Reuse applied to environmental software: Design of an adaptable system for water quality assessment and monitoring

Thesis

For obtaining the academic degree of Doctor by the university of Vigo, with internacional mention
Department of Informatics

Jürgen Heine
January 2013

Supervisors:
Dr. Emilio García Roselló
Dr. Isabel Pardo Gamundi
Dr. Jacinto Gonzalez Dacosta
D. Emilio García Roselló, Dª Isabel Pardo Gamundi y D. Jacinto González Dacosta en calidad de directores de la Tesis Doctoral titulada "Reuse Applied to environmental software: design of an adaptable system for water quality assessment and monitoring" realizada por D. Jurgen Heine, autorizan y dan el visto bueno para su depósito y defensa en la Universidad de Vigo.

Vigo, 22 de Enero de 2012

Emilio G. Roselló  Isabel P. Gamundi  Jacinto G. Dacosta

Jurgen Heine
Contents

List of Figures 9

1 Introduction 11

I State of art 14

2 Software reuse and SOA 15

3 Building SOA 23

3.1 Technology agnostic design ........................................ 24

3.1.1 Platform Independence ........................................ 24

3.1.1.1 Context Independence ..................................... 25

3.1.1.2 Application Independence (Libraries) ..................... 26

3.1.1.3 Programming Language Independence .................... 27

3.1.1.4 Operating System Independence ........................... 28

3.1.1.5 Location Independence ..................................... 28

3.1.1.6 Platform Independence in SOA ............................ 30

3.1.2 Data Independence .............................................. 30

3.1.2.1 The knowledge pyramid .................................. 31

3.1.2.2 Database models ............................................ 32

3.1.2.2.1 Hierarchical data bases ................................ 32

3.1.2.2.2 Network model Databases .............................. 33
3.2.3.1 WS-BPEL ........................................... 80
3.2.3.2 Graphical composition with Workflow Management Systems 81
3.2.3.3 Summary ............................................. 82
3.2.3.4 Separation of components and composition in SOA ........ 82

4 Scientific Workflow Management Systems 84

4.1 Business Workflows vs. Scientific Workflows ......................... 84
4.2 Existing scientific workflow systems .................................... 88
  4.2.1 Kepler ................................................. 88
  4.2.2 Taverna / MyGrid ...................................... 89
  4.2.3 Triana .................................................. 90
  4.2.4 Condor / DAGMan ..................................... 91
  4.2.5 Pegasus .................................................. 92
  4.2.6 Java CoG Kit / Karajan .................................. 93
4.3 Strength and weak points of existing scientific workflow systems: A look into the future ............................................. 94

II Water quality assessment models 99

5 Environmental modeling: Water quality assessment models & SOA 100
  5.1 Environmental modeling for freshwater quality management ........ 100
  5.2 Empirical/statistical models for water quality assessment ........... 103
    5.2.1 Single Metrics and Bimetrics ................................ 104
      5.2.1.1 Saprobic System .................................... 104
      5.2.1.2 Diversity approach .................................. 105
      5.2.1.3 Biotic approach .................................... 106
    5.2.2 Multimetrics ............................................. 107
    5.2.3 Multivariate Analysis .................................... 108
      5.2.3.1 RIVPACS ........................................... 109
      5.2.3.2 Reference Condition Approach ........................ 109
  5.3 The EU - water quality framework directive ........................... 110
V Norti-Online

9 Norti-Online web portal

9.1 The Norti legacy software

9.2 Legacy software reformation steps

9.3 NORTI-Online web portal system design
List of Figures

3.1 Platform independency scale ........................................... 25
3.2 The knowledge pyramid .................................................. 31
3.3 The DAO pattern participants ......................................... 47
3.4 The DAO pattern call sequence ........................................ 47
3.5 Service data object ....................................................... 49
3.6 The Oracle SOA stack .................................................... 60
3.7 The IBM SOA stack ....................................................... 60
3.8 The SAP SOA stack ....................................................... 61
3.9 Basic SOA stack ........................................................... 62
3.10 WebService wiring languages classification ......................... 79

8.1 INodes ........................................................................ 135
8.2 Fork node design primitive ............................................. 136
8.3 Connectors .................................................................... 137
8.4 Control flow lines ........................................................... 138
8.5 WProc example ............................................................. 138
8.6 Viewer and Result treatment selection .................................................. 139
8.7 Mapping assistant ................................................................................. 140
8.8 Loops in NORTIFlow ............................................................................ 141
8.9 WProc construction 1 ........................................................................... 142
8.10 WProc construction 2 .......................................................................... 143
8.11 Run dialog ............................................................................................ 145
8.12 Workflow execution 1 ........................................................................... 146
8.13 Workflow execution 2 ........................................................................... 146
8.14 Workflow execution 3 ........................................................................... 147
8.15 Workflow execution 4 ........................................................................... 147
8.16 Workflow execution 5 ........................................................................... 148
9.1 Area covered by the NORTI project ..................................................... 152
9.2 NORTI data base ................................................................................... 155
9.3 The Norti-Online web portal architecture ............................................ 158
9.4 basic water evaluation process for the invertebrates and diatom model ... 163
9.5 Main component implementation 1 ....................................................... 164
9.6 Main component alternative implementation ........................................ 164
9.7 Evaluation of various stations with calculation of EQR ......................... 165
9.8 Multi metric process .............................................................................. 166
9.9 Multi metric WProc .............................................................................. 166
9.10 Norti-Online accessed via ASP.Net components ................................... 167
1 Introduction

The monitoring of water quality has become increasingly important in recent years, especially propelled by the enactment of the Framework Directive 2000/60/EC [Dir00] on water policy. The ultimate goal of this directive is the protection of European waters to prevent deterioration of their quality and enhance aquatic and terrestrial ecosystems that depend on them. This has motivated the development of monitoring systems based on ICT to control continental water quality and its ecological state. Within a project of R & D promoted by several Spanish hydrographic confederations and conducted by researchers at the University of Vigo, a software prototype called Norti has been designed for this purpose. It allows for evaluating water quality using several statistical models. The present thesis has been developed as a research work focused on the further development of this software, or rather a complete new version redesigned from ground up and based on a service oriented architecture (SOA) [Erl06] approach. This design decision was made to better comply with several requirements like the flexible integration of new water quality assessment models, offering possibilities to adapt them and allow them an evolution, new functionalities like reporting and integration with other software, as well as specific requirements for data independence. SOA fosters this by the usage of XML based platform independent services that can be flexibly composed.

Due to the innovative SOA approach regarding its usage in environmental software, the experiences gained in this project might be valuable for other types of environmental software, which also could benefit from SOA approaches. The advantages promised by SOA based solutions are numerous, although it is hold for difficult to implement and have the reputation of having a very late pay-back, as it is necessary to invest much effort and costs at the beginning to overcoming many arising difficulties. The investigation of these diffi-
culties is one of the topics of this thesis and especially those which arise when domains like environmental modeling use SOA, or in general when SOA is applied in a scientific context. For domains like environmental modeling, this work furthermore proposes the integration of scientific workflow systems into the architecture for modeling and executing scientific workflows via services. The outcomes of this work include a theoretical background on SOA in scientific projects, the design and implementation of a set of services and workflows, which are integrated into the Norti-Online web portal that can be used for environmental modeling software, especially for empirical water quality assessment models, insights and experiences for the modeling of scientific workflows based on web services, and finally the design and development of a new scientific workflow management system with special SOA capabilities and numerous possibilities for user interaction at the execution of workflows.

The remainder of this work is organized as following: The first part introduces the topics of SOA, workflow systems and the connection of both. Basic SOA principles are worked out and put into contrast to traditional programming paradigms together with the possibility of its implementation with existing technologies as well as pointing out remaining limitations. Workflow systems are introduced and a couple of significant existing scientifically workflow systems are reviewed and analyzed regarding its limitations for their usage in projects like the one here described.

The second part gives a short introduction into environmental modeling in general, water quality assessment models in particular and its implementation possibilities and benefits with the SOA paradigm. Water quality assessment models based on metrics, multi-metrics and multivariate analysis techniques are briefly reviewed. Special focus is laid on the coupling of models.

The third part describes the methodology used in this thesis and states research aims.

In the fourth part the design of NORTIFlow, a scientific workflow system with outstanding SOA and user interaction capabilities is described.

The fifth part describes the development of the Norti-Online web portal, that is capable to execute workflows created using the workflow system described in the fourth part. A
collection of services and workflows will be developed, implementing various water quality assessment models.

Finally conclusions of this research work are drawn.
Part I

State of art
The world of software engineering is subject to a rapid development in the ongoing process of an always closer connected world and technological progress. For an easier development and for a better realization of design qualities like flexibility, extensibility, stability, reliability, portability or adaptability, new software development methods are demanded. Very promising here is the migration to service oriented architecture (SOA) [Erl06], which stands for a big switch in software development paradigms and means taking into account and realizing SOA principles at the design of software solutions. To better understand how SOA can contribute to enhance the world of software engineering, here some scenarios and common challenges which frequently arise at the development of software are introduced. First of all we can observe an increased need to connect different software and systems. Some examples:

Different organizations that are doing business together need compatibility of their software systems to share and interconnect data and programs more easily. They even virtually may melt to new emerging "virtual organizations" that span multiple administrative domains or institutions, where collaborating members may last many years or have a fleeting lifetime and management can evolve from a centralized to a decentralized approach. Service orientation can help to realize this.

Cloud computing [Sos11] as a present mayor buzzword is gaining importance and changes the usage of the web. While until now in the main only data have been publically available, now also services (SaaS), hardware (IaaS) or programming environments (PaaS) are offered to be used or rented in a pay-per-use way. This can liberate an individual from the need to maintain multiple desktop software packages for the purpose of a few occasional operations,
by just renting services when needed. It also liberates from the need of having at hand powerful hardware for occasional calculation intensive tasks. Data, programs and hardware are becoming available always and from everywhere. Also here, as all these are offered as services, they perfectly fit into service oriented paradigm. In general, as more and more data is made available to the public in portals to download, in online databases or in clouds, new mechanisms are required to standardize them and to allow them to be discovered in order to use this data efficiently. Those new mechanisms can be implemented by means of SOA services.

