Implementing the FACE Object Model in C++

Masters Thesis
Faculty of Science
University of Berne

by

Matthias Rieger

1997

Supervisor:
Prof. Dr. Oscar Nierstrasz
Institute of Computer Science and Applied Mathematics

Abstract

FACE is an object–oriented, self-descripting data model with first-class types. FACE can be used to model software, e.g. object–oriented frameworks.
We explore techniques and mechanisms to implement the reflective FACE data model in the statically typed, object–oriented language C++. Some comparison of FACE with other meta level approaches like MetaObject Protocols or Open Implementations is done, and a short example modeling software is described.

Keywords: Software Reuse, Frameworks, Data models, Reification, Reflection, Open Implementations.

Contents

1 Introduction **7**

1.1 Software Reuse with Frameworks 8

1.1.1 FACE supports abstraction 9

1.1.2 FACE supports selection 9

1.1.3 FACE supports specialization 10

1.1.4 FACE supports integration 10

1.2 The implementation of the FACE data model 10

1.2.1 The scope of the work in a nutshell 12

1.3 Road map for the reader 12

2 Background **14**

2.1 Conceptual background 14

2.1.1 Frameworks 15

2.1.2 Software Composition 16

2.1.3 From object composition to class composition 18

2.1.4 Reflection and Reification 19

2.1.5 MetaObject Protocols and Open Implementations 21

2.2 Implementational background 24

2.2.1 Generic Programming in C++: Templates and STL 24

2.3 The FACE data model 27

2.3.1 The history of the FACE data model 27

2.3.2 The philosophy of the data model 29

2.3.3 The basic element of the data model: the Object 31

2.3.4 Types 32

2.3.5 The kernel of the FACE data model 37
2.3.6 The Extensibility feature 38

3 Implementation Issues **40**
3.1 Road map for this chapter 40
3.2 Establishing the system of domains and types 42
3.2.1 The formal basis of the domain system 42
3.2.2 Transferring the domain system to C++ 43
3.3 The implementation of the FACE-Object 45
3.3.1 The generic structure of the FACE-Object 45
3.3.2 The basic interface of a FACE-Object 46
3.3.3 The implementation of the FACE-Object 47
3.3.4 The implementation of the type checking mechanism 52
3.4 The implementation of the FACE kernel 54
3.4.1 Mapping the FACE instantiation to C++ 54
3.4.2 The implementation of the instantiation mechanism 59
3.4.3 Discussion 63
3.5 Bootstrapping the system 65
3.5.1 The requirements 65
3.5.2 The method 66
3.5.3 The implementation 67
3.5.4 Discussion 68
3.6 Implementing an example 69
3.6.1 Execution in FACE 69
3.6.2 The implementation of the example 70
3.6.3 Discussion 73

4 Conclusions **75**
4.1 Conclusions from the implementation 75
4.1.1 Specific conclusions for FACE 75
4.1.2 Negative Results 75
4.2 Theoretic Conclusions 76
4.3 Future Work 76

4.3.1 Technical Issues 76
4.3.2 Research Issues 76

A Notation **78**

A.1 The FACE-notation 79
A.1.1 Purpose 79
A.1.2 Motivation of the chosen presentation 79
A.2 The Intermediate notation 80
A.2.1 Purpose 80
A.2.2 Motivation of the chosen presentation 80
A.3 The Unified Modeling Language 81

Acknowledgments

There are a lot of people who have contributed in many direct or indirect ways to the completion of this work.

First and foremost I want to thank my supervisor Theo Dirk Meijler for the long discussions (or should I say monologues?) on FACE and the surroundings. He also teached my scientific reasoning, ripping apart this text over and over again only occasionally showing fatigue. He made me aware of my sloppiness in thinking .[1]

I want to thank Jürg Gertsch and Roland Loser and Sander Tichelaar[2] who more or less by coincidence did the same thing at the same time and thus[3] provided support. (Hey Jürg and Roli: we still failed to achieve the status of last of the mohicans. Amazing, isn't it?) I want to thank Christoph Pappa who helped me organize the final battle. He was a great coach. I want to thank Serge Demeyer for telling me about the fish and constantly reminding me how much fun I was having in writing this thesis.

I want to thank Sonja Muhlert who laughed at me when I took this work too serious. I want to thank Nicole Pfister who also tried to inspire me with a less existential attitude toward this thing (I wonder why it is always women who have to teach us men that there are things in life that might be more important than work.)

Last but not least I want to thank my parents who were so generous to provide me with a shoestring I could live on. It allowed me to concentrate on this work.

Thank you very much.

[1] Hey, I like being sloppy. It's a prerequisite for one of the best things in life: graphic design. But it's good to know where to stop.

[2] Eeeeiii — Hammock!

[3] Of course, they are nice fellows too!

Chapter 1

Introduction

The demand for software is increasing steadily as the micro chip is applied in more and more domains. Hardware is getting faster and more powerful every year, databases are growing by the minute and so do the systems that manage them. But the productivity of the software developer has not kept up with this growth. The widening gap between supply and demand of quality software has been denominated the *software crisis* in an article a long time ago.[1] The problem of the software crisis is twofold: On the one hand there is the already mentioned productivity problem and on the other hand is the complexity problem. One of the basic problems behind the complexity issue are "our very small heads we must live with" (E.J.Dijkstra).

> When we are fighting the software crisis, we are fighting human nature and the human inability to handle complexity. If we possessed perfect minds with perfect recall, programs with one million lines of code or programs written in spaghetti code style would present us with no problems. However, such programs do give us problems because our minds *are* limited. ([Sig96, page 3])

Apart from overwhelming complexity—which is an *essential* problem of modern software systems according to [Bro87]—it is still a non-settled question in the software engineering community on how to implement the technique that lies at the heart of every engineering discipline: *Reuse*.[2] Software reuse can take many forms but the principal goal of every reuse technology is to raise the abstraction level of the artifacts developers deal with. The closer the abstractions are to the informal concepts of the system a software developer has in his mind, the faster the system will be built and the more successful is the reuse technology [Kru92]. As Gamma et. al. in their book on Design Patterns [GHJV95] put it (referring to designers of all kinds of engineering disciplines):

[1] This is the famous article *Mass produced software components* by M. McIlroy, In Naur P., Randell B., *Software Engineering: Report on a conference by the NATO Science Committee*, NATO Scientific Affairs Division, Brussels 1968.

[2] According to [Bro87], *Software Reuse* is one of the promising attacks on the essential problem in software engineering.

> Expert designers don't solve every problem from first principles. Rather they use solutions that have worked for them in the past. A good solution is used over and over again.

In other mature engineering disciplines with a longer history than the software engineering profession, books with "best practices" hand down very specific knowledge about methods and techniques that have proven to be valuable to the young and unexperienced disciples. In the software engineering field, attempts at producing such standardized reference manuals are only beginning to emerge.[3] The efficient reuse of ideas and designs is still a large research area.

Frameworks offer promising capabilities in this respect.[4]

1.1 Software Reuse with Frameworks

An object–oriented framework, or simply *framework*, is a set of cooperating classes that make up a reusable design for a specific class of software.[5] Frameworks are typically designed by domain experts and then used by non-experts to develop applications for that domain. Expert knowledge of the specific domain the framework is written for is expressed in the collaboration patterns between its classes. A framework is therefore much more than a class library. The components of a class library are used individually, whereas the framework is reused as a whole, providing the common skeleton of applications in its domain. A developer who wants to use the framework to build a specific application in the frameworks domain, a process also known as *specialization*, has to 'beef up' the bare bones. In order to specialize a framework correctly he has to have a lot of knowledge. He has to understand the design of the prototype application the framework offers as well as the inner workings of the components he wants to adapt and adjust. Thus, the learning curve for frameworks is very steep. Also, users of a framework are normally not guided in the specialization process, making reuse of the framework in ways not intended by the framework developer a risk.
FACE, a Framework Adaptive Composition Environment, is intended to overcome these problems and make framework reuse more "user friendly". FACE [6] is essentially an object–oriented data model coupled with a visual environment, in which compositions of software artifacts can be described, graphically presented to the user, and changed using the means of direct manipulation.[7] Using a meta-framework, FACE adapts to a specific framework, which means that the framework classes get an explicit representation in the data model. The relations that link the classes of the framework

[3] One of the promising approaches in collections of case studies of exemplary software designs are the *Design Patterns* widely accepted after the publication of [GHJV95].

[4] According to [Kru92, page 178], *software architectures* come closest to an ideal reuse technology among the eight reuse categories reviewed in the article.

[5] Definition cited in [GHJV95, page 26].

[6] For a more detailed introduction refer to [Mei96].

[7] The notion of *Direct Manipulation* refers to interfaces having the following properties:

1. Continuous representation of the object of interest.
2. Physical actions or labeled button presses instead of complex syntax.
3. Rapid incremental reversible operations whose impact on the object of interest is immediately visible. (Shneiderman, 1982)

are also made into explicitly represented components. The whole framework is in this way represented as objects in the FACE data model and can be visually presented in a graphical workspace. Users can then develop applications by instantiating FACE class-components and parameterizing them.
In [Kru92], Charles Krueger discusses eight approaches to software reuse along a taxonomy of four dimensions which are motivated by the four tasks involved in every reuse activity: Abstraction, Selection, Specialization and Integration. In the following four sections I will characterize FACE in discussing how it contributes to each of those four dimensions.

1.1.1 FACE supports abstraction

Abstraction[8] is *the* essential feature in any reuse technique. Successful application of a reuse technique to a software engineering technology is inexorably tied to raising the level of abstraction for that technology ([Kru92, pages 133–134]). FACE tries to raise the abstraction level by the following means:

- **Blackbox parameterization**: The usual approach to framework specialization is subclassing, where the author of the subclass has to understand the implementation of the superclass (whitebox reuse). FACE supports blackbox reuse: components are specialized through parameterization.
- **Reification of relations**: Relations between components can also be represented as domain-specific components which can be parameterized. These relationship informations are normally hidden in the source code of a class. By making them explicit, the abstraction level is raised again.
- **Describing the evolution**: In FACE, an application developer has the possibility to describe not only the initial set of runtime objects but also the *possible evolution* of the object structures during runtime. Making the explicit control over this features possible raises abstraction again.

1.1.2 FACE supports selection

Reusable software artifacts must be cataloged so that developers can locate, compare and select them [Kru92, page 133]. FACE supports the selection process in the following ways:

- **Menu of available components**: In a visual environment, all reified artifacts of a framework can be graphically presented to a user. It is possible to introduce hierarchies to structure the set. With an online help system, features of the components can be explained.
- **Type checking mechanism**: Because relations are made explicit and can be described, the system is able to select candidates for a certain link automatically.

[8] An abstraction for a software artifact is a succinct description that suppresses the details that are unimportant to a software developer and emphasizes the information that is important. ([Kru92, page 134])

1.1.3 FACE supports specialization

The software artifacts of a reuse catalogue are mostly of a *generic* character. In order to use them for a specific solution, they have to be instantiated or specialized through parameters, transformations, constraints or some other form of refinement ([Kru92, page 133]). FACE supports specialization in the following ways:

- **Explicit composition interface**: Every FACE-component has an explicit composition interface. This interface is shown in the visual environment. It is therefore easy for a developer to control the parameterization of such a component.
- **Correctness**: The type checking mechanism ensures that only correct links are established.

1.1.4 FACE supports integration

Integration is the process of combining the selected and specialized components to form a complete, runnable system. FACE has the following possibilities for helping to build a running system out of a FACE-composition:

- **Runtime semantics**: Through the meta-framework of FACE, the framework developer can specify how the components are given runtime semantic. With an underlying programming language, each FACE-component can have an implementation. Compositions can then be translated to a runtime system.
- **Compilation and interpretation**: With the link to the runtime entities, compositions can be tested by interpreting the FACE-structure. It is also theoretically possible to generate stand alone systems by compilation.

1.2 The implementation of the FACE data model

The purpose of the work at hand is to provide an implementation of the kernel of the FACE data model as it would be needed as a back end by a visual composition environment. This kernel is based on the formalization essentially laid down in [Mei93a]. It has, however, evolved in certain aspects since it was published. We will use this formalization as a starting point for the implementation. The elements of the data model that are important for a visual composition environment and that showed the way for the implementation to go are the following:

- the generic FACE-Object, the cornerstone of the model,
- object–oriented mechanisms like instantiation, inheritance and message passing,
- the kernel of the type-system, which allows the creation and parameterization of new types and also enables type checking,

Implementing the formal model means mapping the objects, properties, and methods defined in the model to the programming constructs of an underlying *host programming*

language. Choosing such a host language must be done with respect to the following requirements:

- The concepts of objects, inheritance and polymorphism should be easily realizable in the hostlanguage, the language should obviously be object–oriented itself.
- The graphical user interface (GUI) of the visual environment needs to have an easy interface to the data model implementation. It is also important that the programming of the GUI can be supported by some windowing toolkit. The hostlanguage for the data model should be 'near' such a GUI-language.
- It should be possible to link compositions to the frameworks they are a representation of. It should therefore be possible to interface the hostlanguage to a multitude of different programming languages. This requirement translates to the need for a well known, widely used language with certain low-level programming facilities.
- There should be a range of development tools available for the language.

As a possible set of programming languages that fulfill these requirements the following can be considered: CLOS, Smalltalk, Java, SELF, C++.

- CLOS and Smalltalk were considered too complicated under the aspect of combining the FACE meta model with the meta model already included in each of these languages. Besides, during the initial development of the FACE data model, an implementation was attempted in CLOS with mediocre success.[9]
- Java was at the inception of this work still very new and barely known.
- An implementation in SELF[10] was attempted in parallel to this work.

The choice of implementation language finally fell on the widely used language C++. An advantage of this choice is, for example, that since C++ is being used so frequently for software projects, it would be easy to find a sample framework to which FACE could be adapted for testing purposes without having to cross language borders. Also, since C++ code runs very efficiently, a transformation of the prototype into a production quality program would be possible without a port to another language.[11]
After a closer examination of the implementation task, the following subtasks come into view:

- The type system of the FACE formalization has to be mapped to the type system of the host-language. This comprises built-in types (*basic value types* in FACE) as well as complex types (*object types* in FACE).

[9] Meijler, personal communication.

[10] For an introduction to Self, refer to D. Ungar, R. Smith, *SELF: The Power of Simplicity*, in *LISP and Symbolic Computation*, 4, 3, 1991, Kluwer Academic Publishers, or link your browser to the server `http://self.stanford.edu`.

[11] For a more detailed discussion of the advantages of C++ with respect to our model, see [Mei93b, page 175].

- For the generic structure of the FACE-Object, generic containers have to be provided. These containers will hold the parameters of the component, which can be of every type the data model provides.
- The instantiation process of FACE has to be mapped to the instantiation in the hostlanguage. The same is true for inheritance.
- A bootstrapping procedure has to be designed to start up the kernel of the type system.

The work of this thesis does, however, not comprise the following aspects of FACE:

- It is not our goal to adapt the kernel type system to a sample framework.
- From the last point follows that we do not intend to research possible link mechanisms between the FACE-components and the framework-elements they represent.
- As a further consequence of the first point, we do not provide interpretation or compilation facilities for the compositions developed in the model that go beyond an execution mechanism stemming from a previous step in the FACE evolution.
- The GUI part of the visual composition environment is also out of scope of this work.

What, after all, this work wants to achieve, is just a bare naked implementation of the FACE-kernel with the basic functionalities of the data model.

1.2.1 The scope of the work in a nutshell

FACE is a tool for making structures of object–oriented software explicit. To do so it uses representations for elements of the software, makes them into first class citizens in its own model, and lets the user change and modify these structures as if FACE were a programming language.

The main goal of this work is to explore the possibilities of implementing a reflective object model where types are accessible at runtime in a statically typed object–oriented programming language that has classes which remain implicit at runtime. This requires the implementation of mechanisms that ‘elevate’ classes in the rang of first class citizens.

Additionally, in order to understand the FACE model in more depth, we try to explain the connection between FACE and other reflective models from the realm of MetaObject Protocols and Open Implementations.

1.3 Road map for the reader

The thesis is divided in two parts. In the background chapter (Chapter 2), the work on the concepts of object–oriented data models, reification and reflection, which influenced the development of FACE, are presented. In addition, the theoretical aspects of

some of the methods that were used in the implementation are explained.
In the implementation chapter (Chapter 3), different parts of the implementation are explained in detail giving an overview of the problems and how they were resolved. The description of a small example is presented that was implemented to gain experience with modeling software using the FACE system.

Chapter 2

Background

In this chapter, the problem domain in which FACE is set, is presented in detail. We will give overviews of models and techniques that are used in common solutions to these problems, and we will show how the approach taken with FACE differs from them.
We will proceed in the following general direction. First, we present the ideas behind software reuse with frameworks and the inheritance technique. Next, we are explaining composition as another method of software reuse. Composition is normally done at the object level, but composing at the class level has some advantages which will be presented. Class level composition is the main contribution of the FACE approach. To be able to compose classes in a manner similar to object composition, FACE introduces another level above the class level, the so-called meta level, which allows to describe the compositions on the class level. This can also be seen as configuration of systems. We will explain the theories of reification and reflection that are related to the question of how to configure systems. We will then give an overview of generic programming in C++. The chapter will end with a thorough presentation of the FACE-data model.

2.1 Conceptual background

What is an object?

Objects[1] are entities that combine the properties of procedures and data by performing computations and saving local state. A computation in an object is triggered by sending a message to the object. The object chooses the appropriate function to perform according to a selector in the message. Message sending thus supports data abstraction, as the implementation and the inner workings of the object remain hidden. The set of methods that is visible from the outside is called the *interface* or the *protocol* of the object. Standardized protocols — different kinds of objects have the same protocol — are a prerequisite for *polymorphism*. Polymorphism, in the context of object–oriented programming, refers to "the capability for different classes of objects to respond to exactly the same protocols. ... Protocols extend the notion of *modularity* (reusable and

[1] This introduction on objects is based on [SB86].

modifiable pieces as enabled by data-abstracted subroutines) to *polymorphism* (interchangeable pieces as enabled by message sending)" [SB86, page 41].
The other major idea in object–oriented programming is the feature of specialization. Specialization allows to specify objects incrementally, taking an old specification that nearly fits the current needs and modifying it to suit the needs exactly. Information can be kept in one central place which facilitates change.
Polymorphism and specialization help to keep invariants over change in a program. Information is localized and hidden as much as possible, the program is thus more robust when confronted with extension and modification.

2.1.1 Frameworks

An object–oriented application framework is a set of prefabricated building blocks modeling a certain problem domain. The framework serves as a starting point for application developers. Instead of starting from scratch, developers can rely on

- a set of abstract and concrete classes which offer default behavior and have to be specialized.
- an application architecture or typical collaboration patterns which model common behavior of an application in this problem domain.
- well tested and proven solutions: frameworks are designed for reuse and evolve over many iterations.

In its generic parts (expressed by the abstract classes), a framework encapsulates the aspects of its domain that can vary over the family of applications represented by it. The variability is represented by the 'holes' of the abstract classes, the abstract methods. Developers derive concrete classes from the abstract bases, beefing up the inherited interface with the meat of their specifically needed behavior. Different than reusing code from a simple class library, the user-written classes and methods do not form the backbone of the application. In fact, the relationship between reused code and user code is reversed: the main thread of computation is dictated by the common architecture of the framework, and only at the points where the specific application deviates from the generic one, user code is called. This requires a deep understanding of how the elements of the framework work together; a steep learning curve is normal with framework users [MN96].

