T U M

I N S T I T U T   F Ü R   I N F O R M A T I K

Design and Implementation of FOONET - a Framework for object-oriented Network Design

Volker G. Fischer

TUM-I9919
Dezember 99

T E C H N I S C H E   U N I V E R S I T Ä T   M Ü N C H E N

Design and Implementation of *FooNet*, a Framework for object-oriented Network Design

Volker Gerd Fischer

Institut für Informatik
Lehrstuhl für Rechnerkommunikation
Technische Universität München, Germany

Keywords:

Telecommunication Network Design, Object-oriented Framework, Object-oriented Analysis and Design, Extensible Markup Language

Abstract

This report presents the design and implementation of *FooNet*, an object-oriented application framework for telecommunication network design. The characteristics of *FooNet* in contrast to other planning environments are its consequent object-oriented design and the support of reuse techniques on different levels of abstraction. *FooNet* comes with a library of helpful algorithms used in network design problems. A further important feature of *FooNet* is its support of XML for data exchange. XML offers a new dimension of communication eliminating incompatibilities between various applications.

X-CRClassification

B.4.3, C.2.1, D.1.5, D.2.2, I.7.2

Contents

Motivation

Network design, the task of planning and managing communication networks, comprises a variety of techniques and knowledge evolving from many different fields of science[1]. These sciences include optimization, graph theory, forecasting, simulation and modeling, knowledge representation, decision theory, finance, electrical engineering and computer science. Due to its telephony heritage and the electro-technical problem part, wide area network design has also a long standing history in the field of electrical engineering (remember for example the work of Erlang in the early decades of this century).

The relation between computer science and network planning is at least twofold. First, telecommunication network planning was one of the first applications for which the computational power (supported by advances in mathematical optimization) was and still is used. Second, the explosion of communication service demands (the Internet) causes a vital interest of computer science in telecommunication network design.

However due to its heritage, software engineering aspects have played only a minor role in network design. Practical engineers have traditionally been fond of imperative programming languages, such as FORTRAN and C. The trend in computer communication is towards object-oriented software engineering with its ability to cope with complexity even for large problem sizes and to reuse software. Some projects have already shown the power of object-oriented approaches in communications, for example in the field of network-management (OSI network management) or protocol design ([Böc97]). But the network design task itself lacks tools supporting the object-oriented programming paradigm. It seems that especially in telecommunication the application of object-oriented software design is promising. For example, Jackson showed in [JGJ97] that the reuse of telecommunication software by AT&T was between 40% and 92%.

In this report, a software tool called *FooNet*[2] is presented which is the result of submitting the telecommunication network planning process to an object-oriented design&analysis (OOA&OOD). *FooNet* is an application framework that releases the designer from "reinventing" the parts of the software design that are common to all network design problems. Object-orientation can achieve this without limiting the generality of the design process itself by making restrictions that the designer cannot overrule. This report focuses on software design, but it is assumed that the reader has at least basic knowledge of telecommunication network planning as well as object-oriented design principles.

A further problem in network design is the lack of agreed upon standards for data exchange between applications. Every tool has its own (sometimes even unpublished) interface and data-format. This problem is not a peculiarity of network design, but a more general

[1]In this report the terms "network", "communication network" and "telecommunication network" are used synonymously

[2]Framework for the object-oriented Network Design

one that can be found in many areas of data processing. With the success of Internet technologies, the W3C consortium has introduced a technology called **XML** that allows the exchange of almost arbitrary information between different applications. In this report, a data-format compliant to XML is introduced. Examples of the abilities of this format are given. *FooNet* is designed to produce and process XML compliant output.

The report is organized as follows: The first two chapters give a short introduction in telecommunication network planning and recent advances in object-oriented software design. The third chapter contains the documentation of the design and implementation of *FooNet*. The fourth chapter discusses the capabilities of XML as a data exchange format. The report concludes with a summary and an outlook to future work.

Acknowledgements

I'd like to thank (in alphabetical order) Thomas Erlebach, Manfred Jobmann and Hans-Peter Schwefel for their helpful comments and critics.

Chapter 1

Telecommunication Network Design

1.1 Introduction

It is a non-trivial task to formulate a network design problem by itself. Due to the fact that a network must satisfy the needs of an enterprise and every enterprise has different requirements on communication, network design is a context sensitive problem.

Generally, communication networks are designed using *hierarchical* structures. Two different types of hierarchies can be identified: The **topological** hierarchy imposed by the different **network layers** (see Figure 1.1) and the **logical** hierarchy implied by **tiers** (see Figure 1.2).

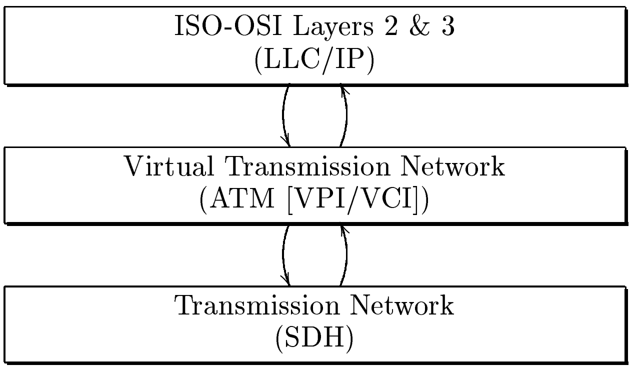

Figure 1.1: Example for a topological Hierarchy

Hierarchical network design is a frequently used strategy to cope with the complexity of the problem. The hierarchical design divides a single network layer into **tier**-levels by grouping several nodes and considering them as one new node of a "higher tier". The B-WiN (see Chapter 1.3) has two tier-levels: The lowest tier contains the **access-** or **end-nodes**. Each end-node represents a canonical source/destination that sends/receives traffic into/from the network. The higher tier, called **backbone-tier**, contains the **backbone nodes**. Each end-node is connected to exactly one backbone node. Figure 1.2 shows a star-topology between the access-nodes and the backbone-nodes.

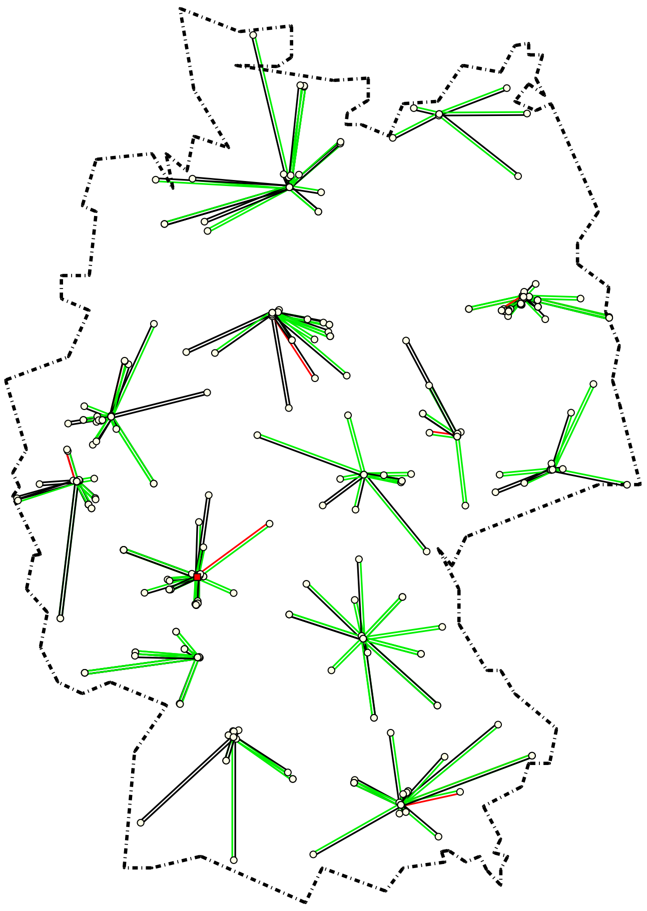

Figure 1.2: B-WiN Access Design Example

1.2 Decomposition Planning

When accepting the following statements:

1. network design as a whole, i.e. *overall network design*, is too complicated to be solved in one step
2. an obvious "optimum" solution does not exist in general due to multi-criteria objective functions and incomplete knowledge
3. network design is an iterative, user controlled procedure
4. networks exist within enterprises and must be adjusted to the goals of the enterprise

Consequently, this leads to **decomposition planning** (see Figure 1.3) with well defined and as much as possible *independent* subtasks combined with *alternate optimization*, i.e. optimization with recourse, and interaction with the designer.

The decomposition planning process must be embedded in the topological hierarchy of the network, e.g. it must be executed for the different topological hierarchies. The more

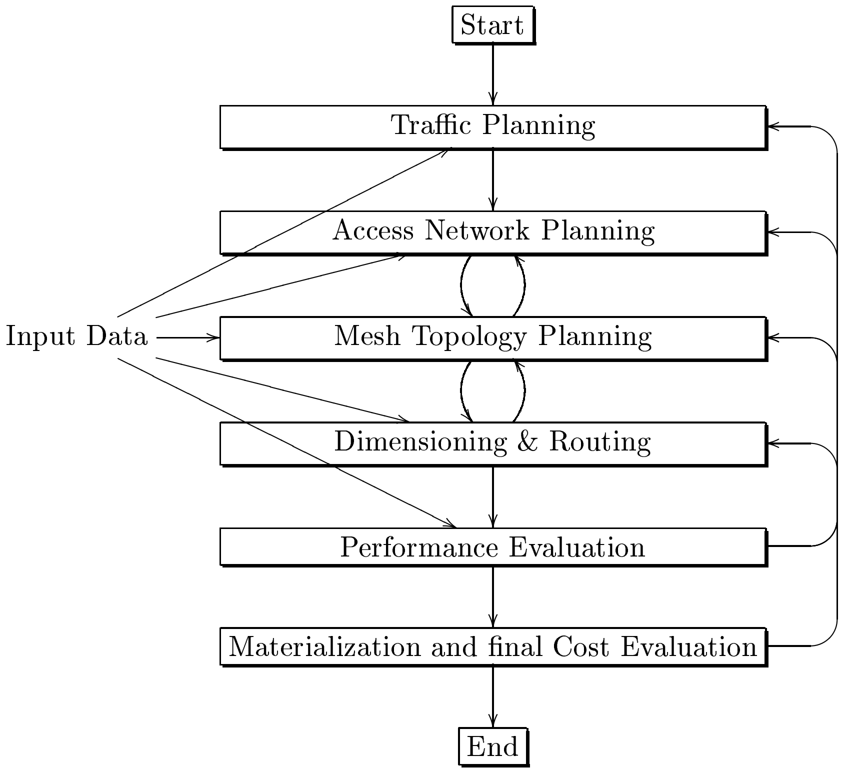

Figure 1.3: Decomposition Planning (taken from [Fri98])

hierarchy levels a designer wants to take into account, the more complex the planning problem becomes. Only a joint optimization over all layers and tiers safeguards an optimal solution, but such an approach usually results in an intractable complexity. Optimality is therefore sacrificed for tractability.