Scientists typically use many different programs for different tasks while doing the linkage between them manually by saving files with one program and importing them with another, maybe after doing some refinements, reformatting and other conversions. For the future it might be desirable to automate those steps to make this time-consuming and error-prone process more transparent and fast. Scientists increasingly might want to have the possibility of defining reusable workflows involving many different programs, conversion steps, etc. Such workflows play an important role in the SOA paradigm [TPD+][LAG03].

eScientists want to do in-silicio experiments, carry out expensive calculations on dynamic grids, taking advantage of all resources available distributed in the web. These experiments may also include a large number of participants all around the world and may include many subtasks and the gathering of data from heterogeneous distributed data sources. Traditionally this is an issue concerning grid computing which currently however are not very easy to build, what could largely benefit from novel SOA approaches [CKRJ05].

All these scenarios have in common that different independently built systems that are not designed to work together should be connected so that interoperability and compatibility problems arise and need to be solved. Finally, it leads to the vision of a component world, where everything can be connected with everything - devices, programs, data bases, whatever available - can be used as reusable artifacts, working together to solve some tasks, using different information sources; and then maybe be disassembled again to be used in another composition, overcoming problems like different operating systems, programming languages with different type systems, incompatible file formats or competing standards.
In fact all those kind of situations can largely be seen as a reuse problem of using existing software artifacts to build new solutions, especially while remote usage often is not foreseen. It also is closely related to Component Based Software Engineering (CBSE) [HC01], the component paradigm of software engineering which deals with how constructing software pieces compatible to each other so they could be assembled like building blocks. Reuse existing solutions always has been an important topic in software engineering. The advantages are obvious: It is cheaper, faster, and less risky to reuse an existing solution than designing and implementing an own one. Many things can become the subject of reuse - code artifacts, architectures, design patterns, processes, specifications - with always the same advantages of less development effort and time, tested and sometimes optimized solutions, standardization, and maybe most important, lower costs. Reuse is needed to share successful solutions and to avoid solving the same problem more than once. A good example is the reuse of components off-the-shell (COTS) [GR10], that are often large proprietary systems around some special task, for example computer algebra systems (CAS) which allow doing mathematical calculations often faster and with higher precision than implementing it by oneself by the means of ordinary programming languages and its type system. So why not use their functionality, if available? Also data base management systems, software capable to generate charts and graphics, physical calculation engines and many more belong to this category and are waiting to be reused.

The basic idea of CBSE on the other side is to create software that is reusable right from the beginning. The goal is to finally establish a component repository so that one day new software in the main can be assembled out of existing components while only new parts of the system have to be implemented. For this, components have to agree to common standards regarding wiring, typing, packaging, composition, execution and others, which generally are defined by a component model. Currently many different competing component models make up component worlds (e.g. CORBA Components [HV99], COM components [Box98] or Enterprise Java Beans [KS06]) that differ in the granularity of components, the support for adaptation of components, if they yield lose or tight coupled systems, if they respect the black box principle, etc. Also they often are limited to a sector
(e.g. GUI-components), a single programming language (EJBs) or operating system (COM components).

Beside the capacity to access and compose components from many different sources and connecting different organizations and platforms, another endeavor goes to build more intelligent program systems which are capable to automatically find compatible components by means of machine readable contracts and metadata description or the automated usage of alternative components in the case of error or version update with enhanced functionality.

But not only components, also databases and user interfaces may wanted to be exchangeable and/or automatically be switched if needed. If a database is not available at the moment another could be consulted or data even could be obtained by doing the automatic consolidation of information of various sources. This would allow a higher grade of data independence than currently possible and also make it easier to change the underlying database of a program, what in present systems uses to be very complicated. But despite various advances in this area, data independence as defined in this work, is extremely complicated to reach and has to be considered an open issue so far as we will see in a later chapter.

The reason of having different user interfaces is that different devices need different user interfaces, for example due to a much smaller screen of mobile devices in comparison to a desktop computer or other differences. Also a web interface differs from the interface that a desktop solution requires, as it needs to work within a browser. Another example is different users with different rights and privileges, where not every user need to access full functionality. Instead of writing different versions of a program for all devices and purposes, it would be much better to be able to just designing different user interfaces for the same program and then switching it as needed. Despite some approaches that are trying to solve this, user interfaces built with traditional programming methods are very tight connected to the underlying system and program logic, so that they normally are neither independent nor exchangeable. Therefore this also has to be considered an open issue.

All these and more are the ideas behind SOA: a switch from tightly coupled systems, written in a certain program language for a certain operating system to solve a clear defined
Software reuse and SOA

task, to loosely coupled open systems that are much more flexible, resistant, sustainable, that can be adapted to newly arising requirements, support evolution, allow including new technologies and components, new data sources, and can continue to be used in the future as development goes on. An important milestone towards that goal is the usage of standardized and autonomous web services instead of complicated bridging technologies that were used to bridge different programming environments and component models before. This alone already makes it much easier to achieve compatibility, as these services can be invoked from any platform and do not depend on a special type system. Together with an enhanced separation of concerns, that is, a very clear modularization and separation of program parts and responsibilities, SOA so far is the most promising approach to realize the idea of an ideal component world.

SOA also means software construction with reuse in mind, where each part, especially its first citizens "services", is specially designed to be reused. The switch of paradigm is an acknowledgement to the fact that software seldom is definitely finished with nothing left to add, and therefore to begin seeing software as just a snapshot in time. In SOA, software is more likely a temporary composition of available configurable and reusable components.

There are also some ongoing problems with the SOA paradigm [Dav07][LS][Ma05]: first, as SOA derives from the business sector, much of SOA terminology and WS-* extensions is directly related to business issues and developed for business needs and until now it only rarely has been used also in scientific projects. The consequence is that it requires some effort to adapt it for scientific purposes. We will examine this problem more in detail when we see the significant different needs of business and scientific workflows in a later chapter, which reveal some general differences of the requirements of scientific and business projects. The differences however do not affect so much basic SOA technology but take place on a higher level of workflows or a need for different WS-* extension demanded due to special scientific working method and requirements.

Next there is the not so easy to be answered question of how to properly build a real SOA system. It is important to underline that just using services does not mean building a SOA. Creating and exposing services is only the first and easiest step. But when doing only this, there remains the danger of later having a traditional system that just is calling web services...
instead of local methods, but without being more compatible, extensible or maintainable than other solutions. The difficulty at building a true SOA based system to really benefit from the promised advantages is to consequently realizing SOA principles. Even though a lot of (often competing) technologies are at hand to help, principals are not realized by simply using some WS-* extensions but need an aware understanding and implementation of those. According to [LS] "There are no effective published strategies for building end-to-end systems based on SOA; there are no approaches for understanding end-to-end quality of service (QoS); the technologies that SOA are based on are still immature and it is not clear what works and what does not work”.

There are for example many possibilities to do the linkage between services, but many of them connect them very tight together exactly like in traditional programming with much knowledge of each other. Strict contract based programming, where components don’t know each other and communicate only by interfaces requires much more effort and needs to aware implement this principle together with the usage of technologies like UDDI [Cer02] to locate compatible services. Data independence sounds easy to demand it as a requirement, but in praxis as we will see later none of the existing technologies can help to support complete data independence. Another example is that there still remains the question of how to connect the services with the user interface or other parts that are not services and remain platform dependent.

So in this work, to help finding a solution to those questions, we first try to summarize and categorize most important SOA principles by extracting some key factors of what is considered most relevant for constructing a true SOA. These SOA principles can be put into two main categories that are summarized in the following:

- Technology agnostic design for better reusability with platform independence and data independence as most important areas.

- Using separation approaches for avoiding too complex and tight connected software by a design using horizontal layers, a strict separation of interface and implementation, and the further separation of components and their composition.
In this work we will pay special attention to the last point regarding the separation of components and composition. The reason for that is because this is the second logical step after developing services and exposing legacy components as services. If one takes into account the above mentioned key factor of a design independent of programming language and other dependencies, an interface-based workflow as the skeleton of the program logic automatically moves to the forefront. Scientific workflow systems would seem a very natural way for the composition and orchestration of those to define workflows and program logic in scientific projects. A remaining problem however is the poor SOA capabilities of existing scientific workflow systems [YYH09][CKRJ05][TCBnAE07]. To fit into the SOA paradigm they would need to be interface based, as well as offering their functionality as (web) services. Also better user interaction capabilities are required. Not satisfied with existing approaches, a new lightweight experimental scientifically workflow system with special SOA capabilities is developed in this work, that allows the easy composition and execution of web-services by means of interface description only, with many possibilities for user interaction and with build-in trader functionality that assists the idea of more intelligent program systems in many ways.

The questions regarding benefits of SOA migration and open issues in praxis are investigated in this work in the light of a case study within the scientific field of environmental modeling, specifically in the sector of water quality assessment, where we had much of the above described requirements:

- Environmental modeling requires especially flexible systems due to the experimental nature of model development, refinement and comparison of results

- Generally different organizations are in charge of water quality assessment in different regions, which may have to work together and possibly want to share data and some models in the future.

- Data independence is usually an important issue in this field, as the whole applications are very data-heavy.

- Several COTS should be integrated: usually Geographical Information System (GIS) capabilities are needed, as well as complex calculus engines, due to the fact that many
of the models need to do complicated calculations that often are already implemented in some COTS packages.

- It is not unusual that there are doubts about if a web application or a desktop solution is better suited to meet the requirements of a software-based environmental model. Sometimes both of them could be needed.

- Working environmental systems currently have different users with different privileges. Most of them are not programmers but nevertheless may want to develop and test new assessment models.

