The following properties of the dependency relation are quite obvious. They are mentioned here because they will be used in the subsequent sections tacitly.

Fact 3.36 *properties of the dependency relation*

1. the dependency relation is **not** reflexive,
2. the dependency relation is **not** transitive.

Proof follows by definition 3.35.

□

The presented notions will be widely used in the sequel. Therefore, this chapter represents a framework for later considerations.

Chapter 4

A design pattern specification framework

In chapter 2, a way was discussed to describe design patterns informally. However, due to the nature of the listed items, such descriptions are often quite vague and ambiguities are not always avoidable. Beside this, soundness and completeness can not be taken for granted.

The notion of a *design pattern* will be introduced formally in this chapter. Its definition is entirely based on the concepts of algebraic specifications as described in chapter 3. In this way, problems like those mentioned above can be overcome. Additionally, this formal approach makes it possible to deduce important properties and to prove that certain requirements on the model are met.

First of all, the notion of a design pattern specification as the corner stone of the *PatternModel* will be presented. Subsequently, relations between design patterns and, based on these relations, concepts for refinements of design patterns will be introduced which are essential for reusability issues.

4.1 The notion of a design pattern specification

According to definition 2.5, a design pattern consists of components, attributes and methods. Before the model will be extended to support all these concepts directly, a much more general notion of a *(formal) design pattern* will be introduced. This notion corresponds to a specification of a class in the object oriented model. In the sequel this basic concept will be extended in order to support features like mentioned above.

Definition 4.1 *design pattern (specification) (formal)*
A structure $psp = \langle \mathbb{C}, C_{HB} \rangle$ is called *design pattern*, if the following holds:

1. $\mathbb{C}$ is finite and $\mathbb{C} \subseteq SPEC$ and $Sum(\mathbb{C})$ must be defined
2. $C_{HB} \in \mathbb{C}$
3. $\forall C \in \mathbb{C} \backslash C_{HB}$ holds $C_{HB} \Vdash C$

□

In subsequent sections, $PATTSPEC$ will denote the set of design patterns, $Comp(psp)$ the set $\mathbb{C}$ of components in psp, $C_{HB}(psp)$ the specification C_{HB} of psp. Furthermore *design pattern specification* will

also be referred to as *design pattern* if the context is clear.

In other words, a design pattern consists of the following components:

- $\mathbb{C}$ — is a finite set of specifications. These specifications represent the components of *psp*. Component specifications can be in relation to each other (cf. chapter 3). These relations altogether represent the internal structure of *psp*.
- C_{HB} — is a specification in the usual sense (cf. chapter 3). This specification represents the *higher behaviour* of the design pattern. It will also be called *controlling component* to emphasize the role of this component in the context of a design pattern. Besides, C_{HB} is contained in $\mathbb{C}$.

It is required that all components depend on the controlling component of a design pattern. This guaranties that a certain set of unintuitive models in components is immediately ruled out.

In analogy to the function $ClassSort$ which has been introduced for the domain $SPEC$, a function $PatternSort$ is defined on $PATTSPEC$. This function associates an element $psp \in PATTSPEC$ with a sort contained in the signature of *psp*.

Definition 4.2 *pattern sort*
Let $psp \in PATTSPEC$. A function $PatternSort$ is defined by:

$$PatternSort : PATTSPEC \to SORT$$
$$PatternSort\,(psp) =_{def} ClassSort\,(Sig\,(C_{HB}\,(psp)))$$

□

In reference to chapter 3, it can be said that a design pattern is a normal specification holding additional information about the components and their relationships to each other. This is an essential property of design patterns. In fact, every design pattern can be *translated* to a corresponding counterpart in $SPEC$. Such a *translation* is defined by the following function.

Definition 4.3 *translation function from $PATTSPEC$ to $SPEC$*
Let $psp \in PATTSPEC$.

A partial function $translate_{P\to S}$ is defined by:

$$translate_{P\to S} : PATTSPEC \to SPEC$$
$$translate_{P\to S}\,(psp) =_{def} \quad Sum\,(Comp\,(psp))$$

□

The application of $translate_{P\to S}$ leads to a specification $sp \in SPEC$ which represents the relevant part of the design pattern in the context of other class specifications or design patterns. Since all other parts of a design pattern are hidden from the oustide, the resulting specification will also be called *external part* of a design pattern.

Definition 4.3 provides a very interesting new point of view. A design pattern can be considered as a class specification in the object oriented sense. This has far-reaching consequences. For instance it is now possible to subclass a design pattern from a class specification and vice versa. The whole significance of this relationship between a design pattern and a class specification will become intelligible in the remainder of this chapter.

In this section, the notion of a design pattern specification was introduced. This concept, considered separately, enables one to encapsulate a structure of components inside a design pattern with the possibility of the definition of some higher behaviour. But how does reusability come into play?

4.2 The refinement relation

Analogously to object oriented relations which operate on class specifications and classes, similar relations can be defined on design patterns. This section will discuss how ideas of object oriented refinements can be adapted and generalized in order to introduce a refinement relation which enables one to reuse a design pattern directly.

Direct reuse of a design pattern means that beside the reuse of *normal* object oriented features it should also be possible to benefit from its internal structure.

In chapter 2, a *refinement relation* on design patterns was proposed in such way that in addition to the refinement of the higher behaviour, the components of the refined design pattern should also be refined versions of the components of the source design pattern. The internal structure of the source design pattern, i.e. the relationships in which the components are, should be preserved. Thus, the possibly complex relationships between components can be reused together with the components themselves as a whole.

Before a formal refinement relation between between design pattern will be introduced, the following notion of a *component morphism* is necessary to deal with the refinement of components under consideration of a given relation R.

Definition 4.4 *component morphism*
Let $\mathbb{C}, \mathbb{C}' \subseteq SPEC$.
A tuple $\delta = \left(\delta^{\Gamma}, \left(\delta^{\Phi}_{C'}\right)_{C' \in \mathbb{C}'}, \left(\delta^{\Phi}_{C'}\right)_{C' \in \mathbb{C}'}\right)$ is called *component morphism* preserving a relation $R' \subseteq R \subseteq SPEC \times SPEC$ refining $C'_{HB} \in \mathbb{C}'$, if the following holds:

- δ^{Γ} is defined by

  $\delta^{\Gamma} : \mathbb{C}' \to \mathbb{C}$ is total and injective.

- Let $C' \in \mathbb{C}'$. For a particular application $\delta^{\Gamma}(C')$ a function $\delta^{\Sigma}_{C'}$ must be defined satisfying the following properties:

  $\delta^{\Sigma}_{C'} : Sig(C') \to Sig(C)$ is a total signature morphism refining sort $ClassSort(C'_{HB})$.

  The union of all these functions, denoted by δ^{Σ}, must be defined, i.e.

  $\delta^{\Sigma} : Sig(Sum(\mathbb{C}')) \to Sig(Sum(\mathbb{C}))$,
  $\delta^{\Sigma} =_{def} \bigsqcup_{C' \in \mathbb{C}'} \delta^{\Sigma}_{C'}$ must be defined, i.e. $\delta^{\Sigma}_{C'}$ are compatible $\forall C' \in \mathbb{C}'$

- Let $C' \in \mathbb{C}'$. For a particular application $\delta^{\Gamma}(C')$ a relation $\delta^{\Phi}_{C'}$ must be defined satisfying the following property:

  $\delta^{\Phi}_{C'} \subseteq Alg(Sig(C')) \times Mod(C')$.

Figure 4.1: *For some $C' \in \mathbb{C}'$ the application $\delta^{\Gamma}(C')$ yields a signature morphism $\delta^{\Sigma}_{C'}$*(solid arrow) and a relation $\delta^{\Sigma}_{C'}$(dashed line).

- $\forall C' \in \mathbb{C}'$ let $C = \delta^{\Gamma}(C')$ be a generalized heir of C'
  via the total signature morphism $\delta^{\Sigma}_{C'}$ and the relation $\delta^{\Phi}_{C'}$ where the following holds:

$$\forall C^* \in \mathbb{C}' : \quad C^* \;\; R' \;\; C' \Longrightarrow \delta^{\Gamma}(C^*) \;\; R \;\; C$$

Furthermore $|R'|_{\delta}$ is defined as

$$|R'|_{\delta} =_{def} \left\{ \left(\delta^{\Gamma}(A), \delta^{\Gamma}(B)\right) : (A,B) \in R' \right\}.$$

The component morphism δ morphing from $\mathbb{C}'$ to $\mathbb{C}$ is denoted by $\delta : \mathbb{C}' \rightarrow mathbbC$.

□

A component morphism refines a subset of $SPEC$ using the generalized inheritance as described in chapter 3. Each component in $\mathbb{C}'$ is associated with a generalized heir in $\mathbb{C}$. Since the function δ^{Γ} is not required to be surjective, there can be more components defined in $\mathbb{C}$ which do not have a counterpart in $\mathbb{C}'$. This property is crucial for extensibility issues in the design pattern context, since it allows one to add new components during the refinement process.

In order to ensure only structure preserving refinements, another restriction is imposed on δ^{Γ}. Existing relationships (as defined in R') between two components in $\mathbb{C}'$ must also hold in R on corresponding components in $\mathbb{C}$. The same is required even if one of the components is not in $\mathbb{C}'$. The heirs of two components which are not in relation in R' can be in relation in R after the refinement. Every relationship has to be preserved, if $R' = R$. However, mostly it is not necessary that every relationship between two components inside a design pattern also holds in the refined design pattern. Supposed, a design pattern describes the semantics of a program written in a pattern oriented language. In many cases it is not possible to deduce all relationships between two components by the syntax of the programming language, so that a refinement operator ist not necessarily able to preserve these non-deducable relationships. Hence, the refinement relation is weakened in order to overcome this problem by modelling R' as a subrelation of R.

Lemma 4.5 Let $\mathbb{C}', \mathbb{C} \subseteq SPEC$. A component morphism $\delta : \mathbb{C}' \rightarrow \mathbb{C}$ preserving $R' \subseteq R$ satisfies the following properties.

1. $|R'|_{\delta} \subseteq R$ and in particular
2. $\forall C'_1, C'_2 \in \mathbb{C}' : C'_1 \;\; R' \;\; C'_2 \Longrightarrow \delta^{\Gamma}(C'_1) \;\; |R'|_{\delta} \;\; \delta^{\Gamma}(C'_2)$.

Proof follows immediately by the definition of $|R'|_{\delta}$.

□

Figure 4.1 depicts the relationships between an application $\delta^{\Gamma}(C')$ and the corresponding signature morphism $\delta^{\Sigma}_{C'}$ respectively relation $\delta^{\Phi}_{C'}$ for some $C' \in \mathbb{C}'$.

For later considerations the *composition of two component morphisms* is defined in order to handle the transitivity property of the refinement relation.

Definition 4.6 *composition of two component morphisms*
Let $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_3 \subseteq SPEC$.

The *composition* $\beta \circ \alpha : \mathbb{C}_1 \to \mathbb{C}_3$ of two component morphisms
$\alpha : \mathbb{C}_1 \to \mathbb{C}_2$ and
$\beta : \mathbb{C}_2 \to \mathbb{C}_3$ is defined by

$$\beta \circ \alpha =_{def} \left(\beta^\Gamma \circ \alpha^\Gamma, \left(\beta^\Sigma_{C'} \circ \alpha^\Sigma_C\right)_{C \in \mathbb{C}_1}, \left(\beta^\Phi_{C'} \circ \alpha^\Phi_C\right)_{C \in \mathbb{C}_1} \right)$$

where $C' =_{def} \alpha^\Gamma(C)$ and

$$\beta^\Gamma \circ \alpha^\Gamma(C) =_{def} \beta^\Gamma\left(\alpha^\Gamma(C)\right)$$
$$\beta^\Sigma_{C'} \circ \alpha^\Sigma_C(x) =_{def} \beta^\Sigma_{C'}\left(\alpha^\Sigma_C(x)\right)$$
$$\beta^\Phi_{C'} \circ \alpha^\Phi_C =_{def} \left\{ \left(F_{|_{\beta^\Sigma_{C'} \circ \alpha^\Sigma_C}}, D \right) : F \in Mod\left(\beta^\Gamma \circ \alpha^\Gamma(C)\right), \left(F_{|_{\beta^\Sigma_{C'}}}, E\right) \in \beta^\Phi_{C'}, \left(E_{|_{\alpha^\Sigma_C}}, D\right) \in \alpha^\Phi_C \right\}$$

□

The composition of two component morphisms satisfies the following property.

Fact 4.7 *transitivity property of compositions of component morphisms*
Let $\mathbb{C}_1, \mathbb{C}_2, \mathbb{C}_3 \subseteq SPEC, R \subseteq SPEC \times SPEC$.

The *composition* of two component morphisms
$\alpha : \mathbb{C}_1 \to \mathbb{C}_2$ preserving $R_1 \subseteq R$ refining $C_{HB} \in \mathbb{C}_1$ and
$\beta : \mathbb{C}_2 \to \mathbb{C}_3$ preserving $R_2 \subseteq R$ refining $\alpha^\Gamma(C_{HB})$ yields a component morphism $\beta \circ \alpha$ preserving $R_2 \circ R_1 \subseteq R$ refining C_{HB},
where $R_2 \circ R_1 =_{def} \left\{(C, C') : \left(\alpha^\Gamma(C), \alpha^\Gamma(C')\right) \in |R_1|_\alpha \cap R_2\right\}$

Proof

- $\beta^\Gamma \circ \alpha^\Gamma$ is total and injective, since α^Γ and β^Γ are total and injective.
- The proposition that $\forall C \in \mathbb{C}_1$ the composition $\beta^\Gamma \circ \alpha^\Gamma(C)$ is a generalized heir of C remains unproven for complexity reasons. In the sequel it will be assumed to hold.
- Let $C, C' \in \mathbb{C}_1$. $C \;\; R' \;\; C'$ implies by α and lemma 4.5 that $\alpha^\Gamma(C) \;\; |R_2 \circ R_1|_\alpha \;\; \alpha^\Gamma(C')$. Furthermore follows by definition of $R_2 \circ R_1$ that $|R_2 \circ R_1|_\alpha \subseteq R_2$, hence $\alpha^\Gamma(C) \;\; R_2 \;\; \alpha^\Gamma(C')$. Since β preserves $R_2 \subseteq R$, $\beta^\Gamma \circ \alpha^\Gamma(C) \;\; R \;\; \beta^\Gamma \circ \alpha^\Gamma(C')$ also holds. Hence $\beta \circ \alpha$ preserves $R_2 \circ R_1$.

□

Using the notion of component morphisms, it is now possible to define a *refinement relation* between two design patterns.

Definition 4.8 *refinement relation between design patterns*
Let $psp, psp' \in PATTSPEC$.

psp is a refinement of psp' via the component morphism δ preserving a relation $R' \subseteq R \subseteq SPEC \times SPEC$ denoted by $psp' \overset{\delta, R' \subseteq R}{\rightsquigarrow} psp$ [1], iff

- the component morphism δ is defined as

$$\delta : Comp(psp') \to Comp(psp) \text{ is preserving } R' \subseteq R \text{ and refining } C_{HB}(psp'),$$

[1] $R' \subseteq$ is omitted in case $R' = R$

- $C_{HB}(psp) = \delta^{\Gamma}(C_{HB}(psp'))$,
- $translate_{P \to S}(psp) \ll translate_{P \to S}(psp')$.

□

The refinement relation uses the notion of component morphisms in order to refine the components of psp'. This concept bases on generalized inheritance. However, the external part of the psp subclasses the external part of psp'. Therefore, this way of refinement takes place on two levels. A design pattern eventually results in a specification which combines both the external part and the components of the design pattern. If these two levels interfere which each other in some way, it is possible that the refined design pattern is not *satisfiable*[2]. Therefore, any special refinement will impose various restrictions on the form of a design pattern in order to handle or to avoid these interferences. In the case of the particular refinement relation which will be defined for design patterns in PP, this will also have consequences regarding its syntax and semantics.

The refinement relation satisfies the following properties.

Fact 4.9 *properties of the refinement relation*
Let $psp, psp_1, psp_2, psp_3 \in PATTSPEC$, $R \subseteq SPEC \times SPEC$.

1. the refinement relation is reflexive, i.e.
   $psp \overset{id,R}{\rightsquigarrow} psp$ where id is the identical component morphism, i.e. $id^{\Gamma}(C') =_{def} C' \quad \forall C \in Comp(psp)$, $id^{\Sigma}_{C'}(x) =_{def} x$ and $id^{\Phi}_{C'} =_{def} \{(M, M) : M \in Mod(C')\}$,
2. the refinement relation is transitive
   if $psp_1 \overset{\alpha, R_1 \subseteq R}{\rightsquigarrow} psp_2$ and $psp_2 \overset{\beta, R_2 \subseteq R}{\rightsquigarrow} psp_3$ then also $psp_1 \overset{\beta \circ \alpha, R_2 \circ R_1 \subseteq R}{\rightsquigarrow} psp_3$.

Proof follows immediately by the definitions of the refinement relation.

□

The dependency relation between the component of the higher behaviour and all other components of a design pattern has to be preserved in any case to ensure that the refined design pattern is in $PATTSPEC$. Additionally, relations, that should be preserved, are usually the clientship-, subclass-, or inheritance relation on $SPEC$. The refinement relation in PP will have to preserve both the the syntactically deducable clientship- and subclass relation.