The following terminology is used throughout this report:

- a **facility** is an arbitrary network element such as a router or an arbitrary service such as a leased line. A different set of facilities exists for each topological network layer. The costs entailed with a facility divide into:
  - **setup costs**, that arise when introducing a new facility in the design (e.g. buying a new router)
  - **reoccurring costs**, that arise in regular intervals (e.g. leasing rates)
  - **termination costs**, that arise when removing a facility from the design (e.g. when terminating a contract)
  - **usage-dependent costs** (e.g. call minutes)
- a **topology** in the context of network design is the physical or logical layout of network facilities. Common topologies include star, ring, bus or mesh. A common representation of the network topology is a graph.
- a **network design**[1] is the specification of topology and configuration forming a productivity network. To put it in other words, the topology specifies where to put

[1]In contrast to the network design task itself here a specific network design, i.e. a realization of a

the facilities and how to interconnect them, the configuration specifies which facility is used for each topological element and how to set it up.

- a **requirement** or **commodity** is a communication demand between two nodes, usually measured in terms of Megabit per second (Mbps).

1.3 The B-WiN Planning Task Example

To illustrate the problems arising in network design the (simplified) planning task of the Breitband Wissenschaftsnetz (*B-WiN*) provided by Deutsches Forschungsnetz Verein (*DFN-Verein*) is shown in Figure 1.4.

Given:

- ca. 400 node locations
- (measured) traffic busy-hour matrix (**IP**-traffic) (quite asymmetric!)
- tariffs of the Deutsche Telekom AG
- OSPF-routing

minimize **cost** subject to

- edge capacities are available in incremental 2 Mbps steps beginning at a minimum capacity of 34 Mbps
- survivability: nodes having $\geq c$ Mbps access rate should be two-edge-connected
- hop-limit: h hops
- backbone lines should not have more then u% utilization

Output:

- number and location of backbone nodes
- access and backbone topology
- line capacities
- OSPF routing parameters

Figure 1.4: Sample B-WiN Planning Task

It has been shown (e.g. in [MS81]) that even subproblems of the network design problem are strongly $\mathcal{NP}$-complete[2]. Another important fact is that there exists no generic algo-

single network, is meant. From the context it should be immediately clear to which of the two meanings it is referred

[2]which means that under the assumption $\mathcal{P} \neq \mathcal{NP}$ this problem is not solvable in an efficient way

rithm that can take into account the various constraints imposed on the problem. Even if formulated as a mathematical program there is no efficient way to solve this problem. Therefore many known algorithms used in network design problems rely on heuristics that have to be adapted by the network planner to meet the requirements of his planning task.

Chapter 2

Introduction to Object-Oriented Software Design

2.1 Overview

This section gives a brief overview of recent advances in object-oriented software design. It is assumed that the reader has some experience with object-orientation, otherwise the author would recommend [Boo91, Cop92, Sto97] for a detailed and application-oriented introduction.

Some important terms and concepts in object-orientation are introduced as follows:

- **Object-Oriented Analysis (OOA)**: Object-oriented analysis is a method of analyzing that examines the requirements on the software from the perspective of classes and objects found in the problem under consideration.
- **Object-Oriented Design (OOD)**: Object-oriented design is a method of design compassing the process of object-oriented decomposition and notation (here in the language of UML) for depicting both logical and physical as well as static and dynamic properties of the problem under design. The most important principles in object-oriented design are abstraction, encapsulation, modularization and hierarchy.
- **Object-oriented Programming (OOP)**: Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class.
- **Inheritance**: Inheritance is a hierarchical relation among classes, in which one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a “is-a” hierarchy among classes in which a subclass inherits from one or more generalized superclasses. A subclass typically specializes its superclass by augmenting or redefining the existing structure or behavior.
- **Polymorphism**: Polymorphism is a concept in type theory wherein a name may denote instances of many different classes (usually related by some common superclass in C++). **Dynamic Binding** is a consequence of polymorphism, which implies that sending the same message to different objects could stimulate different behavior. Technically this is realized by the use of **virtual functions**.

- **Interface**: An abstract base class which provides a set of virtual functions is called an interface.
- **Persistence**: Persistence is the property of an object through which it can transcend time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created). Persistence in *FooNet* works with character streams and relies on two complementing methods: **serialize** (write object in a stream) and **de-serialize** (read object from a stream).
- **Class Library**: A class library is a collection of reusable classes that rely on object-oriented design paradigms such as hierarchy and polymorphism. Usually, class libraries provide support in solving problems belonging to a specific problem category.

2.2 Unified Modelling

The object-oriented software design paradigm does not dictate a methodology of how the classes and the relationships between them are derived from the system. This part has to be performed by the designer[1] supported by a detailed object-oriented analysis.

However, there exist several process models (for an overview see e.g. [NS99]) which guide the designer through this task. The process model that is used here is the **Unified Process** proposed in [Jac99], which is a use-case-driven and incremental approach and consists mainly of the five stages illustrated in Figure 2.1.

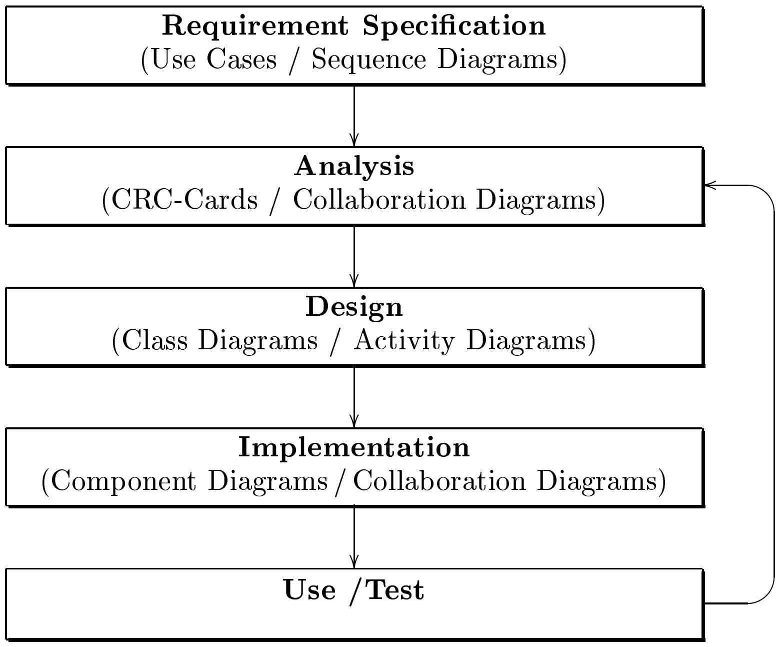
Requirement Specification
(Use Cases / Sequence Diagrams)
Analysis
(CRC-Cards / Collaboration Diagrams)
Design
(Class Diagrams / Activity Diagrams)
Implementation
(Component Diagrams / Collaboration Diagrams)
Use /Test

Figure 2.1: Object-oriented Software Design Process

Connected with the Unified Process is **UML**, the Unified Modelling Language, that supports the process model by providing a set of easy to read but expressive diagrams. UML is

[1]and is therefore often considered as an art

a symbolic language for specifying, constructing and documenting (the semantics of) software systems. It is standardized by the Object Management Group (OMG) in [OMG97] and has become the “Esperanto” of object-oriented design. The terms in the brackets beneath each stage of the diagram in Figure 2.1 denote examples of which part of the UML notation may be appropriate when working in this stage. However, UML does not dictate a particular process, it is more a “blueprint” for software design. A short introduction to UML that is sufficient to understand the diagrams in this report is presented in Appendix A.

2.3 Software Reuse

2.3.1 Motivation

The implementation of complex software systems remains resource expensive and error prone. Already the design of a medium sized computer program is a nontrivial task. Much of the costs stem from the rediscovery and reinvention of core concepts and components.

The concept of software reuse originates from the observation that a software designer should concentrate on the peculiarities of his/her task instead of solving problems that have already been solved many times before. Software reuse is commonly defined as “*the systematic development of reusable components and the systematic reuse of these components as building blocks to create new software systems*”.

A reusable component may be code, denoted as **code reuse**, but the bigger benefits of reuse come from a broader and higher-level view of what can be reused: software specifications, abstractions, design patterns and frameworks. This is commonly denoted as **design reuse**.

FooNet provides an application framework (see Chapter 2.3.3) for network design problems and as such provides **design reuse**, but builds itself on top of two other reuse techniques:

- **Idioms**: Idioms allow **code reuse** by providing higher level datatypes, such as lists, queues, and graphs (see e.g. [Cop92]). The C++ language provides a number of idioms, which are standardized in the Standard Template Library (STL, [Int97]). Additionally the idioms provided by the Graph Template Library (GTL, `http://www.fmi.uni-passau.de/Graphlet/GTL`) are used. In some publications this kind of code reuse is denoted as **generic programming**.
- **Design Patterns**: Design Patterns are a (very successful) **design reuse** technique. A short description is given in Chapter 2.3.2.

From this perspective, *FooNet* has a three level reuse hierarchy: Classes, objects and idioms on the lowest level of abstraction, design patterns on a medium level and frameworks on the highest level.

2.3.2 Design Patterns

A design pattern [GHJV95] describes a (often reoccurring) problem, the core of a simple and elegant solution together with the context in which the solution works, and its cost and benefits. Design patterns serve as the micro-architectural elements of frameworks, but due to their abstractness, they cannot be expressed as classes or lines of code.

The following list gives a short overview of all design patterns[2] that have been used in *FooNet* together with examples of where the design patterns occur:

- **Letter-Envelope** (also known as **Handle-Body** or **Bridge**)
  A *letter-envelope*-pattern decouples an abstraction, denoted as envelope, from its implementation, denoted as letter. The advantage of this design pattern lies in the fact that the letter object can vary independent from the envelope which allows a greater flexibility of usage.