With all these requirements a decision for a SOA based approach was clearly justified, which in our case lead us to realize the idea of an open water quality assessment portal, which allows to use different data sources, introduce new and modify existing models, and offer a stock of web services with methods that can be used to implement new models.
3 Building SOA

This chapter reflects on what to take into consideration when designing a SOA based system. As Earl [Erl06] repeatedly underlines there are no universal rules of how to implement SOA (see also [BLJM08] or [Pet06]). For a long time SOAP services have been by far the most important technology for building web services with the consequence that many SOA technologies and extensions now base on that. But recently also simpler REST services [Til09] are gaining importance and can be used as well for SOA implementation. There are different possibilities for service discovery and using a UDDI registry is only one of them. There is no standardized SOA stack and it is not prescribed which layers and in which order it needs to contain. There are a variety of web services wiring languages, and although at the moment WS-BPEL [ACD+03] is clearly the most used and most important one, this also may not be the last word forever and another one may replace it one time. The key point is that all these technologies may change. SOA is not a closed paradigm with respect to what is used to implement it and on the contrary means being best prepared to react to changing requirements as technology heterogeneity and frequently changing requirements belong to the basic assumptions. In fact, SOA is the first architecture model ever that considers existing system as integrative ingredient of a new system to build [Lie06]. What however remains and therefore constitutes the core of a SOA are the basic SOA principles. In [Erl06] a lot of space is used to work out these principles. In this work another view to most important principles is offered putting them into two main categories of technology agnostic design and the heavily usage of separation approaches which are described in the following chapters. The technology agnostic design principle is most important for compatibility and interoperability issues, whereas separation techniques are mostly important for clarity
of the design and architecture what greatly contributes to enhance improving flexibility, extensibility and similar issues.

### 3.1 Technology agnostic design

The first important SOA principle was identified as a "technology agnostic design", that means making as less assumptions as possible about the concrete technology used to implement the system and a design that does not expect a certain programming language, operating system or a special database or even a type of database. Within the field of technology agnostic design we find platform independence and data independence as the most important aspects, each of which is described more in detail in the next sections. Each concrete technology used normally implies numerous platform dependencies, while there are other technologies which try to overcome such limitations. We will see that there are different levels of platform dependencies and how SOA components at the end of the "platform dependency scale" minimize such dependencies. We also will explain what data independence means, and analyze in how far existing technologies can help to realize it and what limitations however still remain. The goal remains to design a system that allows integrating artifacts from different platforms as well as being used from different platforms. A system that makes it possible to switch from a relational database to an object oriented or XML database, or in other words to depend as little as possible on concrete technologies.

#### 3.1.1 Platform Independence

Before reusing an existing software artifact, it is necessary to isolate it from its context and its original environment, preferably with the support of the underlying platform technology. This ranges from the simple possibility to embed the code in a function, class or module and this again in a compiled library, to some further options like the realization of remote calls via a network. Nevertheless, some dependencies generally remain existent in the isolated artifact. Sametinger [Sam97] calls them platform dependencies, using the term platform in a very wide sense. Sametinger defines the notion of a component platform
as any software or hardware a component is built upon. Examples are operating systems, run-time-systems, window systems, compilers, libraries, network connections, composition platforms and execution platforms. He denotes that the less the platform dependencies remain, the better are the possibilities for reuse. To overview different types of platform dependencies it can be helpful to distinguish different grades of platform independencies and arrange them together with the technologies to reach them onto a scale (Figure 3.1) with decreasing remaining dependencies as done in [AHGR+08].

![Platform independency scale](image)

In the following the different grades of platform independence are described.

### 3.1.1.1 Context Independence

The very beginning of making code reusable is to put it into a sub-program that can be called as often as needed without the necessity to repeat the lines of the sub-program again and again in the source code or in other words to be able to use it from different contexts. There are many different names for the same principle, which more or less refer to the
same: functions, subroutines, methods, procedures - to name a few. The slight differences (although not always consistent in literature) are subroutines that do not have a return value, functions that do have return values, methods are functions or subroutines within classes and procedures that can be found denoting both, subroutines and functions. Also important at the same level of source code reuse are macros and templates. In some programming languages (like C or C++) the use of macros is very widespread, although their excessive usage is widely agreed to make programs prone to errors, for example because they may undermine type safety. Macros and templates work as pure text substitution mechanisms. Templates, in contrast to macros, are mostly designed to reuse a function for different data types. All three have in common that they still work on a reuse level that bases on un-compiled source code.

3.1.1.2 Application Independence (Libraries)

Application independence means that an artifact can be used in applications different from the one it was originally designed for. Normally, this is achieved by placing code into a library or a module. The general difference between both is that normally a module is used as a whole, whereas a library may contain a collection of potentially independent functions and classes from which applications may just a part of it. In practice, various types of libraries are existent which provide this kind of independence. Amongst all, we first point out the source code libraries that may contain collections of definitions of values, functions, classes, generic parts, etc. in readable and editable source code form, sometimes referred to as glass box type of reuse [GR95]. The code is simply copied into the new application thanks to some kind of support mechanism, like the include directive in C/C++. The Standard Template Library (STL) of C++ [Jos99b] is an example for this kind of library. A second type of library is the static library, containing compiled code that can be reused by including it into a new executable application through a binder or linker. Dynamic libraries, as the third type of library, also contain precompiled code that is, in contrast to static libraries, connected with the executable program during runtime. It is the loader responsibility to find the demanded library and load it into the memory at the adequate time. This is the case with Windows DLLs or shared objects in Unix. This type of independence allows for more
flexibility as there is no requirement for the physical inclusion of the reusable artifact within the executable program that reuses it. Anyway, this also involves some disadvantages, i.e. when a dynamic library cannot be found in the specified place or when it was substituted by a newer, partly incompatible version (a problem known as DLL hell).

3.1.1.3 Programming Language Independence

The programming language independence can be explained as the option to reuse a reusable artifact in different programming languages and not only in the one used to create it. In some cases, even existing and precompiled binary files can be reused from a language different from the one that was used to create that file (for example an object file created in C, reused by a program written in Smalltalk). This represents a form of programming language independent reuse, even though differences between languages concerning their data types, procedures calls and parameter passing often complicate its application. Also the above mentioned DLL’s fall into this category. In general they can be used by programs written in other languages, but the mentioned problems remain ¹.

In order to obtain true language independence, the problems of different data types, parameter passing and procedure call have to be solved. This is normally achieved by establishing some language independent, binary compatibility norm. The Component Object Model (COM) [Box98], for example, guarantees the reusability of programming language independent artifacts by describing their types and interfaces with an independent interface definition language (IDL) and the adoption of a specific format for procedure calls and parameter passing. The underlying platform cares for the calling process and the type conversion via the so-called marshalling process. However, during the evolution from COM via DCOM, COM+ to finally .Net (Microsoft), the problem has been solved in a different manner, namely by the Common Type System (CTS) that is used by all .Net compatible languages to define their own set of types [Tro05]. This possibility of using different languages to produce same compiled binary also could be interpreted as a different, although less important meaning of the term programming language independence. With CTS and

¹there are many different dll types. Newer version have some additional information to facilitate this task
CIL mechanisms of .Net, different Net languages can be used to produce the same compiled code. This way every programmer can use the language he is used to without this being a limitation to reuse components written in another language (though they are .Net compliant). Also COM components can be produced using different languages.

### 3.1.1.4 Operating System Independence

This type of independence implies the possibility to reuse a software artifact in different operating systems, commonly designated by the term portability. Portability is achieved in different ways: 1) The portability of source code, what implies that compilers for different operating systems are available to compile the same programming language, often complemented by the option to let the source code include instructions that handle differences between different operating systems, i.e. by preprocessor directives in C/C++. 2) Packages that contain different executable versions for distinct operating systems, choosing automatically the appropriate one. Examples are fat binaries of programs like OpenStep [Sun94], and fat or universal binaries for different Mac OS versions), virtual machines that abstract the underlying operating system, like the Java VM or CIL for .Net. In those cases the compiler creates code just for the virtual machine, not for the physical machine of the operating system. Thus, source code can be executed on any system where a compatible virtual machine exists for.

### 3.1.1.5 Location Independence

This grade of independence leads to the area of distributed systems by reusing artifacts located in any locations in a network. It can be distinguished between a homogeneous location independence where both the origin and the target platform have to be identical (e.g. DCOM), and a heterogeneous independence, where the communication between different components placed on different platforms and machines is possible, (e.g. web services). Only latter one (heterogeneous location independence) also guarantees programming language and operating system independence.
The technology to achieve location independence is mainly some kind of middleware or platform that allows a remote call to a remotely installed artifact. Examples for models that support this kind of independence are Java RMI \cite{Gro01}, DCOM, .Net Remoting \cite{Tro05}, or CORBA \cite{HV99}. This middleware, as need to be installed on both origin and target system can be seen to introduce another (although less heavy) dependency. Comparing for example web service to CORBA, it can be noticed that for the sake of platform independence both offers nearly the same functionality \cite{Pet06}. An important difference is however that CORBA uses its own protocol (IIOP), instead of making use of existing web standards like web services do. The system administrator needs to open an extra port in the firewall in order to use CORBA components. This proprietary protocol constitutes a dependency. Therefore we put on the end of the scale server-sited component models like web services, where there is no need for special middleware or firewall configuration at the client side.

Recent technologies like Windows Communication Foundation can also be placed there, but are furthermore permitting another variant of independence supporting communication using different protocols (SOAP over HTTP, SOAP over TCP or some proprietary Windows protocols like named pipes), that could be configured in the configuration file without need of code modification and recompiling.