The notion of design pattern specifications combined with the introduced relationship on design pattern provide a framework which is general enough to form the basis for a theory which will eventually be used to describe the semantics of PP.

[2]The word *satisfiable* is used in analogy to the *satisfiability* of specifications $SPEC$. A design pattern specification psp is not *satisfiable* iff the set of models of one of its components is empty.

Chapter 5

A design pattern oriented command language

In the previous chapter, a formalism was presented to handle design patterns from a model-theoretic perspective. However, a design pattern specification does not integrate notions which correspond to an imperative perspective. However, they are of importance for the definition of the semantics of *PP*.

The aim of this chapter is to introduce an imperative command language which will eventually be used for method implementations in *PP*. To this end, it is necessary to extend the theoretical framework presented in chapter 3 by design pattern oriented features as mentioned in definition 2.3. These features include concepts for the representation of *instances*, *identities* and *states* of both design patterns and their components.

A design pattern specification consists of components which in turn are represented by specifications (cf. chapters 3 and 4). The notion of the specification is very general in nature. *State based signatures* and, based on these signatures, *object algebras* can be used to describe a (specialized) class of design pattern specifications supporting imperative design pattern oriented features as mentioned above.

Again, at this point it is emphasized that the well known object oriented notion has been adapted in a way that the new definition still comforms to the ideas of the object-oriented world. It is important that the rules of object oriented programming are not violated by the new model. In this way, the *PatternModel* can be conceived as an extension of the object oriented programming model.

5.1 State based signatures

In the sequel, it will be assumed that for sorts $s_1, \ldots, s_n$ contained in a signature Σ there is also a sort $s_1 \times \ldots \times s_n$ defined whose carrier set is interpreted as cartesian product of the corresponding carrier sets of $s_1, \ldots, s_n$ in some particular algebra. Moreover $\left((a_1, \ldots, a_n)^A\right) =_{def} (a_1, \ldots, a_n)$ and $(a_1, \ldots, a_n)_i^A =_{def} a_i$.

In chapter 3, the construct of a signature was introduced in order to describe the structure of Σ-algebras. Basically, a signature comprises a set of sorts and operation symbols on these sorts. A signature, which will eventually be used to model *identities* and *states*, has to provide additional sorts for them. A *state based signature* Σ_O is defined with respect to a *normal* signature Σ. Σ can contain a so-called *basic signature* Σ_V which is used for the specification of basic types like *Integer* or *Boolean* which are not modelled by identities and states but by values. Σ_O can now be constructed by the following definition.

Definition 5.1 *state based signature* (cf. [5] (modified))
Let $\Sigma = (S, \leq, \Vdash, F, class) \in SIG$. A signature $\Sigma_O = (S_O, \leq_O, \Vdash_O, F_O, \underline{class})$ is called a *state based signature with respect to* Σ *with basic type signature* $\Sigma_V = (S_V, \leq_V, \Vdash_V, F_V, class_V)$ iff

1. Σ_V is a closed component in Σ and Σ_O and $class \in S \backslash S_V$.
2. For each $s \in S \backslash S_V$ there are corresponding sorts $\underline{s}, \overline{s} \in S_O$ where $s \leq s'$ implies $\underline{s} \leq_O \underline{s'}$ and $s \Vdash s'$ implies $\underline{s} \Vdash_O \underline{s'}$. In the sequel the following notations will be used. The sort $\underline{s}$ is called the sort of *identities*, the sort $\overline{s}$ is called the sorts of *states* of sort s. The sort s is called an object sort.
3. There is a sort *env* in S_O such that for every operation $f : (s_1, \ldots, s_n) \to s_0$ in $F \backslash F_V$ there is an operation $f : (env, \underline{s_1}, \ldots, \underline{s_n}) \to env \times \underline{s_0}$ in F_O, where for all $i = 0, \ldots, n, \underline{s_i} =_{def} s_i$ if $s_i \in S_V$. The sort *env* is called the sort of *environments*.

□

Notation 5.2 *notations for state based signatures* (cf. [5])
In the sequel the following notation will be used to distinguish particular sorts and operation symbols.

- We use the notations $obj - sorts(\Sigma) =_{def} sorts(\Sigma) \backslash sorts(\Sigma_V)$ and $methods(\Sigma) =_{def} opns(\Sigma) \backslash opns(\Sigma_V)$. We denote in Σ_O by $\underline{s}$ the sort s if $s \in sorts(\Sigma_V)$ and the corresponding identity sort of s in $sorts(\Sigma_O)$, if $s \in obj - sorts(\Sigma)$.
- An operation $f : (s_1, \ldots, s_n) \to s \in opns(\Sigma)$ is called *operation of basic type* if $s \in sorts(\Sigma_V)$; it is called *basic operation* if it is contained in $opns(\Sigma_V)$. An operation $f \in methods(\Sigma)$ (and its corresponding operation in Σ_O) is called *method*.
- In order to obtain a uniform framework of methods, we define for every Σ_O-algebra A and basic operation $f : (s_1, \ldots, s_n) \to s$ a function $f^A : A_{env} \times A_{s_1} \times \ldots A_{s_n} \to A_{env} \times A_s$ by

$$f^A(p, x_1, \ldots, x_n) =_{def} \begin{cases} (p, f^A(x_1, \ldots, x_n)) & \text{if } f^A(x_1, \ldots, x_n) \text{ is defined} \\ \text{undefined otherwise.} \end{cases}$$

□

The sort *env* is defined at the global level. Therefore, *env* is visible from every location and every function $f \in opns(\Sigma_O)$ is visible from $\underline{c} \in sorts(\Sigma_O)$, if $f \in opns(\Sigma)$ is visible from c in $sorts(\Sigma)$.

In order to avoid problems with certain state based signatures, in the sequel only the subclass of so-called $\underline{\Sigma}$-signatures is considered. These signatures can easily be constructed based on a signature Σ with a basic type signature Σ_V.

Definition 5.3 *signature* $\underline{\Sigma}$ (cf. [5])
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature, let Σ_V be a closed component in Σ and $class \in obj - sorts(\Sigma)$. Then the state based signature $\underline{\Sigma}$ with basic type signature Σ_V is defined in the following way.

$$\begin{aligned} \underline{\Sigma} =_{def} \quad (\quad & sorts(\Sigma_V) \quad \cup \{ \quad Id_s, RId_s, State_s : s \in obj - sorts(\Sigma) \} \cup \{Env\}, \\ & \leq_{\underline{\Sigma}}, \\ & \Vdash_{\underline{\Sigma}}, \\ & opns(\Sigma_V) \quad \cup \{ \quad m : (Env, \underline{s_1}, \ldots, \underline{s_n}) \to Env \times \underline{s_0} : \\ & \qquad\qquad\qquad m : (s_1, \ldots, s_n) \to s_0 \in methods(\Sigma), \\ & \qquad\qquad\qquad \underline{s_i} = Id_{s_i} \text{ if } s_i \in obj - sorts(\Sigma), \underline{s_i} = s_i \text{ otherwise for } i = 0, \ldots, n \}, \\ & \underline{class} \\ ), & \end{aligned}$$

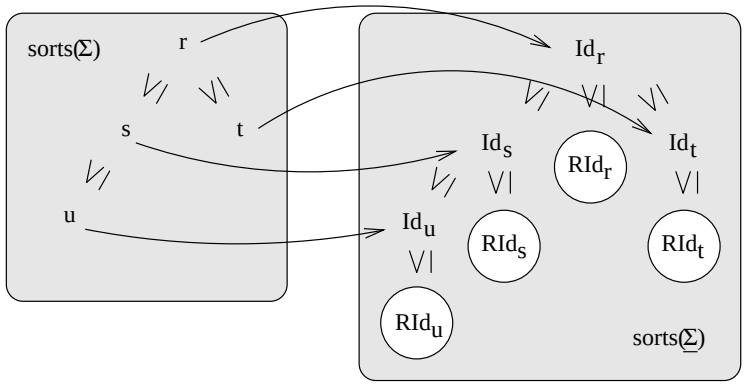

Figure 5.1: *Sorts in Σ with their corresponding sorts in $\underline{\Sigma}$. The sorts with a surrounding circle are minimal sorts of the real identities of a sort in Σ.*

where the sorts $Env, Id_s, RId_s, State_s$ do not occur in S. The partial ordering $\leq_{\underline{\Sigma}}$ is defined by

$$\begin{aligned}\leq_{\underline{\Sigma}} =_{def} \quad & (\{(s,s') : \\ & (s,s' \in sorts\,(\Sigma_V) \wedge s \leq s') \vee \\ & (s = Id_r, s' = Id_{r'} \wedge r \leq r') \vee \\ & (s = Id_r, s' = RId_r \wedge r \in obj - sorts\,(\Sigma)) \vee \\ & (s = s' = Env) \\ & \})^* .\end{aligned}$$

The relation $\Vdash_{\underline{\Sigma}}$ is defined by

$$\begin{aligned}\Vdash_{\underline{\Sigma}} =_{def} \quad & \{(s,s') : \quad (s,s' \in sorts\,(\Sigma_V) \wedge s \Vdash s') \vee \\ & \qquad\qquad (s = Id_r, s' = Id_{r'} \wedge r \Vdash r')\} .\end{aligned}$$

□

The sorts Id_s and $State_s$ will be used to model identities and states of a class or a design pattern, the sort Env is the sort of environments. Note that the coherence of Σ ensures the coherence of $\underline{\Sigma}$.

In a particular $\underline{\Sigma}$-algebra it is often necessary to associate elements of the carrier set of an identity sort with a minimal sort (e.g. to determine the dynamic type of an instance). This can be achieved by introducing sorts RId_s for every object sort s in Σ. Then, an identity in a carrier set A_{Id_r} can always be associated with one particular sort RId_s where $s \leq r$. The idea is illustrated in figure 5.1.

In object oriented program environments, the state of an instance is represented as aggregation of the state of all associated attributes. In this approach, in a given signature Σ, an attribute X associated with a sort s of type t will be modelled by an operation symbol X with a parameter of type s and a result of type t. In a $\underline{\Sigma}$-algebra, the corresponding function returns the value of the attribute in the current environment. Depending on the type of the attribute, this value can either be a basic value or an identity of the corresponding type.

An *attribute signature* consists in its non-basic part only of attribute operation symbols. This notion will later be used to define algebras which have a very special representation of object states and identities.

Definition 5.4 *attribute signature*
A signature $\Sigma = (S, \leq, \Vdash, F, class)$ is called *attribute signature with basic type signature* Σ_V, iff Σ_V is a

closed component in Σ and all non-basic operations in F are of the following form

$$X : (s) \to t \in \mathcal{F}^s.$$

These operations are called *attributes*. For simplicity reasons it will be assumed that attributes are not polymorphic with respect to the dependency structure or overloaded, i.e. $F_{s,r} \cap F_{s',r'} \in opns(\Sigma_V)$ if $(s < s'$ and $(s \not\Vdash r$ or $s' \not\Vdash r'))$ or $(s = s'$ and $r \neq r')$.
The set of attributes associated with the type s is denoted by $attr(\Sigma, s), s \in obj - sorts(\Sigma)$. It is defined by

$$attr(\Sigma, s) =_{def} \{X : (s') \to r \in \mathcal{F}^s : s \leq s'\}$$

The set of valid attribute signatures is called $ATTRSIG$. □

Attributes are visible iff their corresponding operation symbols are visible. If an attribute of s is not only visible from s, then it is also associated with every $s' \leq s$. It is called *object attribute (of sort r)* if $r \in obj - sorts(\Sigma)$, it is called *basic type attribute (of sort r)* if $r \in sorts(\Sigma_V)$.

5.2 Object algebras

State based signatures contain special sorts to represent identities, states and environments. As a matter of fact, every $\underline{\Sigma}$-algebra can be used to describe the semantics of *PP*. However, subsequent sections rely on a particular construction of $\underline{\Sigma}$-algebras. Therefore the class of $\underline{\Sigma}$-environment algebras is introduced. These algebras use records comprising the attributes of a class in order to model the state of objects. Besides, environments have to be of a special form which corresponds to design pattern oriented modelling and implementation ideas as introduced in later sections.

A $\underline{\Sigma}$-environment algebra consists of the following carrier sets and functions:

- The carrier sets A_{RId_s} are sets of identities. Void references are denoted by $void_s$.
- The carrier sets A_{Id_s} which are defined as the union of the carrier sets of all subsorts. Then, all identities of a subsort can also be treated as identities of a supersort which is very important for the applicability of the substitution principle.
- The carrier sets A_{State_s} are modelled as records with a field for each attribute in $attr(\Sigma, s)$. Basic type attributes are associated with values of the correponding basic sort, object type attributes are associated with object identities. For accessing the records, the notations defined in B are used.
- Elements of the carrier set of sort Env are pairs $\left(p^{\mathcal{I}}, (p_s^{\rightarrow})_{s \in S_O}\right)$ representing a particular state of the system. Each pair consists of the so-called *active identities* and mappings between active identities and states. Active identities are represented by a dependency set which is defined on a family of sets containing those identities of a sort which have been associated with an instance in the current state of the system. Additionally, the dependency relation on this familiy of sets defines a dynamic view of visibility of elements. Identities which conceptually belong to a sort c may not refer via states to identities corresponding to sorts which are not visible form c in Σ. In this way, the dependence relation of sorts has its counterpart in the dynamic dependence of instances. Example 5.6 shows a typical environment.
- Applications $X^A(p, n)$ denote the value of the attribute X in the record $p[n]^A$ referenced by an identity n.
- The functions $create^A_{r \Vdash^\circ s}$ are used to model the dynamic creation of instances. The set of active identities is enlarged by one *new* identity which in turn is associated with some initial state. The new identity is created to be dependent on some other identity of sort r. If the sort s is dependent on sort $\perp$ in $\Vdash^\circ$ then the new instance is created directly under $\perp$ in $p^{\mathcal{I}}$.

- The auxiliary functions set_X^A are defined where $set_X^A(p, n, x)$ associates the attribute X in the record $p[n]^A$ with the value or identity x.
- The auxiliary functions $\uparrow^A$ are defined where $\uparrow^A(p, n_s)$ returns the instance which n_s depends on.

Definition 5.5 *$\underline{\Sigma}$-environment algebra*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be an attribute signature with basic type signature Σ_V. Let $S_O =_{def} obj - sorts(\Sigma)$ and let $Min(S_O)$ denote the set of minimal sorts in S_O. Then a *$\underline{\Sigma}$-environment algebra* A is a $\underline{\Sigma}$-algebra with the following properties.

1. All sets $A_{RId_s}, s \in S_O$ are disjoint and contain an element $void_s$. Moreover, $A_{Id_s} = A_{RId_s} \cup \left(\bigcup_{r<s} A_{Id_r}\right)$ and $A_{RId} =_{def} \bigcup_{s \in S_O} A_{RId_s}$.
2. $A_{State_s} \subseteq RECORD(attr(\Sigma, s))$ for all $s \in S_O$, where
$$RECORD(attr(\Sigma, s)) =_{def} \left\{(\sigma_X)_{X:(s')\to r \in attr(\Sigma,s)} : \sigma_X \in [\{X\} \to A_{\underline{r}}]_{fin} \text{ is total}\right\}$$
3. $A_{Env} \subseteq ENV(S_O)$, where $ENV(S_O)$ is the set of pairs $\left(p^{\mathcal{I}}, (p_s^{\rightarrow})_{s \in S_O}\right)$ satisfying the following properties.
   (a) $p^{\mathcal{I}}$ is a dependency set $\left((\mathcal{I}_s)_{s \in S_O}, \Vdash_{\mathcal{I}}\right)$ where $\mathcal{I}_s \subseteq A_{RId_s}, void_s \notin \mathcal{I}_s$ and for all $i_s \in \mathcal{I}_s, i_t \in \mathcal{I}_t$ holds $i_s \Vdash_{\mathcal{I}} i_t \Longrightarrow s \Vdash t$ and for all $i_s \in \mathcal{I}_s$ holds $\bot\,(\Vdash_{\mathcal{I}})^{\circ}\, i_s \Longrightarrow \bot \Vdash^{\circ} s$.
   (b) $p_s^{\rightarrow} \in [\mathcal{I}_s \to A_{State_s}]_{fin}$ is total,
   (c) $\forall s' \in S_O, \forall n \in \mathcal{I}_{s'}$ for all object attributes $X : (s) \to r \in attr(\Sigma, s')$ where $n' = p_{s'}^{\rightarrow}[n][X]$ follows either ($n' \in \mathcal{I}$ and n' is visible from n in $p^{\mathcal{I}}$) or ($n' = void_{r'}$ for some $r' \leq r$).
4. The functions $X^A : A_{Env} \times A_{Id_s} \to A_{Env} \times A_{\underline{r}}$ for each attribute $X : (s) \to r \in F$ satisfy
$$X^A\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), n\right) = \begin{cases} \left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), p_{s'}^{\rightarrow}[n][X]\right) \\ \quad \text{where } s' \in S_O, n \in \mathcal{I}_{s'}, X : (s) \to r \in attr(\Sigma, s') \\ \text{undefined otherwise.} \end{cases}$$
5. $create^A_{r\Vdash^{\circ}s} : A_{Env} \times A_{Id_r} \cup \{\bot\} \to A_{Env} \times A_{Id_s}, s \in S_O, r \Vdash^{\circ} s$ are total functions satisfying
$$create^A_{r\Vdash^{\circ}s}\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), n_r\right) = \begin{cases} \left(\left(p^{\mathcal{I}'}, p^{\rightarrow}[n \to_s init_s]\right), n\right), \text{ if } n_r \in \mathcal{I}_r^{\circ} \\ \text{undefined otherwise,} \end{cases}$$
   where $n \in A_{RId_s}$, $n \notin \mathcal{I}$, $p^{\mathcal{I}'} =_{def} \left((\mathcal{I}'s)_{s \in S_O}, \Vdash_{\mathcal{I}'}\right)$, $\mathcal{I}'s =_{def} \mathcal{I}_s \cup \{n\}$ and $\mathcal{I}'r =_{def} \mathcal{I}_r$ for $r \in S_O : r \neq s$, $\Vdash^{\circ}_{\mathcal{I}'} =_{def} \Vdash^{\circ}_{\mathcal{I}} \cup \{(n_r, n)\}$.
   The constant $init_s \in A_{State_s}$ satisfies $init_s[X] = v_r$ for some given default $v_r \in A_r$ for all basic type attributes X of sort r in $attr(\Sigma, s)$ and $init_s[X] = void_r$ for all object attributes X of sort r in $attr(\Sigma, s)$.
6. The functions $set_X^A : A_{Env} \times A_{Id_s} \times A_{\underline{r}} \to A_{Env} \times A_{Id_s}$ for each attribute $X : (s) \to r \in F$ satisfy
$$set_X^A\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), n, x\right) = \begin{cases} \left(\left(p^{\mathcal{I}}, p^{\rightarrow}[n \to_{s'} p_{s'}^{\rightarrow}[n][X \to x]]\right), n\right) \\ \quad \text{where } s' \in S_O, n \in \mathcal{I}_{s'}, X : (s) \to r \in attr(\Sigma, s') \\ \quad \text{and for all} \\ \quad \text{object attributes } X \text{ holds } x \in \mathcal{I} \text{ and } x \text{ is visible from } n \text{ in } p^{\mathcal{I}} \\ \quad \text{or } x = void_{r'} \text{ for some } r' \leq r \\ \text{undefined otherwise.} \end{cases}$$

7. The functions $\uparrow^A$: $A_{Env} \times A_{Id_s} \to A_{Env} \times A_{Id_r} \cup \{\bot\}$ where $r, s \in obj - sorts(\Sigma), r \Vdash s$ satisfy

$$\uparrow^A \left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), n\right) = \begin{cases} \left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), n_r\right), \text{ if } n \in \mathcal{I} \\ \text{undefined otherwise,} \end{cases}$$

where $n_r \Vdash^{\circ}_{\mathcal{I}} n$.