  A commonly used example for the letter-envelope design pattern is a reference-counted class. The node-envelope of *FooNet* administrates a pointer and a reference-counter to the node-letter. The actual node-letter object is duplicated only if necessary. This significantly reduces resources.
- **Factory**
  A *factory* defines an interface for creating an object, but lets subclasses decide which class to instantiate. Usually a factory provides a method `produce()` which creates an object of a known base class. Factories prevent to include user-specific code in the class design.

  FooNet provides a *factory* pattern for network facilities. To produce a certain kind of a network facility (for example an IP-router), the user sends a message to that factory with a set of requirements (for example having at least a throughput of 100 Megabit per second) and the factory returns the cheapest network facility found (for example a Cisco IP Router having 1000 Megabit per second throughput) which meets the requirements. The user can add more manufacturers dynamically by deriving a new sub-factory. The selection process and the vendor-specific data is shadowed by the design pattern.
- **Strategy**
  A *strategy* defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategies allow to formulate a skeleton of an application which is independent from a concrete realization.

  The (static) routing interface in *FooNet* is a typical example for a *strategy*. For example, testing the network for overload needs the routing functionality, but whether routing is performed by an OSPF, PNNI or some other kind of routing algorithm is irrelevant.
- **Visitor**
  A *visitor* represents an operation to be performed on the "elements of an object structure" (see example below). A visitor allows to define a new operation without changing the classes of the elements on which the visitor design pattern operates.

  The topology of a network layer forms an object structure that consists of topological (node and edge) elements. *FooNet* provides a *visitor* pattern for topological elements. For example, if a user wants to know the average load on the edges of that layer, he could derive this functionality from the visitor interface and simply call the visit-all-edges method.

[2]For a more detailed description and sample implementations please refer to [GHJV95]

- **Composite**
  A *composite* is used to represent part-whole hierarchies of objects, when differences between compositions and individual objects can be ignored.

  The logical hierarchy of a network is a typical example of a *composite* pattern. The two types of nodes (end-nodes and tier-nodes) form a tree structure, but e.g. when calculating the costs of a network, there is no difference between tier-nodes and end-nodes.

- **Warper** (also known as **Decorator**)
  A *warper* dynamically attaches additional responsibilities and functionalities to an object. It provides a flexible alternative to sub-classing.

  Warpers are frequently used for attaching visualizing capabilities to an object. So does the GraphWin-warper of a layer object in *FooNet*. The GraphWin-warper can be transparently used wherever a layer object is expected, but observes any message sent to that layer-object and updates its visualization if necessary.

- **Builder**
  A *builder* separates the construction of a complex object from its representation so that the same construction process can create different representations.

  The XML-Factory class calls on a builder design-pattern which is provided by the SAX-parser (`http://www.jezuk.demon.co.uk/SAX/`). This parser takes an XML document (see Chapter 4) and builds a parse tree from which the internal representation is derived.

- **Prototype**
  A *prototype* specifies the kind of object to create by using a prototypical instance and creates a new object by copying this prototype.

  This design pattern is used wherever a pointer to a base class has to be cloned. Technically, every class supporting the *prototype* pattern provides a method `clone()` which guarantees that a perfect copy is made.

- **Command** and **Observer**
  A *command* encapsulates a request as an object, thereby parameterizing the clients of the *command* pattern with different requests. An *observer* defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

  Both design patterns are typically used in graphical user interfaces. A menu-bar entry could be realized as a command. An observer can guarantee that different views of a single object are consistent. Since both patterns are not used in the base framework of *FooNet*, they are not discussed in greater detail.

- **Virtual Constructor**
  A *virtual constructor* allows to build an object of known abstract type but unknown concrete type, which is important when de-serializing objects.

  A builder that sequentially reads an object out of a stream usually knows the base type of the next expected object, but not its concrete type. For example, the XML-Factory knows that the next object has to be of the type “facility”, but it does not know which concrete type this facility has, i.e. its subclass. The *virtual constructor* pattern solves this problem.

2.3.3 Frameworks

Object-oriented application frameworks are a promising technology for reusing proven software designs (**design reuse**) and implementations (**code reuse**) in order to reduce the costs and improve the quality of software. A framework is defined as follows [Joh97, FSJ99]:

> A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. Frameworks aim at solving a family of similar problems.
>
> The purpose of a framework is to provide the skeleton of an application that can be customized by an application developer.

Frameworks differ from class libraries by their additional reuse of high-level design, since frameworks do not only define classes but also a **model for interaction** between them. A framework is therefore a "semi-complete" application (by the use of **inversion of control**[3]) that can be specialized to produce custom applications. Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Extensibility is supported by providing explicit hook methods that allow applications to extend the interface.

Examples for widely known frameworks are:

- AWT & JavaBeans (`http://java.sun.com`)
- Qt (`http://www.trolltech.no`)
- MFC (`http://www.microsoft.com`)
- Abacus (`http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus.html`)

There exist two different kinds of frameworks:

- **White-box** frameworks rely heavily on object-oriented language features like inheritance and dynamic binding in order to enhance extensibility. To use white-box frameworks, intimate knowledge of their internal structure is needed. *FooNet* is designed to be a white-box framework.
- **Black-box** frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. The functionality of black box frameworks is based on design patterns, such as strategy and command. Black-box frameworks are less flexible than white-box ones, but usually easier to use. The Qt-based visualization `C_qt_layer_warper` in the *FooNet* extensions is an example for a black-box framework that provides a graphical user interface.

Several properties are commonly demanded for frameworks:

- completeness: the framework should provide all necessary functionality.

[3]Sometimes also denoted as the Hollywood principle: "don't call us, we call you"

- efficiency: the framework should provide efficient implementations of the relevant, time-critical parts.
- flexibility (reusability): the framework should be applicable in more than one context.
- ease of use: the user of the framework should only be responsible for the part of the implementation that is software specific.
- extensibility: the framework should have the ability to grow with future requirements.
- portability: the framework should not be restricted to a specific hard- or software.

Chapter 3

A Framework for Object-Oriented Network Design (*FooNet*)

FooNet is a result of the upcoming PHD thesis [Fis00] that deals with various aspects of network design problems.

3.1 Overview

FooNet is a white-box application framework that is designed to significantly reduce the development effort of network design applications. At the moment, its focus is at telecommunication network design, but it could be easily expanded to cope with other network design problems (such as road networks or gas pipelines) as well.

The design of *FooNet* consists of two parts:

- A core framework, which is the result of an object-oriented analysis & design process. It represents the abstractions, interfaces and interaction models of network design problems.
- An extended framework that builds on top of the core framework and enriches it by providing additional functionality and applications. It is expected to grow very fast in the future, providing a library of algorithms helpful for telecommunication network design.

FooNet provides a system of base classes from which the application specific subclasses can be derived. All problem independent parts are invisible to the user, so that he can concentrate on the problem-specific algorithms and data-structures. As a white-box framework, *FooNet* relies heavily on inheritance from base-classes and overloading pre-defined hook-methods, i.e. the user derives his specializations from a set of appropriately designed (interface) classes. Virtual functions provide default implementations that are often useful, but can be overloaded, if required. Sometimes such virtual functions do nothing at all, but they allow the user to add some functionality. The task of "inventing" an appropriate network algorithm and/or representation cannot be completely taken off the designer, since this is problem-dependent[1]. However all other activities in network design (e.g. persistence, displaying, editing, forecasting, loading, performance evaluation, sensitivity testing) are managed by the framework.

The next paragraphs introduce the design of *FooNet* in greater detail. References to the classes or methods in the implementation are in `teletype` font. References to design patterns are in *italics* font.

3.2 The Core Design Classes

The UML static class diagram of the core framework is shown in Figure 3.2. The reader is asked to take a look at this static class diagram regularly, since it helps to clarify the model of interaction between the classes.

In the following sections, the main abstractions are introduced, but the details of the implementation are left to the program documentation that is included in the *FooNet* distribution[2].

3.2.1 Network Nodes

The class `C_node` is an abstraction of a network node. A node is a location (given as `C_coordinate`) of an arbitrary source or destination of a traffic requirement. A node may represent a single workstation or even a whole corporation. Network nodes are identified by their unique name (the **id** of the node). Nodes are elements of the network topology (see Chapter 3.2.6).

A common strategy of network design algorithms is the grouping of nodes and considering them as one new node of a "higher tier". This introduces a hierarchy of the nodes, the **logical hierarchy** of the network, in which it is possible that one node is a component of more than one higher tier node, but no higher tier node is allowed to be component of a lower or equal tier node.

The nodes on the lowest hierarchical level are denoted as **end nodes**, all others are called **component nodes**. An example of an access network design is shown in Figure 1.2.

Nodes are implemented by `C_node_letter` objects using the *composite* design pattern. `C_node` is an envelope according to the *letter-envelope* design pattern that hides a (reference counted) letter object `C_node_letter`.

Each node contains a `C_node_info` object that allows the user to store additional information by deriving his own information class from it.