There are other dependencies that do not fit well into this scheme. The decision for a web application or desktop application constitutes a dependency: although web applications can access the same components and maybe are even better suited to access web services, they require a different implementation style as some features are not available. State management between calls becomes an issue that needs to be resolved, for example with sessions or cookies. Available GUI elements (widgets) to build the user interface differ and the question appears when a postback to the server takes place. As with other types of platform dependencies also here technologies exist that help to overcome and blur differences in the implementation. Examples for this are Windows Communication Foundation (WCF) \cite{Pat11} and Windows Presentation Foundation (WPF) \cite{Mac10} together with Silverlight \cite{Mac12}. First one allows designing components that can be efficiently used for both, desktop applications and web applications. For web applications, the components can for example be exposed as SOAP web service. If we want to use the same component in a
local desktop application it is possible to switch the protocol which allows for much better performance values. WPF and Silverlight permit developing a graphical user interface with help of a markup language. The user interface then can be used in web applications with Silverlight (there are however slight differences between the markup languages of both, although they are largely identical).

3.1.1.6 Platform Independence in SOA

Reusable components and first citizens in SOA are (web) services. They reside at the maximum level of platform independency scale and are usable from any client platform. This way the goal of nearly complete platform independence can be reached for the most important ingredient of a SOA system, services. It is however important to note that web services alone are not sufficient to build a whole system. Every software also needs to have a client side containing for example the user interface that necessarily remains platform dependent. How complex the user interface is depends on the concrete case, but it is not unusual that there are complex interactions between the user interface and the rest of the program, which is very difficult to model in SOA. This is a problem of how to integrate the user interface into the SOA paradigm which until now is given only few space in literature. More about this topic can be found in the chapter of the separation interface and implementation.

Another special type of dependency out of scope of the platform independence scale is frequently a link to a concrete used data base management system, on which a program relies as all processed data reside there. The problem of data independence however differs from platform independence in many aspects and to overcome these, other technologies than shown in this section are needed. This is what the next section is about.

3.1.2 Data Independence

Before examining the question about what data independence means we examine the term "data" itself and try to hedge it off against similar terms like information or knowledge. Then an overview about the different forms of commonly used data representations (data
Technology agnostic design

models) is given and the problem of converting one to another (impedance mismatch) is explained. Secondly the problem of different structuring and organization of data is explained. With this we can go on to define the term data independence. After that a number of technologies that help reaching data independence are analyzed. Last we see remaining open problems.

3.1.2.1 The knowledge pyramid

According to a widely used model (e.g. \cite{Bod05}), data is composed of characters of a character set according to well defined syntax rules. Data begins to be information when assigned a meaning (semantic) and put into a context. Knowledge on the other hand originates from the linkage of information, the interconnection of information pieces and the awareness of the relation between isolated information. It is often argued that the separation of the different steps of the pyramid frequently is more likely a continuum than clearly separable stages. Whereas digital data is most often resident in databases,
which are using different data models or are stored somewhere else like simple flat files, knowledge is generated with the application of different models to the data, which bases on different knowledge-generation approaches like artificial neural networks, case-based reasoning, expert systems or genetic algorithms. In this chapter we are only talking about data independence that is data stored in different data base management systems or other location without any assigned meaning. Different data models might be used to store the data, different implementations of data bases of the same data model may exist and different organization regarding data structures, used tables and record sets are to expect. First we have a look of most important data models.

3.1.2.2 Data base models

Databases rely of different data models to store the data internally. Whereas the hierarchical and the network model lost much of its importance nowadays (although regaining importance with the arising of XML and the semantic web), the relational model is today the most widely used one followed by the object oriented model. Beside this, specialized XML data bases arose because of the increasing need, promoted by the web, of storing XML data directly. Following we will briefly examine the most outstanding data models.

3.1.2.2.1 Hierarchical data bases

The historically oldest database model is the hierarchical model [FS76]. It models the data with a hierarchical tree structure. So each record - except the root node - has exactly one parent node. The data is stored in form of registers that consist of a set of fields. A set of registers with the same fields is called record type, which is the equivalent of a table in the relational model. The relation of data between different layers usually is simple (one-way). Connections between different trees and across various levels within the same tree are not possible. Therefore, it is only possible to model 1:1 and 1:n relationships. The frequently needed m:n relationship have to be modeled with redundancy using m 1:n relationships, which however may easily cause inconsistency problems. Another possibility to model n:m relationships are virtual parent child relationships. Since a child node directly contains the physical address of his parent node, queries against hierarchical data models
often can be resolved very fast, because there is no need to query reference tables like in relational systems. On the other hand, since it is often only possible to define child-parent relationships while there is no link in the other direction, this slows down the answer for some other sort of queries. The main limitation of the hierarchical data model however is the often needed redundancy and the lack of referential integrity. Furthermore it might be very difficult to change the structure of the data base later.

3.1.2.2.2 Network model Databases
The network data model [FS76], presented 1969 by the data base task group (DBTG) of CODASYL (Conference on Data Systems Languages) to avoid the need of redundant nodes of the hierarchical model, can be seen as a generalization of the hierarchical model. Data objects are interconnected in network relations. The tree structure is converted to a directed graph. The network model doesn’t require a strong hierarchy and so it can model m:n relationships (a data object is allowed to have various parents). Also connections between sibling nodes and more than one root node are possible. Often, various search paths exist to come to a data node. The most important drawback of this model is that its structure tends to become unclear and complex very fast.

3.1.2.2.3 Relational Databases
The main characteristic of a relational data base [FS76] is its organization with a collection of tables (=relation). Each line in a table is a record, a tuple of attributes. Arbitrary relations between data are allowed, which are determined by certain table column values. Relational data bases base on the concept of the mathematically well defined relation. Operations to these relations are determined with relational algebra. This also is the theoretical background of SQL, the most important querying language for relational data bases. In spite of the mathematical definition of this data model, it is comparatively easy and flexible to handle, which had a large influence of the success of this data model. The metadata of the structure of the whole database and its tables is defined and stored in so-called schemes. The range of values an attribute is allowed to have can be defined and limited by constraints. The relational data base model was developed 1970 by E. F. Codd (IBM).
In 1979 with Oracle the first commercial implementation came out. Today more than 100 relational data base management systems exist, and it is largely used as a reference model.

### 3.1.2.2.4 Object Oriented Databases

In an object oriented data base \[\text{IGC08}\], data is stored as objects in the sense of object oriented programming. An object contains object attributes, data and methods. An object encapsulates the objects attributes according to the concept of object oriented programming - the access to the data objects is realized via access methods. Objects may be composed of arbitrary further data types and objects. Objects (object instances) belong to an object class (or are copies of a prototype in some languages). Object classes are arranged in class hierarchies. Objects ensure object identity thanks to a system wide unique object ID assigned to each one. The relation between data objects are managed by the data base system. Objects can inherit properties and data from other objects. With late binding, the overridden methods are used for inherited objects. Especially for some "non-standard applications", for example of the domain of CAE (computer aided engineering), computer aided software development or multimedia- and hypermedia applications, relational data bases systems have shown not to be very suitable due to the need of the storage and management of large amounts of data with very complex data structure and having different phases, where object oriented data bases seem to offer a more natural way of data storage. With the usage of object oriented data bases, a mapping between the rational data model and the type system of object oriented programming languages is no longer necessary. Also there is no need to make joins of various data tables to satisfy queries. The object oriented data base model is in fact a combination of approaches of classical data models, object oriented programming and knowledge representation. The goal is to save the structure and behavior of complex objects 1:1 in data bases.

### 3.1.2.2.5 XML Databases

Due to the widespread usage of XML in web applications in the recent years, the use of XML databases \[\text{Bar07}\] became significant. This way, there is no need for perpetual transformations (e.g. from relational data models to XML and vice versa) of the data models. The internal model of Native XML (NXD) databases depends on XML and uses
XML documents as the fundamental unit of storage, which are however not necessarily stored in the form of text files. Especially hierarchical data can be easily serialized in XML, which may be viewed as a renaissance of the hierarchical data model storage. But XML databases also have the disadvantage that they often have much less performance than relational databases in processing large amount of data. XML databases are still very young and immature, because there is little experience with them.

3.1.2.2.6 Hybrid database models
Furthermore there exist a variety of hybrids, for example the object-relational model, where an object oriented access layer is put on top of a relational database and objects and their attributes are loaded not until really needed. With this layer the objects with all characteristics of object orientation can be used although in reality still using a relational model. Also hybrids between relational databases and network databases [Hay81] exist as a further example, with the goal to join the advantages of both models.

3.1.2.2.7 Model mapping and impedance mismatch
Next there is the question, how the different models are related to each other and therefore how far it is possible to convert one model to another without loss of information. Hierarchical models for example always can be converted into network models, because they are a subset of the more general network models while this however not always is possible in the other direction. While the main difference of the hierarchical model and the network data model is the lack of the possibility to model m:n relations, the network data model and the relational model are more or less equally powerful regarding their expressiveness. With respect to the object oriented model, a collection of objects interlinked via pointers of some sort, can be seen as an equivalent to a network database. A network database in turn can be viewed as an extremely de-normalized relational database. XML originally is very close to the hierarchical model due to the tree structure of XML documents, but is flexible enough (e.g. with the usage of XPointers) to be used to serialize other data models as well.
So all of these modes are much related to each other and generally can be converted to each other, especially to the most general ones. In practice however some difficult problems arise when trying to convert one to another. One of the most usual and relevant problems is the often needed conversion of objects to relational models and vice versa is known as the "object-relational impedance mismatch". For the mapping between data of a program written in an object oriented language and a relational data base, for example to store objects in a relational data base for making them persistent, object-relational mapping is needed. In fact, for each access to relational data bases some sort of data structure conversion is necessary and some authors talk of impedance mismatch in any case [Dat06]. In the simplest case, classes are just mapped to tables where each object corresponds to a row and each attribute has its own column. The object identity corresponds to the primary key of the table. If an object holds a reference to another object this can be modeled with a foreign-primary key relation.

But the real problem of object-relational mapping is deeper founded and the difficulty lays in the collision of two fundamental different data model paradigms. Some assumptions of the models are conflicting and other concepts just do not exist in the other model. An object can be characterized by some properties: identity, state, behavior and encapsulation. The rational model on the other hand as derived from relational algebra is storing "truth of statements" in relations (e.g. "there exists a person with the name NAME which is working in the company COMPANY"). This for example causes differences with the following points:

- Reference data types: The relational model strictly prohibits by-reference attributes and is using only value types whereas OO languages embrace and expect by-reference behavior.