□

Example 5.6 Figure 5.2 shows three design pattern instances p_1, p_2 and p_3. p_1 is an instance of the design pattern *GraphicComposite* and contains several component instances according to the definition of the *GraphicComposite* (cf. appendix C.4). p_2 is an instance of a design pattern without components describing an *X-Windows*-screen. An important fact to notice is that attributes of p_2 can not point to component instances inside of p_1. In this case, this scenario is not possible anyway since the type information of the components of the *GraphicComposite* can not be accessed by p_2. But p_3 as instance of *GraphicComposite* has all necessary type information about components in *GraphicComposite*. But even then, attributes of p_3 or attributes of component instances of p_3 are not allowed to point to component instances owned by p_1. E.g. although the the type of the next attribute of c_7 is compatible with the type of c_4, this reference is not allowed since c_7 and c_4 do not belong to the same design pattern instance.
However, the opposite direction is possible. As shown in figure 5.2, the *screen* attribute of c_1 points to p_1.
□

$\underline{\Sigma}$-environment algebras model identities, states and environments in a state based system a very particular way. These algebras are part of the semantic model which will be used for *PP*. In order to support hierarchical construction mechanisms of classes that base on model class inclusion, the following definition provides the more general notion of $\underline{\Sigma}$-*object algebras*. In such $\underline{\Sigma}$-object algebras, the carrier sets of identity sorts can contain additional identities, the records of the carrier sets of state sorts can comprise fields which are not part of the attribute signature. Moreover, environments can map between these addtional identities and states.

Definition 5.7 $\underline{\Sigma}$-*object algebra*
Let Σ be a signature with basic type signature Σ_V and Σ contains an attribute signature $\Sigma_a \subseteq \Sigma$ with basic type signature Σ_V. A $\underline{\Sigma}$-object algebra A is a $\underline{\Sigma}$-algebra such that $A_{|\underline{\Sigma_a}} = B_{|\underline{\Sigma_a}}$ for some $\underline{\Sigma'_a}$-environment algebra B, $\Sigma'_a \supseteq \Sigma_a$.

□

In [5], an additional requirement has to be fulfilled by every function m^A with $m \in methods(\Sigma)$. It limits the effect of functions to what is called the *local state*. The local state is that part of the environment which can be reached from the parameters following the trace of references and attributes. Functions that only modify this local state with respect to an environment are called *local state transition functions*. However, for the remainder of this thesis, such a requirement is not necessary in order to obtain the same results, although methods that satisfy method implementations as defined in the following subsection are local state transition functions.

5.3 Commands and method implementations

In order to define a kernel command language which will later be used for method implementations, every signature has to satisfy some additional restrictions. In the sequel it is assumed that every signature contains at least a sort *Bool* in the global level which is not comparable with other sorts in this signature and the

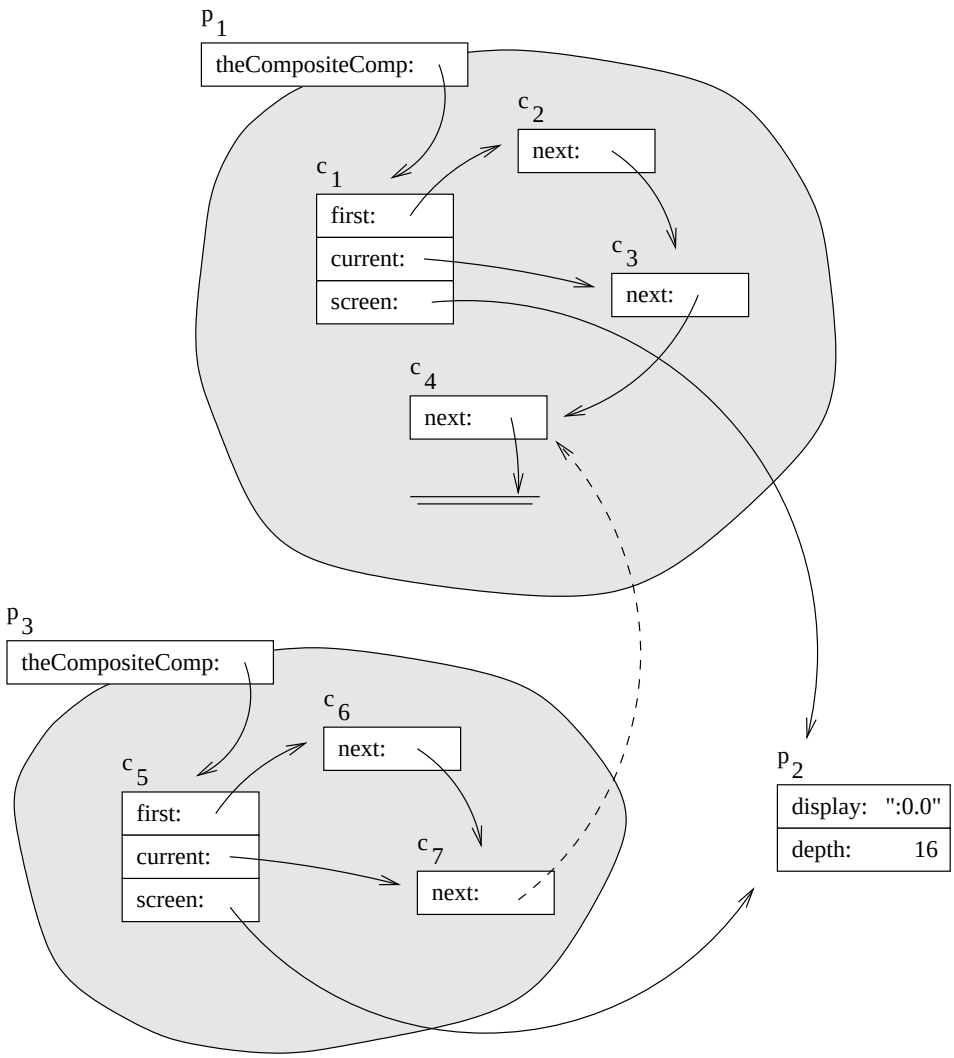

Figure 5.2: *Instances in the* PatternModel. *Identities for design patterns are p_1 and p_2. p_1 contains component instances $c_1, \ldots, c_4$, p_3 contains the component instances $c_5, \ldots, c_7$.*

logical operation symbols $true, false, \wedge, \vee$ and $\neg$ with the usual arities.

Furthermore, the class of *flat signatures* is defined. Every operation symbol in a flat signature has to be visible from its first parameter sort. This property is of special importance for the modelling of selfish methods. In this approach, the implicit parameter *self* is passed as the first parameter. By the definition of a signature (cf. chapter 3) it is required that there must be at least one location which an operation symbol is visible from. Hence, this operation is visible from one of its parameter sorts or the result sort. In a flat signature all operations symbols are required to be visible from the first parameter sort.

Definition 5.8 *flat signature*
A signature $\Sigma = (S, \leq, \Vdash, F, class)$ is called *flat signature with basic type signature* Σ_V, iff Σ_V is a closed component in Σ and all methods in F satisfy the condition

$$f : (c, s_1, \ldots, s_n) \rightarrow s \in \mathcal{F}^c.$$

□

5.3.1 Commands and their execution

Commands are defined over a signature Σ at a location c. Analogously to terms as introduced in definition 3.14, a command is a syntactic construct which will eventually be used to perform specific actions within the context of a $\underline{\Sigma}$-object algebra. These actions are defined by a family of execution functions representing the semantics of commands. The *execution* of commands is indeed similar to the interpretation of terms in Σ-algebras. However, in contrast to the rather functional character of terms, a command is very imperative in nature. The introduced class of $\underline{\Sigma}$-object algebras is capable of expressing a *state of the system*. Thus, a command can be used to alter this state in the desired way yielding a final state as the result.

Commands are used in *PP* to implement the bodies of methods. The aim of language design is to define the syntax of a programming language in a way that is most efficient for the actual purpose of the language. The command language in *PP* is suited for a general purpose but also taylored to use design pattern oriented features. The command language provides constructs for

- the invocation of methods,
- the assignment of values to attributes of the current instance ($self$ - instance),
- the sequential composition of commands,
- the selection of commands,
- commands for the creation of instances, the test for equality and $void$-references and
- the up-command $\uparrow$ which is used to obtain the instance the current instance is created in.

PP is supposed to be a prototype for a design pattern oriented language. It does not support loops which are crucial for every kind of imperative language. The semantics of loops can be described by solutions of fixed point problems. *PP* can be extended by this feature straightforward since these fixed point problems can easily be expressed within the presented framework.

The formal definition of commands over a signature Σ at a location c is as follows.

Definition 5.9 *commands*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature with a basic type signature Σ_V, Σ_f a flat signature $\Sigma_f \subseteq \Sigma$ with basic type signature Σ_V, Σ_a an attribute signature $\Sigma_a \subseteq \Sigma_f$ with basic type signature Σ_V, $c \in S^\circ \cap obj - sorts(\Sigma)$ and X a $\mathcal{S}^c$-indexed family of disjoint sets of variables containing a variable $Self$ of type c. The $\mathcal{S}^c$-indexed family $COM(\Sigma, X)^c$ of *commands* over Σ at the location c with variables X is inductively defined by

1. $f \in COM(\Sigma, X)^c_s$ for all $f :\to s \in \mathcal{F}^c$,
2. $x \in COM(\Sigma, X)^c_s$ for all $x \in X_s$,
3. $f(t, t_1, \ldots, t_n) \in COM(\Sigma, X)^c_s$ for all $f : (d, s_1, \ldots, s_n) \to s \in \mathcal{F}^c$, $t \in COM(\Sigma, X)^c_d$, $t_i \in COM(\Sigma, X)^c_{s_i}, i = 1, \ldots, n$ with the restriction that if

   $$f : (d, s_1, \ldots, s_n) \to s \in \Re\left(\mathcal{F}^d\right) \cap opns(\Sigma_f)$$

   then the term t has to satisfy the condition

   $$t \in \left(\{Self\} \cup \left\{\uparrow t_* : t_* \in COM(\Sigma, X)^c_{s_*}, s_* \in obj - sorts(\Sigma)\right\}\right) \quad \cap \quad COM(\Sigma, X)^c_d,$$

4. $t_1;t_2 \in COM\,(\Sigma,X)^c_{s_2}$ if $t_i \in COM\,(\Sigma,X)^c_{s_i}, i=1,2$,
5. if t_b then t_1 else t_2 end $\in COM\,(\Sigma,X)^c_s$ if $t_b \in COM\,(\Sigma,X)^c_{Bool}, t_1,t_2 \in COM\,(\Sigma,X)^c_s$,
6. $t_1 == t_2 \in COM\,(\Sigma,X)^c_{Bool}$ if $t_i \in COM\,(\Sigma,X)^c_s, i=1,2, s \in obj-sorts\,(\Sigma)$,
7. t.isVoid $\in COM\,(\Sigma,X)^c_{Bool}$ if $t \in COM\,(\Sigma,X)^c_s, i=1,2, s \in obj-sorts\,(\Sigma)$,
8. $Self.(X := t) \in COM\,(\Sigma,X)^c_c$ for all attributes $X : (c') \to r \in attr\,(\Sigma,c)$ if $t \in COM\,(\Sigma,X)^c_r$,
9. create $r \in COM\,(\Sigma,X)^c_r$ for all $r \in \mathcal{S}^c$,
10. $\uparrow t \in COM\,(\Sigma,X)^c_s$ if $t \in COM\,(\Sigma,X)^c_u, s \Vdash u$,
11. $COM\,(\Sigma,X)^c_r \subseteq COM\,(\Sigma,X)^c_s$ if $r \leq s$.

The set of commands at a location c are dentoted by $COMMAND^c$. If the location is not important for a particular consideration then it is omitted and the set of all commands is denoted by $COMMAND$. □

Note that in analogy to the properties of terms (cf. chapter 3), it can be shown that a command at some location is also a valid command at all dependent locations. In particular, the following fact holds for a signature Σ and a variable set X.

$$COM\,(\Sigma,X)^c_s \subseteq COM\,(\Sigma,X)^{c'}_s \text{ if } c \Vdash c'.$$

By induction can be proven that this proposition even holds for all c' where $c \Vdash^* c'$.

The following notion of *command execution* defines the (denotational) semantics of commands, i.e. it defines a function which in turn determines for every command the action that is carried out in a particular $\underline{\Sigma}$-object algebra. For this purpose, the state of the system which is passed to a function as parameter *env* has to be considered especially. Again, commands as well as terms are executed following an *innermost-first* strategy. Only when all parameter commands of a method call have been evaluated, the actual execution of the method call can take place. However, the execution of the first parameter command changes the state of the system. Therefore, the execution of the second parameter command depends on the result of the execution of the first parameter command. This applies to all parameters of the function, i.e. the execution of parameter t_j depends on the execution of parameter t_i, if $i<j$ where i,j specify the parameter position in the method call. The auxiliary functions $(v^c)^*_{s_1\ldots s_n}$ in the following definition handle this dependence problem by passing the resulting environment of the previous parameter execution as current environment into the execution of the following parameter command. In this way, all executions on the same location are evaluated from left to right.

At this point, it is especially emphasized that commands are defined over a signature Σ while command executions take place in corresponding $\underline{\Sigma}$-algebras. The execution functions $(v^c)^*_s$ are dependent on a certain location c and result in a value of type $\underline{s}$. Formally, the family of execution functions is defined as follows.