3.2.2 Commodities

The class `C_commodity` represents the communication demands in terms of bits per second between a set of network nodes, i.e. the **traffic matrix**. Traffic matrices are usually

[1]For a detailed discussion of this subject see [Fis00]

[2]There is an extended version of this report available at `http://wwwjessen.informatik.tu-muenchen.de/~fischerv/foonet` that includes the code documentation

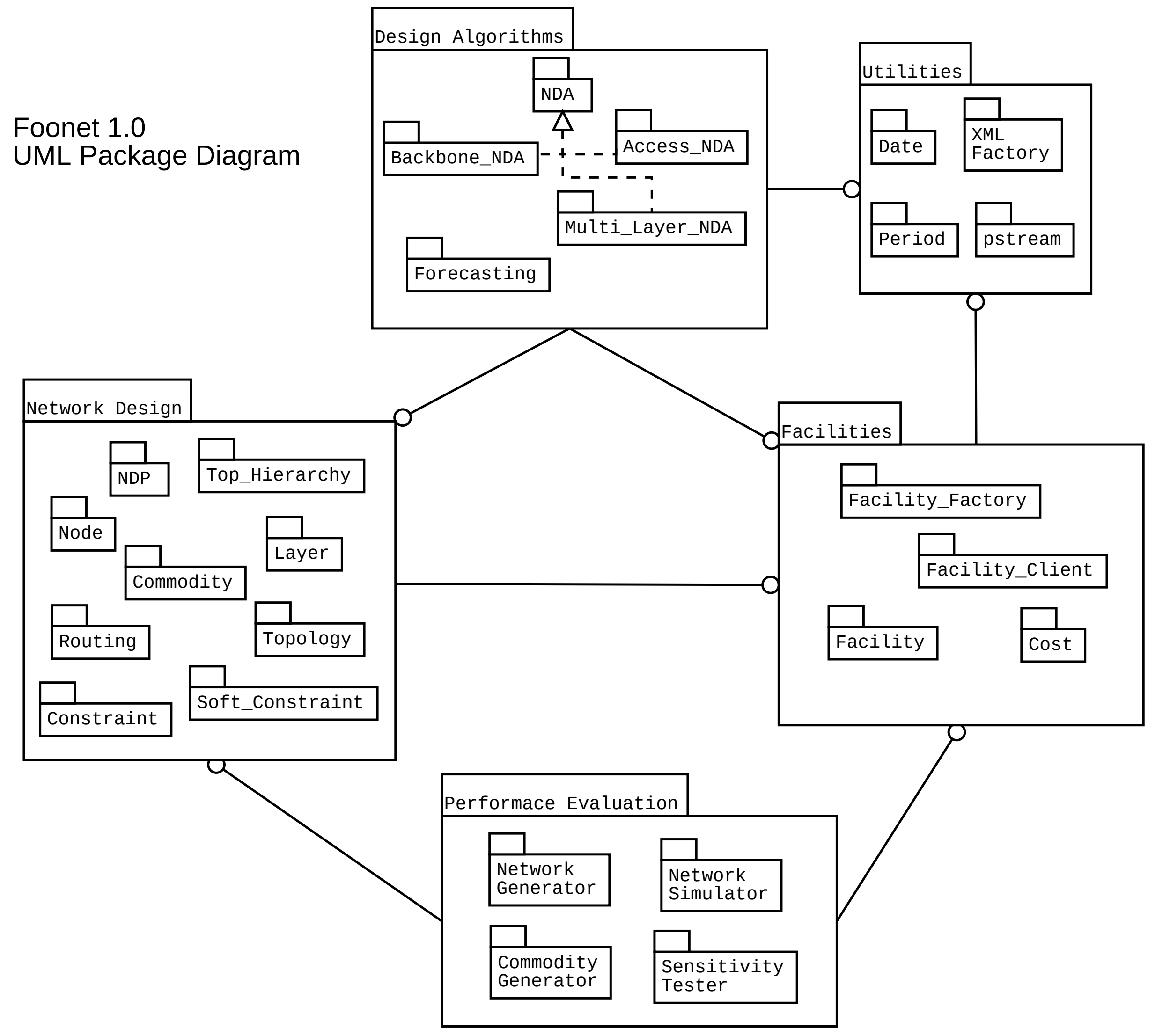

Figure 3.1: UML Package Diagram

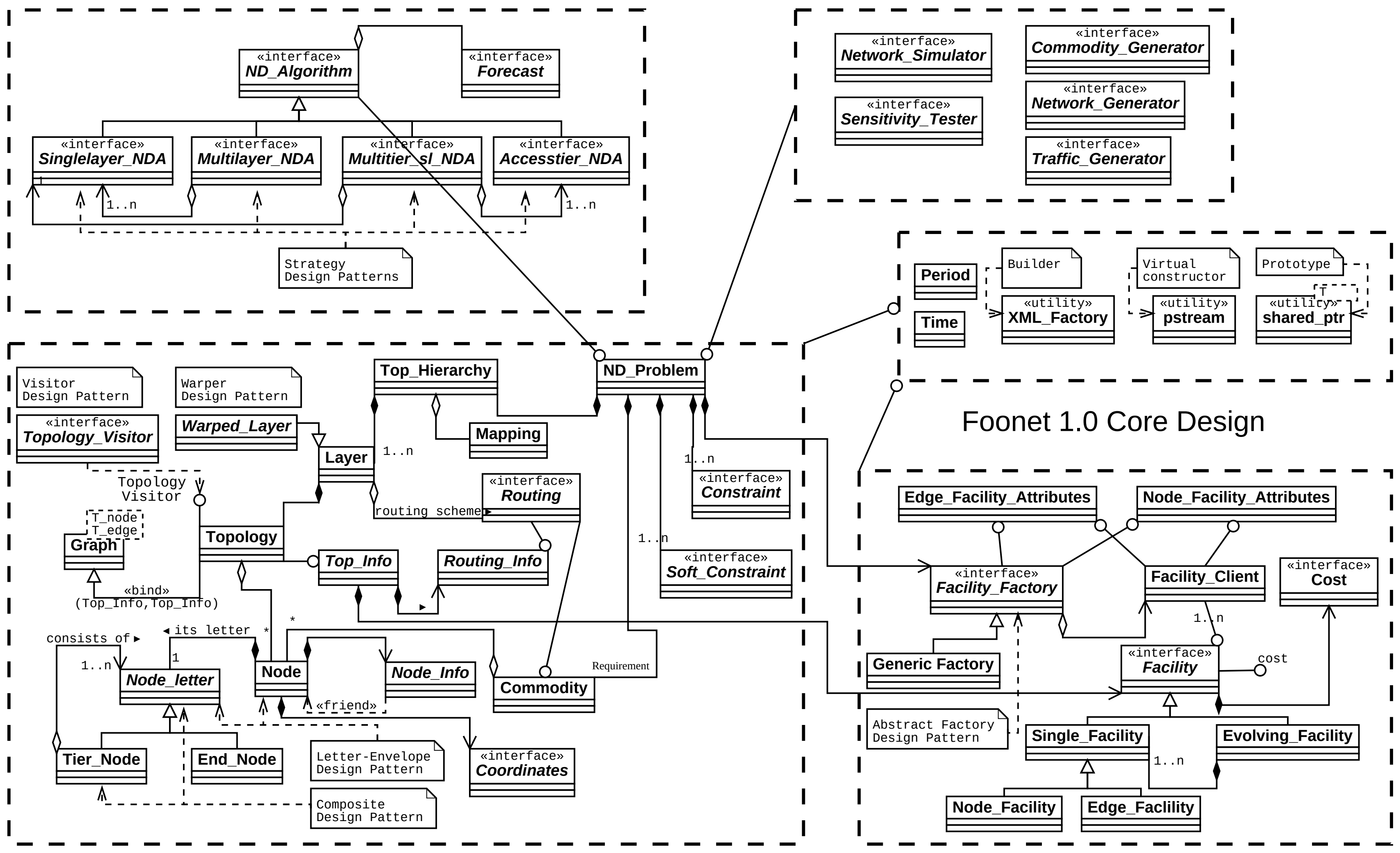

Figure 3.2: Static UML Core Class Diagram

asymmetric. A single commodity is identified (`C_commodity::t_index`) by a pair of network nodes, the source and the destination. The class `C_commodity` provides methods for the computation of the traffic demands on each tier level.

3.2.3 Facilities & Cost

The class `C_facility` is an interface that represents all (edge[3] and node) facilities of a network, whereby a facility is a component belonging to a single network layer (see Section 3.2.5). Each facility has a reliability and a capacity.

The most important attribute common to all facilities is the cost interface that allows to calculate the expenses necessary during their lifetimes. These costs are stored in a `C_cost` object and calculated according to the discounted cash flow (DCF) formula[4] using the setup-, reoccurring- and termination (respectively upgrade) costs. The cost interface is intended to cope with usage-dependent costs in the next software release.

The specializations of `C_facility` include single facilities, `C_single_facility`, on the one hand and evolutionary facilities, `C_evol_facility`, on the other hand. Single facilities divide into edge, `C_edge_facility`, and node, `C_node_facility`, facilities. Evolutionary facilities represent a series of succeeding single facilities over time[5]. Series of facilities are non-overlapping, i.e. at all times, exactly one single facility is present.

3.2.4 Facility Factory & Facility Client

The class `C_facilityclient` represents all available facilities of a single network layer. Internally it stores a list of `C_facilityfactory` classes. Instances of `C_facilityfactory` subclasses may represent a single vendor, product-line, public carrier service etc. To obtain a list of appropriate facilities the user must specify the characteristics by (node or edge) attributes, which are based on `C_ef_attribute` for edge-facilities and `C_nf_attribute` for node-facilities, respectively. `C_facilityclient` is a realization of the *factory* design pattern.

`C_facilityclient` enables the designer to handle different facility factories, but it burdens him with the task of choosing a facility out of a list of appropriate ones, since one facility may have cheaper setup costs while the other has cheaper reoccurring costs.

Sometimes vendors offer special "upgrade-fees" when switching from one facility to another. This can be modeled by over-riding the upgrade-cost method (`C_facilityfactory::upgrade()`). If no appropriate facility is found, an empty object is returned.

An instantiation of a facility factory is given by the class `C_generic_factory` which implements an economy of scale facility factory (see again [Fis00]) with the economy of scale parameter[6] a. Additionally a list of discrete values for the available capacities can be specified. When producing a facility, the cheapest facility that meets the given attributes is returned.

[3]This may also include transport services offered by public carriers

[4]For a more detailed introduction in this subject see [Fis00]

[5]For example, the facility f_1 is known to be replaced by facility f_2 at planning time Δ

[6]also called power-law

3.2.5 Network Layer and Topological Hierarchy

The class `C_layer` is a collection of all information associated with a single network layer, e.g. SDH, ATM or IP. A layer object contains information about the available facilities (in a `C_facilityclient` object), the used routing-algorithm (in a `C_routing` object) and its topology (in a `C_topology` object).

The network layers are organized according to the **topological hierarchy** of the network (`C_topological_hierarchy`). The mapping between peer layers can be supported by a mapping object (`C_mapping`)[7]. It provides routing functionality between peer layers and allows for example to query which facilities of a lower layer (e.g. a SDH-path) are used by a facility on a higher layer (e.g. a ATM-VP) and vice versa.

Additional functionality can be attached to a layer by the class `C_warped_layer` which is an instantiation of the *warper* design pattern.

3.2.6 Topology

The class `C_topology` represents the topology of a single network layer, i.e. the layout of its nodes and edges. `C_topology` is implemented by a parameterized `C_graph` object. Every topological element, i.e. all nodes and edges, contains a `C_topology_info` object. Each topological node-element is associated with a `C_node` object and each topological edge-element is associated with two `C_node` objects, the source and destination node.