- In relational system two tuple with the same attributes are considered to be identically. In OO systems they are not. To solve this it is for example necessary to add a unique identifier to each relation.

- Constraints in OO languages are generally not declared as such, but manifest by throwing exceptions when trying to access encapsulated internal data with values out of the valid range.
The relational model, on the other hand, calls for declarative constraints on scalar types, attributes, relation variables, and the database as a whole. There are also important concepts of object orientation that do not have any counterpart in relational algebra, as for example: Encapsulation/Accessibility/Invariance/Interfaces: In the object oriented model, data is encapsulated, and access to data is restricted and accomplished through access methods. Modifiers like private, public, protected etc. are used to determine the accessibility of data and methods. RDBMS on the other hand tends to use rule-based and role-based protection and security mechanisms instead of direct interface restrictions Inheritance/Polymorphism: The relational model does not know inheritance. To nevertheless store inherited attributes of objects correctly in relational data bases in the main three techniques are used. Some frameworks provide some further variations and mixtures of these basic methods. They are:

1. One table for each hierarchy of inheritance (Single table) All attributes of the base class are stored in one table together with all attributes of all subclasses. Furthermore a so-called "discriminator" is stored in a further column which determines to which class the objects stored in this row belongs.

2. One table for each subclass (Joined) With this method for each base class a table is created and for each subclass a further one. A discriminator is not necessary this way, because the class of an object is determined with a 1:1 relation between the entry in the table of the base class and an entry in one of the tables of the subclasses.

3. One table for each concrete class (Table per class) Here, all attributes of the base class are included in the table of the concrete sub class. There is no extra table for the base class. A disadvantage of this method is that it is not possible to detect instances of different subclasses.

3.1.2.3 Restructuring data

After demonstrating the problem of the co-existence of different data models and the "impedance mismatch" called problem of converting one to another without loss of infor-
mation, now we have a look at the problem of accessing data that is organized differently, although both might use the same data model. Related to that, as it also requires the transformation of different structured data into a uniform scheme, but going a step further, is doing the consolidation of data of various and heterogeneous sources. Typical problems of differently structured data are for example:

- The numeric representation of a date YYYYMMDD differs from other forms of date representation in alphanumerical way, with different separation characters or a different order of the parts year, month and day.

- Keys and encodings differ (e.g. different country encodings into DIN ISO country codes) or other adaptations of data values (e.g. different codes for the sex like 1 (male) 2 (female) and m (male) and f (female)) is necessary.

- One database uses different measuring units than another like miles and kilometers, columns are split up in the other database into multiple columns (e.g. a comma-separated list specified as a string in one column and as individual values in other columns) or the other way round that repeating columns are aggregated into a single table (e.g. single addresses in a set of records in a linked address table are stored as a series of addresses in other record sets).

- Some calculated attributes of a table may not be present in the tables of another database and need to be calculated first (e.g., sale amount = quantity * unit price).

For the consolidation of data of various sources and to join them further operations are necessary, including filtering, sorting, elimination of duplicates, which for example requires mechanisms for object identification. A good place to look for techniques for the homogenization of different structured data sources is data warehouses [KC04]. Data warehouses are commercial standard solutions for processes like data mining or the aggregation of business ratio and analysis within multidimensional matrices called online analytic processing (OLAP). A data warehouse joins and consolidates relevant data from different data sources into a common consistent database with one unified representation. It therefore allows a global view of heterogeneous and distributed data, something very similar to what we want.
to reach. To join data, data warehouses perform the so called ETL-Process: Extract the data from different sources, transform the data into a unified representation and then load it into the unified data bases of the data warehouse system. That is what is also required for every other data independence technologies. The transformation step herby consists of two transformation sub steps. Firstly, syntactical transformations where formal aspects of the data are corrected; the data is modified according to the necessities of the target system. Then semantic transformations were the data is checked with regard to its content including validation of data, error finding or the elimination of duplicates.

But while data warehouses are assuming varying data models, for the easier case when the destination and the target data model is the same, transformation of data can also be performed by querying or transformation languages. With the relational data model for instance, SQL queries can be used for restructuring data and defining views. For XML data, special transformation languages like XSLT [Man02] exist. While the possibilities of SQL for data transformation however is very limited, XSLT on the other hand is very powerful and permits all transformations pointed out in the examples above.

### 3.1.2.4 Data independence definition

We saw that data may reside in different data base management systems based on different data models. We also saw that data may be subject of restructuring, or the data base should be switched to one that is structured differently. To finally give a definition of the term ”data independence” there is only left to mention another possibility: that data resides in different data bases of the same data model, but from different vendors. To summarize, here again all possibilities of changes that data may experience:

**The data base management system (DBMS) changes to a DBMS of the same type:** That could be changing to a DBMS of a different vendor, from a commercial one to freeware or the other way round. The main problem here are different supported features and different dialects of the querying language.
The data is being restructured: Restructuring of data includes the addition and deletion of tables, relations or attributes, splitting the information of one table into various or reformatting data.

The used data model (the DBMS type) changes: This is the case that was explained in the first part of this chapter: The coexistence of different data model types and the possible problems when switching from one to another. It includes for example changing from a relational data base to an object oriented data base, an XML data base or to read from different sources.

It is necessary to mention again that the most frequent problem however is not so much migration of data to another DBMS, but rather having to change a program to access data of another already existing data base, that naturally is organized in a different way or stored in completely different data bases stores. For the problem of data independence however both cases are the same.

The independence of data in this sense is now defined as the immunity of a programs data structures to later changes, evolution or reorganization of data in the underlying data base management systems or including a change of this. This means that programs that rely on a certain data structure do not have to be changed if any of the above mentioned changes occur. To maximize data independence, a program may defines its own data structures (information models) and data independence technologies are used to transform queries and deliver data in the right format.

The most important approaches of those technologies are described in the following.

3.1.2.5 Technologies for data independence

There are several technologies that may be used for that purpose. The approaches work with different technologies:

- Technologies for unifying access to DBMS of different vendors through a unifying API. Also some object-relational mapping tools offer this functionality.
Technoloy agnostic design

- Technologies that work with different data layers and permit defining views and perform a mapping between the views and original data
- The implementation of certain design patterns
- Unifying querying languages

We briefly review them in the following.

3.1.2.5.1 Unifying APIs
There are several APIs that help to access data while blurring differences between different database management systems. Most of these APIs are designed only for accessing relational database management systems although rudimental support of other data models is also offered by some. Some outstanding examples are:

3.1.2.5.1.1 Low level APIs
Many low level APIs permit unified access to mostly realtional data bases. Examples are:

**JDBC:**
JDBC [ORAb] is the standard Java API to connect to databases from different vendors. JDBC is comparable to ODBC in Windows or DBI in Perl. It is only usable with relational data bases. The job of JDBC is to open and manage a database connection, to pass on SQL queries and to convert the result into a form which can be used by Java programs.

**ODBC:**
ODBC [Micb] is a common interface for the access to heterogenous SQL data bases. ODBC is broadly used in Windows platforms, and drivers exist for quite any relational database like MS SQL Server, MS Access, Fox Pro, Dbase, Paradox, Oracle ODBC or even not-relational data sources like MS Excel or text-files. Like JDBC in Java, It handles the SQL request and converts it into a request that the individual database system understands.

**OLE DB:**
OLE DB [Micc] is the higher level replacement and successor of ODBC. It is implemented as a COM component. It is extending the feature set of ODBC to support a wider variety
of non-rational databases, such as OO databases and spreadsheets that does not necessarily implement SQL. Also a modified version especially designed for DotNet exists called DotNet Managed Providers.

3.1.2.5.1.2 ADO

Microsoft ActiveX Data Objects (ADO) [Rof01] was introduced by Microsoft in October 1996. ADO is the successor of DAO (Data Access Objects - the API, not the equally named design pattern) and RDO (Remote Data Objects). ADO enables client applications to access and manipulate data from a variety of sources through an OLE DB provider. Its primary benefits are ease of use, high speed, low memory overhead, and a small disk footprint. ADO is a high level programming interface and therefore provides a layer between programming languages and OLE DB. Programmers must be aware of the specific database they use only when establishing a connection. No knowledge of SQL is required to access a database when using ADO, although one can also use ADO to execute SQL commands. The disadvantage of using SQL directly however is that it introduces a dependency upon the usable databases. The ADO-Data model consists of three main components, ” connection” that is used to establish a connection with the data base to be used using a location and name parameters, ”RecordSet” which is an object dependent from the connection which contains data in the form of tables and queries, and ”Command”, which supports data base commands execution, typically using SQL statements.

3.1.2.5.1.3 ADO.Net

ADO.Net [Ham08] is the latest version of ADO, but with some so fundamental changes that is often considered a completely new product. One of the most fundamental differences indeed is that ADO.Net is a managed library, therefore typed with .Net CIL types etc. The namespaces that allows for interacting with local and remote relational databases are known as ADO.NET. The basic idea of ADO.Net again is to separate the acquisition of data from treatment and display. The developer is able to build a single code that can dynamically pick and choose the underlying data provider via application configuration files without the need to recompile or redeploy the application.
The .Net platform supports numerous data providers (i.e. OLE DB, SQL Server, ODBC Oracle), each of which is optimized to communicate with a specific database management system. Different data providers can be programmed to access any unique feature of a particular DBMS. Each defines a set of class types that provide core functionality.