The problems with Inheritance and How we can try better

The framework idea is based on polymorphism. The specialized classes are subtypes of the general ones, which means that they exhibit (at least) the same interface. Polymorphism assures that the new classes can replace the superclass without the clients noticing it. In an object–oriented language like C++, subtyping is tied to the inheritance mechanism; i.e. we cannot create a subtype without reusing the implementation of the supertype. The problem with this specialization method lies in the amount of detail that is exposed to the programmer who is using inheritance. The superclass to her is a "white box" which she has to understand in great detail in order to extend it without

introducing problems. The knowledge of the inner workings of the superclass that implicitly flows into the design of the subclass results in a strong coupling of the subclass to its parent. The inheritance interface of the class–model in common object–oriented languages is in many cases to wide and weakens the encapsulation of the superclass [Sny86]. This leads to unwanted and costly long–range effects when changing the superclass.
Remedies for these problems of the specialization process in object–oriented frameworks generally follow the path to more modularity. By modularity we mean that the client is decoupled from the server,[2] that the client can access the server only over well defined interface. This interface can be made as narrow as possible. The server exposes no implementation details to the outside, it becomes a "black box". Two important advantages result:

- Changes made at one point of the system do not spread too easily. A class can change its implementation completely as long as it still conforms to its interface.
- It is easier to understand just the interface of an object without having to bother with the implementation details. The learning curve of a user of the class is flattened so she can start reusing the code faster.

The narrower the interface of a software entity is, the more encapsulated the entity is. The most narrow kind of interface is one where a parameter of a simple type is sent to the entity.
In general, parameter granularity can vary from simple values to structured types and even methods. Parameterization is therefore a powerful method to specialize software. The designer of a framework leaves 'holes' in the code which will be filled with parameter values by the framework user. The whole system can be assembled using only parameterization. In that way, parameterization can replace inheritance as a specialization method for software.
In software composition where the components are seen as blackboxes with a number of 'plugs', parameterization is used as a method to 'glue' the components together.

2.1.2 Software Composition

The research done in the field of *Software Composition* is attempting to further develop ideas stemming from object–oriented software engineering, namely reuse of software artifacts and design expertise, as well as flexibility and openness of applications faced with constant evolution. The vision of "component–oriented software technology" [ND95] is an application development environment where systems can be built much like cars or other hardware is developed: by assembling prefabricated components into a finished product along the rules and guidelines that have been established in the engineering history in the respective domain. In addition, software composition wants to pay respect to the special nature of software artifacts which sets them apart from hardware items: their theoretical infinite malleability. Software can be changed, extended, and adapted endlessly without necessary loss of quality or ability to perform its functions. Successful software systems with a long lifespan will undoubtedly be

[2] By server we mean the entity that provides a service to a client. This can be an object that supplies functionality via the methods in its public interface or it can be a superclass that provides its implementation via inheritance to its subclasses.

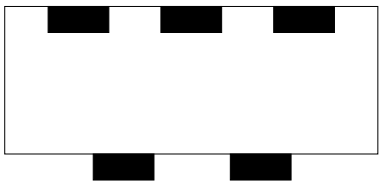

Figure 2.1: A software component and its polarity exhibiting plugs.

subject to changing requirements. Systems built from a set of reconfigurable software components can answer to this needs with a reconfiguration that is relatively easy compared to the approach of reprogramming parts of the system. This kind of reuse has naturally to be planned for from the beginning of the design of such an application. In fact, the software composition approach penetrates the whole range of technological, methodical, tool- and process-oriented aspects of software development.[3]

What is a component?

At the center of the software composition approach lies the notion of a *component*. In a component, the aspect of composability of an entity is emphasized over the computational aspect of the entity. A component is at first an entity in a framework which can be connected to other components and only as such, parameterized and linked to its surroundings, serves its computational purpose. The computational functionality of a component is abstracted away. 'Visible' from the outside is only the interface. A component therefore can be seen as a *"static abstraction with plugs"*[4] (see Figure 2.1). *"Static"* covers the aspect that we are considering entities that can be stored and retrieved so that their lifespan is independent from a runtime environment. The term *"plug"* refers to the ways that are defined to interact with the component (parameters, ports, messages, etc.).
The concept of an object (see section 2.1) can overlap with the concept of a component: sometimes an object is a component and vice versa. There are, however, differences between the two:

- Objects are entities of a fixed granularity. Components can exist at any level where composition makes sense. They may be for example macros, procedures, templates.
- Objects exist only at runtime, components exist at 'composition'–time, which includes run– as well as compile–time. They need not exist at both times, though.
- Objects are software entities. In a component framework, there may be also other entities like specifications or documentation that can be used as composable items.

In order to be able to work together, components can only be assembled if they adhere to a specific contract which settles the terms of their collaboration. If compositions are

[3] See [ND95] for a detailed and thorough exposition of the ideas presented here as an overview.
[4] [ND95, page 6].

to be checked, a type system must make the contracts explicit.[5]

Composition vs. programming

Software composition is the systematic construction of software applications from components. Components can be parameterized by a link to other components over *plugs* (see Figure 2.1). Such a plug represents functionality the server component provides for a client who establishes a link to that plug.
Composition means instantiating the generic component framework by parameterizing its components in a specific manner (according to a set of composition rules). In the composition process, the inner workings of a component need not be understood by the user: components remain blackbox entities that expose only their interface. Therefore, composition is situated on a higher level of abstraction than programming the component itself. This suggests that different skills are needed for programmers that work on the different levels.

- **Component engineering** means building up a component framework. A component framework is a domain specific expert system that incorporates knowledge on how to build applications in this domain. The knowledge consists of a framework of software abstractions that represent a generic application in this domain, but ideally also requirement models and guidelines on how to use components and generic designs.
- **Application development** means using an component framework to build an application for the end-user. This kind of development often happens under strict time and budget constraints. It is also on this level where requirements change frequently and the possibility for adapting a system as opposed to rebuild it from scratch represents the real benefits of component–oriented software technology.

The tools that the two type of programmers use will differ considerably. Component engineers will work with the languages the software components are implemented in, which will be normal 3GL like C++. The application developer however will probably work in quite a different environment, where languages like C++ are considered to be on the machine language level, and the experience of the developer has to be on the domain level instead.[6]

2.1.3 From object composition to class composition

In well–known component-oriented models such as Delphi or Visual Basic, compositions are directly mapped to objects at runtime. A link between two components in the composition then just means that two runtime objects interact in some way. A compositional element refers directly to a runtime object. Composition with such models is limited to specifying initial configurations of objects in an application. But one can give links between entities in a structure (i.e. components) a different semantic. The link, for example, between two classes can mean that instances of these two classes

[5] The question of what kind of type systems can capture the static and dynamic semantics of compositional links is still being discussed. See [ND95, pages 16–18] for details.

[6] This sentence is based on a statement of Ivar Jacobson, cited in [dM95, page276].

have the possibility of being connected [Mei96]. With such an extended link semantic, *classes* can be composed, where the composition of two classes means that they *can* have instances which cooperate. Such a composition does not have to say anything about actual instances being present at some given moment during execution time. In this way, *evolutions* of object compositions can be described, making it possible to compose dynamic applications. An example of a dynamical application is a structural editors for visual composition. The essential feature of an editor is to provide the means to build *arbitrary* structures.
If we want to be able to compose classes (in FACE, they are called *class-components*), those classes must themselves be represented in the composition environment by some sort of objects. They must become *first-class* elements of the system[7] which is achieved by making them instances of meta-classes. A class is then just like a normal object with additional instantiation and subclassing behavior. In traditional object–oriented frameworks, each framework comes with its own set of classes. Also in a FACE setting it should be possible to have specific kind of classes, and specific rules for composing them in order to realize specific frameworks.[8] This is done by having meta class–components describe classes in the same way as class–components describe objects. By specializing a general meta class we can then describe new classes. The meta level is thus used to specify or configure the FACE composition "language".
Meta levels are used in programming languages or systems to configure either the syntactic side of the language or the semantic part of a language or a system. In the theory of MetaObject Protocols techniques were developed to make language implementations modifiable by the user of the language. The work on Open Implementations concerns itself with a similar idea. Subject of interest are in this case any blackbox kind of software abstraction. An Open Implementation tries to introduce a special *meta level* interface over which implementation configuration can be done. This configuration is separated from the normal access to the server functionality which is done over the so-called *base level* interface.
I will give a short overview over the theories of metaobject protocols and reflection and how the FACE approach differs from these well–known ideas.

2.1.4 Reflection and Reification

The term *reflection* in computer science refers to a property of a computing architecture or a language design. A very general definition, taken from [Ibr90], states:

> *Reflection*: an entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter.

Reflection involves having different levels in the system: the domain of the *base level* deals with the outside world the system tries to represent. The domain of the *meta level* is about (some of the aspects of) the system itself. This implies that there is a representation of these aspects in the system.
Now, what kind of aspects of the system one could wish to reflect upon? In every computational system, certain things are made explicit and can be accessed at runtime, while others are inaccessible or *absorbed* [Ste94]. For example in a typical LISP

[7] the system being in this case the FACE "language" with which we are doing the composition.
[8] That is why FACE is called 'framework adaptive'.

system, the programmer has access to the program and the data structures he creates; however, the running program has access only to its data structures and not to a representation of itself. For both the programmer and the program, the LISP interpreter which runs the program is hidden, as well as other elements of the running system like the heap, continuations, local environments, etc.[9] In a language like C++, the language entity *class* is available for explicit manipulation and introspection only to the programmer. During execution, the program cannot access the class of an object; classes are not among the so called *First-Class Entities* of the language.
By *reification*[10] we mean, that such otherwise inaccessible structures are represented in some form on the meta-level. We then can access the representations, read information from them or even manipulate them. The manipulation of objects on the meta-level leads to the key question of reflection: Are the base-level and the meta-level *causally connected*? Lets say for example that we have a robot arm and a representation of the robot arm in a computer system. *Causal connection* means that the representation in the computer system always accurately reflects the state of the real arm and vice versa. When the arm changes its position, the representation is automatically updated. When we edit the representation of the arm in the computer system, then the arm moves to match the new coordinates. A reflective system where base- and meta level are linked in such a way that changes of one of them leads to corresponding changes to the other is called causally connected.
Practical interests of reflective systems differ with the models used. We can identify reflective models by the aspects they make accessible [Fer89]:

Procedural reflection: In procedural reflection,[11] some of the dynamic properties of the system are reified, for example how commands are executed. Programs are not run by an inaccessible interpreter but by another program which represents the interpreter. The user program can then switch deliberately between the two levels.[12] In this way, a procedurally reflective architecture allows the user to alter the interpreter behavior, for example by adding various runtime activities like monitoring function calls, tracing and debugging activities, or even extending the interpreter with new language constructs at runtime, etc.

Structural reflection: In structural reflection, the static parts of the language are reified. For example, the above mentioned *class*–construct in C++ gets a runtime representation. This representation can be accessed and manipulated, which possibly affects the set of instances of the class. As we will see in the detailed discussion of the FACE-data model, making classes explicit consequently leads to explicit meta classes as well. The possibilities of these meta- and metameta-structures lie mainly in the extension of the language. Structurally reflective languages do not need to be executed in a procedurally reflective system [Fer89], one can build normal interpreters that interpret these structures.

It has been stated that object–oriented technology is particularly well suited for implementing reflective facilities [KdRB91, Ibr90]. It seems natural that a reified abstraction

[9] Example taken from [dR88].
[10] To *reify* means to 'thingify': abstractions are treated as material things.
[11] also called *computational* reflection.
[12] The interpreter on the meta-level is written in the same language as the programs at the base level. If this is not the case, a running program cannot switch levels. See [dR88] for a thorough explanation.

takes the 'form' of an object. In an object–oriented language, the meta level functionality can be distributed among a set of individual meta objects which enables users to change meta level elements independently and incrementally. The object models of some object–oriented languages (e.g. Smalltalk) incorporate reified classes and meta classes right away.

2.1.5 MetaObject Protocols and Open Implementations

MetaObject Protocols (MOP's) are an application of reflection to programming languages. The meta level in a computational system allows programmers to "look behind" the curtain[13] of implicit features of the system and even adapt those features to specific needs the designers of the system couldn't and wouldn't anticipate. There are currently two approaches under investigation, which differ in the degree of details used in the representation of the inner workings of the system.

MetaObject Protocols: MetaObject Protocols (MOP's) are mainly used for language design. The implementation of the language is structured as an object–oriented program. The interactions between the meta objects are encoded in a communication protocol (the meta classes that implement the meta objects and their interaction shape a framework). The MOP controls what language constructs are available (how programs may be written) and how those language constructs are mapped to their semantics (what the meaning of the program is). Using the MOP, the programmer can then become more of a language designer and adapt the language implementation to suit his needs. Because of the subclassing and polymorphic features of the object–oriented implementation, he can use arbitrary code to do so, as long as it conforms to the protocol.[14] MetaObject Protocols can vary in what they reify. The CLOS MOP of [KdRB91] is an example where a large part of the language is made accessible to the user.

Open Implementations: Systems other than language implementations can have a kind of a metaobject protocol too. Here, instead of allowing to adapt an implementation in arbitrary ways, a more declarative interface[15] is used to let the user make decisions about the implementation. The range of choices is fixed though. Different than in metaobject protocols, Open Implementations do not allow you to edit how your programs can be written (or how your system can be used), but they let you choose (out of a fixed set of possibilities) what your programs (or your system handling) *means*. Open Implementations are therefore less powerful than metaobject protocols, but they have the advantage of being easier to understand and having less trouble with overhead and efficiency concerns.

In both cases the programmer can use the system on two different levels. On the

[13] Kiczales et. al. use the metaphor of a stage in [KdRB91]: The show being performed on stage stands for the language entities and mechanisms as they are presented to a user, while backstage the meta level exposes the inner workings of these language mechanisms.

[14] It is of course possible to change even the protocol if one is willing to invest the appropriate amount of effort.

[15] "The term 'interface' is used here to refer to any documented means of accessing computational functionality. This is a broader notion than a procedural interface. For example it includes the kinds of commands that one can put into a makefile." [Kic94]

lower level, the basic functionality of the system or of the language is used. On the higher level, the implementation of the functionality is manipulated.[16] The programmer can switch between levels deliberately, writing programs where base level code is interwoven with meta level code.

FACE and Reflection

Basically, the FACE model is an object–oriented data model featuring types, objects, instantiation, subtyping and message passing. Reification and reflection are the techniques used to add genericity and extensibility to the model.
First, reification is used in FACE to turn relationships between objects into first class elements of the language. A relationship in FACE is represented by an object which belongs to the object which is the origin of the relation. A relationship object thus also has a type of which it is an instance, but it cannot exist independently.
With these two elements, the components and the relationships, FACE provides a generic composition model. This model is described using a structurally reflective meta level–architecture. This means that FACE provides meta objects describing the static parts of its language: the objects and their relations. Note that none of the dynamic aspects of object-orientation are reflected on the FACE meta level: inheritance, instantiation and message passing[17] mechanism remain implicit in the language and cannot be changed by the user FACE.[18] The FACE meta objects offer *introspective* access — which means that we can ask them for information about themselves (and thus about their instances) — as well as *intercessory* access — which means that when changing their description we also change the objects that are instantiated from the description. This is illustrated by the following properties of the language:

- **Genericity of the model**. Everything in FACE is an object and present at runtime. Because this is also true for classes, every object has access to its class and can ask it for information (introspective access). This makes it possible to keep the object very generic and hold specific information in the class. This information can be accessed by the object at runtime to adapt its behavior. See [ME96] for an example.

- **Extensibility of the language**. The meta level of the FACE model describes a general composition model. By specializing (subtyping) the meta classes, the user can create a specific composition meta-model (intercessory access). The possibility of introducing specific types allows the model to be adapted to the classes of a certain framework.

The FACE meta level–architecture is described completely using structural reflection. A class is instance of a meta class and a meta class is also instance of its class until in

[16] This distinction between base level and meta level is mirrored in the FACE environment by the idea of two kind of programmers: one using the meta level interface to adapt FACE to a framework and the other using the adapted language to implement applications in the framework.

[17] Although operations can be reified in FACE as well, they are about *what* an object does and not *how* it behaves. The goal of reifying operations in FACE is again to make applicability explicit, which emphasizes the structure of object/operations links. See [McA95] for a meta level architecture which reifies the dynamic aspects of an object–oriented language, i.e. message sending and receiving.

[18] Of course, changing the interpreter is always an option, although it requires knowledge that goes beyond what a simple user of FACE can be expected to learn.

the end the highest meta class is instance of itself (self-description). This makes the FACE language totally open for change.

Comparing FACE and the CLOS MOP

If we compare the FACE meta level–architecture to a metaobject protocol like the CLOS MOP,[19] we see some similarities but also some differences. Both, the FACE meta model and the CLOS MOP define a 'language'. The various elements of this 'language' are reified via meta classes. MetaObject Protocols use structural and procedural reflection to make static as well as dynamic properties of the language accessible to the user. In FACE, on the other hand, only structural parts of the language are reified. In both cases, the semantics of the language can be changed by changing the implementation of the meta classes. In the CLOS case these implementations are performed using CLOS itself, in FACE, however, implementations can be done in any suitable hostlanguage the system happens to be implemented in.
The CLOS MOP reifies a general purpose programming language almost entirely. This makes it possible to program CLOS in CLOS. There are, however, parts in the language that are not reified.
The meta level–architecture of FACE, in comparison, only describes a real simple blackbox configuration 'language'. We therefore cannot use FACE for implementing the compositional and the corresponding reflective behavior but need a hostlanguage for that purpose. On the other hand, since FACE is self-descriptive, every structural element of the language is reified.

We can further compare the FACE approach to the MOP approach on two goals that one wants to achieve with either technique:

Extensibility of the language: In both systems, we are able to specify the structure of the language by accessing the reified objects on the metalevel. The access to the meta level has to follow a structured approach in both cases.

Giving semantics to the language: This is done differently in the two systems. In CLOS, the semantics are given by using the same interface as before when changing the structure of the language. In the FACE language, semantics are currently[20] changed by going to the hostlanguage level and adding new classes and methods.

As a main point of the difference of FACE and the CLOS MOP, we state the following.

- CLOS has a MOP that describes a complete general purpose programming language. This means that the whole language implementation is opened up, including dynamic features like inheritance and instantiation. This results in the MOP being very detailed and thus complicated.

[19] The CLOS MOP of [KdRB91] is taken as the reference metaobject protocol because of its exemplary character and since it is the most widely known.

[20] No further support for adding semantics is yet provided in FACE. This does not mean that this would not be possible, just the work done up to the point where this thesis was written consisted mainly in building the syntactic side of the language. It reflects however the intent of FACE, that support for the application developer should be provided on basis of the blackbox concept.

- FACE on the other hand provides a special purpose MOP which has a clear focus on its problem domain, namely the compositional aspects of software systems. This makes the MOP less complex. Another abstraction or simplification results from the separation of the implementation of the FACE language and its purpose, the configuration of software systems.

2.2 Implementational background

In this section we are going to introduce some of the methodologies and tools that we are going to use in the implementation of the object model.

2.2.1 Generic Programming in C++: Templates and STL

Generic programming in general

A generic solution to a programming problem, e.g. a function, is a solution that can be adapted to elements of different types. Generic software components (e.g. classes, operations) are components that can be parameterized with a type and can thus be used with a range of types.
Generic software is written in terms of operations that are applicable to all of the types the software is going to work with. For example a generic minimum function `min(T a, T b)` that uses the less-operator `<` provides template code that can be instantiated for every type that defines the `<` operator.

Templates in C++

Templates make genericity over data types possible in C++. The template mechanism was introduced into the C++ language by the ANSI C++ standard committee (X3J16) in Juli 1990. A *class* template defines the layout and operations for an unbounded set of related types. The types, which are parameters to the class templates, must all have a set of operations in common[21] in terms of which the methods of the classes are written. Templates often are used to write general container types such as lists, stacks, or arrays. They are then instantiated with a specific type, creating stacks holding integers, stacks holding characters or stacks holding items of any built–in or user–defined type.
The templates in C++ are realized in the following manner: the class template written by the programmer is used as a macro which is expanded by the compiler whenever the template gets instantiated with a specific type as actual parameter. The instantiated class template is then compiled normally. This means of course that on the one hand the programmer saves space writing less source code but this advantage is confronted by a sometimes surprising 'code bloat' in the executable.

[21] Although, no restrictions are enforced on types that match a given type argument. This gives the programmer a lot of flexibility (as always in C/C++) but has the cost of errors — like trying to sort a type which has no comparison operator — being detected but at link time.

```
template <class T>
class vector {
   ...
};
```

```
void foobar()
{
  vector<char> x;
  ...
}
```

Figure 2.2: The definition and an instantiation of the `vector`-class template of the STL.