`C_topology_info` is a base class for (node and edge) information associated with every topological element. It contains a facility (`C_facility`) object, a routing information (`C_routing_info`) object, and a variable to store the current load in terms of bits per second. It is designed to be overloaded if additional information is required.

`C_topology_visitor` represents an abstract interface of a *visitor* design pattern for topological elements. For example a `C_topology_visitor` object is used to calculate the costs of the topology by visiting all topological elements and summing up the associated facility costs.

3.2.7 Routing

The class `C_routing` is an interface for a static routing (respectively loading) algorithm (see e.g. [Cah98]). The main functionality is in the `C_routing::load()` method, which loads a commodity on the topology according to the routing algorithm by setting the `load` variable in each topological element.

To support all possible routing algorithms, each topological element stores a `C_routing_info` object, which should be overloaded to support the needs of the routing algorithm. In its basic implementation `C_routing_info` contains no information.

3.2.8 Network Design Problems, Forecast & Constraints

The class `C_ndp` represents both, a network design problem and its solution. It serves as the parameter to the network design algorithms (`C_nda`).

[7]For a detailed introduction to this subject see [Auc99]

A `C_ndp` object stores

- a topological hierarchy (`C_topological_hierarchy`) object that is subject to the planning
- a commodity (`C_commodity`) object representing the load that the highest network layer has to carry at the time of the planning
- the planning period (`C_period`)
- a forecasting algorithm (`C_forecast`)
- a set of constraints (`C_constraint`)
- a set of soft constraints (`C_soft_constraint`)

The method `C_ndp::objective()` calculates the objective function of the current state of the network design (which is given by the topological hierarchy object). The functionality provided by the base class calculates the objective function by summing up the cost of all used facilities over the planning period plus the (weighted) penalties from all soft constraints.

The class `C_forecast` is an interface class for forecasting algorithms, that calculates an expected commodity at any time in the future given a commodity at the present time.

The class `C_constraint` is an interface for constraints. A constraint takes a network design problem (in the method `C_constraint::is_fulfilled()`) and decides whether constraint is fulfilled or not.

The class `C_soft_constraint` is an interface for soft constraints. A soft constraint takes a network design problem (in `C_soft_constraint::penalty()`) and returns a penalty (i.e. a non-negative number) if the associated condition is not fulfilled, otherwise it returns zero. Usually the penalty grows with the "distance" to the condition that should be fulfilled. The penalties of soft constraints contribute to the objective function of the object.

3.2.9 Network Design Algorithms

`C_nda` is an interface for network design algorithms. There is a wide range of possibilities how to design communication networks that depend heavily on the given facilities, protocols, network layers and many more.

Therefore the class hierarchy derived from `C_nda` implements well known strategies for network design (realized by the *strategy* design pattern) without placing restrictions on new design ideas.

Following strategies are supported:

- `C_singlelayer_nda` is an interface for a single layer design algorithm.
- `C_accesstier_nda` is an interface for designing access networks within a layer.
- `C_multitier_sl_nda` is an interface for a single layer design algorithm consisting of at least one access design algorithm (`C_accesstier_nda`) and a backbone design algorithm (`C_singlelayer_nda`).
- `C_multilayer_nda` is a design algorithm for planning more than one layer at a time by (recursively) propagating the highest layer commodity from the top to the bottom layer and solving each layer by a single layer algorithm.

3.2.10 Additional Interfaces

The whole component `Performance Evaluation` (see Figure 3.1) will be implemented in the next major release of *FooNet*.

3.3 The Extended Design Classes

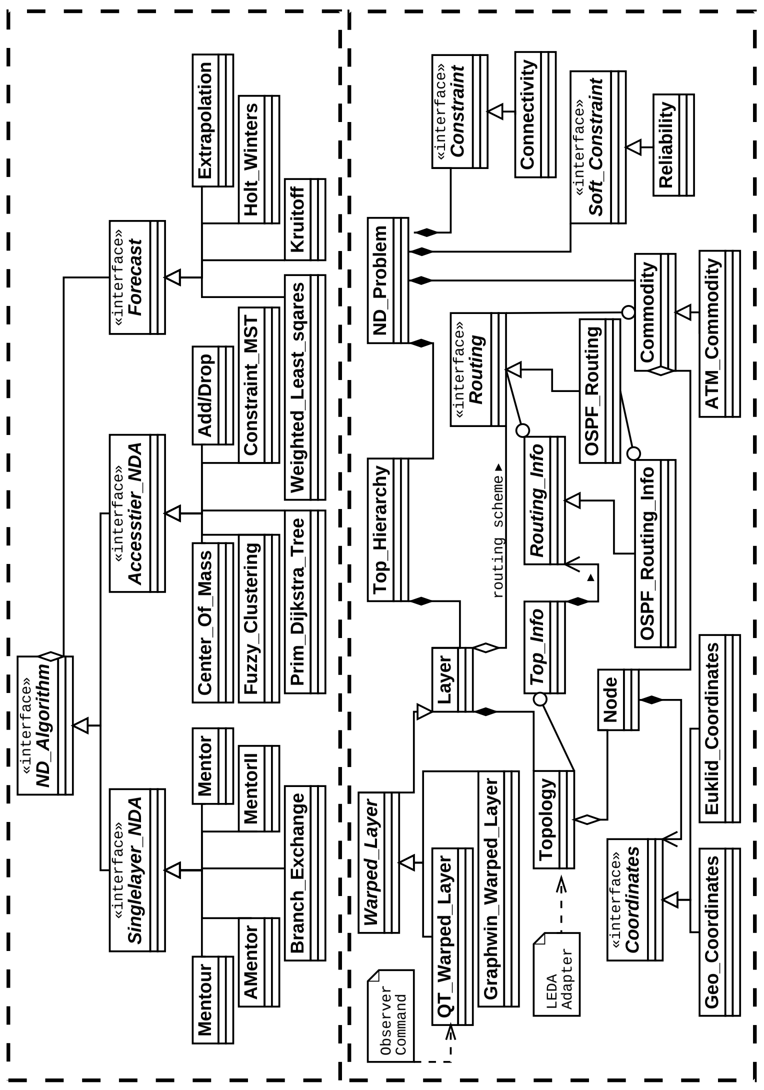

Figure 3.3: Static UML Extended Class Diagram

The second part of the *FooNet* framework extends the core design by a library of useful algorithms and applications. The next paragraphs lists the available extensions and describe the work in progress.

3.3.1 Network Design

The following network design algorithms are provided:

- Add/Drop [Ker93]: access design algorithm
  (`C_add`,`C_drop`)
- Center of Mass [Ker93]: access design algorithm
  (`C_center_of_mass`)
- Prim-Dijkstra Tree [Ker93]: access design algorithm
  (`C_prim_dijkstra`)
- Fuzzy Clustering [Lan99]: access design algorithm
  (`C_fuzzy_clustering`)
- Branch Exchange [GK77]: single layer design algorithm
  (`C_branch_exchange`)
- Concave Branch Elimination [GK77]: single layer design algorithm
  (`C_concave_branch_elimination`)
- Mentor [KKG89]: single layer design algorithm
  (`C_mentor`)
- MentorII [Cah98]: single layer design algorithm with OSPF routing
  (`C_mentorII`)
- MenTour [Cah98]: reliable single layer design algorithm
  (`C_mentour`)
- AMentor [Cah98]: reliable single layer design algorithm
  (`C_amentor`)
- IncreMentor [Cah98]: incremental single layer design algorithm
  (`C_incrementor`)

The following forecasting algorithms are provided:

- Extrapolation: A simple extrapolation based upon various model functions
  (`C_extrapolation`)
- The Kruitoff Algorithm, taken from [ITU92]
  (`C_kruitoff`)
- Weighted Least Squares [ITU92]
  (`C_least_squares`)
- Holt-Winters Method [Har89]
  (`C_holt_winters`)

The following constraints are provided:

- connectivity, returns `true` if the network is connected
  (`C_connectivity_constraint`)
- 2-node connectivity, returns `true` if the network is 2-node connected
  (`C_2node_connectivity_constraint`)
- 2-edge connectivity testing, returns `true` if the network is 2-edge connected
  (`C_2edge_connectivity_constraint`)

The following soft-constraints are provided:

- overload, calculates the amount of traffic that cannot be transported by the network (`C_overload_sconstraint`)
- reliability, calculates the probability of the network to become disconnected [Ker93] (`C_reliability_sconstraint`)

3.3.2 Routing

The following routing algorithms are provided:

- Open Shortest Path First (OSPFv2) Routing [RFC98]
  (`C_ospf_routing`, `C_ospf_routing_info`)

3.3.3 Commodity

- ATM-Commodity: Commodities are calculated from ATM parameters using the Effective Bandwidth formula from [Lin94]
  (`C_atm_commodity`)

3.3.4 Helper Applications

- LEDA-Adapter: LEDA (Library of Efficient Datatypes and Algorithms) is a well-known class library that offers a set of useful graph algorithms. *FooNet* provides a helper application which transforms a `C_topology` object into a parameterized LEDA `GRAPH` object and vice versa.
  (`leda_to_fng()`, `fng_to_leda()`)
- GraphWin-Visualizer: LEDA comes with a graphical user-interface for graphs. Using the *warper* design pattern, a simple visualization is provided by specializing the `C_warped_layer` class. An example of this can be seen in Figure 1.2.
  (`C_graphwin_warped_layer`)
- QT-Visualizer: This is a more elaborate version of a visualizer. It is designed after the model-viewer-control paradigm that bases on the *observer* and *command* design pattern. It allows to display arbitrary information in graphs.
  (`C_qt_warped_layer`)

3.4 Coding Conventions

3.4.1 Namespace

To avoid any collisions with existing names *FooNet* defines its own namespace `FN`.

3.4.2 Persistence

Each base class defines a streaming operator (`operator<<` and `operator>>`) method that calls the private virtual method `serialize`. By overloading the `serialize()` methods it is possible for a class to define its own persisting information, that have to conform the XML conventions. De-serialization is implemented using the *virtual constructor* design pattern.

3.4.3 Cloning Objects

Each base class provides a function `::clone()` that has to be overloaded by each subclass and has to create an exact copy of the object. This approach is identical with the *prototype* design pattern.