Definition 5.10 *command execution*
Let $\Sigma = (S,\leq,\Vdash,F,class),\Sigma_O \in SIG$ with basic type signature Σ_V, $c \in S^\circ \cap obj-sorts\,(\Sigma)$, X a $\mathcal{S}^c$-indexed family of disjoint sets of variables, A a $\underline{\Sigma}$-object algebra and $v = \left(v_s : X_s \to A_{\underline{s}}\right)_{s\in\mathcal{S}^c}$ is a variable assignment. The *execution* of commands in A is described by families of functions

$$\left((v^c)^*_s : COM\,(\Sigma,X)^c_s \to A_{env} \to \left(A_{env} \times A_{\underline{s}}\right)\right)_{s\in\mathcal{S}^c} \text{ and}$$
$$\left((v^c)^*_{s_1\ldots s_n} : COM\,(\Sigma,X)^c_{s_1} \times \ldots \times COM\,(\Sigma,X)^c_{s_n} \to A_{env} \to \left(A_{env} \times A_{\underline{s_1}} \times \ldots \times A_{\underline{s_n}}\right)\right)_{s_1\ldots s_n\in(\mathcal{S}^c)^*}.$$

The functions $(v^c)^*_s$ are inductively defined by

1. $(v^c)^*_s(f)(p) =_{def} f^A_{env\ env\times\underline{r}}(p)$ if defined and if $f :\to r \in \mathcal{F}^c, s \geq r$,

2. $(v^c)^*_s(x)(p) =_{def} (p, v_r(x))$ if $x \in X_r, s \geq r$,

3. if $f : (r_1, \ldots, r_n) \to r \in \mathcal{F}^c, s \geq r$ then $(v^c)^*_s(f(t_1, \ldots, t_n))(p) =_{def}$

$$\begin{cases} f^A_{env\underline{r_1}\ldots\underline{r_n}\ env\times\underline{r}}\left((v^c)^*_{r_1\ldots r_n}(t_1, \ldots, t_n)^A(p)\right) \text{ if } (v^c)^*_{r_1\ldots r_n}(t_1, \ldots, t_n)^A(p) \text{ and} \\ \qquad f^A_{env\underline{r_1}\ldots\underline{r_n}\ env\times\underline{r}}\left((v^c)^*_{r_1\ldots r_n}(t_1, \ldots, t_n)^A(p)\right) \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

4. $(v^c)^*_{s_2}(t_1;t_2)(p) =_{def}$

$$\begin{cases} (v^c)^*_{s_2}(t_2)(p') \text{ if } (v^c)^*_{s_1}(t_1)(p) = (p', x) \text{ and } (v^c)^*_{s_2}(t_2)(p') \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

5. $(v^c)^*_s(\mathsf{if}\ t_b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \mathsf{end})(p) =_{def}$

$$\begin{cases} (v^c)^*_s(t_1)(p') \text{ if } (v^c)^*_{Bool}(t_b)(p) = (p', true^A) \text{ and } (v^c)^*_s(t_1)(p') \text{ are defined} \\ (v^c)^*_s(t_2)(p') \text{ if } (v^c)^*_{Bool}(t_b)(p) = (p', false^A) \text{ and } (v^c)^*_s(t_2)(p') \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

6. $(v^c)^*_{Bool}(\ t_1 == t_2\ )(p) =_{def}$

$$\begin{cases} (p', true^A) \text{ if } (v^c)^*_{ss}(t_1, t_2)(p) = (p', x_1, x_2) \text{ is defined and } x_1 = x_2 \\ (p', false^A) \text{ if } (v^c)^*_{ss}(t_1, t_2)(p) = (p', x_1, x_2) \text{ is defined and } x_1 \neq x_2 \\ \text{undefined otherwise,} \end{cases}$$

7. $(v^c)^*_{Bool}(\ t.\mathsf{isVoid})(p) =_{def}$

$$\begin{cases} (p', true^A) \text{ if } (v^c)^*_s(t)(p) = (p', x) \text{ is defined and } x = void \\ (p', false^A) \text{ if } (v^c)^*_s(t)(p) = (p', x) \text{ is defined and } x \neq void \\ \text{undefined otherwise,} \end{cases}$$

8. $(v^c)^*_{c'}(Self.(X := t))(p) =_{def}$

$$\begin{cases} set^A_X(p', v_c(Self), x) \text{ if } (v^c)^*_r(t)(p) = (p', x) \text{ and } set^A_X(p', v_c(Self), x) \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

9. $(v^c)^*_s(\mathsf{create}\ r\ )((p^{\mathcal{I}}, p^{\to})) =_{def}$

$$\begin{cases} create^A_{\Uparrow(\{r\})\Vdash^\circ r}((p^{\mathcal{I}}, p^{\to}), x) \text{ if defined} \\ \text{undefined otherwise,} \end{cases}$$

where $x\,(\Vdash^\circ_{\mathcal{I}})^*\,v_c(Self), x \in \mathcal{I}_{\Uparrow(\{r\})}$,

10. $(v^c)^*_s(\uparrow t)(p) =_{def}$

$$\begin{cases} \uparrow^A(p', x) \text{ if } (v^c)^*_u(t)(p) = (p', x) \text{ and } \uparrow^A(p', x) \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

The functions $(v^c)^*_{s_1 \ldots s_n}$ are inductively defined by

1. $(v^c)^*_{\varepsilon} =_{def} p$,
2. and if $(v^c)^*_{s_1 \ldots s_n}(t_1, \ldots, t_n)(p) = (p_1, x_1, \ldots, x_n)$ then $(v^c)^*_{s_1 \ldots s_n, s_{n+1}}(t_1, \ldots, t_n, t_{n+1})(p) =_{def}$

$$\begin{cases} (p', x_1, \ldots, x_n, x') \text{ if } (v^c)^*_{s_{n+1}}(t_{n+1}) = (p', x') \text{ is defined,} \\ \text{undefined otherwise.} \end{cases}$$

□

Analogously to the interpretation of terms, the coherence of Σ ensures the well-definedness of the execution functions. Furthermore, a term at a location is executed in all dependent locations of this particular location in the same way (cf. to properties of the interpretation of terms in chapter 3).

5.3.2 Method implementations

Every notion that has been defined so far is designed to deal with signatures of arbitrary depth in their dependency set. For simplicity reasons, however, *PP* provides only three different levels of nesting: the global level, the design pattern level and the component level. Again, this limitation is not too restrictive since it still allows to implement the design patterns presented in [8].

Taking the execution of commands as basis, any command induces a functional in a particular $\underline{\Sigma}$-algebra in the following way.

Definition 5.11 Let A be a $\underline{\Sigma}$-object algebra. A command
$com \in COM(\Sigma, \{Self : s, X_1 : s_1, \ldots, X_n : s_n\})^c_r$ induces a functional

$$com^A : A_{Env} \times A_{\underline{s}} \times A_{\underline{s_1}} \times \ldots \times A_{\underline{s_n}} \to A_{Env} \times A_{\underline{r}} \text{ by}$$
$$com^A\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), x_0, x_1, \ldots, x_n\right) =_{def} (v^c)^*_r(com)\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right)\right),$$

where v is the assignment given by $v(Self) = x_0$, $v(X_i) = x_i, i = 1, \ldots, n$, if $s_i \in obj - sorts(\Sigma)$ implies $x_i \in \mathcal{I} \cup \{void\}, i = 0, \ldots, n$ and $(v^c)^*_r(com)\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right)\right)$ is defined,

$$com^A\left(\left(p^{\mathcal{I}}, p^{\rightarrow}\right), x_0, x_1, \ldots, x_n\right) \text{ is undefined otherwise.}$$

□

Design patterns as well as components comprise method declarations together with their implementations. Methods are implemented using the previously defined command language. In this approach, methods are *selfish*, i.e. there is an implicit parameter $Self$ that is passed to the method at each call. $Self$ contains a reference to the active instance. This instance may be the instance of a design pattern or the instance of a component. A method implementation can be redefined by subclasses[1] as supported e.g. by $C++$. Moreover, a method call should be polymorphic, i.e. an implementation should be selected based on the *dynamic type* of the $Self$ parameter. This requires that the dynamic type of $Self$ is deducable.

A method implementation may contain recursive calls to itself. However, for simplicity reasons, method implementations may not contain mutual recursive method calls.

[1] This process is also known as *overriding*.

Definition 5.12 Let Σ be a signature with basic type signature Σ_V. A *method implementation* imp_m *over* Σ *with arity* $m : (s_0, s_1, \ldots, s_n) \to s \in opns(\Sigma)$ is of the form
$imp_m = m(X_1 : s_1, \ldots, X_n : s_n)$ **return** s **is** com **end**, $s_0 \in obj - sorts(\Sigma)$ and
$com \in COM(\Sigma, \{Self : s_0, X_1 : s_1, \ldots, X_n : s_n\})_s^{s_0}$.

A $\underline{\Sigma}$-object algebra A *satisfies* a method implementation imp_m over Σ with arity $m : (s_0, s_1, \ldots, s_n) \to s \in opns(\Sigma)$ iff $imp_m = m(X_1 : s_1, \ldots, X_n : s_n)$ **return** s **is** com **end** and m^A is the least fixed point of the functional induced by com on the real identities of s_0, in particular $m^A = com^A$[2].

The algebra A *satisfies* a set of method implementations MI over Σ, denoted by $A \models MI$, iff A satisfies each $imp_m \in MI$.

□

In this framework, the definition of $\underline{\Sigma}$-algebras ensures that identities can be associated with minimal sorts. A method in Σ belongs to a sort s determined by the type of the $Self$ parameter. However, the corresponding function in a $\underline{\Sigma}$-object algebra has to satisfy the method implementation only on the subset A_{RId_s} (*real identities*) of A_{Id_s} (*all identities*).

In common object oriented programming languages such as *C++*, a method call $m(t_1, t_2, \ldots, t_n)$ is written in *point notation*, i.e. the command is equivalent to $t_1.m(t_2, \ldots, t_n)$. This notation will also be used in the remainder. Moreover, if t_1 is the variable $Self$, then the command $Self.m(t_2, \ldots, t_n)$ can be abbreviated by $m(t_2, \ldots, t_n)$.

5.3.3 Command translations

One of the benefits of design pattern oriented programming is the structural reuse of components of a design pattern via refinements as described in chapter 4. In *PP*, a specialized kind of refinement has been realized which bases on *command translation*.

Every refinement operator uses the notion of signature morphism to alter the signature within the capsule of a design pattern. When a design pattern is refined in *PP*, the method implementations of the methods of the source design pattern have to be adapted to the namespace of the refined design pattern (as pointed out in section 2.3.3).

Definition 5.13 *command translation*
Let $\Sigma = (S, \leq, \Vdash, F, class)$ be a signature, $\sigma : \Sigma \to \Sigma'$ be a total signature morphism refining sort c^r, $c \in S^\circ \cap obj - sorts(\Sigma)$ and X a $\mathcal{S}^c$-indexed family of disjoint sets of variables. The *translation* of commands via σ is described by families of functions

$$\left(t_s^c : COM(\Sigma, X)_s^c \to COM(\Sigma', X')^{\sigma(c)}\right)_{s \in \mathcal{S}^c}$$

where the set of transformed variables is defined by

$$X' =_{def} (X'_{r'})_{r' \in (\mathcal{S}')^{\sigma(c)}}$$

with

$$X'_{\sigma(r)} =_{def} \begin{cases} X_r & \text{if } r \neq c^r \\ X_{c^r} \setminus \{Self\} & \text{if } r = c^r \text{ and } c = c^r \\ X_{c^r} & \text{if } r = c^r \text{ and } c \neq c^r, \end{cases}$$

[2]In this thesis as well as in [5], function spaces are ordered in the following way. $f^A \sqsubseteq g^A$ iff $f^A(p, x_0, \ldots, x_n)$ defined implies $f^A(p, x_0, \ldots, x_n) = g^A(p, x_0, \ldots, x_n)$ for all $p \in A_{Env}$, $x_0 \in A_{RId_{s_0}}$, $x_i \in A_{\underline{s_i}}, i = 1, \ldots, n$ for any two functions $f, g : A_{Env} \times A_{\underline{s_0}} \times A_{\underline{s_1}} \times \ldots \times A_{\underline{s_i}} \to A_{Env} \times A_{\underline{s}}$. In this approach, the functional induced by a command com is monotonic and continuous (cf. [1]).

and

$$X'_{c^r} =_{def} \begin{cases} \{Self\} \text{ if } c = c^r \\ \emptyset \text{ if } c \neq c^r. \end{cases}$$

and $S' =_{def} sorts\,(\Sigma')$.
t^c_s is defined by

1. $t^c_s\,(f) = \sigma\,(f)$ if $f :\to r \in \mathcal{F}^c, s \geq r$,
2. $t^c_s\,(x) = x$ if $x \in X_r, s \geq r$,
3. if $f : (r_1, \ldots, r_n) \to r \in \mathcal{F}^c, s \geq r$, then $t^c_s\,(f\,(t_1, \ldots, t_n))\,(p) =_{def}$

$$\begin{cases} \sigma\,(f)\left(t^c_{r_1}\,(t_1), \ldots, t^c_{r_n}\,(t_n)\right) \text{ if } t^c_{r_i}\,(t_i) \text{ are defined for } i = 1, \ldots, b \\ \text{undefined otherwise,} \end{cases}$$

4. $t^c_{s_2}\,(t_1;\, t_2) =_{def}$

$$\begin{cases} t^c_s\,(t_1)\,;\, t^c_s\,(t_2) \text{ if } t^c_s\,(t_1) \text{ and } t^c_s\,(t_2) \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

5. $t^c_s\,(\,\mathsf{if}\ t_b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \mathsf{end}\,) =_{def}$

$$\begin{cases} \mathsf{if}\ t^c_b\,(t_b)\ \mathsf{then}\ t^c_s\,(t_1)\ \mathsf{else}\ t^c_s\,(t_2)\ \mathsf{fi} \text{ if } t^c_b\,(t_b), t^c_s\,(t_1) \text{ and } t^c_s\,(t_2) \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

6. $t^c_{Bool}\,(\,t_1 == t_2\,) =_{def}$

$$\begin{cases} t^c_s\,(t_1) == t^c_s\,(t_2) \text{ if } t^c_s\,(t_1), t^c_s\,(t_2) \text{ are defined} \\ \text{undefined otherwise,} \end{cases}$$

7. $t^c_{Bool}\,(\,t.\mathsf{isVoid}\,) =_{def}$

$$\begin{cases} t^c_s\,(t)\,.\mathsf{isVoid} \text{ if } t^c_s\,(t) \text{ is defined} \\ \text{undefined otherwise,} \end{cases}$$

8. $t^c_{c'}\,(Self.\,(X := t)) =_{def}$

$$\begin{cases} Self.\sigma\,(X) := t^c_r\,(t) \text{ if } t^c_r\,(t) \text{ is defined} \\ \text{undefined otherwise.} \end{cases}$$

9. $(t^c)^*_s\,(\,\mathsf{create}\ r\ ) =_{def}$

$$\begin{cases} \mathsf{create}\ \sigma\,(r) \text{ if } c^r \Vdash r \\ \mathsf{create}\ r \text{ otherwise,} \end{cases}$$

10. $(t^c)^*_s\,(\uparrow t) =_{def}$

$$\begin{cases} \uparrow (t^c)^*_u\,(t) \uparrow \text{ if } (t^c)^*_u\,(t) \text{ is defined} \\ \text{undefined otherwise,} \end{cases}$$

□

It is of importance that the variable $Self$ is considered separately from all other variables. $Self$ always refers to an instance that is of the current type. In the refined design pattern, the type of $Self$ is equal to the type of the refined design pattern. Furthermore, all other variables originally referring to instances of the source design pattern are still of that type since they do not call methods of the current instance. The variable $Self$ changes its type as the design pattern morphs. This concept can also be found in the object oriented programming language *Eiffel*. In *Eiffel* it is possible to assign the type **like current** to an attribute of a class. When a new class is inherited from this class, the actual type of that attribute changes to the type of the new class. However, this concept is in its general case as implemented in *Eiffel* not type-safe (cf. [10]). Only the $Self$ parameter can be of type **like current** since it always refers to the current instance.

All notions, terms and concepts that have been introduced so far represent in their entirety the *Pattern-Model*. As a final step, the next chapter will introduce the syntax and semantics of *PP* in order to come full circle.

Chapter 6

A design pattern oriented imperative kernel language

In this chapter, the syntax and semantics of *PP* will be introduced. As mentioned before, *PP* has to be considered as a prototype of a typed design pattern oriented imperative programming language.

In the preceeding sections, the *PatternModel* as a framework was introduced. While the *PatternModel* is a model with rather abstract notions and concepts, the language *PP* addresses the actual programming stage in software engineering. *PP* founds on the *PatternModel* making it possible to actually implement design patterns. It will become intelligible in this chapter that *PP* overcomes the problems and meets the requirements imposed on a design pattern oriented language mentioned in section 2.2.2.

6.1 The syntax of *PP*

In appendix A, the syntax of *PP* is defined in EBNF form. Using this form it is very difficult to understand the correleation between the syntax and the semantics of *PP*. Certain sytactic constructs considered together form an entity. If they are considered separately, however, they lose their context. In these cases it is quite hard to track the actual sense of that construct. Because of this reason, subsequent definitions whill rely on the following abstract syntax of *PP*.

Definition 6.1 *abstract syntax of* PP
In the sequel, the following notations for the data definition part of *PP* will be used.

(I) a design pattern definition is syntactically described by the following construct.

P = design pattern $\mathcal{P}$ is
refinements
$R_1, \ldots, R_k$
subclasses design patterns $\mathcal{D}_1, \ldots, \mathcal{D}_l$
uses design patterns $\mathcal{U}_1, \ldots, \mathcal{U}_n$
components
$C_1, \ldots, C_o$
attributes A
methods M
method implementations IMP
end design pattern

(II) a refine statement $R_i, i = 1, \ldots, k$ in (I) is of the following form.

$R_i =$ $\mathcal{R}_i$ refines $\mathcal{Q}$
refine $\mathcal{V}_1^r$ into $\mathcal{W}_1^r, \ldots, \mathcal{V}_{k^r}^r$ into $\mathcal{W}_{k^r}^r$
rename by $\hat{\sigma}_F^r$
select τ^r
end refinement

(III) a component definition $C_i, i = 1, \ldots, o$ in (I) is of the following form.

$C_i =$ component $\mathcal{C}_i$ is
recast $\mathcal{R}'_1$ $rec'_{\mathcal{R}'_1}, \ldots, \mathcal{R}'_{k'}$ $rec'_{\mathcal{R}'_{k'}}$
subclasses components $\mathcal{D}'_1, \ldots, \mathcal{D}'_{l'}$
uses components $\mathcal{U}'_1, \ldots, \mathcal{U}'_{n'}$
attributes A'
methods M'
method implementations IMP'
end component

(IV) a recast statement $rec'_{\mathcal{R}'_i}, i = 1, \ldots, k'$ in (III) is of the following form.