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1,
                     InputIterator1 last1,
                     InputIterator2 first2,
                     InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

Figure 2.3: The declaration of the generic `merge`-algorithm in STL. Note the different types of iterators that are required.

A collection of reusable, generic data structures and algorithms: The Standard Template Library

The Standard Template Library (STL) [MS96] is a general purpose library providing a rich set of containers and algorithms designed to work together. The library has two dimensions: algorithms and data structures. The data structures that are provided are a small set of of well known containers, such as vectors, lists, sets, and maps. The algorithms cover a broad range of the most common kinds of data manipulations, such as searching, sorting, and merging. The separation between these two dimensions or categories of STL is very strong, as the algorithms only know of the data structures in terms of abstract access methods, so–called *iterators*.[22] The specification of such an iterator type includes also time complexity aspects which enables writing different algorithms that have to meet individual time efficiency requirements by changing the specification of the iterators used. Genericity is achieved because the algorithms work on any container[23] that offers the iterator–access required by the specific algorithm. In other words, generic algorithms and containers can be 'plugged' together if they are 'plug–compatible'. The adaptability of the algorithms leads to increased friendliness towards software reuse: it becomes simpler to adapt than to program from scratch.
STL is implemented in C++.[24] In this language, the generic approach of the library is supported by the C++ template mechanism. An STL container is a class template which takes the data type of the containers' elements as its template–argument (see Fig. 2.2). The generic algorithms in STL are C++ function templates (see Fig. 2.3). They

[22] An iterator is a generalization of a standard C pointer. An iterator represents a position in an abstract data structure in the same way as a pointer represents a position in memory. STL provides a set of several of these pointer types, each having a very well defined behavior which is a subset of the behavior we associate with normal pointers.

[23] Naturally, containers that are not part of the library in the first place but are written by the STL user can also be taken as parameters for the algorithms.

[24] In fact, it has been made part of the ANSI/ISO C++ standard.

are instantiated by specifying concrete iterators that point at positions of a container. The compiler takes care of the implementation of the 'plugging', guaranteeing type correctness. It prevents certain combinations of containers and algorithms and ensures a efficient implementation of the algorithms.

2.3 The FACE data model

FACE is based on ideas from object–oriented data modeling. A *data model* is a model that describes how information may be structured. Object–oriented data models are an extension of the older technique of relational data models. Those models tried to map the structure of data in the real world into a model consisting of tables. The tables were passive data stores with all the functionality that manipulated the data kept in the separate application program. In a next step functionality, constraints and semantics moved from the application program to the data following the object–oriented principle of the encapsulation of data and functionality. This leads to domain specific modeling primitives (e.g. multimedia objects like video movies which not only store the data but also show it, convert it, print stills out of it etc.) where the user finds all the elements he needs for his specific domain. The FACE system is built using an object–oriented data model. A data model may also be described by analogy to the context of programming languages.[25]

1. A data model corresponds to the syntax and the semantics of a programming language, i.e. the grammar that defines *how* programs are written in that language and the runtime meaning of the grammar elements.

   In FACE the data model is defined through so called meta types. By using meta types we can specify what kind of domain specific modeling primitives there are. This set is however not fixed (see section 2.3.6 on page 38).

2. The programs written in the programming language correspond to a schema built using the data model.

   In FACE, the schema is used to model the intentional part of the application (see section 2.3.2 on page 30).

3. The running program corresponds to the instantiation of the schema, i.e. to the objects and relationships that form a structure according to the schema.

   In FACE, the schema defines what objects and what relations can be present in the context of the application.

In what follows, a little bit of the history of the FACE model is exposed, then a general view of the ideas behind the model is presented, and then an in-depth explanation of the model itself is given.

2.3.1 The history of the FACE data model

The history of the FACE system dates back to the YANUS system of [Mei93b]. YANUS stands for Yet Another Unifying System and it tries to integrate multiple applications and their data resources under one graphical user interface. The work on YANUS originated at the department for medical informatics of the university of Rotterdam where statistical research on population data or ECG's is performed. The different software packages that are used there are a database management system, a statistical package for analyzing data, image processing facilities and programs to do statistical pattern

[25] Analogy taken from [Mei93b, page 33].

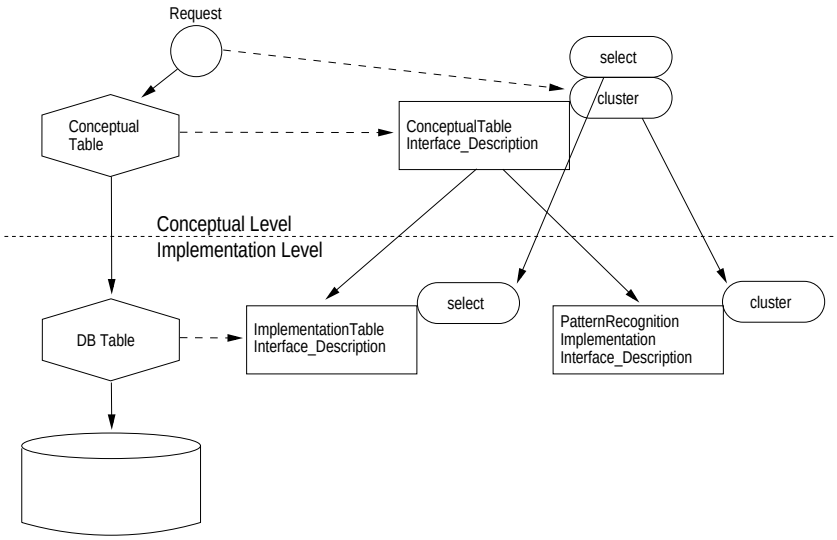

Figure 2.4: Example of a YANUS composition for describing a centralized integration between a database system and a pattern recognition package used to perform statistical information analysis.

recognition. All these packages are stand alone applications with different interfaces, requiring their input-data resources to be in different formats and producing output in still another format. Work in this kind of environment is done by applying a set of operations to data in a certain consecutive order, each operation probably requiring transformations of the data resources to a specific data format, forcing the user to change from package to package, from interface to interface. The idea of the YANUS system is to build a unifying interface on top of such a heterogeneous environment under which the application boundaries and the necessary 'housekeeping' activities would vanish and the user would see only one seamless environment. This was done by defining an object–oriented data model where data and request for operations are the instances. We can divide the data and request instances into two categories:

- Data and request instances that are presented to the user. They represent these elements 'conceptually', independent from underlying representation and implementation in software packages.
- Data and request instances that represent a corresponding implementation of data and the execution of a request in an underlying software package.

The user can create a request and link it to the data object to which it should be applied. By submitting the request, it is executed which corresponds to finding in which software package the request can be executed and then creating the corresponding implementation request; moreover the data has to be represented—and thus, if necessary, transformed—in the data representation which is used by the chosen software package (e.g. a special data representation for statistical analysis). In the terminology given above, the system creates the right implementation object.
The domain specific 'model' of the application describes how the interface that is pre-

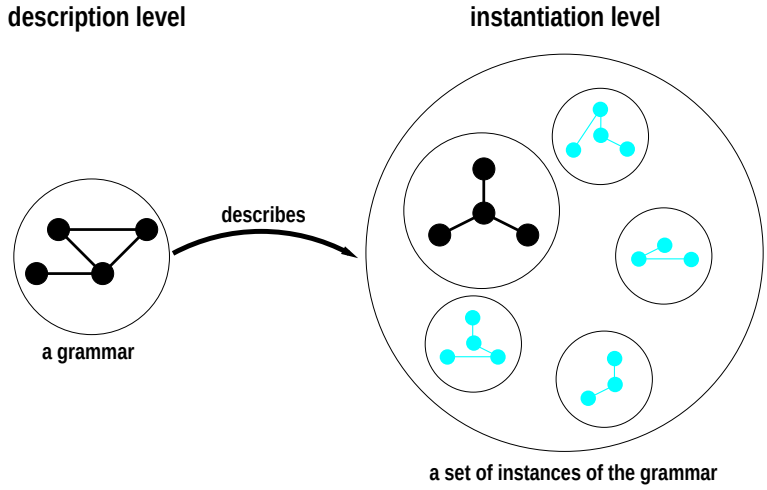

Figure 2.5: Object structures describing object structures.

sented to the user is related to the interface of the underlying packages. This is shown in Figure 2.4. The link between ConceptualTable Interface_Description and the two types on the implementation level, ImplementationTable Interface_Description and PatterRecognitionImplementation Interface_Description, show how the operation presented to the user corresponds to implementation operations as provided by the underlying software packages.

The idea of FACE is to use the principle of domain specific modeling primitives in the context of frameworks. The use of meta types already introduced in YANUS allows to define new modeling primitives. YANUS also shows how a semantic can be given to those primitives.

2.3.2 The philosophy of the data model

Syntax

The purpose of the FACE data model in the most general terms is to represent arbitrary object structures consisting of elements that have various connections among each other. In order to know for a specific structure what the rules are that govern its construction, a basic feature of the model is to have a separate *description* level 'above' the structure. The description level contains the rules which are basically telling which objects can relate. The rules contained on the description level form a grammar. A grammar does not describe one single object structure but a whole set of structures (see Figure 2.5). It can be checked if a specific structure conforms to the rules of the grammar. If it does,[26] we will speak of an *instance* of the grammar. In what follows, the level of the rules will be called *description level* and the level with the object structures described by the grammar will be called *instance level*.

[26] We are not interested in structures that do not conform to the rules so we are not considering them any further.

Now the important feature of FACE is that the grammar on the description level is again represented as an object structure. Even simpler, there is no fundamental difference between object structures on the instance level and object structures on the description level. On both levels the same construction mechanisms are used, and the objects on both levels have a common basis. The described and the describing are essentially *uniform*.
The obvious advantage of this uniformity is that the description level can have a description of its own in just the same way as the instance level. On the level 'above' the description level we find a grammar for grammars, the description of a general grammar structure, which can be instantiated to yield a set of specific grammars. Therefore, the description level can be adapted, which is the basis of the extensibility of the data model.
We have seen so far that the FACE model consists of three levels—an instantiation-, a description- and a *meta level*—which have all a uniform syntactical structure: objects connected in various ways.

Syntax and Semantics

The idea of FACE is, that an object structure has a certain runtime-meaning, that structures have semantic.To give semantics to an object structure, the structures—the objects, and their connections—have to be interpreted somehow. The uniformity of structures on the different levels of the model is a clue as to how independent the object structures are of their interpretation: the meta level has completely different meaning—being a meta-meta-description—as the instance level, but it has the same structure. The connection between syntax and semantics of the object structures in FACE, i.e. the interpretation, must therefore be implemented in a very flexible manner. The flexibility such an implementation has to exhibit implies openness to further semantic extensions, e.g. domain specific types that are introduced when adapting the model to a specific application domain.

Since we are building FACE to give arbitrary semantics to the object structures, when we introduce types that stem from another domain, i.e. not the domain of the FACE model itself, these types will have semantics that are added to their behavior as a FACE object. FACE then becomes sort of a scripting language[27] for these systems. How the FACE types are connected to the semantics of the foreign domain is not fixed. It can be done by interpretation but it can also be done by compiling the object structure, thereby loosing the FACE behavior. So far, only the interpretative approach has been worked out.[28] The appropriate method will have to be decided from case to case and further work is required to research the possibilities.[29]

In the following sections we will present the details of the data model in as much accuracy as is needed for the reader to understand the problems that have to be solved by an implementation. The details can also be found in [Mei93b] and [Mei93a], although the model has evolved in certain aspects in the time since those texts were written down.

[27] A scripting language is a language for controlling and composing components of a system.
[28] See the explanation of the FACE execution mechanism in section 3.6.1 on page 69.
[29] See [ME96] for an experiment on using an interpretative approach to model the domain of network protocols.

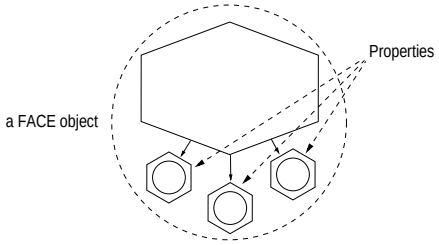

Figure 2.6: A FACE object with three Properties.

2.3.3 The basic element of the data model: the Object

At the heart of the FACE data model lies the notion of an *object*. We will describe how those objects are synthesized.

The structure of an object

An object is an entity present at runtime of the FACE system. An object can have an arbitrary number of properties, among them for example a 'Name' Property, i.e. the Property that contains the name of an object. Each property contains values of a specified kind, in our example this would be a string. The identity of the objects is independent of the values of its properties, so for example changing the name of an object does not change the object, and other objects still refer to the same object.
An object refers to one type of which it is an instance. This means that the type describes a set (or class) of similar objects out of which an instance is one specimen. The information the FACE-Type holds about its instances concern the structure of the object as well as the operations that are applicable to the object (for details on the type–instance relation see chapter 2.3.4).

Properties

An FACE-Object has a certain number of Properties. A Property has a name and can be accessed through this name. Each Property has a Property Value which can hold one or more elements of a specified type, called the Element Type of the Property. These elements can either be references to other objects or they can be "simple" values from different domains like integer numbers, booleans, strings, etc. A Property is described by a Property Descriptor in terms of Element Type, number of elements that can be held and other descriptions (see section 2.3.4 on page 33).

Associations

Associations between types are higher level abstractions that represent structural relationships between instances of those types. Higher level means that associations have a certain semantic the goes beyond one object referencing the other. A relationship between objects can be called an instance of an association between the types of the

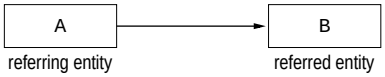

Figure 2.7: A binary link (A,B).

two objects. Such a relationship is realized in FACE using one or multiple links. A link is, theoretically spoken, a tuple of object references, e.g. a binary link is a pair of references (A,B) (see Figure 2.7). We call the first element of the pair the *referring* part and the second element the *referred*. The referring object is the distinguished part of the relation which establishes and destroys the relation. A link is modeled in FACE by putting a reference to object B in a Property of object A. Using links we model the various associations in FACE. All associations in FACE are binary. There are three different kinds of associations:

- The simplest association is a unidirectional Reference where *unidirectional* means that only the referring side is aware of the relation. The association has no additional semantic.

- The Component association is a bidirectional reference. Both participants of the relation are aware of it, i.e. the referred object has a so-called *backreference* to the referring object. In a Component association, the referring object owns the referred object. This means for example that the lifetime of the referred object depends on the lifetime of the referring object.

- The Cross-Reference association is again bidirectional. Cross-references are used to prevent dangling references since before destroying an object B, the back-references of B can be used to notify objects that refer to B to abolish the relation. Cross-references have their name since they essentially refer *across* the acyclic tree of the component structure established by component associations.

2.3.4 Types

The object as described by its type

Every object refers to a type from which it is an instance. Through this reference the object has permanent (at runtime) access to its type. The type describes its instances in terms of their properties. This means that the type specifies a list of all the properties that its instances must have. The list of properties is in fact a list of references to Property Descriptors [30] (see Figure 2.8). At the moment of the instantiation, the object is formed by a routine that creates a Property for each Property Descriptor in the list of the type.

[30] Those Property Descriptors can be reused by other types. We therefore can say that *composition* is used to create the types.

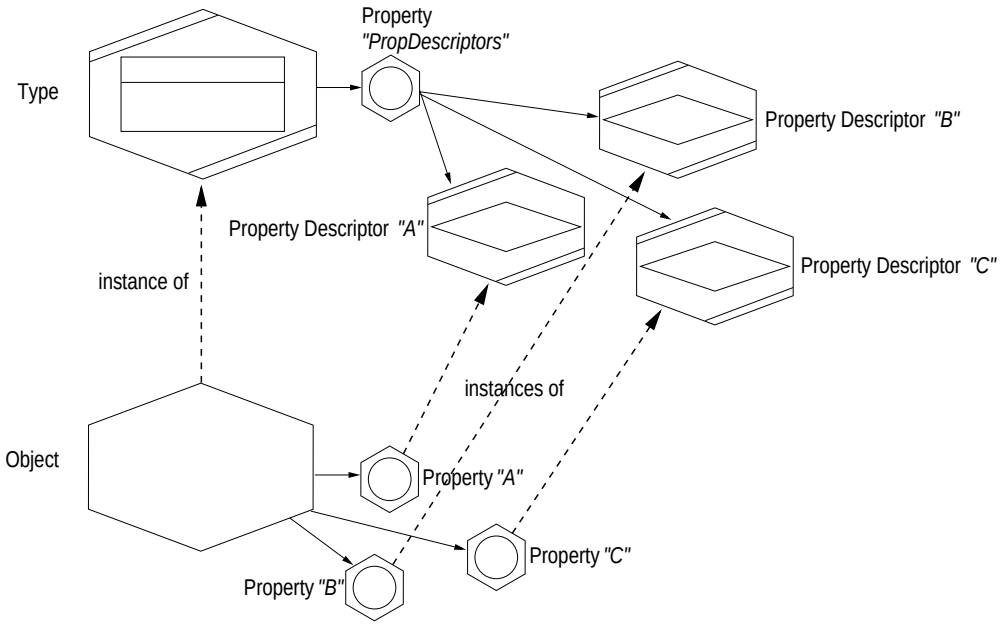

Figure 2.8: A type describing an object through Property Descriptors.

Property Descriptors

A Property Descriptor is a special type object which describes Properties. A Property Descriptor is structured with a fixed set of slots, each with a specified meaning. These slots, which are realized as Properties like in a normal object, can be parameterized so that the Property Descriptor completely describes the corresponding property. Some of the slots of a Property Descriptor, i.e the categories in terms of which the Property is described, can be found in the following list:

Elttype: Determines the type of the property values. This property enables the type checking mechanisms (see section **Type checking and Validity** on page 35).

Minelt: Determines the minimal number of elements the property can have. Together with Maxelt, Minelt controls the arity of the relationship.

Maxelt: Determines the maximal number of elements the property can have. Maxelt must be equal or larger than Minelt.

Through property descriptors the type exerts control over the *structure* of the object: it determines the number of the properties and some aspects of the property values in prospect. As mentioned in section **Associations** on page 31, Property Descriptors are used to represent the associations between types in the model. In the YANUS model,[31] which was formalized in [Mei93a], there was a range of different types for Property

[31] This section about the outdated categorization of Property Descriptor types in YANUS and how they translate into the FACE model is part of this text since the transition and the reasons for it are documented nowhere else.

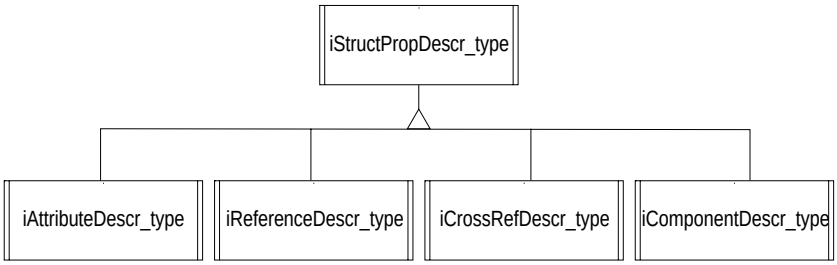

Figure 2.9: The different categories of Property Descriptors in the YANUS model.

Descriptors (i.e. meta types relative to a Property, see Figure 2.9). Each of the types represented a different relation:

iAttributeDescr_type: Property Descriptors that are instances of this meta type have basic values as the Element Type of the Properties they describe. Since basic values do not have the status of objects, elements of an attribute-Property cannot be called relations. This implies that there is no special semantic attached to these Property Descriptors.

iReferenceDescr_type: Property Descriptors with this type represent a unidirectional relation. An object B is referenced in a Property of object A without being aware of it. This type does not have any special semantics.

iCrossRefDescr_type: Property Descriptors with this type represent relationships that are bidirectional. Establishing a link from A to B under this relation results in a backlink from B to A. This link would be set automatically in a Property of B called RefObjs.

iComponentDescr_type: Property Descriptors of the iComponentDescr_type describe component relationships. An object B that is referenced in a component Property of A automatically gets a link to A in its Property SuperObj. If A would be destroyed, B would be destroyed also.