3.4.4 Heap Objects

All dynamically created objects are managed by a reference counted `shared_pointer<>`-template that guarantees to destroy the object when the last pointer to the object is destroyed.

3.5 Network Design using *FooNet*

The design principles that are realized by *FooNet* are more than a simple implementation of a set of base classes for network design problems. The following non-trivial functionality is entailed with *FooNet*:

- The design relies on a set of well known and well understood design patterns. The internal implementations profit from this fact, but also the software designer who uses *FooNet*. Examples for design patterns that may be used by the software designer are:
    - the *visitor* design pattern for topological elements
    - the *factory* design pattern for facility creation
    - the *warper* design pattern for adding functionality to network layers
    - the *strategy* design pattern for the creation of new network design algorithms
    - the *prototype* design pattern for transparently cloning objects
    - the *virtual constructor* and *builder* design pattern for persistence
- All important functionality is implemented in virtual base classes, i.e. interfaces. The interfaces provided by *FooNet* are:
    - `C_node_info` - extends a network node by additional information
    - `C_topology_info` - extends a topological element by additional information
    - `C_coordinate` - allows the use of arbitrary coordinate systems
    - `C_routing` - allows the software designer to realize arbitrary routing which is used whenever routing functionality is required
    - `C_routing_info` - extends each topological information by the necessary routing information
    - `C_facility`, `C_node_facility`, `C_edge_facility` & `C_evol_facility` - interfaces for families of facilities
    - `C_facilityclient` - interface for selecting facilities having certain attributes
    - `C_constraint` & `C_soft_constraint`- interface for arbitrary (soft) constraints
    - `C_nda` - interface for arbitrary network design algorithms (including the interfaces of the derived classes)

  The software designer can use the default functionality given by each interface, but no restrictions are imposed on him. He can always overrule them by deriving his own class that realizes the desired functionality. For example, the objective function of a network design problem and the selection & upgrade of facilities are useful candidates that can and should be overloaded by the software designer.
- The design of the persistence guarantees that each piece of information is stored only once. This is similar to the "normal forms" of relational database systems.
- The cost structure realized by *FooNet* bases on a well known and widely accepted theory of finance, the discounted costs. *FooNet* is the first tool that explicitly takes this notion into account. The support for evolving facilities `C_evol_facility` is a direct consequence.

- The extended design provides a repository of well-known algorithms used in network design. These algorithms may be helpful for solving a concrete design problem.
- Last but not least, *FooNet* is designed to offer the highest possible degree of code reuse. However, only the future will show how good this design is in practical use.

Chapter 4

Data Exchange using XML

4.1 Introduction and Motivation

The Extensible Markup Language (XML) [XML98] is a proposed recommendation from the WWW-Consortium (`http://www.w3c.org`) for a file format to support the distribution of electronic documents. XML is a subset of the SGML (Standard Generalized Markup Language) and data is processed in human readable form. The outstanding feature of XML is the fact that unlike other formats it contains also information about how to process the data, i.e. the data describes its own format. Like HTML, XML is a markup language, which relies on the concept of rule-specifying tags and the use of a tag-processing application that knows how to deal with the tags. In contrast to HTML, XML is a meta markup language which allows to define application-specific markup-tags. A software module, called XML processor, is used to read an XML document and to provide access to its content and structure.

The advantages of XML are:

- Searching information in the data is comparatively easy and efficient, by simply parsing the description-bearing tags. Even complex relationships like trees or graphs and inheritance[1] can be included.
- Extensibility is supported, while maintaining the legibility of the code by self-describing tags.
- the GUI is not embedded by the data. Thus changing the display does not influence the data.
- "Extensible Stylesheet Language Templates" allow to convert XML data in almost any format without the necessity to write additional programs.
- XML processors are available as free software (see e.g. `http://www.jclark.com/xml/xt.html`).
- As soon as the standardization of [SOX99] is finished, XML can solve the problems of exchanging object-oriented datatypes between different application, which is still an unsolved problem.

XML is supported by XEmacs, Netscape 5.0, Internet Explorer 5.0.

[1]work in progress by the W3C

4.2 Processing XML Data

The file that describes the syntax of a well-formed document (i.e. the tag-names and their hierarchical relationships) is called **Document Type Definition** (DTD). A DTD contains the meta information that is necessary to check whether a given XML document is syntactically correct. To put it in a more formal way, the DTD describes the (context-free) grammar of an XML document. Typically DTDs are stored in separate documents. A short introduction to the DTD can be found in Appendix B.1.

To access or display the relevant information included in an XML-document, additional information is necessary. The **Extensible Stylesheet Language** (XSL) [XSL99b] will define a set of formatting and processing instructions that allow the conversion of XML documents. The transformation part, **Extensible Stylesheet Language Templates**, is already standardized in [XSL99a] and contains rules for patterns that are matched against elements in the source tree of the document and templates that construct a portion of the resulting output. During this transformation the data can be modified (e.g. reordered) and processed (e.g. accumulated).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE node SYSTEM "Node_DB.dtd" [
]>

<FN_NODE_DB number="323">
 <FN_NODE id="b-win-gw.rrz.uni-koeln.de" type="E" childs="0" tier="0">
  <FN_COORD type="geo" x="50.9272" y="6.9213"></FN_COORD>
  <FN_NODE_INFO></FN_NODE_INFO>
  <FN_NODE_CHILDREN></FN_NODE_CHILDREN>
 </FN_NODE>
 <FN_NODE id="bam-berlin" type="E" childs="0" tier="0">
  <FN_COORD type="geo" x="52.4479" y="13.2994"></FN_COORD>
  <FN_NODE_INFO></FN_NODE_INFO>
  <FN_NODE_CHILDREN></FN_NODE_CHILDREN>
 </FN_NODE>
 <FN_NODE id="bast.koeln1" type="E" childs="0" tier="0">
  <FN_COORD type="geo" x="50.9522" y="7.17136"></FN_COORD>
  <FN_NODE_INFO></FN_NODE_INFO>
  <FN_NODE_CHILDREN></FN_NODE_CHILDREN>
 </FN_NODE>
 ...
</FN_NODE_DB>
```

Figure 4.1: Fraction of a Node Database

The following (very simple) example demonstrates how this works. In Figure 4.1, a fragment of an XML-document (compliant to the DTD in Appendix B.2.1) that contains a set of nodes is shown. If a user wants to access all node-ids together with their coordinates, he can use the XLST script shown in Figure 4.2 that provides this functionality. Figure 4.3 presents the result of the transformation.

As a conclusion, XML seems to be a very promising approach for data exchange between different tools and applications in the context of network design.

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/">
     <HTML><HEAD>
     <TITLE> FOONET Processing Output </TITLE>
     </HEAD><BODY>
     <UL>
     <xsl:apply-templates />
     </UL>
     </BODY></HTML>
 </xsl:template>

 <xsl:template match="FN_NODE">
     <LI>
     <xsl:value-of select="@id" />
     <xsl:text>: </xsl:text>
     <xsl:apply-templates />
     </LI>
 </xsl:template>

 <xsl:template match="FN_COORD">
     <B>
     <xsl:value-of select="@x"/>
     <xsl:text> </xsl:text>
     <xsl:value-of select="@y"/>
     </B>
 </xsl:template>

</xsl:stylesheet>
```

Figure 4.2: XSL-Transformation

```
<HTML><HEAD>
 <TITLE> FOONET Processing Output </TITLE>
</HEAD><BODY>

<UL>
 <LI>b-win-gw.rrz.uni-koeln.de: <B>50.9272 6.9213</B></LI>
 <LI>bam-berlin:  <B>52.4479 13.2994</B></LI>
 <LI>bast.koeln1:  <B>50.9522 7.17136</B></LI>
 ...
</UL>

</BODY></HTML>
```

Figure 4.3: Transformation Result

4.3 XML in *FooNet*

FooNet produces and processes XML compliant data, i.e. all streaming operators (see Chapter 3.4.2) work with XML documents. The Document Type Descriptions of the in- and output data are listed in Appendix B.2.

The following XSLTs that provide mainly filtering functionality are already used or at least in a development stage:

- HTML-Conversion: For all *FooNet* DTDs there are XSLTs that display the data contained in the document in HTML format.
- GNU-Plot-Conversion: This XSLT transforms the output of a `C_Layer` to a GNU-Plot compatible input file.
- GML-Conversion: GML is a wide-spread data format for (parameterized) graphs [Him96]. This XSLT transforms a `C_Layer` output into a GML-document.
- VRML-Conversion: Similar to the previous two points this XSLT transforms a `C_layer` output into a VRML document that can be viewed by any WEB-browser supporting VRML[2].

The development of traffic measurement tools that produce XML-compliant data is a further step in this direction. Such measurements could for example easily be converted into a `C_commodity` compliant output.

[2]Virtual Reality Modelling Language, a standard from the W3C

Chapter 5

Summary and Outlook

5.1 Summary

In this report, *FooNet*, an object-oriented application framework for telecommunication network design, is presented. The characteristics of *FooNet* in contrast to other planning environments are its consequent object-oriented design and the support of reuse techniques on different levels of abstraction. *FooNet* comes with a set of algorithms used in network planning and this library is expected to grow in the future. Additionally, the capabilities of XML as a data exchange format between various applications are discussed.

5.2 Outlook

The author is aware of the fact that the present version of *FooNet* is not nearly covering all facets of network design problems. It lacks a lot of features that are useful or even necessary for some problems.

The following work is intended to be done in the near future:

- Include support for usage dependent costs.
- Finish the interfaces and provide default implementations for the network performance analysis part.
- Include extensions for planning mobile networks.
- Increase the number of network design strategies derived from `C_nda`.
- Increase the number of algorithms and utilities in the extended framework.
- The XML part of *FooNet* bases exclusively on proposed standards of the W3C. However the DTDs of XML lack a support of object-oriented datatypes. The W3C is currently working on a “Schema for object-oriented XML” [SOX99] that will solve this problem. A soon as this specification is available as a proposed standard, it will be supported by *FooNet*.

List of Figures

Bibliography

[Auc99] Ben Auch. Design und prototypische Implementierung einer integrierten Plattform zur Planung von hierarchischen Netzen. Master's thesis, Technische Universität München, Institut für Informatik, 1999.

[Böc97] Stefan Böcking. *Objektorientierte Netzwerkprotokolle - Grundlagen, Entwurf und Implementierung.* Addison Wesley Longman Publishing Company, 1997.