$rec'_{\mathcal{R}'_i} =$
rename by $\hat{\sigma}_F^c$
select τ^c
end recast

□

Additionally, it is assumed that in a given system of design pattern expressions the following holds. Most of these items follow common sense. Others are listed for convenience reasons.

Assumptions 6.2 *assumptions for the data definition language part of* PP

1. Typically, (meta-) variables $C, C_1, D, E, \ldots$ denote design patterns or components (i.e. expressions of type (I) or (III). Expressions of type (II) (refinements) will be denoted by $R, R_1, \ldots$. Identifiers for these expressions will be denoted by calligraphic letters $\mathcal{C}, \mathcal{C}_1, \mathcal{D}, \mathcal{E}, \mathcal{R}, \ldots$.

2. Given a system $\mathbb{P}$ of design pattern expressions, there is a one-to-one correspondence between design pattern identifiers and design pattern expressions denoted by $P =_{def} \mathbb{P}[\mathcal{P}]$ or $\mathbb{P}[\mathcal{P}]$ is undefined if there is no such design pattern expression. It is moreover assumed that a component identifier is unique within that particular design pattern it is defined in. Therefore, the association between a component identifier $\mathcal{C}$ and a component expression C also depends on a design pattern and is denoted by $C =_{def} \mathbb{P}^P[\mathcal{C}]$ or $\mathbb{P}^P[\mathcal{C}]$ is undefined if there is no such component expression. The same applies to refinements for a design pattern. The one-to-one correspondence between refinement identifiers of a design pattern and a refinemment expression is denoted by $R =_{def} \mathbb{P}^P[\mathcal{R}]$ or $\mathbb{P}^P[\mathcal{R}]$ is undefined if there is no such refinement expression.

3. *PP* allows both the modelling of objects and values. Values are modelled by a basic system of satisfiable class specifications $\mathbb{B}$ at least containing specifications for $Bool$ and Int with term generated semantics and non-empty carrier sets (cf. [5]).

4. Expressions of type <command> are directly treated as elements of the set $COMMAND$. Expressions $\mathcal{P}$ of type <design pattern id> will be treated as sorts denoted by $\mathcal{P}_S$. Expressions $\mathcal{C}$ of type <component id> which occur inside a design pattern expression are treated as elements of $SORT$ in the following way. Since sorts have to be unique throughout a signature regardless any context, the corresponding sort for an expression of type <component id> is obtained by combining the corresponding design pattern identifier $\mathcal{P}$ and the component identifier denoted by $\mathcal{C}_S^{\mathcal{P}} =_{def} \mathcal{P} :: \mathcal{C}$. The design pattern identifier is determined by the context of the <component id>. In some cases, identifiers can refer to both components and design pattern. In order to obtain a uniform framework, it is defined that $\mathcal{Q}_S^{\mathcal{P}} =_{def} \mathcal{Q}_S$ for some design pattern $\mathcal{Q}$.

5. Attributes in PP are identifier. They have to be transformed in order to treat them as operation symbols in an environment of signatures.

   (a) $X_F^{\mathcal{P}} =_{def} \mathcal{P}$ ::X: $\left(\mathcal{C}_S^{\mathcal{P}}\right) \to s_0'$ iff $X =$X: s_0 is a component attribute,

   (b) $X_F^{\mathcal{P}} =_{def} \mathcal{P}$ ::X: $(\mathcal{P}_S) \to \mathcal{E}_S^{\mathcal{P}}$ iff $X =$X: $\mathcal{E}$ is a design pattern attribute and $\mathcal{E}$ is a component identifier,

   (c) $X_F^{\mathcal{P}} =_{def}$X: $(\mathcal{P}_S) \to s_0'$ iff $X =$X: s_0 is a design pattern attribute and s_0 is not an component identifier.

   Then the set $A_F^{\mathcal{P}}$ is defined by

   $$A_F^{\mathcal{P}} =_{def} \left\{X_F^{\mathcal{P}} : X \in A\right\}.$$

6. A method declaration is part of the DDL of PP. Therefore, only identifiers (not sorts) are used for the definition of parameter- and the result types. However, the corresponding operation symbols in a signature use sorts instead of identifiers. Besides, the name of a method may not be unique throughout the whole design pattern system. In order to avoid confusions regarding this namespace problem, a <method> in PP defined inside a design pattern P will be treated in the sequel as elements of $OPNS$ in the following way.

   (a) $m_F^{\mathcal{P}} =_{def} \mathcal{P}$ ::m: $\left(\mathcal{C}_S^{\mathcal{P}}, s_1', \ldots, s_q'\right) \to s_0'$ iff $m =$m$(X_1 : s_1, \ldots, X_q : s_q)$ returns s_0 is a component method,

   (b) $m_F^{\mathcal{P}} =_{def} \mathcal{P}$ ::m: $\left(\mathcal{P}_S, s_1', \ldots, s_q'\right) \to s_0'$ iff $m =$m$(X_1 : s_1, \ldots, X_q : s_q)$ returns s_0 is a design pattern method and contains component identifiers,

   (c) $m_F^{\mathcal{P}} =_{def}$m: $\left(\mathcal{P}_S, s_1', \ldots, s_q'\right) \to s_0'$ iff $m =$m$(X_1 : s_1, \ldots, X_q : s_q)$ returns s_0 is a design pattern method and does not contain any component identifiers.

   where $s_i' =_{def} \mathcal{E}_S^{\mathcal{P}}$ if $s_i = \mathcal{E}$ is a component identifier and otherwise $s_i' =_{def} s_i$ for $i = 0, \ldots, n$. Then the set $M_F^{\mathcal{P}}$ is defined by

   $$M_F^{\mathcal{P}} =_{def} \left\{m_F^{\mathcal{P}} : m \in M\right\}.$$

7. Every method defined in a supercomponent of a component is implicitly reimplemented in that component using the method's original implementation if it is not overridden by the developer. In this way, all methods of a supercomponent are overridden and can therefore be treated homogeneously. The same applies to design pattern methods in super- and sub- design pattern.

8. For the corresponding expressions, the following properties are assumed to hold.

   (I) $\mathcal{D}_i$ is a design pattern identifier and $\mathbb{P}\left[\mathcal{D}_i\right]$ must be defined for $i = 1, \ldots, l$. $\mathcal{U}_i$ is a design pattern identifier and $\mathbb{P}\left[\mathcal{U}_i\right]$ must be defined for $i = 1, \ldots, n$.

(II) $\mathcal{Q}$ is a design pattern identifier and $\mathbb{P}[\mathcal{Q}]$ must be defined, $\mathcal{V}_i^r$ and $\mathcal{W}_i^r$ are component identifiers where $\mathbb{P}^{\mathcal{Q}}[\mathcal{V}_i^r]$ must be defined for $i = 1, \ldots, k^r$. Therefore these identifier will be considered in the context of Q. The rename statement renames only internal methods and attributes in Q. The select statement selects only (renamed) methods of Q.

(III) ${\mathcal{R}'}_j$ is a refinement identifier and $R'_j = \mathbb{P}^P[{\mathcal{R}'}_j]$ must be defined for $j = 1, \ldots, k'$ where P is the design pattern expression this component belongs to.
Moreover, for every $\mathcal{D}' \in \{\mathcal{D}'_1, \ldots, \mathcal{D}'_{l'}\}$, $\mathbb{P}^P[\mathcal{D}']$ must be defined. For every $\mathcal{U}' \in \{\mathcal{U}'_1, \ldots, \mathcal{U}'_{n'}\}$, $\mathbb{P}^P[\mathcal{U}']$ must be defined.

(IV) Let R' be the corresponding type (II) refinement expression of $\mathcal{R}'$ and let V be the component in Q which this component has been refined to. The rename statement renames only methods and attributes in V. The select statement selects only (renamed) methods of V.

□

6.2 The semantics of *PP*

The definition of the semantics of *PP* is divided into two separate parts. In a given a system of design pattern expressions, there are design pattern expressions that are written from scratch (flat design patterns expressions) and design pattern expressions that refine from other design pattern expressions. In a first step, all refinements of a non-flat design pattern expressions have to be eliminated. The result of this process is a flat design pattern expression. In this way, refinements are handled on the level of *PP* since the the flattening yields valid design pattern expressions.

Definition 6.3 *The flattening of the refinement structure in* PP *(semantics of* PP *part I).*
The flattening of a design pattern is described by a partial function

$$F^* : \langle\text{design pattern sys}\rangle \to \langle\text{design pattern}\rangle \to \langle\text{design pattern}\rangle$$

which transforms the specified design pattern to a design pattern which is flat in the refinement structure, i.e. the resulting design pattern expression does neither contain any refinement expressions nor component expressions inside the specified design pattern which contain recast statements.

In the first place, the following auxiliary functions are introduced.

1. There are three sorts of signatures Sig. The first one composes the signature of a component expression within a given design pattern system $\mathbb{P}$ and a particular design pattern expression P'. The second family of signatures Sig composes the signature of the higher behaviour of a design pattern expression P' within a given a given design pattern system $\mathbb{P}$. The third class of signatures Sig comprises the first two cases. It represents the total signature of a design pattern expression.
First, P' has to be transformed into a design pattern expression that is flat in its refinement structure, i.e. there are no refinement expressions in P. This can be achieved by the definition $P =_{def} F^*_{\mathbb{P}}[P']$.
In the following definitions all identifiers refer to expressions in the flattened design pattern expression P.

(a) The signature of a component expression is defined by

$$
\begin{array}{lll}
Sig_{\mathbb{P}}^{P'}[C] & =_{def} & \\
Sig_{\mathbb{P}}^{P}[C] & =_{def} & \left(Sig(Gen(\mathbb{B})) + \sum_{i=1}^{l} \overline{Sig}_{\mathbb{P}}[\mathbb{P}[\mathcal{D}_i]] + \sum_{i=1}^{n} \overline{Sig}_{\mathbb{P}}[\mathbb{P}[\mathcal{U}_i]] + \right. \\
 & & \left. \sum_{i=1}^{l'} Sig_{\mathbb{P}}^{P}\left[\mathbb{P}^{P}[\mathcal{D}'_i]\right] + \sum_{i=1}^{n'} Sig_{\mathbb{P}}^{P}\left[\mathbb{P}^{P}[\mathcal{U}'_i]\right] \right) \oplus \\
 & & \left(\begin{array}{l} \mathcal{C}_S^{\mathcal{P}}, \\ \mathcal{C}_S^{\mathcal{P}} < \left\{ (\mathcal{D}'_1)_S^{\mathcal{P}}, \ldots, (\mathcal{D}'_{l'})_S^{\mathcal{P}} \right\}, \\ \mathcal{P}_S \Vdash \mathcal{C}_S^{\mathcal{P}} \Vdash \emptyset, \\ (M')_F^{\mathcal{P}} \cup (A')_F^{\mathcal{P}} \end{array} \right).
\end{array}
$$

(b) The signature of the higher behaviour of a design pattern expression is defined by

$$
\begin{array}{lll}
Sig_{\mathbb{P}}[P'] & =_{def} & \\
Sig_{\mathbb{P}}[P] & =_{def} & \left(Sig(Gen(\mathbb{B})) + \sum_{i=1}^{l} Sig_{\mathbb{P}}[\mathbb{P}[\mathcal{D}_i]] + \sum_{i=1}^{n} Sig_{\mathbb{P}}[\mathbb{P}[\mathcal{U}_i]] \right) \oplus \\
 & & \left(\begin{array}{l} \mathcal{P}_S, \\ \mathcal{P}_S < \{(\mathcal{D}_1)_S, \ldots, (\mathcal{D}_l)_S\}, \\ \perp \Vdash \mathcal{P}_S \Vdash \left\{ (\mathcal{C}_1)_S^{\mathcal{P}}, \ldots, (\mathcal{C}_o)_S^{\mathcal{P}} \right\}, \\ M_F^{\mathcal{P}} \cup A_F^{\mathcal{P}} \end{array} \right).
\end{array}
$$

(c) The total signature of a component expression is defined by

$$
\begin{array}{lll}
\overline{Sig}_{\mathbb{P}}^{P'}[C] & =_{def} & \\
\overline{Sig}_{\mathbb{P}}^{P}[C] & =_{def} & Sig_{\mathbb{P}}^{P}[C] + Sig_{\mathbb{P}}[P].
\end{array}
$$

(d) The total signature of a design pattern expression is defined by

$$
\begin{array}{lll}
\overline{Sig}_{\mathbb{P}}[P'] & =_{def} & \\
\overline{Sig}_{\mathbb{P}}[P] & =_{def} & \sum_{i=1}^{o} \overline{Sig}_{\mathbb{P}}^{P}[C_i].
\end{array}
$$

2. A signature morphism specified by a refinement R inside a design pattern expression P is induced in the following way. $\sigma_{\mathbb{P}}^{P}[R]$ morphs the sorts and operation symbols of a design pattern specified by a refinement R into the elements of the current design pattern P. It is defined as

$$\sigma_{\mathbb{P}}^{P}[R] : \overline{Sig}_{\mathbb{P}}[\mathbb{P}[\mathcal{Q}]] \rightarrow \overline{Sig}_{\mathbb{P}}[P] \text{ refining sort } \mathcal{Q}_S$$

with $\sigma_{\mathbb{P}}^{P}[R] =_{def} \left(\left(\sigma_{\mathbb{P}}^{P}[R]\right)_S, \left(\sigma_{\mathbb{P}}^{P}[R]\right)_F \right)$ by

$$
\left(\sigma_{\mathbb{P}}^{P}[R]\right)_S(s) =_{def} \left\{ \begin{array}{ll}
(\mathcal{W}_i^r)_S^{\mathcal{P}} & \text{if } s = (\mathcal{V}_i^r)_S^{\mathcal{Q}}, i = 1, \ldots, k^r \\
\mathcal{P}_S & \text{if } s = \mathcal{Q}_S \\
s & \text{if } s \neq (\mathcal{W}_i^r)_S^{\mathcal{P}}, i = 1, \ldots, k^r, s \neq \mathcal{Q}_S \\
\text{undefined otherwise.} &
\end{array} \right.
$$

Furthermore, let $op = f : (s_1, \ldots s_n) \to s_0$.

$$\left(\sigma^P_{\mathbb{P}}[R]\right)_F(op) =_{def} \begin{cases} \hat{\sigma}_F[f](s'_1, \ldots, s'_n) \to s'_0 & \text{if } \hat{\sigma}_F \text{ is defined, } f \in dom(\hat{\sigma}_F) \\ & \text{and all } \left(\sigma^P_{\mathbb{P}}[R]\right)_S(s_i) \text{ are defined} \\ f(s'_1, \ldots, s'_n) \to s'_0 & \text{if } \hat{\sigma}_F \text{ is defined, } f \notin dom(\hat{\sigma}_F) \cup im(\hat{\sigma}_F) \\ & \text{and all } \left(\sigma^P_{\mathbb{P}}[R]\right)_S(s_i) \text{ are defined} \\ \text{undefined otherwise,} & \end{cases}$$

where $\forall i \in \{0, \ldots, n\}$

$$s'_i =_{def} \begin{cases} \left(\sigma^P_{\mathbb{P}}[R]\right)_S(s_i) & \text{if } s_i = (\mathcal{C})^{\mathcal{Q}}_S, \mathcal{C} \text{ is a component identifier in } \mathbb{P}[\mathcal{Q}] \\ & \text{and } \left(\sigma^P_{\mathbb{P}}[R]\right)_S(s_i) \text{ is defined,} \\ & \text{if } s_i = \mathcal{Q}_S, op = m^{\mathcal{Q}}_F, m \text{ is a design pattern method in } \mathbb{P}[\mathcal{Q}], \\ & i = 1 \text{ and } \left(\sigma^P_{\mathbb{P}}[R]\right)_S(s_i) \text{ is defined,} \\ s_i & \text{if the above case is not true and } \left(\sigma^P_{\mathbb{P}}[R]\right)_S(s_i) \text{ is defined} \\ \text{undefined otherwise,} & \end{cases}$$

and

$$\hat{\sigma}_F =_{def} \hat{\sigma}^r_F \sqcup \bigsqcup_{C \in \mathbb{C}} \hat{\sigma}^C_F[rec'_{\mathcal{R}}],$$

$\mathbb{C}$ is the set of components defined in P which contain recast expressions for the refinement referenced by $\mathcal{R}$ and

$$\hat{\sigma}^C_F[rec'_{\mathcal{R}}] =_{def} \left\{ \left(m^{\mathcal{Q}}_F, (m')^{\mathcal{P}}_F \right) : (m, m') \in \hat{\sigma}^c_F \right\}.$$