While ADO was primarily designed for tightly coupled client/server systems, ADO.Net was built with the disconnected world in mind, where the connected mode however is still supported. In the connected mode one has to handle Connection, Transaction, Command and DataReader objects to first build a connection to the database and then read information or send commands. DataReader objects are the replacement of Cursors in ADO. Command objects support an internal parameter collection, which can be used to add some type safety to your SQL queries and are quite helpful when triggering stored procedures. ADO.Net introduces a disconnected mode where, unlike the connected layer, data obtained via a data adapter is not processed using data reader objects. Rather, data adapter objects make use of DataSet objects to move data between the caller and data source. Conceptually, a DataSet object can be seen as a small in-memory relational database in its own right that allows for manipulation of data in any direction (whereas a RecordSet was a forward-only reader). It represents a local copy of any number of related data tables. The user is able to manipulate and update a DataSet contents while disconnected from the data source, and send any modified data back for processing using a related data adapter.

The DataSet type is a container for any number of DataTable objects, each of which contains a collection of DataRow and DataColumn objects. The data adapter object of the used data provider handles the database connection automatically. As an attempt to increase scalability, data adapters keep the connection open for the shortest amount of time possible. Once the caller receives the DataSet object, the calling tier is completely disconnected from the data base and left with a local copy of the remote data until the caller explicitly passes the DataSet to the data adapter for updating.

ADO.Net can easily save and load data as XML, which is another mayor difference between ADO and ADO.Net. In fact, the data obtained from a data store is serialized as XML by default. This also helps to avoid problems with firewalls.
3.1.2.5.2 Data Layers

This is an approach used in many DBMS to permit some grade of data independence. The basic concept is the usage of external schemes (views) which are not necessarily identical to the real organization of data. This way, real organization of data is allowed to change what allows some grade of data independence. The following data schemes are commonly used [FS76]:

**Internal scheme**

The internal scheme describes data from the view closest to the machine. It extends the logical scheme with aspects like saving methods and helping constructs for more efficiency. The internal schema is intended to reflect efficiency considerations by describing the structure of the database in terms of an abstract model of storage. Data representations, access paths, etc. are defined at this level.

**Logical scheme**

The logical scheme describes a real-word-view to the data in the data description language (DDL) of a concrete data base management system. The external and internal schemes extend the logical scheme.

**External scheme**

The external scheme describes different user views for different user groups. It usually includes only a part of the logical scheme, hence more than one external scheme is possible. The external level of description contains any number of external views of the database, each of which is a collection of data objects representing the entities, properties, and relationships in the enterprise which are of interest to a specific application. Each external view of the database is associated with an external schema describing the objects in the external view, as they are to be presented to that application. Although ”real” tables of the logical scheme may be visible in the external schemes, it can be used to hide changes made in the logical scheme so that applications that rely only on the external schemes do no notice the changes.
Among those three schemes, the concept is frequently enriched by another scheme called the conceptual scheme:

**Conceptual scheme**

The conceptual data scheme is the platform independent description of data, that is, a description of data independent of a concrete data base and computer system. The toolkit to create a conceptual scheme consists usually on diagrams of entities and relations. The logical scheme later is derived from the conceptual scheme. The conceptual schema serves as an information model of the enterprise which the database is to serve, and as a control point for further database development. Information of interest to the enterprise is described in terms of relevant entities, their properties, and their interrelationships.

Depending on the used schemes, in literature different types of data independence are distinguished. We are talking about physical data independence when the physical organization of the data remains hidden for the programs working on the data base. Changes in the internal scheme are possible as long as the logical schema remains unchanged to hide physical organization issues, like access paths. Physical data independence describes the ability to change the internal scheme without having to modify the conceptual or external schemes.

Logical data independence on the other hand means that each program is working with its own specific views to the data. So it might be possible that tables showed in a program differ from the "real" tables used in the data base, (i.e. having more or less attributes or various tables melted to a single one). The "real" tables therefore may change without affecting the program. Logical data independence therefore describes the ability to change the logical scheme without having to modify the external scheme or the applications using these external views.

The implementation of logical data independence by the definition of views is usually limited to a certain data base type or even a special implementation of it. Only later changes and evolution of the data organization of this single data management system is foreseen, but not a possible change of the whole data base management system itself. In fact, normally views are realized simple by defining queries. A further, more practical problem with this approach is the effect that changes made on views have, as they not always do have effect
to the layer below. Especially critical here are deleting operations. So in practice, views
often are only read only and one-way.

**Data layers as design pattern**

The approach of using different data layers however can be applied as generic design pattern
to grant data independence also in technologies other then RDBMSs. The different data
layers here not only extend the logical layer but perform a real mapping between them.
The Ado.Net entity framework [Ler09] is an example for this to allow for logical data
independence by defining two different data schemes and a mapping description. The basic
idea of data layers still is that a change of the data scheme in the lower layers never affects
the higher layers, which can be continued to be used as they are. The difference of realizing
view by queries or by the definition of a mapping between the layers is that latter approach
is more powerful and possibly permits some extended mapping possibilities.

**3.1.2.5.3 Design patterns**

Despite using different data layers and a mapping between them, there are further design
pattern that can be used to allow for data independence. In the following some of the most
important approaches are presented:

**Data Access Object (DAO)**

In 2001 Sun Microsystems released the specification of the data access object (DAO)
design pattern [ORAA], which is considered a J2EE core design pattern and describes, in
a technology independent way, how the architecture of a system may be to grant data
model independent data access. The data models may vary from relational databases,
OO database, flat flies, XML documents and more. All different possibility is abstracted
by the class data source. The data access is only accomplished through the data access
object, while for the transfer of the data transfer objects are used. Am UML diagram of the
participants and the typical call sequence when implementing the DAO pattern are shown
in figures 3.3 and 3.4.
The basic idea of the DAO pattern\(^2\) is to encapsulate all data access in a data access object, instead of directly communicating with the physical data source. The data access object has to implement methods like `GetAllCustomers()` or whatever data there may be and is responsible of acquiring these data from the concrete data source. A transfer object may be used to return data also encapsulated in an object. It is also propose to use the DAO

\(^2\)DAO is also the name of a Microsoft API created for a similar purpose, which now however is already obsolete and not used any longer.
pattern together with the Abstract Factory [GHJV95] or the Factory Method [GHJV95] pattern, where the creation and consulting of the data access objects works according to the factory pattern. The specification also mentions the possibility for tools to generate a large amount of the necessary DAO code automatically as it is very time-consuming to write it by oneself but on the other hand can be automated relatively easy. All DAOs of a program form the data access layer (DAL), from which all data access is performed and therefore is the only place changes have to be made when the underlying data base changes.

**Service Data Object (SDO)**

A more sophisticated specification to solve the same problem was proposed in 2006 as a collaboration work of IBM and BEA [IB]. Basically, it founds on the same ideas as DAO. The data sources also can be heterogeneous and should not be limited to relational database management systems. The difference however is a much higher flexibility of supported scenarios and some additional new features. A problem with DAO for example is its static nature, so methods like GetAllCustomers() are static and strongly typed. Some other potential data sources like JDBC ResultSet and RowSet however provide only dynamic and untyped data. Furthermore it sometimes is unavoidable to support dynamic queries that are not known a-priori. So, in SDO the DataMediatorService Object takes the place of the DAO DataAccessObject and is supposed to support for example also dynamic queries. The TransferObject is substituted by the DataGraph Object, which is a tree-like and XML based representation of the data that connects different DataObjects, which hold the real data as public properties (references to other DataObjects are allowed). The client that receives the DataGraph can modify its structure and return it to the DataMediatorService, which then is responsible to update the data source using for example the optimistic concurrency strategy. This implies a mechanism to log the changes made to the DataGraph in a change history. SDO therefore supports a disconnected model of data access. The DataMediatorService also has to implements methods that are capable to read different types of metadata and allow for introspection. In SDO the querying language that is used to do queries is not prescribed.
Until now, SDO specifications exist for Java, C, C++ and Cobol. The specification however does not include how to implement DataMediatorServices, which can be very difficult to do especially with complex data objects where the change history. Service component architecture (SCA) [MR09], a specification that should help implementing SOA systems, makes heavily use of SDOs and following this specification data always should be implemented as SDOs.

3.1.2.5.4 Object Relational Mapping frameworks
Object relational mapping frameworks (OR/M’s) are made primarily to help overcoming the impedance mismatch problem between objects of objects oriented programming models and relational data bases as described in a previous section of this chapter. But as most of them also support many RDMS as data backend, they also help to establish some grade of data independence. The Ado Net Entity framework furthermore potentially supports other DBMS types as well as it supports different data layers and a mapping definition between them like described above, however only permitting some grade of logical data independence.

Hibernate
Hibernate [BK06] is an open-source project for persistence and object-relational mapping
for Java and .Net (NHibernate [Den10]). It supports compatibility with various relational data bases. The querying language is the Hibernate Query language, an own dialect of SQL for Hibernate, which is translated automatically to the SQL dialect of the underlying data bases system. Alternatively object-oriented queries via the Hibernate Criteria-API are used. Hibernate realizes the mapping between Java-Classes and data tables either via mapping files or via Java annotations. All three methods of treatment of inheritance mentioned in chapter 2.1.3 (One-table-for-each-hierarchy-of-inheritance, one-table-for-each-subclass and one-table-for-each-concrete-class) are supported. It supports the relations types 1:1 (object references), 1:n and m:n for collections as well as reflexive references to the own class.

**ADO.Net Entity Framework**

The name of the ADO.Net Entity Framework [Ler09] already shows its close relations to ADO.Net. Although one in ADO.Net already also is working with objects to access a relational database, there is no real mapping between the relational and the object oriented data modeling. The data sets objects with which one works in ADO.Net still are tables and no real objects. The goal of the ADO.Net Entity Framework is to give developers the capability to create applications that access data in an authentic OO fashion, or in other words perform an object relational mapping. Beside only doing object relational mapping, the ADO.Net Entity Framework furthermore features a mapping between conceptual/external and internal data scheme following the proposed layered architecture for achieving data independence described in chapter 2.1 ADO.Net is a further development of the object persistence framework "Object Spaces". It offers a similar functionality like Linq2SQL [Mica]. Until version 4 the automatic creation of business objects from the data base is only possible in this direction, but not the creation of a data base from business objects. ADO.Net Entity Framework communicates with the same ADO.Net data providers that ADO.Net already uses (with some minor limitations). Although currently only some relational data bases are supported, Microsoft’s vision is to work with any relational store and even non-relational ones, for example an XML file with known scheme. One very important feature of the ADO.Net Entity Framework is that is allows querying against an external data model, while the logical remains hidden to the user. Two files are used to represent the logical and external (in Ado.Net entity framework they are called conceptual and logical
models) models and a third file one contains the mapping definitions between them. The mapping possibilities however are very limited and do not allow to define mappings for all the cases we listed in the section about restructuring of data.