[Boo91] Grady Booch. *Object oriented Design with Applications.* The Benjamin/Cummings Publishing Company, 1991.

[Cah98] Robert S. Cahn. *Wide Area Network Design - Concepts and Tools for Optimization.* Morgan Kaufmann Publishers Inc., 1998. The Morgan Kaufmann Series in Networking.

[Cop92] James O. Coplien. *Advances C++ Styles and Idioms.* Addison Wesley Publishing Company, 1992.

[Fis00] Volker Gerd Fischer. *Evolutionary Design of Corporate Networks under Uncertainty.* PhD thesis, Technische Universität München, Institut für Informatik, 2000. work in progress.

[Fri98] Jochen Frings. Eine kurze Einführung in die Netzplanung. Technical report, Technische Universität München, 1998. Lehrstuhl für Kommunikationsnetze.

[FSJ99] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building Application Frameworks : Object-Oriented Foundations of Framework Design.* Horizon Publishers & Distributors Inc., 1999.

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

[GK77] Mario Gerla and Leonard Kleinrock. On the topological Design of Distributed Computer Networks. *IEEE Transactions on Communications*, COM-25(1):48–60, January 1977.

[Har89] Andrew C. Harvey. *Forecasting, structural time series models and the Kalman filter.* Cambridge University Press, 1989.

[Him96] Michael Himsolt. GML: A portable Graph File Format. Technical report, Universität Passau, 1996.

[Int97] International Standards Organization. *ISO/IEC Final Draft International Standard 14882 - Programming Language C++*, 11 1997.

[ITU92] International Telecommunication Union. *Forcasting International Traffic*, 1992. E.506 (rev.1).

[Jac99] Ivar Jacobsen. Applying UML in The Unified Process. Technical report, Rational Software Inc., 1999. `www.rational.com`.

[JGJ97] Ivar Jacobsen, Martin Griss, and Patrik Jonsson. *Software Reuse : Architecture Process and Organization for Business*. ACM Press, 1997.

[Joh97] Ralph E. Johnson. Frameworks = Components + Patterns - How frameworks compare to other object-oriented reuse techniques. *Communications of the ACM*, 40(10):39–42, October 1997.

[Ker93] Aaron Kershenbaum. *Telecommunications Network Design Algorithms*. McGraw-Hill, 1993.

[KKG89] A. Kershenbaum, P. Kermani, and G. Grover. Mentor: An Algorithm for Mesh Network Topological Optimization and Routing. Technical Report RC 14764 7/14/89, IBM Research Division, T.J. Watson Research Center, 1989.

[Lan99] Daniel Lang. Plazierung von Backboneknoten mit Fuzzy-Clustering. Master's thesis, Technische Universität München, 1999. Institut für Informatik.

[Lin94] Karl Lindberger. Dimensioning and Design Methods for Integrated ATM Networks. *International Teletraffic Conference*, 14:897–906, 1994.

[MS81] Andranik Mirzaian and Kenneth Steiglitz. A Note on the complexity of the Sta-Star Concentrator Problem. *IEEE Transactions on Communications*, COM-29(10):1549–1552, October 1981.

[NS99] Jörg Noack and Bruno Schienmann. Objektorientierte Vorgehensmodelle im Vergleich. *Informatik-Spektrum*, (22):166–180, 1999.

[Oes97] Bernd Oesterreich. *Objekt-Orientierte Softwareentwicklung mit der Unified Modelling Language*. Oldenbourg Verlag, 2. edition, 1997.

[OMG97] OMG - Object Management Group. *UML Notation Guide - Version 1.1*, September 1997.

[RFC98] OSPF Version 2. Technical report, Internet Engineering Task Force - Network Working Group, April 1998. RFC 2328.

[SOX99] W3C Discussion Paper. *Schema for Object-Oriented XML 2.0*, 7 1999.

[Sto97] Bjarne Stoustrup. *The C++ Prorgamming Language*. Addison-Wesley Publishing, 3rd. edition, 1997.

[XML98] W3C Recommendation. *Extensible Markup Language (XML) 1.0*, 10 1998. REC-xml-19980210.

[XSL99a] W3C Recommendation. *XSL Transformations (XSLT) Version 1.0*, 8 1999. REC-xslt-19991116.

[XSL99b] W3C Working Draft. *Extensible Stylesheet Language (XSL) Specifiaction*, 4 1999.

Appendix A

The Unified Modelling Language - UML

A.1 Overview

UML, the Unified Modelling Language, is a diagram-oriented language for analyzing and designing object-oriented systems. UML notation comprises several types of diagrams:

- Use Case Diagrams
- Class Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- State-chart Diagrams
- Activity Diagrams
- Implementation Diagrams

This report uses two types of diagrams - static class diagrams and implementation diagrams - and only these are described in the following section to the necessary level of detail. The interested reader may refer to [Oes97] for a detailed introduction.

A.2 Static Class Diagram

Static class diagrams show the static structure and relations of the abstractions (i.e. classes) of the software design. Class diagrams can be used to show the attributes and operations of a class and the constraints for the way objects collaborate. The UML notation of a static class diagram consists of a set of nodes and edges. The nodes have the form of rectangles and the size and relative position does not matter. An overview of the used symbols is shown in Figure A.1.

Classes are symbolized by rectangles that have three compartments with the following properties: the first compartment contains the name and stereotype of the class, the second its attributes, and the third the operations. For convenience, the second and third compartment can be hidden in a diagram. If a class is **abstract**, its name is displayed in *emphasized* letters. UML supports also **parameterized classes** (i.e. template classes), whose parameter is specified in a dashed rectangle on the upper right.

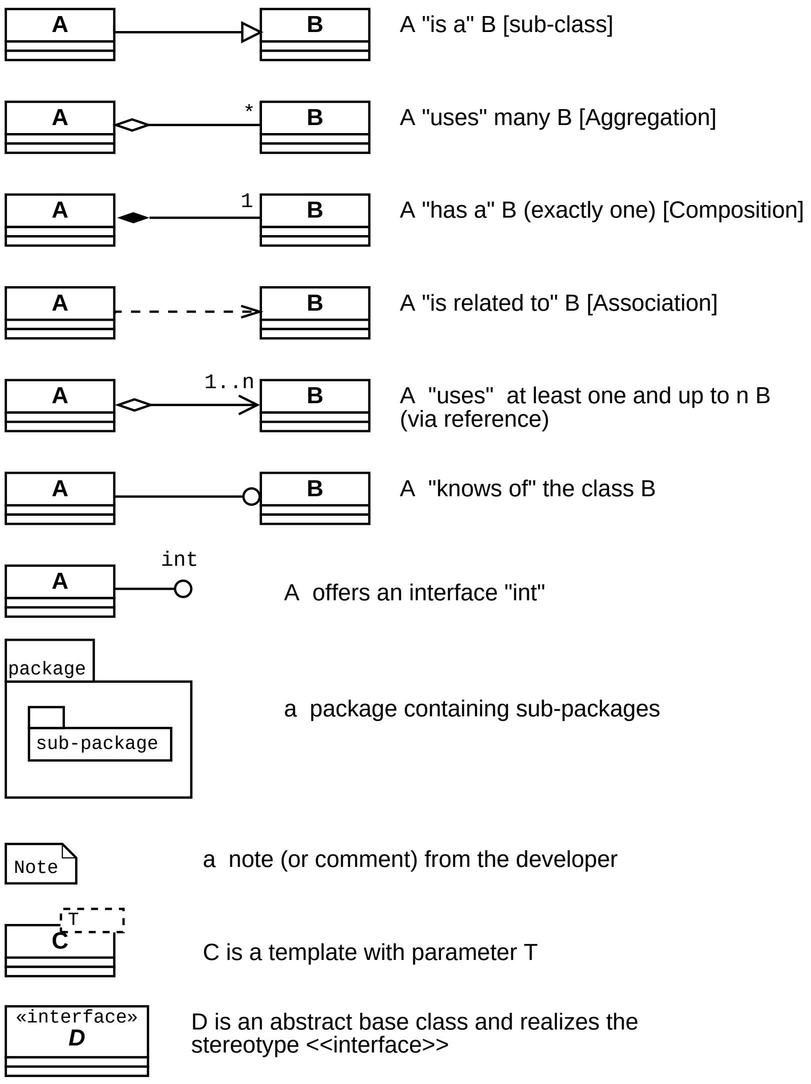
A
B
A "is a" B [sub-class]
A
*
B
A "uses" many B [Aggregation]
A
1
B
A "has a" B (exactly one) [Composition]
A
B
A "is related to" B [Association]
A
1..n
B
A "uses" at least one and up to n B (via reference)
A
B
A "knows of" the class B
int
A
A offers an interface "int"
package
sub-package
a package containing sub-packages
Note
a note (or comment) from the developer
T
C
C is a template with parameter T
«interface»
D
D is an abstract base class and realizes the stereotype <<interface>>

Figure A.1: UML Notation Guide

Stereotypes, denoted by matched double brackets (also called guillements) ≪≫, are used to extend a construct at modeling time. Generally stereotypes represent usage distinctions. Examples are the stereotype ≪interface≫, which denotes a pure abstract base class, or the stereotype ≪bind≫, which instantiates a template with a parameter.

Notes and comments are supplied by a rectangle with its upper right corner folded down. Although newer versions of UML support *design patterns* by additional annotations in the class, here they are included as comments.

The associations between classes are shown by various types of lines between the classes. An association can have a cardinality which is expressed by the number (or range) at the end of the line. If the association has a name it is written on the top of the connecting line.

The following types of associations are used:

- **Composition** - means that an object of class `A` "owns" ("has-a" relationship) an object of class `B`, i.e. class `A` is responsible for its associated objects of class `B`, and if an object of class `A` is destroyed, all owned objects of class `B` are also destroyed. The graphical symbol is a line with a filled diamond on the side of class `A`. To express that the association is of a referenced type, the side of the associated class `B` has an arrow.
- **Aggregation** - is a weaker form of composition ("uses-a" relationship). It means that an object of class `A` has a (temporary) acquisition of an object of class `B` without ownership and responsibility for its lifetime. To express that the aggregation is of a referenced type, the side of the aggregated class `B` has an arrow.
- **Association** - If the modeller wants to express an association that is not specified in detail, he can use a dashed line between the associated diagram elements.
- **(Public) Inheritance** - ("is-a" relationship) is indicated with a triangle pointing up to the class from which the other is derived.
- **Interface** - ("knows of a" relationship) expresses that the objects of a class `A` know the interface of class `B`. Technically this implies that the class `A` cannot be compiled without importing the class `B`. The graphical notation is a line with a circle at the end pointing to class `B`.