3. Based on $\sigma^P_{\mathbb{P}}[R]$ a morphism $\iota^P_{\mathbb{P}}[R] = \left(\left(\iota^P_{\mathbb{P}}[R]\right)_I, \left(\iota^P_{\mathbb{P}}[R]\right)_A, \left(\iota^P_{\mathbb{P}}[R]\right)_M, \left(\iota^P_{\mathbb{P}}[R]\right)_{IMP}\right)$ is introduced. It is used to morph identifiers, methods and method implementations in PP. It is defined by

$$\begin{aligned} \left(\iota^P_{\mathbb{P}}[R]\right)_I(\mathcal{E}) &=_{def} \begin{cases} \mathcal{E}' & \text{if } (\mathcal{E}')^{\mathcal{P}}_S = \left(\sigma^P_{\mathbb{P}}[R]\right)_S\left(\mathcal{E}^{\mathcal{Q}}_S\right) \text{ is defined} \\ \text{undefined otherwise,} & \end{cases} \\ \left(\iota^P_{\mathbb{P}}[R]\right)_A(X) &=_{def} \begin{cases} X' & \text{if } (X')^{\mathcal{P}}_S = \left(\sigma^P_{\mathbb{P}}[R]\right)_F\left(X^{\mathcal{Q}}_F\right) \text{ is defined} \\ \text{undefined otherwise,} & \end{cases} \\ \left(\iota^P_{\mathbb{P}}[R]\right)_M(m) &=_{def} \begin{cases} m' & \text{if } (m')^{\mathcal{P}}_S = \left(\sigma^P_{\mathbb{P}}[R]\right)_F\left(m^{\mathcal{Q}}_F\right) \text{ is defined} \\ \text{undefined otherwise.} & \end{cases} \end{aligned}$$

For the morphing of method implementation it is necessary to distinguish between a component method implementation and a design pattern method implementation because the corresponding implementations have to be translated to the correct context. Let imp be the method implementation of a component method or a design pattern method m of the form $imp = \mathsf{m}(X_1 : s_1, \ldots, X_n : s_n)$ returns s is com end. Then $\left(\iota^P_{\mathbb{P}}[R]\right)_{IMP}$ is defined by

$$\left(\iota^P_{\mathbb{P}}[R]\right)_{IMP}(imp) =_{def} \begin{cases} m' \text{ is } t^c_{s'}(com) \text{ end if } m' = \left(\iota^P_{\mathbb{P}}[R]\right)_F(m) \text{ is defined} \\ \text{undefined otherwise,} \end{cases}$$

where

$$c =_{def} \begin{cases} \mathcal{E}^{\mathcal{Q}}_S & \text{if } m \text{ is a component method in component } E \text{ in } Q \\ \mathcal{Q}_S & \text{if } m \text{ is a design pattern method,} \end{cases}$$

$t^c_{s'}$ is the translation of commands via $\sigma^P_{\mathbb{P}}[R]$ and $s' =_{def} \mathcal{E}^{\mathcal{P}}_S$ if $s = \mathcal{E}$ is a component identifier and otherwise $s' =_{def} s$,

4. Based on $\sigma_{\mathbb{P}}^{P}[R]$ a component morphism $\delta_{\mathbb{P}}^{P}[R]$ is defined as follows[1].

$$\delta_{\mathbb{P}}^{P}[R] : Comp(DP_{\mathbb{P}}[Q]) \to Comp(DP_{\mathbb{P}}[P])$$
$$\left(\delta_{\mathbb{P}}^{P}[R]\right)^{\Gamma}(sp) =_{def} \begin{cases} M_{\mathbb{P}}^{P}\left[\left(\iota_{\mathbb{P}}^{P}[R]\right)_{I}(C)\right] & \text{if } sp = M_{\mathbb{P}}^{Q}[C] \\ M_{\mathbb{P}}[P] & \text{if } sp = M_{\mathbb{P}}[Q]. \end{cases}$$

with

$$\left(\delta_{\mathbb{P}}^{P}[R]\right)_{sp}^{\Sigma} =_{def} \underline{\sigma_{\mathbb{P}}^{P}[R]}_{Sig(sp)}{}^{2}$$

and

$$\left(\delta_{\mathbb{P}}^{P}[R]\right)_{sp}^{\Phi} =_{def} \left\{ \left(E_{\left|\left(\delta_{\mathbb{P}}^{P}[R]\right)_{sp}^{\Sigma}\right.}, D \right) : \begin{array}{l} E \in Mod\left(\left(\delta_{\mathbb{P}}^{P}[R]\right)^{\Gamma}(sp)\right), D \in Mod(sp), \\ E_{|Sig(Gen(\mathbb{B}))} = D_{|Sig(Gen(\mathbb{B}))} \end{array} \right\}$$

Let P be a design pattern expression (type I) within the design pattern system $\mathbb{P}$. Then the function F^* is defined as k consecutive applications of F. F^* is defined by

$$F_{\mathbb{P}}^{*}[P] =_{def} \underbrace{F_{\mathbb{P}}[\ldots F_{\mathbb{P}}[P]\ldots]}_{k \text{ times}}$$

F itself eliminates the first refinement expression R_1 in P and all recast statements in components in P that refer to R_1. In order to distinguish between identifiers of the specified design pattern, the flattened design pattern $Q =_{def} F^*[\mathbb{P}[\mathcal{Q}]]$ that is specified by R_1 and the new, refined design pattern $F_{\mathbb{P}}[P]$, the following name convention is used. Identifiers and variables in P appear normally as proposed in notation 6.1, in Q they appear with a hat (i.e. like $\hat{\mathcal{C}}$). Other identifiers and variables in the refined design pattern appear with a check mark (i.e. like $\check{\mathcal{C}}$). F is defined by

$F_{\mathbb{P}}[$design pattern $\mathcal{P}$ is
refinements
$R_2, \ldots, R_k$
subclasses design patterns $\mathcal{D}_1, \ldots, \mathcal{D}_l$, $\mathcal{Q}$
uses design patterns $\mathcal{U}_1, \ldots, \mathcal{U}_n$
components
$\check{C}_1, \ldots, \check{C}_{\check{o}}$
attributes $A \cup \check{A}$
methods $M \cup \check{M}$
method implementations $IMP \cup I\check{M}P$
end design pattern

The components $\check{C}_1, \ldots, \check{C}_{\check{o}}$ have to be completly rewritten. They are defined in the following way.

1. $\check{C}_i, i = 1, \ldots, \hat{o}$ represents the refined component of $\hat{C}_i$ via $\check{\mathcal{C}}_i =_{def} \left(\iota_{\mathbb{P}}^{P}[R_1]\right)_{I}\left(\hat{\mathcal{C}}_i\right)$ under consideration of the component $\mathbb{P}^{P}\left[\check{\mathcal{C}}_i\right]$, if defined. All identifiers and variables which occur inside a component expression according to 6.1 refer to this component.

[1] The functions which are referred to are part of semantics (II) (cf. definition 6.4). However it is necessary to introduce δ at this point since it logically belongs to the process of the flattening of a design pattern.

[2] If σ is a signature morphism defined on Σ, then the corresponding signature morphism on $\underline{\Sigma}$ is denoted by $\underline{\sigma}$.

$\check{C}_i =_{def}$ component $\check{\mathcal{C}}_i$ is
recast $\mathcal{R}'_2\ rec'_{\mathcal{R}'_2}, \ldots, \mathcal{R}'_{k'}\ rec'_{\mathcal{R}'_{k'}}$
subclasses components $\mathcal{D}'_1, \ldots, \mathcal{D}'_{l'}, \hat{\mathcal{D}}'_1, \ldots, \hat{\mathcal{D}}'_{\hat{l}'}$
uses components $\mathcal{U}'_1, \ldots, \mathcal{U}'_{n'}, \hat{\mathcal{U}}'_1, \ldots, \hat{\mathcal{U}}'_{\hat{n}'}$
attributes $A' \cup \check{A}'$
methods $M' \cup \check{M}'$
method implementations $IMP \cup I\check{M}P'$
end component

W.l.o.g. it is assumed that $\mathcal{R}'_1 = \mathcal{R}_1$ for each component. Then, the sets $\check{A}', \check{M}'$ and $I\check{M}P'$ are defined by

$$\begin{array}{lll}
\check{A}' & =_{def} & \left\{ \left(\iota^P_{\mathbb{P}}\left[R_1\right]\right)_M \left(\hat{X}\right) : \hat{X} \in \hat{A}' \right\} \\
\check{M}' & =_{def} & \left\{ \left(\iota^P_{\mathbb{P}}\left[R_1\right]\right)_M (\hat{m}) : \hat{m} \in \hat{M}' \right\} \cap \tau^c, \\
I\check{M}P' & =_{def} & \{ i\check{m}p : \quad i\check{m}p =_{def} \left(\iota^P_{\mathbb{P}}\left[R_1\right]\right)_{IMP} \left(i\hat{m}p\right), i\hat{m}p \in I\hat{M}P', \\
 & & \qquad i\check{m}p \text{ implements a component method } m \text{ and } m \in \check{M}' \}.
\end{array}$$

2. $\check{C}_i, i = \hat{o}+1, \ldots, \check{o}$ represents a newly introduced component $C_j, j \in \{1, \ldots, o\}$ with respect to R_1 in P, i.e. $\mathcal{C}_j \neq \mathcal{W}^r_h, h = 1, \ldots, k^r$. In this case, $\check{C}_i =_{def} C_j$.

The sets $\check{A}, \check{M}$ and $I\check{M}P$ are defined by

$$\begin{array}{lll}
\check{A} & =_{def} & \left\{ \left(\iota^P_{\mathbb{P}}\left[R_1\right]\right)_M \left(\hat{X}\right) : \hat{X} \in \hat{A} \text{ is internal} \right\} \\
\check{M} & =_{def} & \left\{ \left(\iota^P_{\mathbb{P}}\left[R_1\right]\right)_M (\hat{m}) : \hat{m} \in \hat{M} \right\} \cap \tau^r, \\
I\check{M}P & =_{def} & \{ i\check{m}p : \quad i\check{m}p =_{def} \left(\iota^P_{\mathbb{P}}\left[R_1\right]\right)_{IMP} \left(i\hat{m}p\right), i\hat{m}p \in I\hat{M}P, \\
 & & \qquad i\check{m}p \text{ implements a design pattern method } m \text{ and } m \in \check{M} \}.
\end{array}$$

□

In a second step, the semantics of a system of flat design pattern expressions can be described.

Definition 6.4 *The semantics of* PP *(part II).*
The semantics of a component defined inside a design pattern is described by a partial function

$$M : \langle \text{design pattern sys} \rangle \to \langle \text{design pattern} \rangle \to \langle \text{component} \rangle \to SPEC.$$

The semantics of the higher behaviour of a design pattern is described by a partial function

$$M : \langle \text{design pattern sys} \rangle \to \langle \text{design pattern} \rangle \to SPEC.$$

The semantics of a design pattern in $PATTSPEC$ is described by a partial function

$$DP : \langle \text{design pattern sys} \rangle \to \langle \text{design pattern} \rangle \to PATTSPEC.$$

The semantics of a design pattern (system) in $SPEC$ is described by a partial function

$$\overline{M} : \langle \text{design pattern sys} \rangle \to \langle \wp_{fin}\left(PATTSPEC\right) \rangle \to SPEC.$$

In the first place, another class of signatures $AttrSig$ is defined.

There are three sorts of signatures $AttrSig$. The first one composes the attribute signature of a component expression within a given design pattern system $\mathbb{P}$ and a particular design pattern expression P'. The second family of signatures $AttrSig$ composes the attribute signature of the higher behaviour of a design pattern expression P' within a given a given design pattern system $\mathbb{P}$. The third class of signatures Sig comprises the first two cases. It represents the total attribute signature of a design pattern expression.

First, P' has to be transformed into a design pattern expression that is flat in its refinement structure, i.e. there are no refinement expressions in P. This can be achieved by the definition $P =_{def} F^*_{\mathbb{P}}[P']$. In the following definitions all identifiers refer to expressions in the flattened design pattern expression P.

1. The attribute signature of a component expression is defined by

$$
\begin{array}{lll}
AttrSig^{P'}_{\mathbb{P}}[C] & =_{def} & \\
AttrSig^{P}_{\mathbb{P}}[C] & =_{def} & \left(AttrSig(Gen(\mathbb{B})) + \sum_{i=1}^{l} \overline{AttrSig}_{\mathbb{P}}[\mathbb{P}[\mathcal{D}_i]] + \sum_{i=1}^{n} \overline{AttrSig}_{\mathbb{P}}[\mathbb{P}[\mathcal{U}_i]] + \right. \\
 & & \left. \sum_{i=1}^{l'} AttrSig^{P}_{\mathbb{P}}\left[\mathbb{P}^P[\mathcal{D}'_i]\right] + \sum_{i=1}^{n'} AttrSig^{P}_{\mathbb{P}}\left[\mathbb{P}^P[\mathcal{U}'_i]\right] \right) \oplus \\
 & & \left(\begin{array}{l} \mathcal{C}^{\mathcal{P}}_S, \\ \mathcal{C}^{\mathcal{P}}_S < \left\{ (\mathcal{D}'_1)^{\mathcal{P}}_S, \ldots, (\mathcal{D}'_{l'})^{\mathcal{P}}_S \right\}, \\ \mathcal{P}_S \Vdash \mathcal{C}^{\mathcal{P}}_S \Vdash \emptyset, \\ (A')^{\mathcal{P}}_F \end{array} \right).
\end{array}
$$

2. The signature of the higher behaviour of a design pattern expression is defined by

$$
\begin{array}{lll}
AttrSig_{\mathbb{P}}[P'] & =_{def} & \\
AttrSig_{\mathbb{P}}[P] & =_{def} & \left(AttrSig(Gen(\mathbb{B})) + \sum_{i=1}^{l} AttrSig_{\mathbb{P}}[\mathbb{P}[\mathcal{D}_i]] + \sum_{i=1}^{n} AttrSig_{\mathbb{P}}[\mathbb{P}[\mathcal{U}_i]] \right) \oplus \\
 & & \left(\begin{array}{l} \mathcal{P}_S, \\ \mathcal{P}_S < \{(\mathcal{D}_1)_S, \ldots, (\mathcal{D}_l)_S\}, \\ \bot \Vdash \mathcal{P}_S \Vdash \left\{ (\mathcal{C}_1)^{\mathcal{P}}_S, \ldots, (\mathcal{C}_o)^{\mathcal{P}}_S \right\}, \\ A^{\mathcal{P}}_F \end{array} \right).
\end{array}
$$

3. The total signature of a component expression is defined by

$$
\begin{array}{lll}
\overline{AttrSig}^{P'}_{\mathbb{P}}[C] & =_{def} & \\
\overline{AttrSig}^{P}_{\mathbb{P}}[C] & =_{def} & AttrSig^{P}_{\mathbb{P}}[C] + AttrSig_{\mathbb{P}}[P].
\end{array}
$$

4. The total signature of a design pattern expression is defined by

$$
\begin{array}{lll}
\overline{AttrSig}_{\mathbb{P}}[P'] & =_{def} & \\
\overline{AttrSig}_{\mathbb{P}}[P] & =_{def} & \sum_{i=1}^{o} \overline{AttrSig}^{P}_{\mathbb{P}}[C_i].
\end{array}
$$

If all $Sig[C]$ and all $Sig[C]$ are defined then the functions M are defined as follows. Again, the definition of the functions M relies on the fact that the design pattern expression P is flat in its refinement structure. Therefore an arbitrary design pattern expression P' has to be flattened into P using the above definition $P =_{def} F^*_{\mathbb{P}}[P']$. In the following definitions all identifiers refer to expressions in the flattened design pattern expression P.

1. The semantics of a component expression is defined by

$$
\begin{array}{ll}
M_{\mathbb{P}}^{P'}[C] & =_{def} \\
M_{\mathbb{P}}^{P}[C] & =_{def}
\end{array}
\left(
\begin{array}{l}
Gen[\mathbb{B}] + \overline{M}_{\mathbb{P}}[D_1,\ldots,D_l] + \overline{M}_{\mathbb{P}}[U_1,\ldots,U_m] + \\
M_{\mathbb{P}}^{P}[D_1'] + \ldots + M_{\mathbb{P}}^{P}[D_{l'}'] + M_{\mathbb{P}}^{P}[U_1'] + \ldots + M_{\mathbb{P}}^{P}[U_{m'}'] + \\
M_{\mathbb{P}}[P] + \\
\langle \quad \underline{\overline{Sig}_{\mathbb{P}}^{P}[C]}, \\
\quad \left\{ A \in Alg\left(\underline{\overline{Sig}_{\mathbb{P}}^{P}[C]}\right) : A \text{ is a } \underline{\overline{Sig}_{\mathbb{P}}^{P}[C]}\text{-object algebra with} \right. \\
\quad \text{basic type signature } Sig(Gen[\mathbb{B}]) \text{ and an attribute} \\
\quad \text{signature } \underline{\overline{AttrSig}_{\mathbb{P}}^{P}[C]} \text{ and } A \text{ satisfies} \\
\quad \left. \text{the component method implementations } IMP' \text{ in } C \right\} \rangle
\end{array}
\right)\Big|_{\underline{Sig_{\mathbb{P}}^{P}[C]}}
$$

2. The semantics of the higher behaviour of a design pattern expression is defined by

$$
\begin{array}{ll}
M_{\mathbb{P}}[P'] & =_{def} \\
M_{\mathbb{P}}[P] & =_{def}
\end{array}
\left(
\begin{array}{l}
Gen[\mathbb{B}] + \overline{M}_{\mathbb{P}}[D_1,\ldots,D_l] + \overline{M}_{\mathbb{P}}[U_1,\ldots,U_m] + \\
M_{\mathbb{P}}^{P}[C_1] + \ldots + M_{\mathbb{P}}^{P}[C_o] + \\
\langle \quad \underline{\overline{Sig}_{\mathbb{P}}[P]}, \\
\quad \left\{ A \in Alg\left(\underline{\overline{Sig}_{\mathbb{P}}[P]}\right) : A \text{ is a } \underline{\overline{Sig}_{\mathbb{P}}[P]}\text{-object algebra with} \right. \\
\quad \text{basic type signature } Sig(Gen[\mathbb{B}]) \text{ and an attribute} \\
\quad \text{signature } \underline{\overline{AttrSig}_{\mathbb{P}}[P]} \text{ and } A \text{ satisfies the} \\
\quad \left. \text{design pattern method implementations } IMP \text{ in } P \right\} \rangle
\end{array}
\right)\Big|_{\underline{Sig_{\mathbb{P}}[P]}}
$$

By the above definitions it can easily be seen that $M_{\mathbb{P}}[P]$ is dependent on $M_{\mathbb{P}}^{P}[C_i], i = 1,\ldots,o$ and in turn every such $M_{\mathbb{P}}^{P}[C_i]$ is dependent on $M_{\mathbb{P}}[P]$ for a design pattern expression P. Therefore, these two definitions have to be considered as a system of equations. Its solution results in the semantics of the design pattern expression P as well as in the semantics of all component expressions C_i.