The type of the Property Descriptor of a Property accessed with a write operation helped to decide, which additional semantics had to be respected (see [Mei93a, pages 46–50]).
The FACE data model has evolved from this older ideas. In FACE only the meta type iStructPropDescr_type remained. Two new Properties were added to the Property Descriptors which let us express the same as before in a more declarative way:

Ownership: Determines if the referred objects are owned by the object the Property belongs to. Owned objects are destroyed when their owner is destroyed and are copied when their owner is copied.

BackRefPropDescr: Determines which Property in the referred object is used to link back to the referring object. If the value of the BackRefPropDescr is set to `NIL`, no back link is established.

We can model the relation between a type and its subtype for example by setting Ownership to `false` in the Property Descriptor SubTypes and BackRefPropDescr to Supertype. This would result in a cross-reference relationship where the supertype refers to its subtypes via the Property SubTypes and the non-owned subtypes automatically refer back to their supertype through the Property Supertype. This achieves the same semantic as did the Property Descriptors of type iCrossRefDescr_type in the YANUS model.
The main advantage of the new model over the old is that in the case of the BackRefPropDescr it allows to specify individually for every relation where the back link should be set. In the old model, the back link for a subtype relation would have been set in RefObjs together with all the back links of the other crossreference relations.

Type checking and Validity

Type checking is a means to ensure that objects are parameterized and composed in the right manner. Before two objects A and B are related by referencing the object B through a property p of A , the system must check if B has the right type, which it does by looking at the Elttype property of the Property Descriptor of p.

An object is only legitimate when it refers to a valid type. Only then can type checking be applied. The *validity* of an object tells how good an object satisfies the description of its type. Validity is divided into three levels, each higher level encompassing the requirements of the lower levels. On the first level, which Properties the object has conforms to the property list, or the template, of its type. On the lowest level of validity the object is still labeled *invalid*. On the second level, the values of the properties fulfill the syntactic requirements of the type's property descriptors, like being of the right type and the right arity. The object is said to be *syntactic valid*. On the highest level, the object conforms to prescriptions of a set of extra requirements[32] each object type has. An object which satisfies the extra requirements of its type is labeled *valid*. See pages 46-50 in [Mei93b] for a detailed explanation.

Types and Meta types

In the FACE data model, types are also objects. They are therefore also described by types of their own, the so called *meta types*, which, being types themselves, again have to be described. This seemingly never ending recursion finds its fix point in a meta-meta type which describes itself.
The relationship between object and type is found throughout the model, and because everything in the data model is an object, everything has also a type. The model is thus totally self-descriptive.

Types come in different categories:

- **Object types**
  Object types describe objects. Instances of object types are FACE objects.

[32] *Extra requirements* are predicates that range over the property values. Extra requirements will be used to prescribe relations of different property values, for example the value 9 of a hypothetical property Month has the consequence for the value of the hypothetical property Date not to exceed 30. See [Mei93b, page 49].

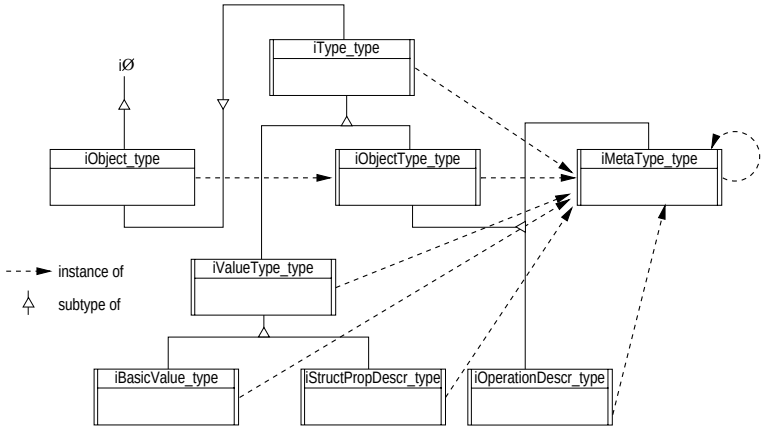

Figure 2.10: The FACE kernel (partial view).

- **Value types**
  Value types describe values that can be elements of object properties. There are two sorts of value types:

  - **Basic Value types**
    BasicValue types describe elementary data elements of such domains as integers, booleans, characters, strings, etc.

  - **Property Descriptors**
    Property Descriptor are used to describe properties of objects. An instance of a Property Descriptor is a property object.

- **Type types**
  Type types describe types. An instance of a type type is a type (therefore, type types are meta types).

Supertypes and subtypes

Between instances of the same meta type, therefore between types, there may exist a supertype/subtype relationship.

A subtype *inherits* the lists of Property Descriptors and Operation Descriptors of its supertype and incrementally modifies them. That means that the subtype can add new values to the set of inherited values, or that it can modify some of those inherited values. The instances of the subtype are then also instances of the supertype because the set of their properties is a superset of the properties of the instances of the supertype.

Each type has one (and only one) supertype which is referred through the property Supertype. If the property is empty, the supertype is the *empty type* i∅.

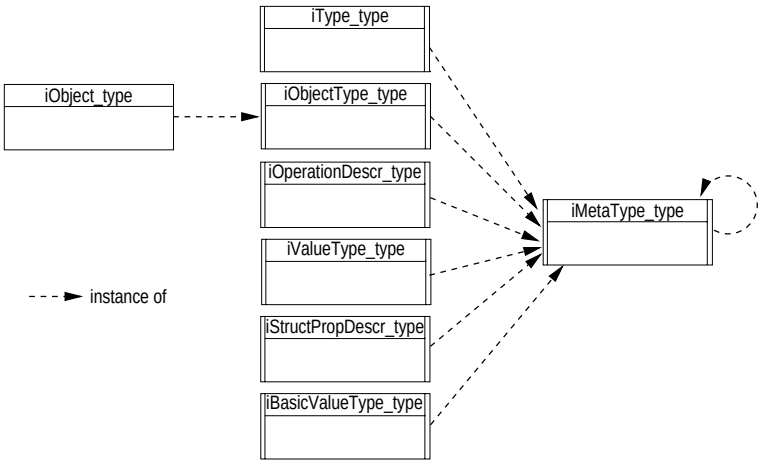

Figure 2.11: Part of the instantiation hierarchy of the FACE kernel.

2.3.5 The kernel of the FACE data model

The kernel of the FACE data model contains the descriptions or types of all the basic structure and elements of model itself. Because the model is self-descriptive, the types in the kernel are simultaneously elements of the model *and* descriptions of the elements of the model. Thus the kernel inevitably evolves into a set of objects that have multiply interwoven instance- and inheritance relationships. (see Figure 2.3.5)
To give an overview over the kernel structure which is clarifying rather than confusing, we split the depiction in two parts, emphasizing the aspect of "Who is instance of whom?" in the one drawing and the aspect "Who is subtype of whom?" in the other drawing.

The instantiation hierarchy

The instantiation hierarchy in Figure 2.11 shows the relationship between the types and their instances which are part of the kernel. The most remarkable thing is that the meta type iMetaType_type is an instance of itself, thus ending the infinite recursion of description.

The inheritance hierarchy

The inheritance hierarchy in Figure 2.12 shows the relationships between the supertypes and the subtypes in the kernel. At the top of the hierarchy we find the type iObject_type which describes the basic behavior of an object. All the other types and meta types are subtypes of iObject_type, which means that everything in the model is first of all an object.

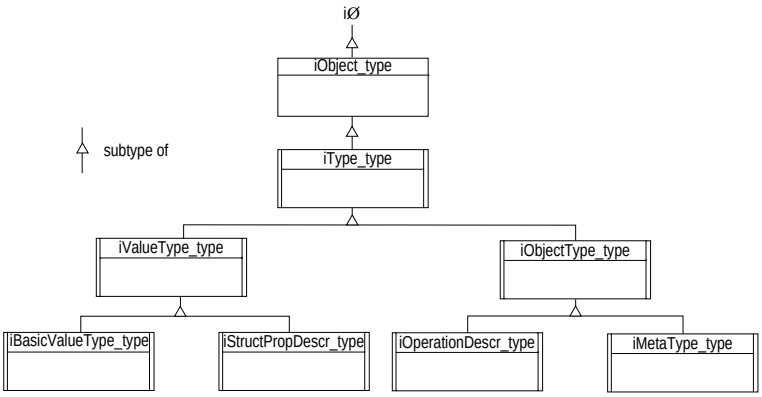

Figure 2.12: Part of the inheritance hierarchy of the FACE kernel.

2.3.6 The Extensibility feature

The following quote defines extensibility for programming languages. Because FACE can be seen as an programming language as well, the definition can be used in the case at hand.

> Let us define an *extension* to be a set of definitions which augment a language with an entirely new facility that can be used in the same way that preexisting facilities are used. ... Any time we define a new object, a new function, or a new data type we are extending the language. Each such definition extends the list of words that are meaningful and adds new expressive power. By building up a vocabulary of defined functions and/or procedures, we ultimately write programs in a language that is much more extensive and powerful than the bare language provided by he compiler. ([FG93, page 101])

Extensibility is an essential feature of the FACE data model. Thanks to it we can adapt FACE to describe systems of a wide variety. This is done by extending the FACE kernel with special types which describe, for example, the classes of a framework. The very general FACE kernel model thus gets specialized to a specific data model which lets the user then model exactly those systems that are realizable with the framework. The self-description of the model offers a flexibility for extensions which is remarkable: Because every item of the model has also its description in the model, and these descriptions are accessible to the user—this means that the user can derive specialized versions from the descriptions—we can specialize every item of the model and, theoretically, build a completely new model.

The specialization interface

The FACE data model, being object–oriented, offers the possibility to create new types by *deriving* from older types. Derivation first means that the new type is determined in certain aspects by the old. The main reason to derive is of course to extend the old

type in the process, so the derivation can be influenced by the user over a so called *specialization interface*. This interface enables her to incrementally change the new type with respect to the old.
Specialization is called *subtyping* in the FACE data model [Mei93b, page 50 ff]. The type that is going to be specialized is called the *supertype* and the specialized type is called the *subtype*. A subtype inherits the values of two properties of its supertype: the value of PropDescriptors and the value of Operations. PropDescriptors determines the structure of the instances of the type as it contains a list of all the properties.[33] Operations determines which methods are applicable to the instances of the type as it contains a list of all the operations.[34] The specialization can now be applied to those two lists and their elements. It has two possible dimensions:

1. *Horizontal extension:* New[35] Property Descriptors are added to the subtype's list, so that instances of the type have more properties compared to instances of the supertype.
   These Property Descriptors are added as a link to the OwnProperties–property.

2. *Vertical modification:* Specific elements of the list of Property Descriptors are changed in certain restricted ways. For example, the number of elements that the property can hold may be (further) constrained.
   Vertical modification is realized by substituting the link to the 'old' Property Descriptor with a link to the more constrained Property Descriptor, which must be a subtype of the old Property Descriptor.

Specialization in FACE is thus done by composing or re-composing a set of Property Descriptors. The specialization interface is well defined and allows only a small set of operations to be performed. Still, it can be controlled down to a very detailed level, i.e. the user can influence the creation of new types down to the basic elements of the structure and the behavior of the objects.

[33] In fact, the list holds the types of the properties, the Property Descriptors. See section **The object as described by its type** on page 32.

[34] Again, the list holds the types or descriptions of the operations and not the operations (the instances of the operation descriptors) itself.

[35] *New* respective to the list of Property Descriptors of the supertype of which the type in question is a specialization.

Chapter 3

Implementation Issues

In this chapter, we are going to present some key problems that had to be solved in the implementation of the FACE data model.
The implementation work performed could follow in large parts the formalization of the data model in [Mei93a]. The problem was not so much to find a correct algorithm to implement the ideas of the model but more to find the means in the chosen language to translate the formalization adequately.
The main problems that arouse during the implementation where therefore of pragmatic nature,[1] although in the course of the implementation some problems with the formalization were discovered which led to adaptions of the formalization.

3.1 Road map for this chapter

This chapter consists of 5 sections, each treating a separate problem that emerged during the work. Each section starts with a description of the problem and then goes on to describe the work that has been performed.
The sections are loosely connected as is the case of the problems that were treated. The order of the sections follows the construction of the system, starting with the low level parts and gradually building up.
The sections are the following:

- The basis of the FACE model: the system of basic types. How was it implemented on top of the type system of C++? (See section 3.2 on page 42)
- The cornerstone of the FACE model: the FACE-Object. How was the basic object with attributes and interface built? (See section 3.3 on page 45)
- How are FACE-Objects and types mapped to C++ classes? Which basic C++ classes are needed to implement the functionality of specific FACE-Objects? (The set of basic types is called the kernel of FACE) (See section 3.4 on page 54)
- The bootstrap. How does FACE perform its lift-off? (See section 3.5 on page 65)

[1] As a subtitle of this work, “Teach yourself C++ in 8 month” was shortly in the discussion.

- A small example. How can we use FACE to model software? (see section 3.6)
- Last but not least: Who killed John F. Kennedy? Finally the answer. Discovered accidentally while ensuring `const`-correctness of the `faceObj`code. (See page Ψ)[2]

Since the chapter is mainly organized around the different subjects mentioned, we will the different sections will be a mixture of technical documentation and discussion of the contribution.

[2] HaHaHa, very funny!

3.2 Establishing the system of domains and types

In the FACE object model we work with typed items. All elements that appear in the context of the model either stem from a specific domain of basic values or have a type of which they are an instance. Types are introduced to be able to check and support the user in creating correct compositions. Using types FACE can control if values of the right kind are being entered or operations are applied to the right objects.

In the following section we explain how the formalization of the YANUS model introduced domains and types and to what extent we created a representation of the system in our implementation.

3.2.1 The formal basis of the domain system

The YANUS model establishes one main dichotomy of the elements that appear in the formalization. This is the one between

- basic values from different 'primitive' domains,
- and elements from a set of object identifiers.[3]

The primitive domains[4] include generally known sets like that of all integer numbers ($\mathbb{Z}$), that of all boolean values ($\mathbb{B} = \{true, false\}$), that of all strings ($STRING$), and also some other domains that are more specifically tailored to the YANUS model like the set of all possible values that indicate the validity status of an object ($VALIDITY = \{invalid, syntactically\ valid, valid\}$). For each primitive domain in YANUS, a type object is introduced which represents that domain [Mei93a, page 25-26, Definition 23a].

The set of all object identifiers ID[5] that is defined in the YANUS model is structured by a system of subsets. Each different subset of ID is defined as the set of instances of a specific YANUS type [Mei93a, page 28, Definition 24]. The set $IMT \subset ID$ for example is the set of all instances of the type iMetaType_type.

Additionally, the YANUS formalization introduced some domains that were unions of the primitive sets. The union of all primitive domains is the set D ([Mei93a, page 5, Definition 1]). The set EV of all the values that can be elements of Property Values is defined like follows:

$$EV = D \cup ID \cup NIL$$

D and EV are not only used to define the domains of specific Property Values but consequently also for the definitions of the domains of functions that access the Properties of an object. Note that a function that has a parameter $ev \in EV$ is defined for all elements out of $\mathbb{Z} \cup \mathbb{B} \cup \cdots \cup ID \cup NIL$.

[3] A third category is actually defined on page 6, Definition 3: It consists of the set NIL which contains only the element Nil.

[4] See [Mei93a, page 5, Definitions 1-1b] for details.

[5] See [Mei93a, page 6, Definition 2].

3.2.2 Transferring the domain system to C++

When implementing the FACE data model, we mapped the domains of the formalization onto the type system of our hostlanguage C++. The correspondence between the formalization and the implementation preferably should be as close as possible, since the formalization will be a guide for the implementation also in other areas. The following three tasks result:

- Map primitive domains to C++ data types.
- Create a representation for object identities.
- Build support for the union of domains, D and EV.

The goal of the of the union data types D and EV is to support multiple data types in one interface. C++ is a statically typed language, meaning that the types of variables and arguments to functions have to be known at compile time. A function has to be declared with the exact types of the arguments it is going to get which is not possible when the argument can be of a data type out of a range of types.

Implementation of primitive domains

The mapping of the primitive domains is very straightforward. Some examples:

- $\mathbb{Z}$ was mapped to the builtin C++ data type `long`.
- $\mathbb{B}$ was mapped to an enumeration set `bool = {true, false}`.
- $STRING$ is represented by a user-defined class `string`.

Implementation of Object Identifiers

The requirements for the set ID and its elements are the following:

- ID is a countably infinite set of symbols.
- For each FACE-Object that is created, one object identifier is claimed from ID. When the object is deleted, the identifier is given back to ID.
- Via its identifier an object can be referenced, i.e. retrieved from the set of all objects currently in existence.

We mapped the set ID to the set of all C++ pointers, i.e. an object is identified by the memory location its storage space starts at. Note that we do not move the objects to a different location in memory after they are created. The management of free store done by the implicit C++ runtime system thus models the identity concept well enough (except that the storage space is limited), so no further work on our side is required to let the C++ pointers have the semantics of FACE object identifiers.

Implementation of the union sets

The sets of data types D and EV are used to treat different types of a common category at once. This is useful also for an implementation so we reproduced the type EV. Using such a union type we can pass objects of different data types around under the hood of a single type. This helps in keeping the amount of code that has to be written to handle the data of the different types small. Instead of writing a function for each of the different data types we know as little as possible about the types of the data objects until we really need to be exact. At that moment, we convert the union type to the true type of the value.

A class `SimpleType` acts as a wrapper for the different domains or C++ data types, including the built-in types `long`, `char`, `bool` and also the user-defined data types `String` and `faceObj`, the base class for all FACE-Objects (see Figure 3.6 on page 58). An instance of the class `SimpleType` acts as a variable which can have values of different domains. Such an object keeps one element of an arbitrary data type using the `union` construct [ES90, page 181] to save space. A tag identifies the type of the element being kept in the object so type information is available in the object itself. The object is initialized with a specific value (and a specific type) by a constructor for each of the different data types. Once created, the value of the object can change, the data type of the values kept in that particular object however cannot. Conversion functions [ES90, page 272] are used to transfer the value of a `SimpleType`-object into its proper format.

Discussion of the implementation

The implementation of the object identifiers may not be optimal in the long run. A redesign would probably necessary when implementing a permanent object store. At the moment of saving an object it is moved from memory to a different storage medium and thus pointers to memory locations become invalid as identifiers.

The problem of hiding data types behind a generic mask to save code will return when we implement the FACE-Object (see next section). The solution we choose in this case moves type checking from the hands of the C++ compiler into the hands of the programmer since the programmer has to do the right downcasting. This means more efficiency in space but also more overhead at runtime (since type checking is no longer done at compile time but at runtime). The tradeoff is in favor of our solution since efficiency concerns are for the moment at lower priority for the FACE system.

3.3 The implementation of the FACE-Object

The FACE-Object is the basic entity of the FACE data model. FACE, being a self-descriptive data model, has explicit types and meta types and represents types as objects: FACE-Types are also FACE-Objects. The FACE-Object is therefore the basis for most of the elements (objects, types, meta types) of the data model. In this section we are going to present the implementation of the FACE-Object.
FACE-Objects have a generic implementation, consisting of containers for its Properties and Property Values. Although FACE-Objects are (runtime) typed, all these types use this same implementation.

Road map for this section

This section discusses the implementation of the generic structure of the FACE-Object in detail. We will start with a description of the formal definition and the interface of the object which is a summary of the formal definition of the FACE-Object in [Mei93a]. We will then describe how the FACE-Object was implemented in C++, how the Properties and the Property Values are implemented. At the end of this section, some basic details on the implementation of type checking will touch the typed structure of the FACE-Object.

3.3.1 The generic structure of the FACE-Object

> Objects have an identity independent of their value, and refer to their type. Objects have named *properties*. Via a property name, the value of that property can be accessed. This so-called *property value* is either a sequence of identifiers of other objects, i.e. references to other objects, or a sequence of (other kinds of) values. [Mei93b, page 42]

These are the informal requirements for the FACE-Object. In order to define a FACE-Object formally[6] correct, we need the following ingredients:

- The set of all identifiers ID. Identifiers play the role of object references.
- The set of all identifiers identifying object types:[7] $IOT \subset ID$.
- The set of all basic types:

$$D = Integer \cup Boolean \cup String \cup \ldots$$

- A set of symbols representing the validity state of objects:

$$VALIDITY = \{invalid, synt\_valid, valid\}$$

- The set of all Property names:

$$PNAMES$$

[6] For the full formalization see [Mei93a, pp. 5-8]
[7] Object types are types describing objects. See section 3.4.1 on page 57.