3.1.2.5.5 Querying Languages

Last, to complete the chapter of data independence, we need to have a look at querying languages, as they are the most common way to access data. In principle querying languages are more likely to belong to the data models and therefore to be described there: for each data model there is a certain (or more) querying language to query and update data from this model. But, as the example of LINQ [PR09] shows, unification is possible also on this level so that query languages can be used in a similar fashion or as an alternative to the unifying APIs to access data in a uniform way. Querying languages normally belong to one of three different language types, each used to perform different tasks:

- Data querying and manipulation (DML)
- Administration of the data base and the data structures (DDL)
- Rights management (DCL)

Some languages like SQL assemble these categories into one single language, while in other systems a different language is used for each task (as for example XPath, XQuery and XUpdate for XML queries). It follows a brief description of the most important querying languages:

**SQL**

SQL [Mol05] is a language for queries against relational data bases. It is originally based upon relational algebra. Its scope includes data query and update, schema creation and modification, and data access control. SQL was one of the first languages for Edgar F. Codd’s relational model in his influential 1970 paper, ”A Relational Model of Data for Large Shared Data Banks” [Cod83] and became the most widely used language for relational databases.
**OQL**

Object Query Language (OQL) [Obj98] is a query language standard for object-oriented databases modeled after SQL. OQL differs from SQL in that OQL supports object referencing within tables. Objects can be nested within objects. Furthermore not all SQL keywords are supported within OQL. Keywords that are not relevant have been removed from the syntax. OQL can perform mathematical computations within OQL statements. For defining objects in the OO data base the Object Definition Language (ODL) can be used. This language’s purpose is to define the structure of an Entity-relationship diagram.

**XQuery, XPath and XUpdate**

The aim of the XML Query [MB06] project is to provide flexible query facilities to extract data from real and virtual documents on the world wide web, providing this way the needed interaction between the web world and the database world. Ultimately, it will support collections of XML files to be accessed like databases. XQuery provides the means to extract and manipulate data from XML documents or any data source that can be viewed as XML, such as relational databases or office documents. XPath, the XML Path Language, is a query language for selecting nodes from an XML document. In addition XPath may be used to compute values (e.g. strings, numbers, or Boolean values) from the content of an XML document. XUpdate is a lightweight XML query language for modifying XML data.

**LINQ**

LINQ [PR09] is a unifying querying language applicable for objects, relational databases, XML documents and more. Currently, it is only available for the .Net platform. It allows for data processing for all possible data types and models in a uniform and symmetric manner. LINQ expressions are strongly typed, in contrast to SQL weak type system. Until now, LINQ extension-APIs are provided for ADO.Net DataSet objects (Line2ADO.Net), relational databases & ADO.NetADO.Net Entity Framework (Lindq2SQL) and XML data sources (Linq2XSD) This way, LINQ queries can be used to query against a variety of different data models with the advantage of only having to learn the syntax of one single querying language. Its usage for data independence as defined here, meaning not having to adapt queries when changing data models or restructuring data is however only limited,
as it does not contain a possibility to define data mappings etc neither, suffering from the same limitation as most competing technologies here presented.

For completeness we have to note that besides the mentioned querying languages here, many more querying languages exist. Some examples are QLP/1100 [Spe80] from Sperry Rand for network data model, the Java Persistence Query Language (JPQL) [WBG10] as a platform-independent object-oriented query language defined as part of the Java Persistence API specification or JPQL, which is used to make queries against entities stored in a relational database. Latter one is heavily inspired by SQL, and its queries resemble SQL queries in syntax, but operate against JPA entity objects rather than directly with database tables. LINQ however is the only languages so far which transcends the limitation to be bound to a certain data model.

3.1.2.6 Summary: Unsolved problems

As seen in this section, the problem of data independence is to write programs or reusable components and services that do not rely on a specific data source. While this is easy for components where all necessary data is passed by parameters, it becomes a difficult problem when the component requires doing data queries by itself; and becomes even more complicated when theses queries need to be allowed having a dynamic nature. We saw three main aspects that need to be taken into consideration: different data models, different query dialects and different data organization. While for all of these points separately technologies exist to overcome differences, it is hard bringing it all together to, for example, be able to write a component that way that passing the component mapping and transformation rules would be enough to make it know how to query and transform the result into the required form. Maybe future data access frameworks will solve this better. There are some easy to implement technologies like DAO that do not suffer from any restriction regarding data models or possible data restructuration. But it is first very costly to implement changes (in the case of any change data access is isolated but all adaptations of the queries have to be made by hand); and secondly, they cannot handle dynamic queries. The usage and definition of external views allows for dynamic queries, but do not allow for data model changes and
often not even to change to a data base of a different vendor. OR/M’s are widespread used in praxis and mostly include the possibility to change to a data base of a different vendor, but it also does not prepare the program for a change of the data model. The main purpose however of OR/M’s is not data independence. Moreover, as stated before, for full data independence it is for various reasons not possible to only combine some of the technologies presented above. Ayende Rahien, one of the developers of NHibernate, for example argues in an article in his blog entitled ”The false myth of encapsulating data access in the DAL” \[Aye\] strongly against using an OR/M together with an encapsulated data access layer. In fact to allow for complete data independence one single seamless technology would be helpful that supports at least two mapping steps: one for logical data independence taking care of data structure transformation and restructuration problems and second for physical data independence for differences in the data modes deriving from different DBMS types, finding a solution for impedance mismatch problems. After that the question of which querying language should be used to access all data still needs to be answered.

3.1.2.7 Data independence in SOA

Conceptually, data independence is a very important issue in SOA. The technologies helping here however are still immature, allowing only for limited data independence. Several of the technologies previously described can be usefully transposed to SOA, but as seen, they neither entirely provide data independence yet. Currently, the SOA world is split a little regarding to what programming language is used to implement SOA services and therefore also to implement data access. For the Java world, Service component architecture (SCA) \[MR09\] and SDO should facilitate the implementation of SOA services. The SCA specification heavily recommends the usage of SDOs for any data access operations. For the Microsoft world, the Microsoft answer to SCA, Windows Communication Foundation (WCF) \[Pat11\] and Microsoft Data Access Components are widely used to implement SOA services and to access data, first of all one of the recent developments, the ADO.Net Entity framework.
SDO is conceptually similar to the DAO pattern, but it also allows dynamic queries. It lacks however mapping possibilities to compensate for problems derived from restructuring. ADO.Net for the Microsoft world also shares many properties with SDO like the disconnected mode, good XML support, etc. The mayor difference however is that ADO.Net only supports relational data bases. The ADO.Net entity framework as last evolution step of ADO.Net still suffers from this limitation, although the specification mentions a possible support for other data models in the future. ADO.Net entity framework, in contrast to SDO, does contain a data structure mapping, but the mapping possibilities are very limited and cannot nearly handle all the cases described here in the subsection of restructuring of data. Furthermore, as XML plays an outstanding role in SOA, in praxis also XML transforming languages like XSLT [Wil09] are used to transform data that is structured differently to be compatible with the required format. Both, SDO and Ado.Net Entity framework also strongly support XML.

In SOA also the enterprise service bus (ESB - see later chapters) often has to take over a great part of the work of transforming data and messages into the required form. However, a single seamless technology to solve the data independence problem in SOA, as in software development in general, does not exist yet.

### 3.2 Separation Approaches

Separation of concerns (SoC) is one of the most essential and important principles at the core of software engineering. The term was probably first coined by Edsger W. Dijkstra in his 1974 paper "On the role of scientific thought" [Dij82]. Its primer goal is to achieve order within a system, similar to the "divide & conquer" problem solving strategy. Through a proper separation of concerns, complexity becomes manageable. All programming paradigms aid developers in the process of improving separation of concerns. The first fundamental SoC discipline is the process of separating a computer program into distinct features that overlap as little as possible in functionality, which leads to a vertical modularization of the program, that is, the division into procedures, functions, modules, objects, components, processes, threads and services.
Separation Approaches

Beside this vertical separation there is also a horizontal separation of program parts, which leads to a layered architecture. The horizontal separation does not exclude but is complementary to vertical separation. Each vertical component can internally be horizontal layered or the other way round. The layered design, as crucial ingredient in SOA, is explained more detailed in the next section.

Furthermore, more recent approaches such as adaptive programming [Kar96], aspect-oriented programming [CCHW05], composition filters [ABV92], role-modeling [BK96] or subject-oriented programming [HO93] have enhanced programming methodologies by providing separation of concerns along additional dimensions. The idea behind aspect oriented programming for example is to separation an applications cross-cutting concerns from its core concerns. An example for a cross-cutting concern is logging, which is a feature needed by nearly any software component. The problem is that mixing the cross-cutting concerns with core concerns adds unnecessary complexity to the applications code. By separating these concerns, both core concerns and cross-cutting concerns are made much easier to manage. One approach also tries to generalize all of these separation approaches into hyperspaces - a multi-dimensional separation of concerns [HP00].

Another important aspect of SoC is information hiding. The separation of interface and implementation(s) as well as the separation of the components and their composition are both reducing complexity and augmenting understandability, by hiding implementation details that are not necessary for understanding the architecture of a program or to use a reusable software artifact.