In order to solve this system of equations, it is necessary to find the greatest fixed point of the system starting from the following conditions[3]:

$$
\begin{array}{lll}
\left(M_{\mathbb{P}}^{P}[C]\right)_0 & =_{def} & \left\langle \underline{Sig_{\mathbb{P}}^{P}[C]}, Alg\left(\underline{Sig_{\mathbb{P}}^{P}[C]}\right)\right\rangle, \\
\left(M_{\mathbb{P}}[P]\right)_0 & =_{def} & \left\langle \underline{Sig_{\mathbb{P}}[P]}, Alg\left(\underline{Sig_{\mathbb{P}}[P]}\right)\right\rangle.
\end{array}
$$

The semantics of a design pattern expression in $PATTSPEC$ is defined by

$$
\begin{array}{lll}
DP_{\mathbb{P}}[P'] & =_{def} & \\
DP_{\mathbb{P}}[P] & =_{def} & \left\langle \left\{ M_{\mathbb{P}}[P], M_{\mathbb{P}}^{P}[C_1], \ldots, M_{\mathbb{P}}^{P}[C_o] \right\}, M_{\mathbb{P}}[P] \right\rangle.
\end{array}
$$

The semantics of a design pattern expression (system) in $SPEC$ is defined by

$$
\overline{M}_{\mathbb{P}}[P_1,\ldots,P_n] \quad =_{def} \quad translate_{P\to S}\left(DP_{\mathbb{P}}[P_1]\right) + \ldots + translate_{P\to S}\left(DP_{\mathbb{P}}[P_n]\right).
$$

□

[3]For this purpose, a partial ordering $\succeq$ on $SPEC$ is introduced by $sp \succeq sp'$ iff $Sig(sp) = Sig(sp')$ and $Mod(sp) \supseteq Mod(sp')$.

By induction can be shown that starting from the conditions above, the number of models is monotonously decreasing. The basic idea of the proof is as follows.

The specifications $M_{\mathbb{P}}^{P}[C]$ and $M_{\mathbb{P}}[P]$ are iteratively computed by the following calculus.

$$\begin{array}{llll}
\left(M_{\mathbb{P}}^{P}[C]\right)_0 & = \left\langle \underline{Sig_{\mathbb{P}}^{P}[C]}, Alg\left(\underline{Sig_{\mathbb{P}}^{P}[C]}\right)\right\rangle & \left(M_{\mathbb{P}}[P]\right)_0 & = \left\langle \underline{Sig_{\mathbb{P}}[P]}, Alg\left(\underline{Sig_{\mathbb{P}}[P]}\right)\right\rangle. \\
\left(M_{\mathbb{P}}^{P}[C]\right)_1 & = \left(\ldots + \left(M_{\mathbb{P}}[P]\right)_0 + \ldots\right)_{\left|\underline{Sig_{\mathbb{P}}^{P}[C]}\right.} & \left(M_{\mathbb{P}}[P]\right)_1 & = \left(\ldots + \left(M_{\mathbb{P}}^{P}[C]\right)_0 + \ldots\right)_{\left|\underline{Sig_{\mathbb{P}}[P]}\right.}{}^{4} \\
\vdots & \vdots & \vdots & \vdots \\
\left(M_{\mathbb{P}}^{P}[C]\right)_n & = \left(\ldots + \left(M_{\mathbb{P}}[P]\right)_{n-1} + \ldots\right)_{\left|\underline{Sig_{\mathbb{P}}^{P}[C]}\right.} & \left(M_{\mathbb{P}}[P]\right)_n & = \left(\ldots + \left(M_{\mathbb{P}}^{P}[C]\right)_{n-1} + \ldots\right)_{\left|\underline{Sig_{\mathbb{P}}[P]}\right.}
\end{array}$$

It immediately follows that $\left(M_{\mathbb{P}}^{P}[C]\right)_0 \succeq \left(M_{\mathbb{P}}^{P}[C]\right)_1$ and $\left(M_{\mathbb{P}}[P]\right)_0 \succeq \left(M_{\mathbb{P}}[P]\right)_1$. Based on the assumption $\left(M_{\mathbb{P}}[P]\right)_{n-2} \succeq \left(M_{\mathbb{P}}[P]\right)_{n-1}$ and the calculus above can be shown that $\left(M_{\mathbb{P}}^{P}[C]\right)_{n-1} \succeq \left(M_{\mathbb{P}}^{P}[C]\right)_n$. It can also be shown that based on the assumption $\left(M_{\mathbb{P}}^{P}[C]\right)_{n-2} \succeq \left(M_{\mathbb{P}}^{P}[C]\right)_{n-1}$ and the calculus above follows that $\left(M_{\mathbb{P}}[P]\right)_{n-1} \succeq \left(M_{\mathbb{P}}[P]\right)_n$. By induction follows that the given calculus is monotonous on $\succeq$.

Therefore, the worst case would lead to an unsatisfiable specification. This implies that a greatest fixed point exists and that the given problem converges into that fixed point.

6.3 A deduction system for components and design patterns in *PP*

Based on the definition of the semantics of *PP*, it is now possible to deduce relations syntactically. To this end, it has to be shown first that syntactically intended relationships also hold semantically.

Fact 6.5 Let $\mathbb{P}$ be a given system of design pattern expressions. Then, the following properties hold for a arbitrary design pattern expression P and a component expression C contained in P.

1. $DP_{\mathbb{P}}[P]$ is a design pattern specification,
2. Relations on component level:
   (a) $M_{\mathbb{P}}^{P}[C] \ll M_{\mathbb{P}}^{P}[D_i']$ for all $i = 1, \ldots, l'$,
   (b) $M_{\mathbb{P}}^{P}[U_i] \longrightarrow M_{\mathbb{P}}^{P}[C]$ for all $i = 1, \ldots, n'$.
3. Relations on design pattern level:
   (a) $\overline{M}_{\mathbb{P}}[P] \ll \overline{M}_{\mathbb{P}}[D_i]$ for all $i = 1, \ldots, l$,
   (b) $\overline{M}_{\mathbb{P}}[P] \ll \overline{M}_{\mathbb{P}}[Q]$ where Q is the design pattern expression referenced in R_i for all $i = 1, \ldots, k$,
   (c) $\overline{M}_{\mathbb{P}}[U_i] \longrightarrow \overline{M}_{\mathbb{P}}[P]$ for all $i = 1, \ldots, n$.

Proof follows immediately by the definition of the semantics of *PP*.

□

The following definition introduces a deduction system on the level of components expressions. It enables one to deduce semantic relations based on the syntax of *PP*.

[4] The remaining components are abbreviated by ... since they do not change througout the whole process.

Definition 6.6 *a deduction system for component expressions*
Let $\mathbb{P}$ be a given system of design pattern expressions. Then, a deduction system for component expressions can be defined as follows.

(SC-C) For all design pattern expressions P containing a component expression C

1. $\vdash^P_{\mathbb{P}} C \ll D'_i$ for all $i = 1, \ldots, l'$,
2. $\vdash^P_{\mathbb{P}} U'_i \longrightarrow C$ for all $i = 1, \ldots, n'$,

(Ref-C) for all component expressions E in P

1. $\vdash^P_{\mathbb{P}} E \ll E$,
2. $\vdash^P_{\mathbb{P}} E \dashrightarrow E$,

(Trans-C) for all component expressions E, F, G in P

1. if $\vdash^P_{\mathbb{P}} E \ll F$ and $\vdash^P_{\mathbb{P}} F \ll G$ then $\vdash^P_{\mathbb{P}} E \ll G$,
2. if $\vdash^P_{\mathbb{P}} E \longrightarrow F$ and $\vdash^P_{\mathbb{P}} F \longrightarrow G$ then $\vdash^P_{\mathbb{P}} E \dashrightarrow G$,

(Rel-C) for all component expressions E and F in P

1. if $\vdash^P_{\mathbb{P}} E \ll F$ then $\vdash^P_{\mathbb{P}} F \dashrightarrow E$,

(DPR-C) for all $R_i, i = 1, \ldots, k$ where Q is the design pattern expression R_i refines from and for all components $\hat{E}$ and $\hat{F}$ in Q

1. if $\vdash^Q_{\mathbb{P}} \hat{E} \dashrightarrow \hat{F}$ then $\vdash^P_{\mathbb{P}} E \dashrightarrow F$
2. if $\vdash^Q_{\mathbb{P}} \hat{E} \ll \hat{F}$ then $\vdash^P_{\mathbb{P}} E \ll F$

where $E =_{def} \mathbb{P}\left[\left(\iota^P_{\mathbb{P}}[R_i]\right)_I\left(\hat{\mathcal{E}}\right)\right]$ and $F =_{def} \mathbb{P}\left[\left(\iota^P_{\mathbb{P}}[R_i]\right)_I\left(\hat{\mathcal{F}}\right)\right]$.

□

Fact 6.7 The deduction system presented in definition 6.6 is sound, i.e. the following holds. Let $\mathbb{P}$ be a given system of design patterns. Then, the following properties hold for a arbitrary design pattern expression P and contained component expressions E and F.

1. if $\vdash^P_{\mathbb{P}} E \ll F$ then $M^P_{\mathbb{P}}[E] \ll M^P_{\mathbb{P}}[F]$,
2. if $\vdash^P_{\mathbb{P}} E \dashrightarrow F$ then $M^P_{\mathbb{P}}[E] \dashrightarrow M^P_{\mathbb{P}}[F]$.

Proof follows by fact 6.5 and the properties of the clientship- and subtype- relations (cf. [5] and chapter 3).
□

The refinement operator in *PP* has been defined using concepts of program transformation. In many situations, it is necessary to prove that certain properties hold throughout the refinement. Regarding the refinement in *PP*, it is especially of importance to show that the object oriented clientship- and the subtype relations between the components of the source design pattern are preserved by the refinement. This ensures that method implementations that are the result of the command translation (cf. definition 5.13) can be executed in the context of the refined design pattern. As a consequence follows that this specialized kind of refinement maintains runtime safety of the implementation in the refined design pattern.

The following fact ensures that the refinement used in *PP* is a valid refinement as defined in chapter 4 preserving the syntactically deducable clientship- and the subtype relations between components.

Fact 6.8 Let $\mathbb{P}$ be a given system of design pattern expressions. Then, the following property holds for a arbitrary design pattern expression P.

$$DP_{\mathbb{P}}[Q] \overset{\delta_{\mathbb{P}}^{P}[R_i], R' \subseteq R}{\rightsquigarrow} DP_{\mathbb{P}}[P]$$

where Q is the design pattern expression referenced in R_i for all $i = 1, \ldots, k$ and

1. R' are deducable clientship relationships in Q and R is the clientship relation in $SPEC$ or
2. R' are deducable subtype relationships in Q and R is the subtype relation in $SPEC$.

Proof follows by the definition of the flattening of design patterns expressions (cf. 6.3).

□

In analogy to the deduction system presented in definition 6.6, a deduction system for design patterns expressions can be defined. Again, the aim of the deduction system is to deduce relations syntactically.

Definition 6.9 *a deduction system for design patterns expressions*
Let $\mathbb{P}$ be a given system of design pattern expressions. Then, a deduction sytem for design patterns expressions can be defined es follows.

(SCR-DP) For all design pattern expressions P

1. for all R_i with $i = 1, \ldots, k \vdash_{\mathbb{P}} Q \overset{\delta_{\mathbb{P}}^{P}[R_i], R' \subseteq R}{\rightsquigarrow} P$ where
   (a) R' are deducable clientship relationships in Q and R is the clientship relation in $SPEC$ or
   (b) R' are deducable subtype relationships in Q and R is the subtype relation in $SPEC$,
2. $\vdash_{\mathbb{P}} P \ll D_i$ for all $i = 1, \ldots, l$,
3. $\vdash_{\mathbb{P}} U_i \longrightarrow P$ for all $i = 1, \ldots, n$,

(Ref-DP) for all design pattern expressions E

1. $\vdash_{\mathbb{P}} E \overset{id, R}{\rightsquigarrow} E$ for some relation R,
2. $\vdash_{\mathbb{P}} E \ll E$,
3. $\vdash_{\mathbb{P}} E \longrightarrow E$,

(Trans-DP) for all design pattern expressions E, F, G

1. if $\vdash_{\mathbb{P}} E \overset{\delta_1, R_1 \subseteq R}{\rightsquigarrow} F$ and $\vdash_{\mathbb{P}} F \overset{\delta_2, R_2 \subseteq R}{\rightsquigarrow} G$ then $\vdash_{\mathbb{P}} E \overset{\delta_2 \circ \delta_1, R_2 \circ R_1 \subseteq R}{\rightsquigarrow} G$
2. if $\vdash_{\mathbb{P}} E \ll F$ and $\vdash_{\mathbb{P}} F \ll G$ then $\vdash_{\mathbb{P}} E \ll G$,
3. if $\vdash_{\mathbb{P}} E \longrightarrow F$ and $\vdash_{\mathbb{P}} F \longrightarrow G$ then $\vdash_{\mathbb{P}} E \longrightarrow G$,

(Rel-DP) for all design pattern expressions E and F

1. if $\vdash_{\mathbb{P}} E \overset{\delta, R' \subseteq R}{\rightsquigarrow} F$ then $\vdash_{\mathbb{P}} E \ll F$.
2. if $\vdash_{\mathbb{P}} E \ll F$ then $\vdash_{\mathbb{P}} F \longrightarrow E$.

□

Fact 6.10 The deduction system presented in definition 6.9 is sound, i.e. the following holds. Let $\mathbb{P}$ be a given system of design patterns. Then, the following properties hold for a arbitrary design pattern expressions E and F.

1. if $\vdash_{\mathbb{P}} E \overset{\delta, R' \subseteq R}{\rightsquigarrow} F$ then $DP_{\mathbb{P}}[E] \overset{\delta, R' \subseteq R}{\rightsquigarrow} DP_{\mathbb{P}}[F]$
2. if $\vdash_{\mathbb{P}} E \ll F$ then $\overline{M}_{\mathbb{P}}[E] \ll \overline{M}_{\mathbb{P}}[F]$,
3. if $\vdash_{\mathbb{P}} E \longrightarrow F$ then $\overline{M}_{\mathbb{P}}[E] \longrightarrow \overline{M}_{\mathbb{P}}[F]$

Proof follows by fact 6.5 and the properties of the clientship- and subtype- relations (cf. [5] and chapter 3). □

Note that the deduction system introduced above are not complete. In [5], this lack is compensated by a deduction rule that includes all relations that are not deducable syntactically but semantically. In this thesis, however, there is no need for such a rule. Therefore, it is ommitted.

6.4 The satisfiability of design patterns

The semantics of a system of design pattern expressions results in a specification. In order to obtain useful programs, it has to be guaranteed that this specification is satisfiable. Especially, the refinement relation that is used by *PP* requires a non-empty set of models in order to maintain the model relation between source design pattern and refined design pattern.

As defined in chapter 3, a specification is satisfiable if the associated set of models is not empty. However, the satisfiability of a design pattern system implemented in *PP* can not simply be deduced by syntactic criterias. Moreover, additional restrictions have to be imposed on *PP* programs. Basically, these restrictions rule out unintuitive cases and cases in which the resulting signature is not coherent. Applied to *PP*, these restrictions could informally be expressed as follows.

1. Each non-basic operation in $Sig\left(\overline{M}_{\mathbb{P}}[P_1, \ldots, P_n]\right)$ is associated with exactly one method or attribute in $\mathbb{P}$.
2. Each non-basic operation in $Sig\left(\overline{M}_{\mathbb{P}}[P_1, \ldots, P_n]\right)$ is associated with at most one method implementation in $\mathbb{P}$.
3. If there are two methods $m(s_1, \ldots, s_n) \to s$ and $m(t_1, \ldots, t_n) \to t$ in $Sig\left(\overline{M}_{\mathbb{P}}[P_1, \ldots, P_n]\right)$ with the same arity and comparable parameter sorts then their parameter sorts beside the $Self$ parameter have to be equal, i.e. $s_2 = t_2, \ldots, s_n = t_n$. This restriction can be relaxed for *non-selfish* methods. In this approach, however, covariant sorts lead to problems of type-safety.
4. It has to be ensured that $Sig\left(\overline{M}_{\mathbb{P}}[P_1, \ldots, P_n]\right)$ is coherent. E.g. when multiple subtyping is used, it has to be ensured that the definition of the subcomponent- or design pattern expression contains corresponding **select** statements in order to select the appropriate method implementations.