- For each object a finite subset A that is determined by the type of the object:

$$A = \{Pn_1, \ldots, Pn_k\} \subset PNAMES$$

- The set of all possible Property Values:

$$PV = ID^* \cup D^*.$$

  Note that a Property Value consists of a sequence of data elements of one type, the so called Element Type.

- The set OV of all mappings t:

$$\begin{array}{rcccl} t & : & A & \rightarrow & PV \\ & & Pn_i & \mapsto & v_i \quad \forall i, 1 \leq i \leq k \end{array}$$

A FACE-Object $\mathcal{O}$ is then defined in the following way [Mei93a, page 7, Definition 6]:

$$\mathcal{O} = (value, type, validity)$$

where

$$type \in IOT, validity \in VALIDITY, value \in OV$$

We note that each FACE-Object has a distinct number of Properties which depends on the type of the object. We further note that the number of elements that form a Property Value is not fixed. We finally note the Properties cannot be discerned by the type of their elements, they all have the same structure.

3.3.2 The basic interface of a FACE-Object

In FACE, the definition of a Property for an object automatically results in the availability of operations to read and write the values of the Property [Mei93b, page 38]. The user wanting to access the Properties of the object does so by sending an access-message via the object which is the owner of the Property. The basic interface of a FACE-Object therefore consists of a `get` and `set` method for each Property the object has. There are two `get` methods which are slightly different [Mei93a, page 8, Definition 9]:

1. `get`: returns the whole Property Value in the form of a set. The set can then be queried for its contents.
2. `singet`: returns only the first[8] element of the Property Value.

Access to the properties must be realized in a *generic* manner. This means that instead of having a method

```
setName("Idefix")
```

to change the contents of the Property "Name" we should be able to parameterize a generic `set` method with the name of the Property, e.g.

[8] 'First' is meaningful since Property Values are a sequence (see previous section).

```
set("Name","Idefix").
```

The reason for the genericity-requirement is that we want to be able to create new FACE-Types at runtime [Mei93b, page 38] which will then be instantiated without recompilation of the system. We thus are not able to predict at compile time which Properties any FACE-Object has. It is possible to build such a generic access mechanism, i.e. to know at runtime which Properties the object has and which not, because a FACE-Object always has a reference to its type. The type is where the information about the Properties of the instances can be retrieved. Note that we must take precautions that we do not run into an endless recursion when, in order to access an object's properties, we first have to access the type-object's properties.

3.3.3 The implementation of the FACE-Object

The major characteristic of a FACE-Object that influences the implementation is that it is created at runtime following a description that is not known at compile time.[9] It is therefore obvious that a FACE-Object cannot be implemented by a simple class in the hostlanguage C++. For an implementation of the FACE-Object we have to consider three major points:

1. How are the Properties of an object going to be realized?
2. How does the fact that there are Properties with different Element Types affect the implementation?
3. How does the fact affect the implementation that the number of Properties in a FACE-Object as well as the number of elements in each Property Value is unspecified at compile time?

We will examine the threee points in detail in the following sections.

The Implementation of the Properties

We propose to implement a Property as an object in its own right, the Property Object. The Property Value is contained as an aggregation in the Property Object. The characteristics of this choice are the following:

- A Property Object has its own identity, i.e. it can be treated as a handy item. We can build easily support for collections or sets of Property Objects using container classes from libraries (see next section).
- The behavior of the Property is encapsulated. The Property Object has a distinct interface of its own. This allows to create different levels of abstraction.
- A Property Object is generic. This means that for every element type of the Property Value, the Property Object "looks" the same from outside. The Property Object hides the Element Type of its value.

[9] For a detailed description of the instantiation mechanism see section 3.4.2 on page 59.

The Element Type of the Property Value will be fixed at creation time of the Property Object. The information, which Element Type to choose for a specific Property Object is taken from the Property Descriptor. The Property Object therefore is seen as an instance of the Property Descriptor. It is, however, not a FACE-Object and thus not a FACE-instance of the Property Descriptor (see page 62 for details about the instantiation of Property Descriptors).

Using containers

Both the set of Properties and the set of elements of a Property Value are unknown at compile time. We therefore have to devise a mechanism to deal with the dynamic allocation of items during runtime. This is a standard task that occurs when implementing software systems. Reusable code for this kind of functionality can be found in every source code library. We used container components from the STL [MS96] in the two cases.

The implementation of the FACE-Object called for the possibility of allocating any number of Properties, which are items that can be identified by their name. To build a container for these Properties (see the left side of Figure 3.1 on page 49) we used an associative array of the STL, the template container

```
map<class Key, class T, class Compare>.
```

where a `map` contains elements of class `T` which are indexed by keys of some arbitrary type `Key`. It offers fast retrieval of information based on the key, using a `Compare`-function provided by the user.[10]
In our implementation, the objects of class `T` are the Property Objects. The `map` organizes storage and retrieval by using the name of the Property as the `Key`. The information which Properties the object has is thus implicitly contained in the `map` and does not have to be retrieved from the object's type. The access to the objects Properties can thus be generic *and* fast. We furthermore avoid the trouble when, in order to access the Properties of an object, we first have to access the Properties of the type.

For the implementation of the Property Value (see the right side of Figure 3.1) we used a sequential container from the STL, the

```
vector<class T>
```

where `T` is the type of the elements that can be stored in the container. A `vector` offers fast random access to sequences of varying length in addition to fast insertions and deletions at the end.[11]

The interfaces of the STL containers are very "heavy" and allow the user to do almost everything. To tailor the interface to our specific needs, we wrapped both containers in classes of our own (classes `propsContainer` and `propValue<T>` in Figure 3.1). Note that we can do this with a non-template class in the case of `propsContainer`

[10] See [MS96, page 164].
[11] See [MS96, page 118].

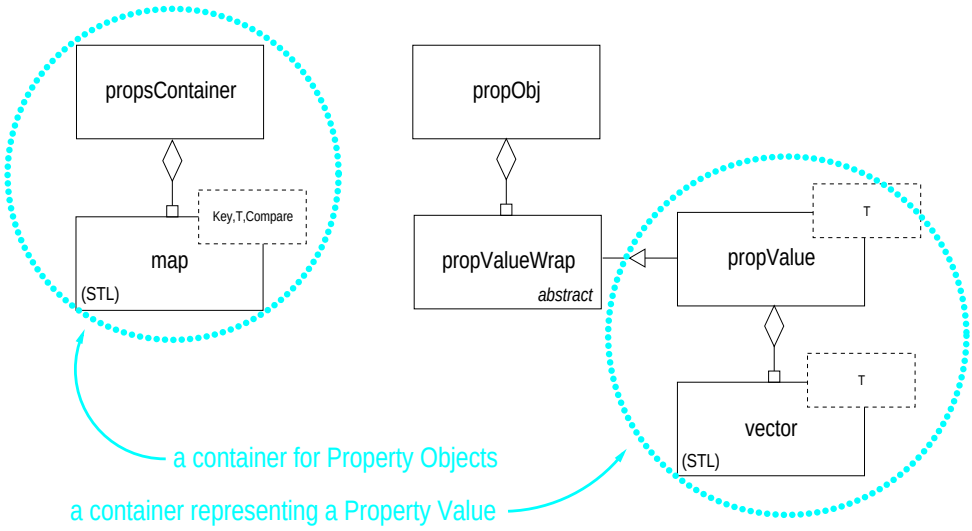

Figure 3.1: Wrappers around STL container classes.

but have to use a template for `propValue<T>` since this class must provide an interface to the elements of type `T` stored in the `vector`.

Two problems arise when we implement the Property Value as a template class:

1. In order to store all the Property Objects of a FACE-Object in one and the same `map`, the type of implementation of the Property Object—the class `propObj` (see Figure 3.2)—cannot vary with the elements that are stored in it, it has to be the same for all the types. This means that we have to aggregate the template `propValue<T>` in the non-template class `propObj`.

2. Another question to ask at this point is how the access functions of the FACE-Object are realized in the class structure that we have developed. All three of these operations must have access to the Property Value.

   - `get` returns the whole Property Value,[12]
   - `singet` returns only the first element of the Property Value,
   - and `set` adds another element to the Property Value.

   `get`, `singet`, and `set` must be available for each of the different types that can be stored in a Property Value. In a statically typed programming language like C++ we can achieve this by overloading the `set` and `singet` methods for each type. This results in the interface that has to provide these access functions being "heavy" in the sense of having lots of methods, which implies lots of code. The number of those overloaded interfaces has to be minimized.

These problems can be overcome by the introduction of a non-template, abstract base class for the the value container of the Property Object: this is the class `propValueWrap`

[12] Note that `get` does not return the Property Object but only its value part (see section 3.3.2 on page 46).

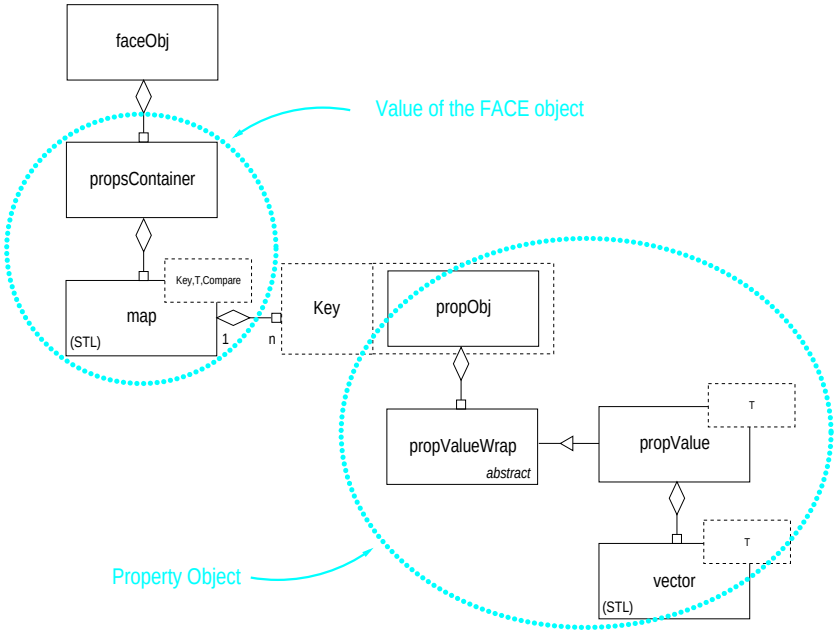

Figure 3.2: The classes of the implementation of the FACE-Object.

(see Figure 3.2). This class enables the following respective solutions to the problems mentioned above:

1. We can easily store a reference to the base class in the class `propObj` and thus hide the type of the value container.
2. Access requests that 'enter' the FACE-Object over the broad, overloaded interface of the class `faceObj` can be easily multiplexed into a single function that transfers the `propValueWrap` container from the `propObj` through the different classes (`propObj`, `propsContainer`) to the access functions in `faceObj`:
   - `get` returns the whole `propValueWrap` container to the caller which has to do the downcasting itself. This is possible since the client either knows what to expect or the Element Type can be asked from the type of the Property. An iterator that is built in `propValue<T>` helps accessing the elements of the container sequentially.
   - `set` and `singet` which, as mentioned above, are implemented using overloading and are thus typed, can downcast the container and access it over the typed interface of `propValue<T>`.

We thus solve the problem of the unspecified numbers of elements using two template containers. Note: The way we use the STL template classes is different in the two cases:

- The `map<Key,T,Compare>` template will only be instantiated with the class `propObj` as actual argument for `T`.

- The `vector<T>` will be instantiated with the range of different types that are going to be stored in a Property Value. The template functionality of `vector<T>`is thus essential for our implementation.

Layering of structure

Summarizing the implementation of the generic FACE-Object we can see a layer structure. The behavior of the FACE-Object is distributed in the implementation over the different levels of abstraction. Starting 'bottom up', the levels are the following:

1. The Property Value
   At the lowest level (implemented by the class `propValue`) we find the behavior for storing and retrieving data elements.

2. The Property Object
   The Property Object (implemented by the class `propObj`) encapsulates the functionality to create the right type of value container when the Property Object is created (see section 3.4.2 on page 62). It additionally stores and provides access to Property-specific data like the access-control flags `addallowed`, `removeAllowed`, `changeAllowed`.

3. The Object value
   The value of a FACE-Object consists in a set of Properties and their values. The class `propsContainer` implements the management of the set of Property Objects of the object by using the STL associative `map`-container.

4. The FACE-Object
   The basic behavior of a FACE-Object is implemented by the class `faceObj`. With basic behavior we mean, for example, how an object behaves during instantiation (see section 3.4.2 on page 59), how basic access is provided.

These different levels are a clean separation of tasks and responsibilities in the FACE-Object. Following the object–oriented paradigm we implemented each of these layers in a separate class. The implementation profits of the advantages of encapsulation and abstraction.

Discussion of the Implementation of the Properties

An alternative implementation of the Properties was proposed in [Mei93b, page 177]. The idea was to implement the object value as an array. Each position of the array would hold a Property Value (in the form of a sequence). In order to access a Property, the offset of its array-position would have to be known. This information would be gathered by asking the Property Descriptor. The mechanism thus proposed would guarantee genericity, because it would only be based on the runtime availability of type information.
The implementation proposed in this work, while keeping the genericity feature of the access mechanism, has the advantage that it eliminates the call to a Property Descriptor as Properties are searched for not by offset but by name. The drawback is obviously that more space is needed for the overhead of an object.

3.3.4 The implementation of the type checking mechanism

In the following, we assume the reader to know what a FACE-Type is, namely a description for an object in terms of the Properties the object has and the operations that can be applied to the object (see section 2.3.4 on page 32). Since every object in FACE refers to its type, runtime type checking can be performed for every operation that is called. The basic example of an operation in FACE where type checking is needed is `set`: does the data element that is about to be entered in a Property match the Element Type that the Property Descriptor prescribes? In fact, before a new value can be entered into a Property Value, a number of checks have to be performed:

- Is the Property allowed to change its contents? (Checking the flags `addAllowed`, `removeAllowed`, `changeAllowed`)
- Is the Element Type of the Property the same as the type of the new value? (Type checking)
- If the new value is an object reference, do we have to set a backreference or not? (look at the Property BackrefPropDescr of the Property Descriptor)
- After the value has been added/removed/changed: does the object still conform to the syntactic requirements of its type? Note that this check does not decide if an operation can be performed or not.

These different checks are very dependent on runtime type information and are designed to keep an interactive user from doing the wrong thing. If, however, a FACE-Object is accessed by the means of a program, and we assume that the correctness of the operation was ensured before compilation time of the program (an example for such an program is the bootstrap process, see page 65), such tests could be a waste of time (or not even possible, see again page 65). We therefore divided the interface of the FACE-Object into two layers.

1. On the upper layer, access to the objects' Properties is guarded by the different checks mentioned above. To perform the checks on this level, the access methods need runtime type information.
2. On the lower level, the only checks that are performed are ensuring that the requested Property exists and that the domain of the new value matches the domain of the Property's Element Type. To perform these checks, no other runtime information is needed than what can be found in the object itself.

The lower level is implemented as part of the class `faceObj`, which has the result that the implementation of `faceObj` does not have to bother with the typesystem. The upper layer of the interface is implemented in the class `SCObject_type` (see Figure 3.6 on page 58). This class, using the information of an established run time type system, can then offer the full type checking capabilities.

As an example, we display the different phases of an access of Property p of an object O to store a new data item d.

1. The add-check for the Property p and the item d is performed. The add-check consists of the following parts.

- Does the Property p exist?
- Is adding to p allowed?
- Is the type of O valid, i.e. is type checking possible?
- Check if the type of d conforms to the Element Type that is prescribed by the Property Descriptor of p.
- If d is an object reference, check if d is not is not already in the backreferences of O (to avoid circular references).

2. Call the lower level interface function. On the lower level, the following checks are performed:

   - Is the Property p part of the object value of O?
   - Does d have the right data type for the Property Value container of p?

   If these checks are passed successfully, d is being added to p. We return to the upper level.

3. If d is an object reference: check with the Property Descriptor of p if a backreference has to be set and set it if necessary.

4. Compute the validity of O which has now an altered value: does it still satisfy the prescriptions of its type as before?

3.4 The implementation of the FACE kernel

The kernel of the FACE system consists in those types and meta types that describe the basic FACE-Object and its relations (see section 2.3.5 on page 37). The question we have to answer here is how these types and meta types are implemented in C++. The main point we have to consider when implementing the kernel of the FACE model is the relationship between FACE types and classes in C++.

Our hostlanguage C++ has an object model as well as FACE. They share basic features of object orientation:

- Objects are instantiated following a description of a class (or type).
- Classes (or types) can be specialized through inheritance.

The most remarkable difference between the two objects models is however the following: Types (and subsequently meta types as well) are reified in the FACE runtime environment. Everything in the FACE model is an object. A type therefore has again a type and this leads to an instantiation tree with a depth of four: from the self-descripting meta class iMetaType_type which has meta types as instances which in turn instantiate types which instantiate simple objects.

In C++, on the other hand, classes are no reified entities of the runtime environment and meta classes do not exist. We have an instantiation tree with a fixed depth of one: a C++ class has objects as its instances.

What we want to achieve with our implementation is to map the FACE object model on the object model of C++, to map FACE instantiation to instantiation in C++, and to map inheritance or subtyping in FACE to inheritance in C++. This may be summarized in the following single goal: We want to map FACE-Types to C++ classes.

3.4.1 Mapping the FACE instantiation to C++

The goal of the implementation of the FACE instantiation mechanism is to create an instantiation mechanism for FACE which is built upon the instantiation mechanism of C++. This implies that instances of FACE-Types are then real C++ objects. Intuitively this means that each FACE-Type has to be mapped to a C++ class. To map the FACE-Types to C++ classes we use a mechanism [Mei93b, page 177] which equates all instances of a FACE-Type $\mathcal{T}$ with the instances of a C++ class SCT. This means that every (FACE) instance of $\mathcal{T}$ is a C++ instance of SCT. The class SCT[13] is called the Shadowclass of the type $\mathcal{T}$ (see Figure 3.3).
The Shadowclass implements the FACE-Type:

- All the methods that are defined for instances of $\mathcal{T}$ are implemented as methods of the Shadowclass: the generic access methods that implement the behavior of a normal FACE-Object as well as specific methods that implement specific behavior of that type.

[13] We use the following naming convention in our implementation: A FACE-Type with the name of "$AType$" corresponds to a Shadowclass named "`SCAType`".

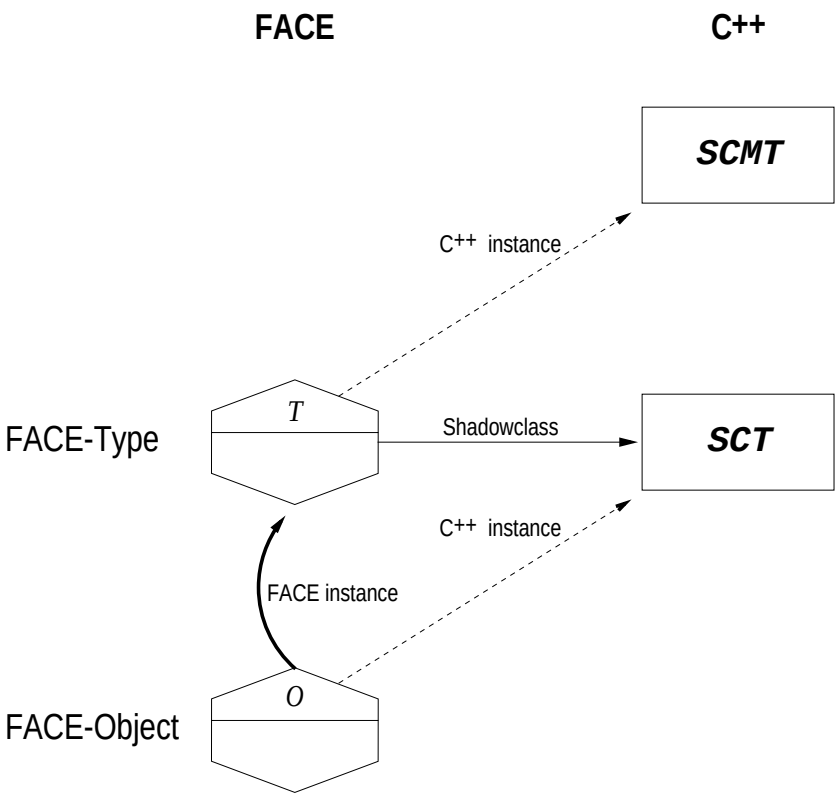

Figure 3.3: The principle of Shadowclasses. [Mei93b, page 178]

- The Shadowclass does not, however, implement the Properties of the instances of $\mathcal{T}$. Since the Properties are realized as dynamically allocated Property Objects (see section 3.3.3 on page 47), the Shadowclass does not provide any member variabless to represent the individual Properties.