  Sometimes more than one class support a certain kind of interface. This can be indicated with a (dangling) line that has a small circle at the end of the line.

A.3 Implementation Diagram

A component diagram (which belongs to the class of implementation diagrams) is used to break down a larger software system into logical grouping of smaller systems. It can also show the dependencies of classes and their dependencies within a component. It serves as an orientation where to find which functionality.

Packages are used to group a set of classes having a common purpose. They are displayed by large rectangles with the name of the purpose (i.e. the name of the package) in a small rectangle on top of the upper-left corner. Classes belonging to the package are visualized by smaller rectangles grouped within. Packages can be nested.

Appendix B

Document Type Description

B.1 Introduction to the Document Type Documentation

The syntax of a (well-formed) XML document is structured by tags that can be projected into a tree structure. Each element in this tree consists of a start tag, a body and an end tag as well as a set of attributes associated with that element. Syntactically, a tag is anything between "<" and ">". Tags are case sensitive. End tags are marked by a leading "/". The following construction is an example for a valid tag:

```
<string length="17"> This is a string </string>
```

Sometimes it makes sense to have an empty tag simply by putting the slash at the end of the tag "`<EMPTY_TAG/>`". Empty elements usually have a number of attributes to give them usefulness. The actual names of tag-elements are arbitrary, i.e. can be chosen by the document designer (usually guided by their meaning and therefore often denoted as "semantic tags"). All documents begin with a "root of document" entity, all other entities are optional.

The Document Type Definition (DTD) defines the syntax of the XML-document. It contains meta information about valid elements, valid attribute names and values, and information how elements can nest in each other. One can think of a DTD as defining the overall structure and syntax (i.e. the grammar) of the document. Typically DTDs are stored in separate documents.

Here, the syntax of a DTD is demonstrated with the help of the following example:

```
<!DOCTYPE FOONET [

 <!-- ENTITIES HERE -->

 <!ENTITY \% LRK \'Lehrstuhl f&uuml;r Rechnerkommunikation\'>

 <!-- ELEMENTS HERE -->

9   <!ELEMENT FN_NODE_DB (FN_NODE+, #PCDATA)>
10  <!ATTLIST FN_NODE_DB
11            date      CDATA   #IMPLIED
12            creator   CDATA   #REQUIRED
13            id        ID      #REQUIRED >
14
15  <!ELEMENT FN_LAYER "SDH" | "ETHERNET" | "ATM" | "IP" >
16
17  <!ELEMENT FN_NODE EMPTY>
18 ]>
```

This document makes use of the following DTD features:

- **Root Tag**
  The line 1 of the DTD defines the root element of the DTD, i.e. in this example all documents conforming to this DTD must be encompassed by "`<FOONET>`" and "`</FOONET>`".
- **Comments**
  Comments can be placed using the following syntax "`<!-- COMMENT -->`".
- **Entities**
  Entities are aliases for more complex functions. For example, the entity "`&LRK;`" defined in line 5 represents the term "`Lehrstuhl f&uuml;r Rechnerkommunikation`" in the document. Entities can reduce the file size and they prevent error-prone repeating.
- **Elements**
  An element defines a tag and the syntactically correct usage of that tag. For example, line 9 defines the tag "`<FN_NODE_DB>`" and demands that this element must contain at least one element (denoted by "`+`") of the type "`<FN_NODE>`" followed by arbitrary character data ("`#PCDATA`") in its body. The rules for building the body of the element are similar to regular expressions. Line 15 defines that the body of the tag "`<FN_LAYER>`" may contain one of the terms "`SDH`", "`ETHERNET`", "`ATM`" or "`IP`". Line 17 defines an element with an empty body, i.e. "`<FN_NODE/>`".
- **Attributes**
  Attributes allow to associate an element with additional parameters. For example, the rule beginning at line 10 allows the element "`<FN_NODE_DB>`" to have an attribute "`date`" and requires the attributes "`creator`" and "`id`". A valid realization could be for example "`<FN_NODE_DB creator="VOLKER" id="SDH">`".

  The following types of attributes are defined:

  - `CDATA` - any value
  - `ID` - unique identifier within the XML document
  - `IDREF` - reference to an element with a specific ID
  - `IDREFS` - sequence of IDREFs
  - `XPOINTER` - a relative path through the XML tree (e.g. a child or parent)

The DTD can be included in the XML document by inserting it after the processing instructions:

```
<?xml version = "1.0" encoding=’’UFT-8’’ standalone="yes"?>

<!DOCTYPE ROOT [
 <!-- HERE COMES THE DTD -->
]>

<ROOT>
<!-- HERE COMES THE BODY -->
</ROOT>
```

or it could be included by referencing a file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FOONET SYSTEM "FOONET.dtd" [
]>
<FOONET>
<!-- HERE COMES THE BODY -->
</FOONET>
```

B.2 *FooNet* DTDs

The following paragraphs present the document type descriptions of the in- and output of *FooNet*. Please note that this becomes obsolete as soon as the object-oriented schemata are standardized.

B.2.1 Node Database

```
<!--Copyright 1999 V. Fischer -->
<!-- Network Node DTD -->

<!ELEMENT FN_NODE_DB (FN_NODE)*>
<!ATTLIST FN_NODE_DB creator CDATA  #IMPLIED
                     date    CDATA  #IMPLIED
                     number  CDATA  #REQUIRED >

<!ELEMENT FN_NODE (FN_COORD , FN_NODE_INFO , FN_NODE_CHILDREN )>
<!ATTLIST FN_NODE id         ID     #REQUIRED
                  type       CDATA  #REQUIRED
                  childs     CDATA  #REQUIRED
                  tier       CDATA  #REQUIRED >


<!ELEMENT FN_COORD (#PCDATA )>
<!ATTLIST FN_COORD type      CDATA  #REQUIRED
                   x         CDATA  #REQUIRED
                   y         CDATA  #REQUIRED>
```

```
<!ELEMENT FN_NODE_INFO (#PCDATA)>

<!ELEMENT FN_NODE_CHILDREN (FN_NODE_ID)*>

<!ELEMENT FN_NODE_ID EMPTY>
<!ATTLIST FN_NODE_ID id         ID     #REQUIRED >
```

B.2.2 Facilities

```
<!--Copyright 1999 V. Fischer -->
<!-- Network Facility DTD -->

<!ELEMENT FN_FAC (FN_COST | ( FN_PERIOD , FN_FAC* , #PCDATA?) )>
<!ATTLIST FN_FAC  type      ("N"|"E"|"EVOL") #REQUIRED
                  layer     CDATA  #REQUIRED
                  cap       CDATA  #REQUIRED
                  vendor    CDATA  #REQUIRED
                  id        CDATA  #REQUIRED >


<!ELEMENT FN_COST (FN_PERIOD , #PCDATA?)>
<!ATTLIST FN_COST   setup   CDATA     #REQUIRED
                    term    CDATA     #REQUIRED
                    reoc    CDATA     #REQUIRED >

<!ELEMENT FN_PERIOD EMPTY>
<!ATTLIST FN_PERIOD d       CDATA     #REQUIRED
                    h       CDATA     #REQUIRED
                    m       CDATA     #REQUIRED
                    s       CDATA     #REQUIRED >
```

B.2.3 Layer

```
<!--Copyright 1999 V. Fischer -->
<!-- Network Layer -->



<!ELEMENT FN_LAYER ( FN_F_CLIENT, FN_ROUTING, FN_TOPOLOGY )>
<!ATTLIST FN_LAYER id         ID     #REQUIRED>

<!ELEMENT FN_CLIENT (#PCDATA )>
<!ATTLIST FN_CLIENT name    CDATA  #REQUIRED>

<!ELEMENT FN_ROUTING (#PCDATA )>
<!ATTLIST FN_ROUTING name    CDATA  #REQUIRED>
```

```
<!ELEMENT FN_TOPOLOGY ( FN_TNODE+ , FN_TEDGE* ) >
<!ATTLIST FN_TOPOLOGY nodes  CDATA     #REQUIRED
                      edges  CDATA     #REQUIRED >

<!ELEMENT FN_TNODE ( FN_TOP_INFO , #PCDATA ) >
<!ATTLIST FN_TNODE  id  IDREF     #REQUIRED >

<!ELEMENT FN_TEDGE ( FN_TOP_INFO , #PCDATA ) >
<!ATTLIST FN_TEDGE  source  IDREF     #REQUIRED
                    dest    IDREF     #REQUIRED >

<!ELEMENT FN_TOP_INFO ( ( FN_FAC | FN_NULL ), ( FN_ROUTING_INFO | FN_NULL) , #PCDATA ) >
<!ATTLIST FN_TOP_INFO  load  CDATA  '0' >

<!-- NULL-Pointer -->

<!ELEMENT FN_NULL EMPTY >
```

B.2.4 Commodity

```
<!--Copyright 1999 V. Fischer -->
<!-- Network Commodity DTD -->

<!ELEMENT FN_COMMODITY (FN_NODE_ID* , FN_TM* ) >
<!ATTLIST FN_COMMODITY nr_nodes=CDATA #REQUIRED
                       author  =CDATA #IMPLIED >

<!ELEMENT FN_NODE_ID EMPTY>
<!ATTLIST FN_NODE_ID id      IDREF  #REQUIRED >

<!ELEMENT FN_TM (#PCDATA , FN_T)*>
<!ATTLIST FN_TM      id      IDREF  #REQUIRED >

<!ELEMENT FN_T EMPTY>
```

Appendix C

Software Requirements

FooNet is developed under LINUX 2.2.7 using the GNU C++ Compiler Suite gcc-2.95.2 (`http://www.cygnus.org`). The core design should work with any standard compliant C++ compiler that supports namespaces, exceptions and templates.

Software requirements for the core framework are:

- STL [Int97]
- GTL (`http://www.fmi.uni-passau.de/Graphlet/GTL`)
- SAX (`http://www.jezuk.demon.co.uk/SAX/`)

Software requirements for the extention framework are:

- LEDA (`http://www.mpi-sb.mpg.de/LEDA/`)
- Qt 2.0 (`http://www.trolltech.no`)

Index