Although it is not the goal of this thesis to prove the satisfiability of a specification that result from the semantics of *PP*, a theoretical proof could be found on the construction of a particular model of that is contained by this specification.

Chapter 7

Final remarks

7.1 Related work

The book [8] represents a mile stone in software engineering. Design patterns are introduced as abstract structural entities that base on an object oriented view of things. The notion of a design pattern provides a powerful way to improve the efficiency of the development of large-scale software. However, since design patterns in [8] base on an informal paradigm, they can not be applied to a specific problem in an automatized process. Soon after the time when [8] was published first, the need for a support on the side of programming languages had been expressed.

In [4], the importance of a language support of design patterns was underpinned by common problems that occur when design patterns are implemented in object oriented programming languages. These problems include *traceability*, *reusability*, the *self-problem* and the implementation overhead. However, the approach of the **LayOM** as introduced in [4] is rather different to the ideas that have been applied in this thesis. The **LayOM** extends the object oriented programming model by so-called layers. The concept of layers which enables one to intercept and alter messages from and to objects provides a versatile programming facility making it possible to represent the nature of design patterns. On the one hand, design patterns can be implemented in this way and even be reused by extended object oriented mechanisms, but on the other hand, their original component structure as proposed e.g. in [8] gets lost in this process.

The article [9] provides an interesting new perspective of design pattern language support that is much more similar to the approach presented in this thesis. [9] strictly distinguishes between two different levels: the *program level* and the *extract level*. Usually, the software development in conservative object oriented programming languages entirely takes place on the program level whereas the actual design pattern implementation resides on the extract level. Design patterns on this level can be applied leading to programs in the usual sense residing on the program level. In contrast to the *PatternModel* and *PP*, these notions have a static character since the resulting programs are object oriented. Furthermore, problems of type safety have not been worked out yet.

7.2 Future work

The results obtained by this thesis can be applied and extended in various areas. Especially from the perspective of the *PatternModel*, there are many different aspects that have not been discussed yet.

One of these aspects is the so-called *design by contract* as it is realized in *Eiffel*. Design by contract provides powerful means for the development of maintainable and error-free software. The developer can specify such a *contract* in form of e.g. pre- and postconditions that are evaluated when a method, etc. is

entered or exited. If the contract is broken, special error-handling routines can be invoked. Especially, when design patterns are included in the development process, design by contract can be benefical since it allows to specify properties on a finer grain.

Another future extension could deal with distributed environments and parallel computing. Since the *PatternModel* and *PP* provides a strong notion of visibility and encapsulation, it is possible to handle single design pattern instances independently. Therefore, these instances can compute results of calculations, etc. autonomously and parallelly. Presently, products like *CORBA* deal with the distribution of objects in networks. One approach could be to integrate the notion of the design pattern into such a system. Another approach could directly base on a language extension of *PP*.

On the technical side, all these theoretical considerations have to accompanied by the development of corresponding tools on a computer. In the case of *PP*, a compiler for *PAL* has already been implemented in [6]. As a second step, the *GoF* design patterns have to be implemented in *PAL* in order to provide a basis for design pattern oriented software engineering.

7.3 Conclusion

The introduction of the *PatternModel* as a design pattern oriented programming model has been a main aim of this thesis. It provides a framework for the definition of the semantics of the design pattern oriented programming language *PP* based on the design mechanisms of abstract data types. The notions used in the *PatternModel* extend conventional object oriented notions, the *PatternModel* itself is an extension of the object oriented programming model on a conceptual level.

Due to encapsulation- and reusability problems, the class of object oriented programming languages can not be used for the implementation of design patterns. *PP* overcomes these problems by using new design pattern oriented features. The means provided by *PP* are sufficient to implement and apply design patterns. On the other hand, *PP* also supports common object oriented concepts. Therefore, *PP* represents a powerful design pattern oriented language that does not contradict the object oriented paradigm.

Appendix A

The syntax of *PP* in EBNF

<design pattern> ::= <design pattern> {, <design pattern>}*

<design pattern> ::= design pattern <design pattern id> is <design pattern exp> end design pattern

<design pattern exp> ::= refinements <refinement> {, <refinement>}*
subclasses design patterns <design pattern id> {, <design pattern id>}*
uses design patterns <design pattern id> {, <design pattern id>}*
components <component> {, <component>}*
attributes <attributes>
methods <method head> {, <method head>}*
method implementations <method imp> {, <method imp>}*

<refinement> ::= refinement id> refines <design pattern id> <refinement exp> end refinement

<refinement exp> ::= refine <component ref> {, <component ref>}*
rename by <method renaming> {, <method renaming>}*
select <method head> {, <method head>}*

<component ref> ::= component id> into <component id>

<method renaming> ::= method id> → <method id>

<method head> ::= method id> (<entities>) return <type>

<entities> ::= <entity> {, <entity>}*

<entity> ::= <entity id> : <type>

<type> ::= <design pattern id> | <component id> | <class spec id>

<component> ::= component <component id> is <component exp> end component

<component exp> ::= refinement id> <recast> { <refinement id> <recast> }*
subclasses components <component id> {, <component id> }*
uses components <component id> {, <component id> }*
attributes <attributes>
methods <method head> {, <method head>}*
method implementations <method imp> {, <method imp>}*

<recast> ::= <recast exp> end recast

<recast exp> ::= rename by <method renaming> {, <method renaming>}*
select <method head> {, <method head>}*

<attributes> ::= <entities>

<method imp> ::= <method head> is <command> end

<command> ::= as presented in definition 5.9

<design pattern id>,
<component id>,
<entity id> ::= identifiers

<class spec id> ::= identifier of a basic type

Appendix B

Basic notions of partial finite mappings (cf. [5])

Definition B.1 Let M, N be two sets. For relations $R \subseteq M \times N$ we define

$$dom\,(R) =_{def} \{m : (m,n) \in R\}, im\,(R) =_{def} \{n : (m,n) \in R\},$$

A *partial finite mapping* p is a finite subset of $M \times N$ satisfying the property $(m,n),(m,n') \in p \Longrightarrow n = n'$. The set of partial finite mappings with respect to M and N is denoted by $[M \to N]_{fin}$. We use the following notations.

$$[] =_{def} \emptyset, \quad id =_{def} \emptyset, \quad p\,[m \to n] =_{def} \{(m',n') : (m',n') \in p, m' \neq m\} \cup \{(m,n)\},$$
$$[m_1 \to n_1, \ldots, m_k \to n_k] =_{def} []\,[m_1 \to n_1] \ldots [m_k \to n_k],$$
$$p\,[m] =_{def} \begin{cases} n, \text{ if } (m,n) \in p \\ \text{undefined otherwise.} \end{cases}$$

p is *total* iff $dom\,(p) = M$ (which requires M to be finite).
p_1 and p_2 are *compatible* iff $p_1\,[m] = p_2\,[m]$ for all $m \in dom\,(p_1) \cap dom\,(p_2)$.
p_1 and p_2 are *disjoint* iff $dom\,(p_1) \cap dom\,(p_2) = \emptyset$.
$p_1 \sqsubseteq p_2$ iff p_1 and p_2 are compatible and $dom\,(p_1) \subseteq dom\,(p_2)$.
If p_1 and p_2 are compatible then the mapping $p_1 \sqcup p_2$ is characterized by

$$p_1 \sqcup p_2\,[m] =_{def} \begin{cases} p_1\,[m], \text{ if } m \in dom\,(p_1) \\ p_2\,[m], \text{ if } m \in dom\,(p_2) \\ \text{undefined otherwise.} \end{cases}$$

Let $(p_i)_{i \in I}$ and I-indexed family of partial finite mappings $p_i : M_i \to N_i$. We extend the notations for partial finite mappings to families of partial finite mappings in the following way.

$$dom\,((p_i)_{i \in I}) =_{def} \bigcup_{i \in I} dom\,(p_i), im\,((p_i)_{i \in I}) =_{def} \bigcup_{i \in I} im\,(p_i),$$
$$[] =_{def} (p_i)_{i \in I}, \text{ where } p_i =_{def} [] \text{ for all } i \in I,$$
$$p\,[m \to_i n] =_{def} (\tau_j)_{jinI} \text{ where } \tau_j =_{def} p_j \text{ if } j \neq i \text{ and } \tau_i =_{def} p_i\,[m \to n],$$
$$[m_1 \to_{i_1} n_1, \ldots, m_k \to_{i_k} n_k] =_{def} []\,[m_1 \to_{i_1} n_1] \ldots [m_k \to_{i_k} n_k].$$

$p = (p_i)_{i \in I}$ is *total* iff for all $i \in I$, p_i is total. $p = (p_i)_{i \in I}$ and $\tau = (\tau_i)_{i \in I}$ are *compatible* (*disjoint*, respectively) iff p_i and τ_i are compatible (disjoint, respectively), and $p \sqsubseteq \tau$ iff $p_i \sqsubseteq \tau_i$ for all $i \in I$. If p and τ are compatible then $p \sqcup \tau =_{def} (p_i \sqcup \tau_i)_{i \in I}$.
As usually, we omit indices if the context is clear.

Appendix C

Selected design pattern implementations

C.1 *List*

```
design pattern List

    components

        component Item

            attributes

                next : Item

            methods

                setNext(anItem : Item) returns Item

            method implementations

                setNext(anItem : Item) returns Item is
                do

                    self.next := anItem

                end

        end component

        component ListComp

            uses components

                Item

            attributes

                first : Item,
                current : Item

            methods

                add(anItem : Item) returns ListComp,
                delete returns ListComp,
                getCurrent returns Item,
                rewind returns ListComp,
                next returns ListComp,
                previous returns ListComp,
                isLast returns Boolean,
                isEmpty returns Boolean
```

```
method implementations

    add(anItem : Item) returns ListComp is
    local

        tempItem: Item

    do

        if self.isEmpty
        then
            self.first := anItem;
            self.rewind
        else
            tempItem := self.current;
            self.current := anItem;
            anItem.setNext(tempItem.next);
            tempItem.setNext(anItem);
            self
        end

    end,

    delete returns ListComp is
    local

        tempItem : Item

    do

        if not self.isEmpty
        then
            if self.first == self.current
            then
                self.first := self.current.next;
                self.rewind
            else
                tempItem := self.previous(self.current);
                tempItem.setNext(self.current.next);
                self.current := self.current;
                self
            end
        else
            self
        end

    end,

    get returns Item is
    do

        self.current

    end,

    rewind returns ListComp is
    do

        self.current := self.first

    end,

    next returns Item is
    do

        if isLast
        then
            self.rewind
            void
        else
```

```
                    self.current := self.current.next;
                    self.current
                end

            end,

            previous returns Item is
            local

                tempItem : Item

            do

                if self.isFirst
                then
                    void
                else
                    from
                        tempItem := self.current;
                        self.rewind
                    until
                        self.current.next == tempItem
                    loop
                        self.next
                    end;
                    self.current
                end

            end,

            isEmpty returns Boolean is
            do

                first.isVoid

            end,

            isFirst returns Boolean is
            do

                (isEmpty) or (self.First == self.current)

            end,

            isLast returns Boolean is
            do

                (isEmpty) or ((self.current.next).isVoid)

            end

        end component

    attributes

        theListComp : ListComp

    methods

        make returns List

    method implementations

        make returns List
        do

            self.theListComp := create List::ListComp

        end

end design pattern
```

C.2 *Subtyping*

```
design pattern Subtyping

    components

        component Parent

        end component

        component Child

            subclasses components

                Parent

        end component

end design pattern
```

C.3 *Composite*

```
design pattern Composite

    refinements

        SubtypingLeafRef refines Subtyping

            refine

                Parent into Component,
                Child into Leaf

        end refinement,

        SubtypingCompositeRef refines Subtyping

            refine

                Parent into Component,
                Child into CompositeComp

        end refinement,

        ListRef refines List

            refine

                Item into Component,
                ListComp into CompositeComp

            rename by

                theListComp -> theCompositeComp

        end refinement

    components

        component Component

            methods

                operation returns Component,
                add(anItem : Component) returns Component,
                delete returns Component,
                getCurrent returns Component,
                rewind returns Component,
                next returns Component,
                previous returns Component

            method implementations
```

```
            operation returns Component is do self end,
            add(anItem : Component) returns Component is do self end,
            delete returns Component is do self end,
            getCurrent returns Component is do self end,
            rewind returns Component is do self end,
            next returns Component is do self end,
            previous returns Component is do self end
        end component
        component Leaf
            methods
                operation returns Leaf
            method implementations
                operation returns Leaf is do self end
        end component
        component CompositeComp
            methods
                operation returns CompositeComp
            method implementations
                operation returns CompositeCom is
                do
                    from
                        self.rewind;
                        self.current.operation
                    until
                        (self.next).isVoid
                    loop
                        self.current.operation
                    end
                end
        end component
end design pattern
```

C.4 *GraphicComposite*

```
design pattern GraphicComposite
    refinements
        LeafRef refines Composite
            refine
                Component into Graphic,
                Leaf into Line,
                CompositeComp into Picture
        end refinement,
        CompositeRef refines Composite
            refine
```

```
            Component into Graphic,
            Leaf into Circle,
            CompositeComp into Picture

    end refinement,

components

    component Graphic

        recast LeafRef

            rename by

                operation -> draw

            select

                draw returns Graphic,
                add(anItem : Graphic) returns Graphic,
                delete returns Graphic,
                getCurrent returns Graphic,
                rewind returns Graphic,
                next returns Graphic,
                previous returns Graphic

        end recast

    end component

    component Line

        recast LeafRef

            rename by

                operation -> draw

            select

                add(anItem : Graphic) returns Graphic,
                delete returns Graphic,
                getCurrent returns Graphic,
                rewind returns Graphic,
                next returns Graphic,
                previous returns Graphic

        end recast

        attributes

            x1 : Integer,
            y1 : Integer,
            x2 : Integer,
            y2 : Integer

        methods

            draw returns Line

        method implementations

            draw returns Line is
            do

                -- draw a line using x1, y1, x2, y2
            end

    end component

    component Circle

        recast CompositeRef
```

```
            rename by

                operation → draw

            select

                add(anItem : Graphic) returns Graphic,
                delete returns Graphic,
                getCurrent returns Graphic,
                rewind returns Graphic,
                next returns Graphic,
                previous returns Graphic

        end recast

        attributes

            x : Integer,
            y : Integer,
            radius : Integer

        methods

            draw returns Circle

        method implementations

            draw returns Circle is
            do

                – draw a circle using x, y, radius
            end

    end component

    component Picture

        recast CompositeRef

            rename by

                operation → draw

            select

                draw returns Picture,
                add(anItem : Graphic) returns Picture,
                delete returns Picture,
                getCurrent returns Picture,
                rewind returns Picture,
                next returns Picture,
                previous returns Picture

        end recast

    end component

end design pattern
```

Bibliography

[1] J.W. de Bakker: *Mathematical Theory of Program Correctness.* Prentice-Hall, 1980

[2] Helmut Balzert: *Lehrbuch der Software-Technik*
ISBN 3-8274-0042-2 Spektrum Akademischer Verlag, 1996

[3] Jan Bosch, Goerel Hedin and Kai Koskomies. *Language support for Design Patterns and Frameworks*, 1997

[4] Jan Bosch: *Design Patterns & Frameworks: On the Issue of Language Support*,
In Bosch et al. [3].

[5] Ruth Breu: *Algebraic Specification Techniques in Object Oriented Environments*,
ISBN 3-540-54972-2 Springer-Verlag Berlin Heidelberg New York, 1991

[6] Stefan Bünnig: *Entwicklung einer Sprache zur Unterstützung von Design Patterns und Implementierung eines dazugehörigen Übersetzers*
Master's Thesis, University of Rostock, Department of Computer Science, in preparation, 1999

[7] Stefan Bünnig, Peter Forbrig, Ralf Lämmel and Normen Seemann: *Design pattern oriented programming* University of Rostock, Department of Computer Science, 1999

[8] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*,
ISBN 0-201-63361-2 Addison Wesley Publishing Company, 1994

[9] Eyoun Eli Jacobsen: *Design Patterns as Program Extracts*
Aalborg University, Department of Computer Science. In Bosch et al. [3].

[10] Pete Thomas, Ray Weedon: *Object-Oriented Programming in Eiffel*,
ISBN 0-201-59387-4 Addison Wesley Publishing Company, 1995

List of Figures

Eklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 31.05.1999 Normen Seemann

Thesisses

1. Using object oriented programming techniques, it is possible to apply a design pattern to a special problem, however, it is not possible to implement the design pattern itself. For this purpose, it is necessary to use more advanced, design pattern oriented programming techiques.

2. The introduced design pattern oriented model *PatternModel* together with the design pattern oriented programming language *PP* directly support the notion of a design pattern, its refinement and instantion which allows the reuse of whole class structures and the actual implementation of design patterns.

3. The introduced design pattern oriented model *PatternModel* together with the design pattern oriented programming language *PP* improve the development of software in terms of reusability, traceability and maintainability.

4. The *PatternModel* represents a framework defining basic design pattern oriented concepts and notions using the approach of algebraic specifications.

5. The design pattern oriented imperative programming language *PP* provides constructs for supporting design pattern and refinements. Its semantics is defined in a denotational way using the notions contained by the *PatternModel*.