A Shadowclass is thus not a complete representation of a FACE-Type since it does not represent the structure of the type that is only created at runtime.

In FACE, subtyping adheres to the principle of subtypes corresponding to subsets [Mei93a, page 13ff, Definitions 18-19]. This principle—meaning that the set of instances I^S of a supertype $\mathcal{T}^{\mathcal{S}}$ is a superset of I, the set of all instances of type $\mathcal{T}$—is also fulfilled by inheritance in C++ since every instance of a subclass is also an instance of the superclass. Therefore, when two FACE-Types $\mathcal{T}$ and $\mathcal{T}^{\mathcal{S}}$, which correspond to the respective Shadowclasses SCT and SCTSup, have a subtype relationship—$\mathcal{T}$ is subtype of $\mathcal{T}^{\mathcal{S}}$—then SCT will be a subclass of SCTSup (see Figure 3.4).

However, FACE subtyping is not equal to C++ inheritance. As mentioned before, which Properties a FACE-Object has is prescribed by the FACE-Type of the object and not by the Shadowclass. We therefore need a special subtype relationship between FACE-Types to copy Properties and add new ones. FACE subtyping is needed to introduce new structure while C++ subclassing is needed to add fundamental new behavior to FACE-Objects. If a type $\mathcal{T}_{\mathcal{S}}$ has not a specific behavior which would need

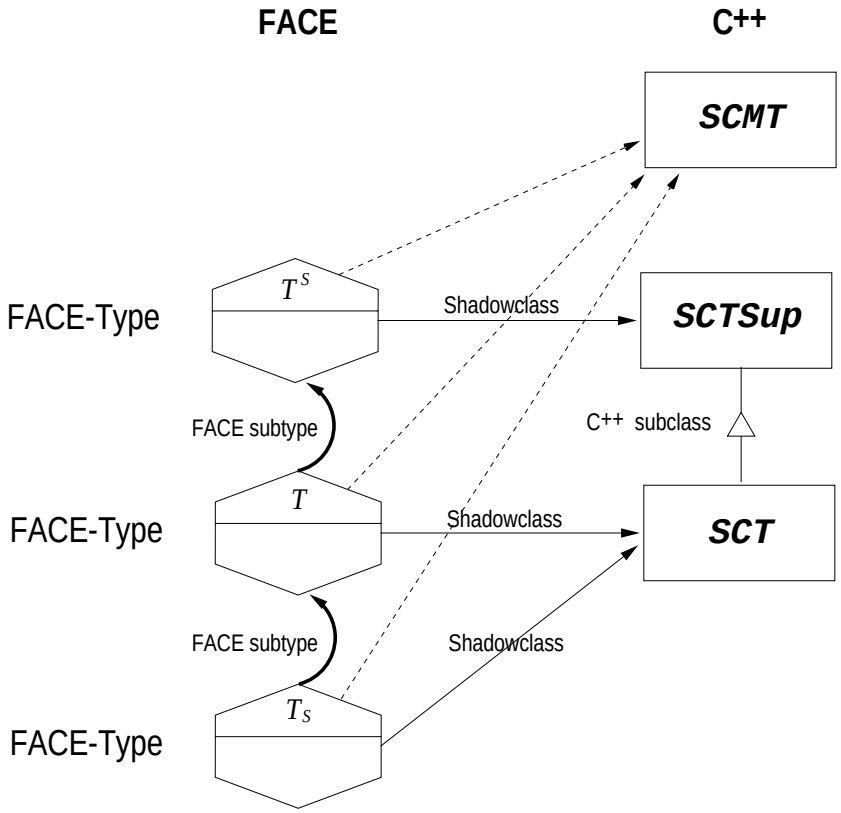

Figure 3.4: Subtyping corresponds to subclassing.

a special implementation in a Shadowclass of its own, it can as well just correspond to the Shadowclass SCT of its FACE supertype $\mathcal{T}$. All FACE instances of $\mathcal{T}_S$ are then C++ instances of the class SCT [Mei93b, page 179] (see Figure 3.4). Note that this enables us to create new FACE-Types at runtime and immediately instantiate them without recompiling the system with a new Shadowclass.

FACE-Objects that are not FACE-Types do not have a Shadowclass since they have no instances and thus cannot fulfill the left part of the equation "FACE instance *implies* instance of the C++ Shadowclass".

In summary, one can state that the Shadowclass mechanism is a direct mapping of FACE-Types to classes in C++, including a correspondence of the two instantiation mechanisms. Subtyping in FACE can also be mapped to inheritance in C++. The Shadowclass mechanism allows us to implement FACE-Types using the object–oriented paradigms of C++, i.e. data abstraction and encapsulation. The closeness of the mapping of the basic object–oriented features allows to further build a good representation of the FACE type system in C++, as will be described in the next section.

The implementation of the kernel

Given the idea presented in the last section, the question now is: When implementing the kernel, where do we need Shadowclasses?

When we look in Figure 3.5 at the part of the meta type-hierarchy of the FACE model which describes types, we see that there are different categories of types. All the types in FACE are FACE-Objects. They do not, however, describe similar 'things' and do not all have the same behavior; e.g. they cannot all be instantiated. In what follows we give a short explanation of the relevant categories. This will make clear which types have a Shadowclass and which do not.

Object Types (Instances[14] of iObjectType_type)
: Object types describe FACE-Objects. When we instantiate an object type, we get a FACE-Object (see function New [Mei93a, page 42, Definition 37]).

Basic Value Types (Instances of iBasicValueType_type)
: Basic Value types describe elements of primitive domains like strings, integer numbers or booleans (see section 3.2 on page 42). Elements of these domains do not have the status of objects. They do not refer to a type, their type can only be known by virtue of their context [Mei93b, page 42]. Since Basic Values are not objects, Basic Value types do not have Shadowclasses.

Property Descriptors (Instances of iStructPropDescr_type)
: Property Descriptors are used in the formal model to describe Properties (see for example predicate post_New [Mei93a, page 42, Definition 37] where the list of Property Descriptors is translated into Properties) but the Properties have no object status. In our implementation, however, Properties are a special kind of objects (see page 47). We thus introduced a type-object relation between Property Descriptor an Property: sending a Property Descriptor the message `instantiate` results in a Property Object being created.

All elements of the two categories, object types and Property Descriptors, have a corresponding Shadowclass. But object types and Property Descriptors have two

[14] Note that the types in Figure 3.5 are on the meta level: their instances are again types which can be instantiated themselves.

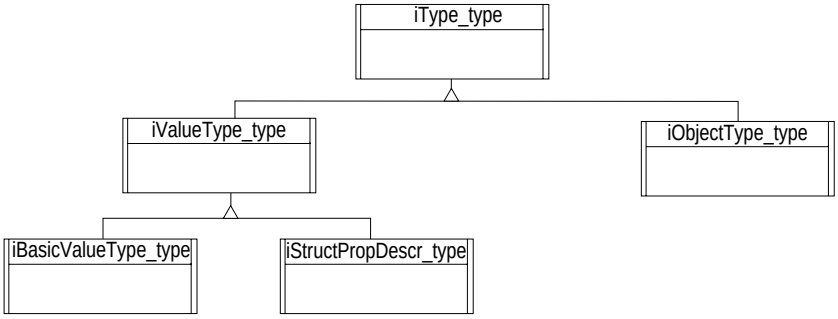

Figure 3.5: The meta type hierarchy showing different kinds of types.

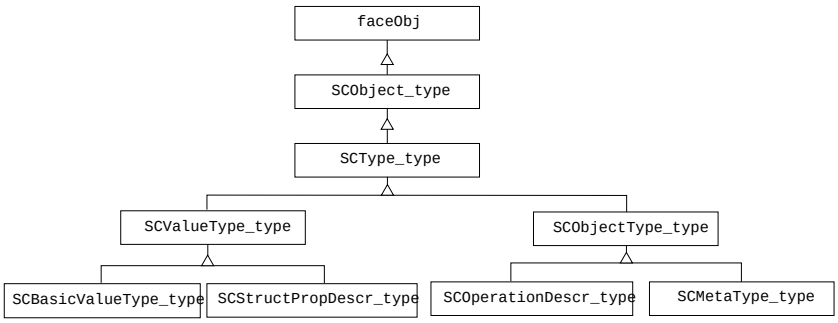

Figure 3.6: The system of Shadowclasses which correspond to the FACE kernel types.

completely different set of instances:

- the instances of iObjectType_type [15] describe FACE-Objects.
- the instances of iStructPropDescr_type describe Property Objects.

Since Property Objects and FACE-Objects have no common behavior, the Shadowclasses of these two items are not related. This leads to two different Shadowclass-hierarchies with different root classes (see Figures 3.6 and 3.7).

In Figure 3.6 we see the hierarchy among the Shadowclasses of object types. The layout is similar to the one showing the subtyping relationships among the kernel types (see Figure 2.12 on page 38). The topmost class in the hierarchy is not a Shadowclass. The class `faceObj`(see section3.3.3 on page 51) implements generic object behavior which has nothing to do with the the type system of FACE. The top Shadowclass is the class `SCObject_type` which corresponds to the FACE-Type iObject_type. Every type in the FACE model is a subtype of iObject_type and that makes every object type Shadowclass a subclass of `SCObject_type`.
The different mechanisms of FACE are implemented in the Shadowclass that corresponds to the type that introduces the mechanism in the FACE model.
For example:

- The meta type iObjectType_type introduces the method `instantiate`[16] Objects which are instances of iObjectType_type can therefore be sent the message `instantiate`. The class `SCObjectType_type` therefore has the method

  ```
  faceObj *instantiate() const
  ```

  and introduces the instantiation mechanism (see next section).

- The class `SCType_type` introduces the function

[15] The instances of iObjectType_type include also the instances of iMetaType_type which is a subtype of iObjectType_type.

[16] In the formalization ([Mei93a, page 42, Definition 37]), the function NEW is defined for all elements of the set IOT, i.e. for all object types.

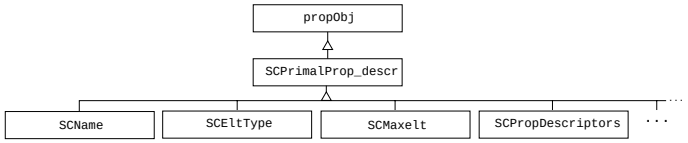

Figure 3.7: The hierarchy of Shadowclasses of Property Descriptors.

```
bool is_subtype_of(const SCType_type& supertype) const
```

which checks for a subtyping relationship between two types.

- The class `SCBasicValueType_type` introduces the function

  ```
  static bool DomainChecking(const SimpleType&)
  ```

  which enables clients to check for affiliation of data elements to a specific basic value type.[17]

The hierarchy of Shadowclasses that correspond to Property Descriptors can be seen in Figure 3.7. The topmost class `propObj` (see section 3.3.3 on page 51) implements the main behavior of a Property Object. It is implemented with the least possible knowledge of the FACE type system. The subclasses are real Shadowclasses of FACE Property Descriptors.
The existence of Shadowclasses for Property Descriptors opens up the possibility to implement a special behavior for a Property Object, like triggering certain actions when data is stored in the Property Value or retrieved from there. This behavior could then be made explicit in the model by introducing a new meta type (which would then describe the new kind of Property Descriptors).

3.4.2 The implementation of the instantiation mechanism

As we have seen in section 3.4.1 we can map instantiation of a FACE-Type directly to instantiation of the type's Shadowclass. In this section we present a mechanism that realizes this mapping.

What we want when instantiating a FACE-Type is an instance of the type's Shadowclass. The `instantiate`-message received by a FACE-Type must somehow trigger a call to a constructor in the Shadowclass. Just calling a constructor of a Shadowclass is not enough, though, to establish the FACE-Object fully, since for example the Properties of a FACE-Object are not represented by member variables of the Shadowclass (see section 3.4.1 on page 54). In fact, the call to the specific constructor of the FACE-Type's Shadowclass is only used to ensure the proper implementation of the FACE-Object (i.e. its C++ based behavior) and not sufficient to build the generic structure of the FACE-Object based on the description of it's FACE-Type. This done generically in a second stage of the instantiation procedure.

[17] See section 3.2.2 on page 43 for details on the implementation of primitive domains.

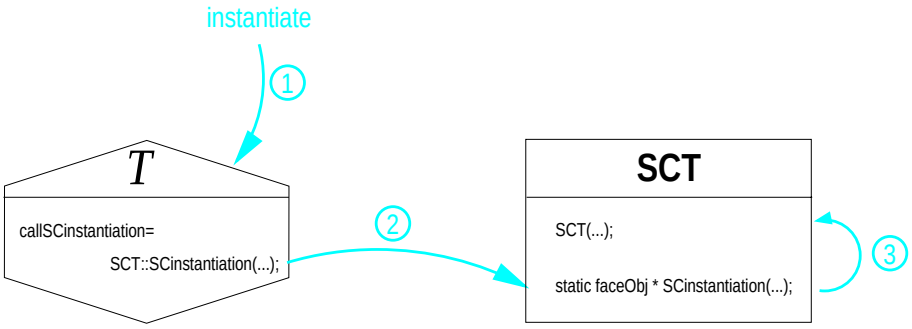

Figure 3.8: How a FACE-Type calls the constructor of its Shadowclass.

The connection between a type and its Shadowclass

The first stage of the instantiation process consists of the call to the constructor of the type's Shadowclass. In order to be able to do that, we must link the runtime entity 'FACE-Type' to the compile time entity 'Shadowclass'. They are inherently not connected in any way. Thus, access from the type to the constructor functions of the Shadowclass is not immediately possible and has to be established by the implementation.

To connect the type object to its Shadowclass we would like to reference the constructor function of the Shadowclass via a function pointer. This is not possible since in C++ it is not allowed to take the address of a constructor [ES90, page 265]. We are therefore creating a 'stand-in' for the constructor in each Shadowclass—a *static* function since we will not call it through an instance—and reference a pointer to this function in the associated FACE-Type. When the `instantiate` message is sent to the FACE-Type, it will call the static stand-in which in turn calls the constructor of its class (see Figure 3.8).
Attaching a Shadowclass to a FACE-Type is normally done right after the creation of the type object. The advantage of using a dynamically attached Shadowclass is flexibility: we can change the Shadowclass at runtime, thus altering the implementation of the FACE-Type's instances on the fly.
The constructor of the Shadowclass is the place to put initialization routines for class members or behavior that is specific to the implementation of the type.

After the first stage of the instantiation process, the C++ object has been created, but none of the type specific structure of the FACE-Object has been instantiated. This is done in the second stage of the instantiation process. The second stage can be performed in two different ways. On the one hand there is a mechanism to build the Property structure of a FACE-Object by taking its structural description (found in the type's Property PropDescriptors) as an input for the algorithm. On the other hand, we can use a prototype mechanism where a prototypical instance of the type that is attached to the type is cloned.
Since the creation of the inner structure of a FACE-Object needs access to data structures that are hidden[18] in the class `faceObj`, we implement the functions that perform the creation of the FACE-Object structure not as method of the type or the prototype

[18] that is, they are `private` members of the C++ class.

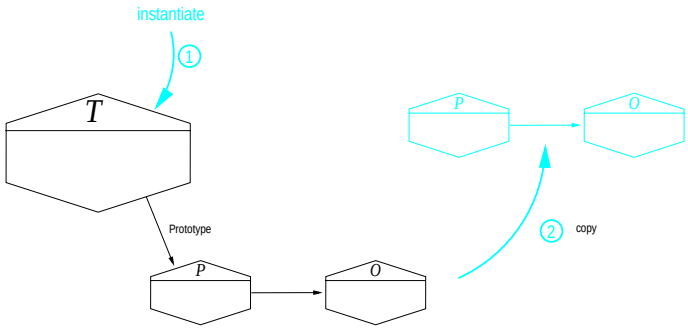

Figure 3.9: How a FACE-Type creates a new instance of itself by copying a prototype.

that instantiates an object, but as method of the object that gets instantiated. Either of these functions is thus called on the freshly created object with the FACE-Type as argument in one case and the prototype in the other case.

The following sections detail the two implementations of the second stage of the implementation process.

Instantiation following the description of the FACE-Type

The function `faceObj::getinstantiatedbydescription()` takes the type $\mathcal{T}$ of the object as an argument and creates a container `propsContainer` to hold the Property Objects. The routine then requests the list of all Property Descriptors from $\mathcal{T}$ and one after the other calls the `instantiate`-method of the Property Descriptors (see section 3.4.2 on page 62). This results in the different Property Objects being created and the object is thus filled with the Properties that are prescribed by $\mathcal{T}$. The Properties are empty, no Property Value can be filled by this algorithms since $\mathcal{T}$ prescribes only the structure of its instances but not their values.

Instantiation using a prototype mechanism

The function `faceObj::getinstantiatedbycopying()` takes the prototypical instance of the type $\mathcal{T}$ as an argument and calls the C++ copy-constructors on the components of the prototype.[19] Using a deep copy algorithm, the Property Objects including the content of their Property Values are duplicated. Not only the prototype object is thus copied but also objects that are eventually attached to it. This is the main difference to the instantiation by description that was discussed in the previous section. This feature of the prototype mechanism allows us to instantiate whole object structures at once.
Prototypes like Shadowclasses can be changed on the fly.

[19] Note that we do not call the copy-constructor of the prototype-object itself.

The instantiation of Property Descriptors

The instantiation mechanism that was described in the previous sections concerned only the instantiation of object types, i.e. only FACE-Objects were created. But since we established a type-object connection between the Property Descriptor and its Property Object, we use a similar instantiation mechanism to create Property Objects. The only difference to the instantiation mechanism for FACE-Objects is that the Shadowclasses of Property Descriptors are subclasses of the class `propObj` and thus calls to their constructors create Property Objects.
As opposed to the instantiation of FACE-Objects, the whole creation of the Property Object is done in the constructor of the Shadowclass (or mainly in the constructor of the Shadowclass's superclass `propObj`). Since a Property Object, in contrast to a FACE-Object, has not a structure that consists of an number of Properties that is only known at runtime like a FACE-Object, but has a fixed member structure, instantiation is easier. Using the Element Type prescribed by the Property Descriptor, the domain (i.e. the C++ data type) is derived with which the Property Value container is instantiated. This is realized using a static `case`-statement which has a instantiation-command for each container-type.

The prototype mechanism

In the YANUS data model, an implementation proposal already made use of so called 'templates', i.e. prototypes (see [Mei93a, section 6.4, page 181]). In addition to that the model also featured the so-called PrimalType (see [Mei93b, page 68ff]). A PrimalType was a special instance of a meta type and was attached to its type. All the other instances of the meta type had to be subtypes of the PrimalType (see the extra-requirement req_{t1} in [Mei93a, page 27, Definition 23c]). The PrimalType thus allowed its meta type to exert control over the instances of its instances.
In FACE, the PrimalType and the 'template' are unified[20] since they had essentially the same features: In FACE, types can have a Prototype. A Prototype is similar to a PrimalType as it has to be an instance of the type it is attached to. A Prototype is different from a PrimalType as it does not have to be a type: Since they are not instantiated but copied, Prototypes can be everything down to normal FACE-Objects or even Property Objects. Prototypes surpass the abilities of PrimalTypes as they can represent an object structure which encompasses multiple objects and connections between them and which can then be copied as a whole.

The formalization of the prototype mechanism

The formalization for the Property Descriptor of the Prototype-Property is the following:

[20] Meijler, personal communication.

```
iPrototype inst_of iStructPropDescr_type
        name(iPrototype) =
                        iPrototype
        supertype(iPrototype) =
                        iPrimalProp_descr
        elttype(iPrototype) =
                        iObject_type
        minelt(iPrototype) =
                        0
        maxelt(iPrototype) =
                        1
        changeable(iPrototype) =
                        true
        unique(iPrototype) =
                        true
```

The iPrimalType was introduced as Property of iMetaType_type, meaning that every meta type had the Property PrimalType and could thus have a primal type. Instead of PrimalType, we introduce the Property Descriptor iPrototype as OwnPropdescriptor of iType_type, meaning that every type (Object types as well as Property Descriptors) can have a prototype.

3.4.3 Discussion

The instantiation

For specific Shadowclasses specific initializations may be needed for C++ (non-FACE) properties. This code should always be called even when copying the FACE Properties of the prototype. Since both instantiation mechanisms—using the description or using the prototype—are going through the same constructor in the Shadowclass, we can assure such a similar initialization. This is the main advantage over an implementation which would have followed the *Prototype* design pattern in [GHJV95, page 117] more closely. In the prototype design pattern, a new instance of a specific class `C` can be created not only by calling the constructor of `C` but by calling a `clone()` method of a prototype instance of `C`. The `clone()` method then calls the copy constructor of `C` to copy itself. Thereby a second location in the class definition (besides the normal constructor) would be created where initializing would have to be done.

The Shadowclass mechanism

The Shadowclass mechanism has some advantages that we want to discuss in this section. As an alternative to the Shadowclass mechanism we can envisage an implementation where only one C++ class is used to represent all the FACE-Objects at compile time. The whole FACE type system would be based on dynamical structures. The advantage of this hypothetical solution would be, that since we would create new types only dynamically with no new classes, we would never have to recompile the system after extending the type system.

The advantages of the Shadowclass mechanism are obvious.

- We model the FACE type system in C++. A FACE-Object has two identities, one in FACE and one in C++. We can thus program FACE easily from the C++ level and use the C++ type checking mechanism to ensure type safety of programs.
- Since FACE-Object always has an implementation as a C++ object, FACE object structures—which are models of software systems—can be executed. In the hypothetical solution with no shadowclasses, a FACE structure would first have to be translated into a executable format.
- When we think the other way round, a framework class can directly be the implementation of the FACE-Type that is its representation.

The Shadowclass mechanism allows a tight integration of the FACE type system with the type system of the underlying hostlanguage.

3.5 Bootstrapping the system

> A natural and fundamental question to ask, on learning of these incredibly interlocking pieces of software and hardware is: "How did they ever get started in the first place?" It is truly a baffling thing.
>
> Douglas R. Hofstadter
> in: *Goedel, Escher, Bach: An Eternal Golden Braid*

3.5.1 The requirements

During the bootstrap of the system, the kernel types and meta types of FACE are created. Bootstrapping FACE is tricky since it is a self-descriptive model. Self-description implies that in order to build the model we have to have the model at hand. Several functions which are crucial in building the system, in order to function have to read information contained in the structure. At the beginning of the bootstrap, none of this information is available.
In the following list, the dependencies of the operations on information contained in the structure are listed. The survey is done in a cursory manner, for greater detail refer to the formalization.

Instantiation: When we instantiate a FACE-Type, we use the list of Property Descriptors the type holds to build the object (see function New [Mei93a, page 42, Definition 37]).
Thus, for automatic instantiation we have to have a link to a type object, the type has to have a Property called PropDescriptors of which the value has to hold the list of Property Descriptors.

Inheritance: During the subtyping process the list of Property Descriptors as well as the list of Operation Descriptors is copied from the supertype to the subtype (see functions mk_propdescriptors and mk_operations in [Mei93a, page 28, Definition 25]). Both of these lists are merged with the respective list from OwnPropDescriptors and OwnOperations of the subtype.
Thus, for automatic inheritance a type has to have the Property Supertype which has to be instantiated with a link to its supertype, the supertype has to have the two Properties named "PropDescriptors" and "OperationDescriptors" filled with links to its Property Descriptors, and the subtype has to have the Properties OwnPropDescriptors and OwnOperations, possibly initialized with its own Property Descriptors.

Adding elements to Properties: When we want to add an element to a Property, the system performs a series of type- and other checks which rely heavily on the runtime type-information (see section 3.3.4 on page 52). These functions also compute the validity of the object based on the new elements of the Property Value (see predicates set_cond_synt_valid and set_cond_valid in [Mei93a, page 41, Definition 36]).
Thus, when adding elements to Properties under full control of the type checking mechanism, we have to have the typesystem more or less fully operational.

Establishing crossreference- and component-relationships: When we link FACE-Objects together we have to check the Property Descriptor if it requires a back-reference to be set (see section 2.3.4 on page 33).
Thus, for automatic linking of objects the Property Descriptors have to have the Properties "Owner" and "BackrefPropdescr" initialized.

Since we cannot use the built-in functions during the bootstrap phase, the efforts that are normally performed automatically, when for example instantiating a FACE-Object, have to be coded 'by hand': create the C++ instance, create the necessary Properties for the object, enter the elements in the Property Values and establish backreferences where necessary. Doing this for each and every type in the kernel is of course an unacceptable way of wasting (programming) time and space.
Based on these considerations we formulate the requirements for bootstrapping in the following way:

- The bootstrap process creates a fully functional set of kernel object instantiated and initialized according to the formalization in [Mei93a].
- The code that has to be written to substitute for functions and methods of the normal system that cannot be used due to lack of runtime structures has to be kept to a minimum.

3.5.2 The method

The main goal when finding a method for the bootstrap process is to minimize the additional code that has to be written specifically for this process. This means that we should use the built-in functions as much as possible, i.e. as early in the process as possible. We can do the following things to reach that goal:

First of all we have to find a minimal set of kernel objects which has to be instantiated and initialized fully as a start, using ad-hoc methods. This set of types should then allow to create the rest of the kernel with the normal methods.

Second, we have to find an order to instantiate the chosen objects which tries to use the built-in function for instantiation and inheritance as much as possible. Note that there is a tradeoff between these two actions : The topmost object in the instantiation hierarchy is iMetaType_type (see Figure 2.11 on page 37) which serves as type for all the meta types of the kernel (including itself). This means that in order to use the normal instantiation mechanism as soon as possible, we should create iMetaType_type very early. In the inheritance hierarchy however, iMetaType_type is located towards the bottom (see Figure 2.12 on page 38). This implies that if we want to use the built-in inheritance function to assemble the list of Property Descriptors for iMetaType_type, we must first instantiate all the objects that are above iMetaType_type in the hierarchy.

Two strategies can be applied here:

- We must bypass the type checking mechanism when accessing the Properties of objects. This is without danger since the bootstrap is a controlled process and thus type checking is not necessary.

The interface of the FACE-Object is implemented with two levels (see section 3.3.4 on page 52) so when we use just the lower level of access no type checking is done.

- Instead of instantiating an object at once with one function call (which implies that the type of the object has to have a complete list of all the Property Descriptors) or to use the inheritance function only once (which implies the supertype to have a complete list of all the Property Descriptors), we use a 'stepwise' instantiation and inheritance mechanism.
  With stepwise we mean that we can run the instantiation or inheritance procedure several times on the same object, each time with a extended list of Property Descriptors in the type. The respective mechanism is clever enough to find out which Property Descriptors have already been instantiated (as Property Objects) or copied (in the case of inheritance) and leaves them out of the process if necessary. Using this method, we can start with a small set of essential Property Descriptors in the type, instantiate the object in a first step, then let the type inherit new Property Descriptors and re-instantiate the object.

3.5.3 The implementation

At the beginning of the implementation we choose the set of objects that have to be instantiated first. These are the objects that define the basic behavior of a FACE-Object, including the topmost object in the instantiation hierarchy iMetaType_type, and the topmost object in the inheritance hierarchy iObject_type as well as the objects that lie between them in the type hierarchy. The various Property Descriptors have to be included in the initial set too since they are essential to the creation of FACE-Objects. This implies their type iStructPropDescr_type being also created early in the bootstrap process. When these objects are all instantiated and correctly connected, the normal methods of the implementation can be used.

The actual implementation can be divided in two parts: preparation and actual bootstrap.
During the preparation phase we do following things in a very 'dirty' way, i.e. we do it all by hand:

- First we create 'empty' C++ instantiations of all the objects of the initial set and connect them to their Shadowclass. We have to create the C++ objects at the very beginning since their addresses will be needed for references.

- Then we (FACE-) instantiate and initialize the set of Property Descriptors. Instantiation is done 'by hand' since the type iStructPropDescr_type is still an empty hull. Initializing the Property Descriptors is done with a specially written function `initializePropertyDescriptor(...)` which sets certain backreferences automatically. But since this mechanism requires again runtime information to be gathered from certain other Property Descriptors, we have to start with an initialization of these other Property Descriptors done without the function.

- As the last point in preparing we establish the instantiation- and the type-hierarchy among the kernel-objects by setting the links to types. Since the reference of an

object to its type is not realized via a Property (see formal definition of the FACE-Object in section 3.3.1 on page 45), we do not have to create a Property Object for the object for this purpose.

After the preparation phase, we begin instantiating the objects in the kernel.

- First, the two properties PropDescriptors and OwnPropDescriptors for the type iMetaType_type are created by hand. Then they are initialized with the following Property Descriptors: iPropDescriptors and iOwnPropDescriptors.
- Next we instantiate the type iObjectType_type with the normal instantiation procedure: the Properties PropDescriptors and OwnPropDescriptors are created automatically in iObjectType_type. We initialize these two Properties by hand and instantiate iObject_type with the automatic instantiation mechanism.
- After initializing iObject_type we begin to use the inheritance function down the inheritance tree until iMetaType_type has the necessary list of Property Descriptors.
- We then re-instantiate iObjectType_type and the other objects.

From now on, the instantiation process can be used in a more or less normal way.

3.5.4 Discussion

The algorithm implemented here to bootstrap the FACE system is, to speak frankly, a hack. It still uses a lot of space (around 800 lines of pure code) since we practically translated the specification of the kernel into code. This is a waste of memory since the code is used only once at the startup of the system; some kind of dynamic loading of the code for bootstrap would be a good solution here. We could then throw the code out again after generating the kernel.

Another problem is clearly that we have duplicated a lot of knowledge about the behavior of objects and the model as a whole. This knowledge is on the one side part of the implementation of the FACE-Object and on the other side contained here in the code that was specially written for the bootstrap procedure. This will get problematic when the model is changed and we subsequently have to change the bootstrap procedure too.

Another approach to bootstrapping which could save a lot of code lines (but would require a sophisticated interpreter) would be to read a specification of the kernel from a file. This idea can be integrated into the prospect of a permanent object store. An object store would allow us to save object structures to permanent memory (i.e. disk or tape) and recover it from there. We could generate the kernel once, save it in a file and every time the systems boots it would just read from the object store. This way, the extra code for the bootstrap would not be part of the system.

Since a big problem is the setting of backreferences, we could imagine forgetting about the backreferences until the end of the process when a checker program would be run over the object structures ensuring that all the backreferences were set.

3.6 Implementing an example

To gain experience with the modeling of software that FACE is supposed to enable, we designed and implemented a small example. The example was inspired by the situation presented in Figure 2.4 (see page 28), which is essentially the reference problem of [Mei93b].

The general setting of the example is the following: we have a number of existing programs with a command line interface that perform operations on simple ASCII data streams (see section 3.6.2 for details). We want to build a model consisting of FACE types and operation descriptors which provide an interface to these programs. These types and operation descriptors and the ways they are composed can be seen as defining a 'language'. Users can then compose structures of instances of the types and operation descriptors. These structures can be seen as scripts for the underlying programs, and we should be able to run the scripts. This means that we have to take advantage of an execution mechanism.

The questions we ask with respect to the example are mainly concerned with the experience gained in implementing the example.

- Is it feasible to build such a software model in FACE?
- How much work is necessary to model the example, i.e. how many types have to be introduced to describe the elements of the example?

The section has three parts: First, we are going to explain the execution mechanism which takes care of the connection of the FACE objects and operations with the underlying software. We are then going to present the situation of the example and the design of the model in FACE. The section will conclude with a comparison of the FACE execution mechanism with other generic implementations of behavior in object–oriented systems and a presentation of experience data gathered during the implementation of the example.

3.6.1 Execution in FACE

An operation in FACE generally[21] consists of a request which has a number of operands. A request is an instance of an Operation Descriptor and thus an object. A request is parameterized by establishing connections between the request object and the operand objects. Additionally, arguments that are less important, called *settings* of the operation, can be entered as strings or numbers. The parameterization is done by the user on the conceptual level. Execution is initiated by sending the conceptual request an `execute`-message.
The way this `execute` is performed stems from the YANUS model [Mei93b, Chapter 4]. The YANUS request execution creates an object structure on the implementation level that corresponds to the composition on the conceptual level (see Figure 3.10). The data represented by the conceptual operands is brought into the required implementation formats by the transformation functions. Finally the sequence[22] of implementation

[21] i.e. operations on the conceptual level and on the implementation level alike.
[22] The sequence may consist of any number of implementation requests including 1.

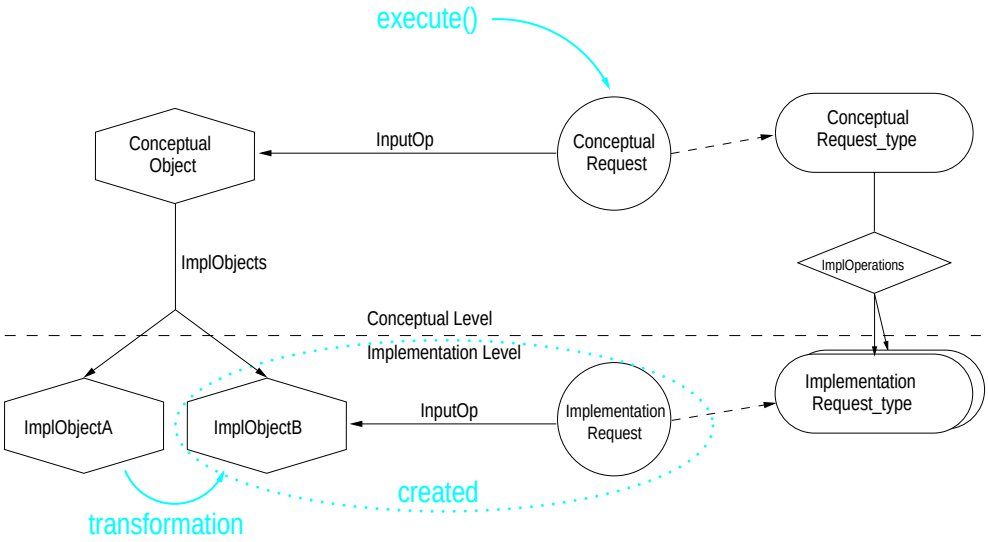

Figure 3.10: The execution of a conceptual request creates an analogous structure on the implementation level for each implementation operation.

requests is executed by sending `execute` messages to the implementation requests. Each implementation operation encapsulates the knowledge on how the underlying software packages are controlled. Note that the execution mechanism is not fixed at compile time. It is guided by the the connections established between the FACE-Types and acts thus as an interpreter of FACE compositions.

3.6.2 The implementation of the example

The situation of the example

Our example is, as mentioned above, a simplification of the YANUS reference problem. Standard UNIX tools play the role of 'software packages'.

We are modeling the functionality[23] of the following statements that can be executed in every UNIX shell:

```
cd <path>
grep -c <searchstring> <filenamemask>
```

The `path`, `searchstring`, and `filenamemask` are settings of the operations in our model.
To make the parsing of the output of the `grep -c` command easier,[24] our implemen-

[23] Our goal is to count the number of occurrences of a string in a specific type of files that are stored in a specific directory.

[24] When searching a single file, `grep -c` gives a single number as output. When searching multiple files `grep -c` gives a list of the names searched files followed by the number of occurrences.

tation achieves the above behavior by actually implementing the following statement sequence:

```
cd <path>
cat <filenamemask> > tmp.text
grep -c <searchstring> tmp.txt
```

In the following two sections we will describe the FACE-Types that were introduced to model this situation.

On the conceptual level

On the conceptual level of the example we have two kinds of elements: the chosen set of files and the number which is the result of the count. These two elements are represented by conceptual objects:

- A set of selected files is represented by an instance of the type iDirectory_type. Instances of iDirectory_type have the Properties Path and FileMask to identify the file selection.
- The result of the count operation, an integer number, is represented by an instance of iIntegerObject_type.

The functionality of the example is captured by two conceptual operations:

- Instances of iCSelect_descr upon execution take a filemask-string and an instance of iDirectory_type and then create a new iDirectory_type-instance with a FileMask that combines the masks of the arguments.
- Instances of iCCount_descr upon execution take an instance of iDirectory_type, a string to search for and return an instance of iIntegerObject_type.

On the implementation level

On the implementation level we introduce the types that connect the conceptual level to the actual data and actual functionality. We therefore introduce implementation types for data representations and operations. We additionally need transformation functions that prepare data in the required formats for input to the underlying software.

Implementation representations for conceptual data are the following:

- Instances of iDirectory_type play a double role of conceptual and implementation objects (see [Mei93b, Section 4.6.4, p. 129] for details on this principle). This means that instances of iDirectory_type can be used as operands for the conceptual select as well as for the select on the implementation level. For the count operation, however, we need to represent the data of the selected files as an instance of iUnixFile_type. Instances of iUnixFile_type refer to a file in an UNIX file system. This file is created upon each instantiation of iUnixFile_type and removed when the object gets destroyed.

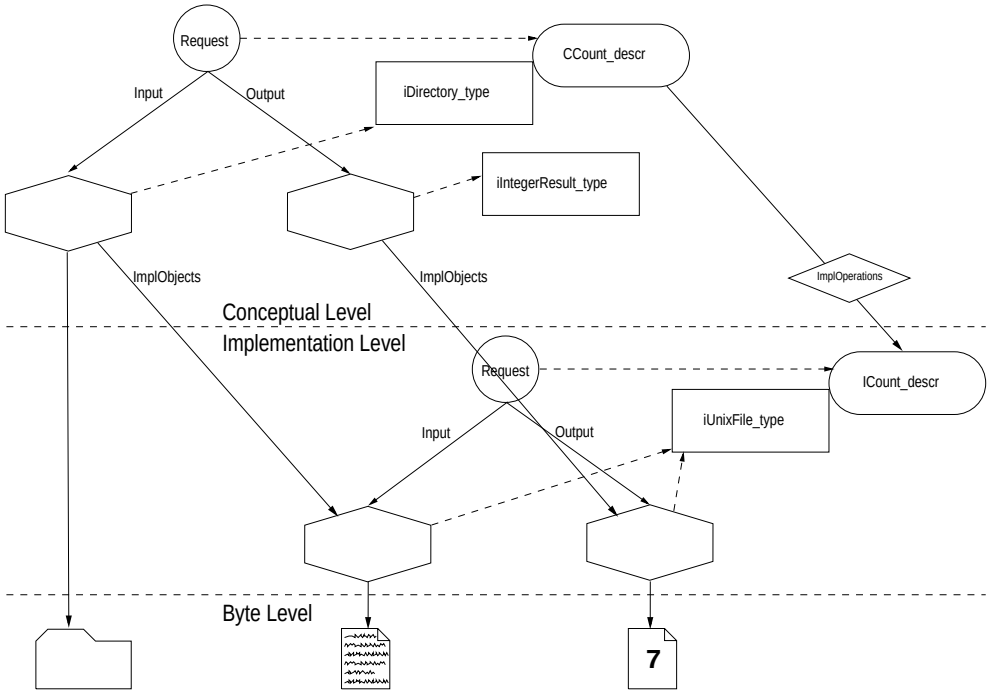

Figure 3.11: The structure of objects and types during the execution of the count-operation.

- The conceptual iIntegerObject_type is also represented by iUnixFile_type on the implementation level.

The implementation types that correspond to the conceptual operations iCSelect_descr and iCCount_descr encapsulate the knowledge of how the operations are actually performed:

- Selection requests, instances of iISelect_descr, combine the FileMask of the iDirectory_type operand with the SelectCriteria setting of the request. This is done with a simple string-concatenation operation.
- Count requests, instances of iICount_descr, take an ASCII-file represented by an instance of iUnixFile_type and use a call to the UNIX `grep` utility to count occurrences of the string of the SelectCriteria setting of the request. The output of the call to `grep` is redirected to another ASCII-file. In Figure 3.11 the situation of objects at the execution of the count operation is depicted.

In order to transform between the data representations we had to create two transformation functions. A transformation function is represented by an instance of a transform operation descriptor (see [Mei93b, section 4.6, p. 126 ff].

- The set of selected files is represented using a path and a filemask by instances of iDirectory_type. The input to the `grep` utility is done with a single file. We therefore `cat` the selected files into a single file. The function that does this is represented by the type iCatTransform_descr.
- The result of the iICount_descr operation is another ASCII file. In order to transform this data into an instance of iIntegerObject_type we build a trivial reader function which is represented by the type iReadIntTransform_descr.

3.6.3 Discussion

Comparison of the FACE execution mechanism to other approaches

The execution mechanism in FACE allows to put requests together at runtime. This can be seen in contrast to many message passing mechanisms in object–oriented programming. In the normal message-passing syntax of, for example, C++, the client is coupled directly to the server and the server has to be known at compile time. Moreover, the connection that is implicitly established when the client calls a method of the server, is hidden in the code.

A well known approach to dynamically determining which operation must be executed are the behavioral design patterns found in [GHJV95, Chapter 5]. The `Command` pattern, for example, encapsulates requests in generic command-objects allowing the coupling of client and server to be deferred to runtime. The execution mechanism in FACE does however more.[25] It allows not only to specify at runtime which server will execute the operation, but we can specify an operation at runtime in more detailed terms, i.e. we can compose a sequence of subordinate operations to perform a conceptual task.

[25] In fact, the `Command` design pattern is used in the implementation of the FACE execution mechanism.

Object-Types		
iDirectory_type	Yes	Conceptual types
iIntegerObject_type	No	
iUnixFile_type	Yes	Implementation type
Operation Descriptors		
iCSelect_descr	No	Conceptual operations
iCCount_descr	Yes	
iISelect_descr	Yes	Implementation operations
iICount_descr	Yes	
Operand Descriptors		
iOldDir	No	Conceptual and implementation
iNewDir	No	operands for the select operation
iCCountInput	No	Conceptual operands
iCCountOutput	No	for the count operation
iICountInFile	No	Implementation operands
iICountOutFile	No	for the count operation
Setting Descriptors		
iSelectCriteria	No	specifies filemasks and searchstrings
Property Descriptors		
iFileMask	No	
iPath	No	
iResult	No	
Transformation Descriptors		
iCatTransform_descr	Yes	`cat`'s a series of file into another file
iReadIntTransform_descr	Yes	Reads an integer out of a file

Figure 3.12: The list of types that were introduced when modeling the example (Yes and No refer to the necessity of an own Shadowclass).

Another specific aspect of FACE message execution is that it is not pre-defined and implicit. The mechanism being explicit it has in common with certain reflective approaches [Chi95]. In our point of view this is essential for a compositional environment: by making request execution explicit, the objects can be fitted in different environments (in which different non-functional requirements are posed) without having to change the object itself.

Experience data from the implementation of the example

To measure the effort that has to be taken to model the example, we display in Figure 3.12 a list of all the types that had to be introduced into the FACE type system. A distinction is made between types that introduce some new behavior and therefore have to have a Shadowclass of their own and types that use the Shadowclass of their supertype. Types having their own Shadowclass have a *Yes* in the second column of the table. Note that we also list the Property Descriptors and Operand Descriptors that were introduced to describe the Properties of the data representations and requests.

Chapter 4

Conclusions

4.1 Conclusions from the implementation

We have presented mechanisms to implement a reflective object–oriented data model in a non-reflective object–oriented language.
We have explored mechanisms that allow us to make objects behave as classes or types, i.e. we have implemented mechanisms that make instantiation and inheritance possible with these type-objects. Due to the runtime availability of types we can have runtime type checking.

Using the Shadowclass principle, we have integrated the FACE object model very closely to the object model of C++. This means that we can profit from the compile time and runtime systems of C++. Also programming with FACE types and objects is not different from programming with normal C++ objects.

We have devised a mechanism that bootstraps the self-descriptive FACE model.

To demonstrate the expressiveness of the FACE modeling capacities, we implemented a short example.

4.1.1 Specific conclusions for FACE

In contrast to the implementation proposals of [Mei93b, Chapter 6], we devised a mechanism for accessing a specific Property of a FACE-Object without querying the type of the object on the existence of the Property first. This simplification avoids the danger of an endless recursion.

4.1.2 Negative Results

Our experience when writing the bootstrap of the system in an ad hoc manner lead to the conclusion that bootstrapping the model should be done in a more algorithmic[1] way,

[1] To illustrate what we mean with 'more algorithmic' we give a simple example: When the task is to write a program that counts to ten and writes all the encountered numbers on the screen, a non-algorithmic way

i.e. the specifications of the FACE type system should be stored in a machine readable format and be translated into a system of runtime objects and types algorithmically. We think that this development could go hand in hand with an investigation of persistent objects (see section 4.3.1) .

4.2 Theoretic Conclusions

We have investigated the connection between the FACE data model and the theory of MetaObject Protocols. Since both of these techniques use meta levels to configure systems or languages, we tried to establish the relationships between these techniques in greater detail.

We draw the following conclusions:

1. While CLOS provides procedural reflective mechanisms in much of its runtime behavior, in FACE such reflective mechanisms are only introduced when needed (e.g. the execute mechanism, see section 3.6.1).
2. While CLOS
   a) provides a full programming language that can be used to implement reflection to realize, for example, new semantics of its own language elements,
   b) has a relatively fixed framework of the language elements that can be changed and extended in this way,

   the language elements of FACE, in contrast, are defined when needed to define new forms of composition; giving semantics to these elements is done using an underlying programming language. The reason for this is that FACE focuses on being a software composition environment and not providing a full fledged programming language itself.

4.3 Future Work

4.3.1 Technical Issues

Investigate the possibilities of persistent stream mechanisms that would enable us to store compositions of FACE objects to and retrieve them from permanent streams.

Devise facilities to efficiently report errors encountered when running the FACE system. Since for example runtime type checking is an important part of FACE, the user should be informed properly about type checking errors.

4.3.2 Research Issues

Gain experience in modeling software with FACE. Prime candidates are rather small frameworks like for example the STL.

would be to actually write ten `printf` statements and a more algorithmic way would be to use a `for`-loop.

Explore the connections between compositions done with FACE and the semantics they represent in a more formal setting. Investigate further how runtime semantics can be given to FACE compositions. The current approach is based on giving semantics through interpretation. Another interesting approach would be the use of compilation techniques.

Continue on an extension of FACE as a visual composition environment (see [ME96]). Define a FACE framework for that purpose in order to support visualizing objects and their connections in a graphical user interface.

Appendix A

Notation

Three different notations are used in this document:

1. The FACE-notation to describe sets of connected FACE types and meta types (e.g. the kernel) on a high abstraction level.

2. An ad hoc notation to explain on a medium abstraction level the coherence between the basic elements of the model.

3. The Unified Modeling Language (UML) [BJR96] for low level descriptions of the implementation.

I will present a short description of the three notations and a legend describing the drawing elements for each of them.

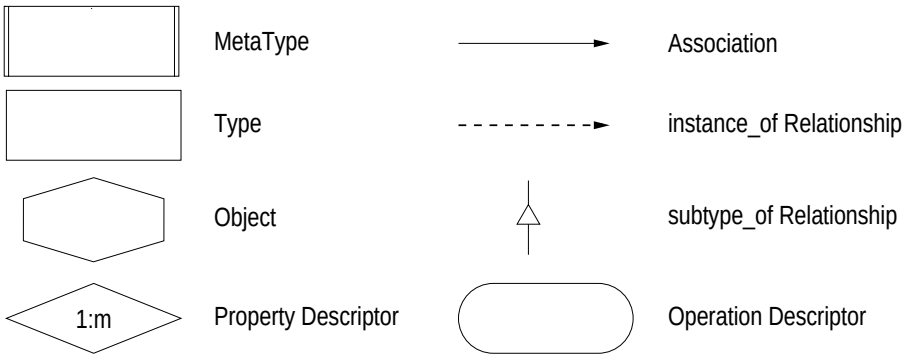

Figure A.1: The elements of the FACE notation.

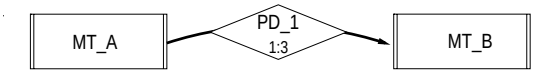

Figure A.2: An association in the FACE notation.

A.1 The FACE-notation

A.1.1 Purpose

The FACE notation was developed in the course of the evolution of the FACE data model. FACE is destined to be the basis of a visual composition tool. A notation to visualize compositions with a graphical user interface is therefore an essential part of the development. Compositions with FACE will happen especially on the class level, so classes must have a representation. Important in the FACE notation is that it makes clear that an association in FACE is reified or represented by a type-object. In the example of Figure A.2, a meta type MT_A has a connection to a meta type MT_B. This connection is described by the Property Descriptor PD_1. All the information that defines the association is concentrated in PD_1 and can be asked for or adapted there. The labeling 1:3 indicates that instances of MT_A can have up to three associate instances of MT_B in this relation.
Since the FACE notation shows links between elements, it emphasizes a compositional viewpoint. Structures that consist of associated types can be displayed with a balanced amount of drawing elements, as unimportant information is not shown.

A.1.2 Motivation of the chosen presentation

The FACE notation holds a symbol for every reified component. For each different type there is a different symbol, e.g. for types, meta types, property descriptors, etc. (see Figure A.1). A FACE composition is close to a design. The difference is that it is a) a composition which b) uses domain specific connections. The closeness to design diagrams has led us to take as much as possible from the UML symbols (see Figure A.5). The FACE type is thus equivalent to a UML class, the FACE meta type is a class which is slightly modified. For a Property Descriptor, which has no direct counterpart in the UML, a new symbol was created.
The FACE notion is not fixed. It must be extensible because the FACE meta model allows for introduction of new, domain specific types and possible connections between types and objects. The notation must be adaptable as well to depict these new connections.
Since the FACE notation is built upon an object- and class diagram technique, it has an immediate mapping to the Intermediate Notation (see Figure A.4).

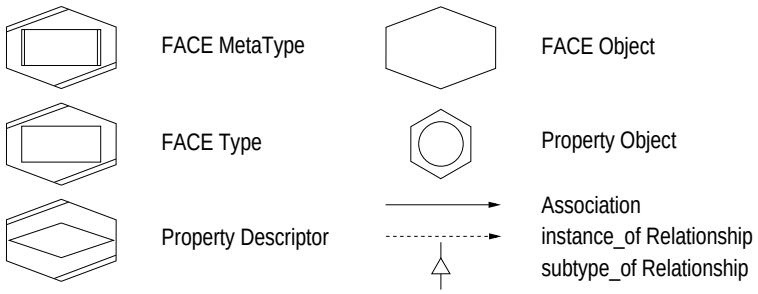

Figure A.3: The elements of the intermediate notation.

A.2 The Intermediate notation

A.2.1 Purpose

The intermediate notation was developed for the purpose of this work. It was felt that for certain explanations a notation would be needed which had more detail than the FACE notation but still did not convey *all* the details of the implementation. Especially to explain the complicated structure of the object-type connection we need a certain intermediate (hence the name) abstraction level which emphasizes the different kinds of objects that are involved in this relationship.

A.2.2 Motivation of the chosen presentation

The intermediate notation depicts four kinds of objects (see Figure A.3) :

- normal FACE-Objects.
- FACE-MetaTypes and FACE-Types which describe FACE-Objects.
- Property Descriptors which describe property objects.
- Property objects.

The hexagon of the UML object (see Figure A.5) is used for all the elements in the notation. This emphasizes that intermediate diagrams are basically object diagrams.
Type objects (FACE-MetaTypes, FACE-Types, Property Descriptors) are marked with double lines at the upper left and at the lower right sides and they have their respective symbol from the FACE notation (see Figure A.1) in the middle. Property objects are marked with an inscribed circle.
A mapping from the Intermediate notation to the FACE notation (see Figure A.4) can be done by eliminating the property objects and annotating associations between types with the association descriptor. In this way, we suppress unnecessary detail which clutters the view on the compositional structure

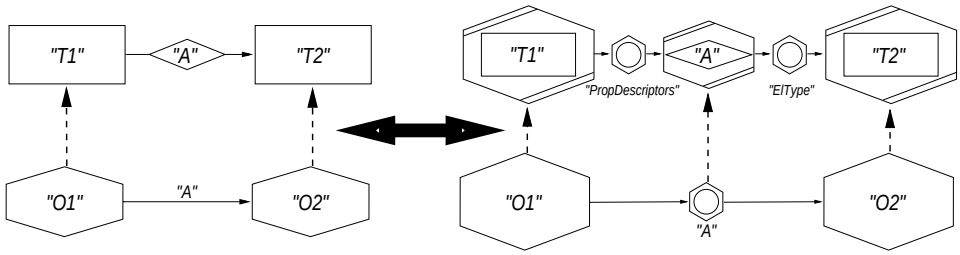

Figure A.4: The FACE and the Intermediate notation can be mapped onto each other.

A.3 The Unified Modeling Language

The *Unified Modeling Language* (formerly *Unified Method*) is the unification of the *Booch* method from Grady Booch, the *Object Modeling Technique* (OMT) method from James Rumbaugh, and the OOSE method from Ivar Jacobson as well as some ideas from other methodologies. It was created for specifying, visualizing, and documenting the artifacts of an object–oriented system under development. Since the methods it unifies are the leading object–oriented methods, the UML represents a *De facto* standard in the domain of object–oriented analysis and design. At the time of this writing, the version 1.0 was still under way.
The method offers a number of diagrams which span most of the aspects of an object–oriented system. Design is supported on different levels of abstraction, from the level of objects and classes up to modules and platforms. The list of diagrams is the following:

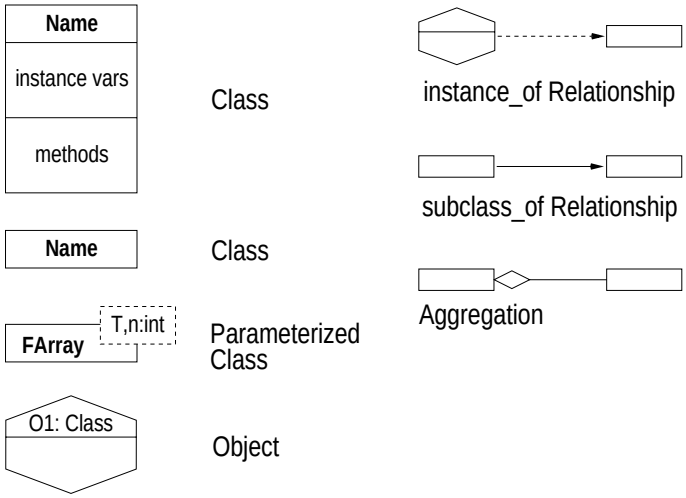

Figure A.5: Classes and objects in the UML-Notation.

- **Class Diagram**
  A class diagram shows the set of logical elements that make up a system: classes

and objects. Object diagrams are a snapshot of a particular moment of the systems runtime, while class diagrams show descriptions of possible systems.

- **Use Case Diagram**
  Use cases are generic descriptions of a transaction in the system. Emphasis is put on the the set of objects that are involved in the transaction, not in the sequence of sub-transactions that are performed.

- **Message Trace Diagram**
  Message trace diagrams shows the interactions among a set of objects along a time line. It is used to emphasize timing issues.

- **Object Message Diagram**
  An object message diagram displays the series of objects and the messages sent between them that implement a certain transaction. In contrast to the message trace diagrams, relationships between objects are emphasized.

- **State Diagram**
  ”The state diagram describes the temporal evolution of an object of a given class in response to interactions with other objects inside or outside the system” [BR95, Version 0.8, page 31]. The state diagram specifies the behavior of a class.

- **Module Diagram**
  A module diagram permits a view on the system under the aspect of physical modules (or files) of source code. It can display compilation dependencies between the files.

- **Platform Diagram**
  A platform diagram specifies the physical topology upon which the software system executes, which compromises processors (computing devices) and devices with no computational power. Displayed are connections along which information passes.

In the work presented here, class and object diagrams will be used. A legend with the most important drawing elements can be found in Figure A.5. For a complete reference see [BJR96] or look for the release 1.0 of the UML on

`http://www.rational.com/ot/uml.html`.

The UML was chosen as design notation for this document because it is expected to become a quasi standard and therefore will be understood by most of todays computer scientists. With a fine grained vocabulary and great expressive power, it is capable of depicting a design in much detail.

Bibliography

[BJR96] Grady Booch, Ivar Jacobson, and James Rumbaugh. Unified Modelling Language for Object-Oriented Development, September 1996. Version 0.91.

[BR95] Grady Booch and James Rumbaugh. Unified Method for Object-Oriented Development, 1995. Version 0.8.

[Bro87] Frederick P. Brooks. No Silver Bullet. Essence and Accidents of Software Engineering. *IEEE Computer*, pages 10–19, April 1987.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings OOPSLA 95*, ACM SIGPLAN Notices, 1995.

[dM95] Vicki de Mey. Visual Composition of Software Applications. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 10, pages 275 – 303. Prentice Hall, 1995.

[dR88] Jim des Rivières. Meta-Level Facilities in LISP. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustroup. *The Annotated C++ Reference Manual (ARM)*. Addison Wesley, 1990.

[Fer89] Jacques Ferber. Computational Reflection in Class based Object-Oriented Languages. In *OOPSLA 89 Proceedings*, ACM SIGPLAN Notices. ACM, ACM Press, November 1989.

[FG93] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall International Editions. Prentice-Hall, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Ibr90] Mamdouh H. Ibrahim. Report on the Workshop Reflection and Metalevel Architectures in Object-Oriented Programming. In *Addendum to the Proceedings OOPSLA/ECOOP 90*, ACM SIGPLAN Notices, pages 73–80, 1990.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[Kic94] Gregor Kiczales. Foil for the Workshop on Open Implementations. WWW, October 1994.

[Kru92] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):133 – 183, June 1992.

[McA95] Jeff McAffer. Meta-Level Programming with CodA. In *OOPSLA Proceedings*, 1995.

[ME96] Theo Dirk Meijler and Robert Engel. Making Design Patterns explicit with FACE, a Framework Adaptive Composition Environement. In *EuroPLoP Conference Proceedings*, May 1996.

[Mei93a] Theo Dirk Meijler. *User-Level Integration of Data and Operation Resources by Means of a Self-Descripting Data-Model, Part II: Formalization*. PhD thesis, Erasmus Universiteit Rotterdam, September 1993.

[Mei93b] Theo Dirk Meijler. *User-Level Integration of Data and Operation Resources by Means of a Self-Descripting Data-Model*. PhD thesis, Erasmus Universiteit Rotterdam, September 1993.

[Mei96] Theo Dirk Meijler. An Overview of FACE, a Framework Adaptive Composition Environment. unpublished techreport, September 1996.

[MN96] Simon Moser and Oscar Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, September 1996.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Professional Computing Series. Addison-Wesley, 1996.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 1, pages 3 – 28. Prentice Hall, 1995.

[SB86] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 1986.

[Sig96] Stefan Sigfried. *Understanding Object-Oriented Software Engineering*. IEEE Press, 1996.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPSLA Proceedings*, ACM SIGPLAN Notices, pages 38–45, September 1986.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, 1994.