Beside modularization and information hiding SoC has even more aspects. For example the demand at object orientation modeling that objects should be modeled as close as possible to real existing objects in the end also derives from SoC, as it helps to avoid mixing of concerns. This way, negative consequences of a bad SoC in all its flavors very often do not become apparent for long time after an application have been released. Problems with scalability and limited reuse possibilities are frequently directly due to bad design choices not obeying SoC demands. Achieving good SoC on the other hand is often an iterative process. The primary behavior of a system is generally conceived during a design phase, but the
specific implementation of a system design often requires several iterations of refactoring as fine-grained concerns become more apparent [der08].

In SOA we can find SoC first of all at the vertical separation of components into services, and thanks to platform independence technologies treated in the last chapter even components residing on other platform can be included into the architecture and easily be invoked. Next, a horizontal separation into layers also is mandatory. This is what will be described in the next section. Further as loose coupling, flexible composition and discoverability of services, which all rely on contracts, are important SOA principles, the separation of service implementation and service description is the next SoC technique, that is used by SOA. Last, the separation of components and composition together with novel ways to perform that composition, for example by workflow management systems is maybe the most important innovation that SOA adds to software engineering.

3.2.1 Layered Architecture

Layered architecture, also called multitier architecture or n-tier architecture (the terms layer und tier are used here interchangeable) is a SoC technique that helps to reduce complexity. Modules or program parts are ordered into horizontal layers, where communication between the layers is allowed only according to strict rules, from up to down. With this a dependency direction is established with the consequence of a lower coupling of the components and at the same time an enhanced cohesion of the layers. The establishment of a dependency direction also helps to avoid cycles in the dependency graph. This brings advantages as well for the comprehensibility of the system as for maintainability and a better scalability. Data transfer between layers is part of the architecture. It can differ regarding whether communication across various layers is allowed or not (this last option is called ”strict layering”). The technologies used for inter-layer communication may include CORBA [HV99], Java RMI [Gro01], .Net Remoting [Tro05], Windows Communication Foundation [Pat11], sockets, UDP, SNMP, web services or others, or also some kind of middleware can be used to connect the separate layers. But the layered architecture is also very frequently realized within one single system, i.e. the layers do not necessarily have to run on separate physical
servers. Most systems that use a layered architecture follow the two- or three-tiered architecture described below. But also other solutions make use of layered architectures. The seventh-tiered Open Systems Interconnection (OSI) model [DZ95] is an example for that, or the organization of operating systems into a hierarchy of layers, which is one of the three most important architecture models of operating systems, beside the monolithic kernel and microkernel approaches.

3.2.1.1 Traditional layered architecture

The simplest layered architecture is the two tier architecture, which consists of only two layers, one upper and one lower layer. As, like in every layered architecture, only the upper layer is allowed to access the lower layer, it is also called a client-server-architecture. This kind of architecture is very common in distributed systems, and even most simple internet services architectures (e.g. browser client and web server; ftp, etc.) follow this model. Three tier architectures are also very common and usually consist of presentation layer, business layer and data layer. The presentation layer, also called client tier or front-end), is responsible for displaying data. It receives data from the lower layers and presents it to the user. The business layer, also called application layer, middle tier or enterprise tier is the place where the execution of business logic, the applications intrinsic functionality and processing is realized. The data layer, also called resource access layer, data-server tier or data back end normally consists of a database server, where information is stored and retrieved (but also normal files can be used to store data) and often a data access layer. Putting data in its own layer improves scalability and performance. The data access layer, if used, encapsulates data access. This can be seen as part of the data layer or as an additional (sub)-layer. Object-relational mapping, the DAO pattern and so on (see data independence chapter) are all placed in the data access layer.\(^3\)

\(^3\)The three tier architecture must not be confounded with the model-view-controller pattern, what often happens by associate View with Presentation layer, controller with business layer and model with data layer. But in MVC there is no horizontal layering and the elements are allowed to communicate with each other in a very different way. In fact, MVC is a design pattern for implementation within the presentation tier.
Separation Approaches

While layered architecture in traditional programming mostly work with two or three layers also a higher number of layers are possible. The three layers of the three-layered architecture for example each can be further sub-divided into additional layers, or other example are that from the introduction of this section OSI model or hierarchical operating system organization. In this case we generally talk about a multitier or n-tier architecture.

### 3.2.1.2 Layered architecture in SOA

Layered architecture of SOA is often seen as the natural next step in the evolution of business automation. The same way in which mainframe systems were succeeded by client-server applications, and client-server environments then evolved into distributed solutions based on web technology, SOA is succeeding traditional distributed architecture. The importance of layering in SOA is made very clear by this quote by Earl [Erl06]:

“Service orientation presents an ideal vision of a world in which resources are cleanly partitioned and consistently represented. When applied to IT architecture, service-orientation establishes a universal model in which automation logic and even business logic conform to this vision. This model applies equally to a task, a solution, an enterprise, a community, and beyond. By adhering to this vision, past technical and philosophical disparities are blanketed by layers of abstraction that introduce a globally accepted standard for representing logic and information. This level of standardization offers an enormous benefit potential for organizations, as many of the traditional challenges faced by ever-changing IT environments can be directly addressed through the application of these standardized layers”

While the importance of layered architecture in SOA is undoubted, the designs for a concrete SOA stack vary significantly. Each SOA implementation for each project or company designs its own SOA stack according to the conditions found in the company/project. In the following we present some examples of SOA stacks of different vendors (Oracle, IBM, SAP, see also [Lie06]):
Separation Approaches

Figure 3.6: The Oracle SOA stack

Figure 3.7: The IBM SOA stack
Other examples for designs and proposals for SOA stacks are [AZE+07] where a 9 layered stack is proposed or [Til05], which describes the a collaboration work of 21 companies who joined to design an exemplary SOA blueprint for a imaginary company named GeneriCo, which again differs from the SOA stacks listed here.

An attempt to find the “lowest common denominator” of SOA stacks may be the following taken from [Lie07]. According to Liebhart the basic SOA stack consists of the following 6 layers, where as the rest of the here shown stacks may be considered as variations of the basic SOA stack and especially with the vertical differentiation of each layer much more fine-grained.
Basically, the SOA model and all SOA stacks extends the three tiered architecture by some more layers, although as shown the concrete number of layers as well as the possibilities of layers configuration is not prescribed. The layers also use to be much more differentiated than the very general layers in three-tiered architecture. Frequently for example a SOA stack includes the abstracted infrastructure as layer on the base (see figure), as an abstraction of the underlying hardware and its components.

Very widespread in SOA stacks also is the existence of an integrations layer - the so called enterprise service bus (ESB) [Cha04]. The ESB is responsible for a number of things: It has to manage the services regarding concerns of safety, policies, privileges and provide possibilities for monitoring, logging and debugging. But most important is the integration aspect: the ESB replaces the complicated web of direct physically coupling by a communication infrastructure that is shared and used by all service providers and service clients. The core of an ESB is its communication bus, where messages are exchanged. Services connect the interfaces by endpoints with the bus. Service clients communicate with the service providers only by exchanging messages through the bus. The technological properties of service providers and service clients usually are very heterogeneous. Neither soft- and hardware platforms, nor communication protocols nor data formats- or structures are
Separation Approaches

usually directly compatible. That is the ESB needs to perform some transformations of data from one format into another and into other models to bridge differences between the service provider and the service client. Another important part of the ESB is the routing service. The routing service accepts messages from the ESB and redirects it according to predefined rules. To do that, different routing strategies are common, like itinerary-based routing, where the message is forwarded to a sequence of predefined paths and destinations; or Content-Based Routing, where the routing decision is made based on the content of the message.

We see that the ESB takes over a lot of different functionalities in many SOA systems. It is often the "technological backbone" of the whole SOA implementation. Maybe it is even a little bit overloaded of different functions. In some implementations the ESB for example is also responsible for services orchestration, where it is reasonable to ask, if this really should be a task for the ESB (see also [SS]). We see furthermore, that the most important integration aspect contains much of the concerns what in this thesis is called the "data independence problem", of translating messages and transforming data, which due to a lack of a standard solution or technology for this issue needs to be solved somehow. There are good reasons to warn, trying not to depend too much on the ESB. To say it with the words of Richard Turner, a program manager from Microsoft who discusses in his blog with David Chappell about his book about ESBs: "Each ESB is implemented in an entirely proprietary manner, with no guarantees that the messages transmitted across the bus are actually based on any form of open standard protocol, there is absolutely no guarantee that any technology offered by a company other than the ESB platform vendor will be able to communicate freely via the bus. So, not only am I held to ransom by the ESB platform vendor because I cannot easily replace one ESB with another, but I am also likely to only be able to integrate systems which the ESB vendor provides specific adaptors for. Isn’t this precisely what we’re trying to get away from?" [Tur05].

Last, the orchestration layer (also called process layer) is another layer new in SOA and is responsible for much of the enhanced flexibility that is promised by SOA solutions. The process layer in SOA is responsible to coordinate the service calls of the lower layers. In
comparison to traditional programming this is a novelty in SOA and it adds and enhanced flexibility with much better opportunities to redesign and extend a program afterwards.

3.2.2 Separation of interface and implementation

The next SoC method which at the same time is an important SOA principle is the separation of interface and implementation. It is a very powerful technique to improve understandability, increase flexibility of a program and helps to design complex systems by only concentrating on the expected functionality and not on how to implement it. First there is to clarify the term "interface", which is used denominating quite different things in different contexts. Here are some examples of interfaces:

- Graphical user interface (GUI), which can be considered a specific type of human machine interface (HMI) allow for an interaction of the user with a component through a graphical front-end. For the communication a computer mouse is mostly used but also other input devices are used. The GUI elements are commonly standardized components (widgets) and small symbols (icons).

- Command line interface (CLI): command line interfaces demand from the user textual input of commands. Command line interpreter are one example for this. At the beginning of computer history CLIs have been the only available user interface. Today command line interfaces are mostly used when a component should be called periodically without the assistance of the user, for example for triggering periodically recurring tasks.

- Text user interface (TUI): Text user interfaces are also text based but in contrast to command line interfaces don’t require typing commands. Instead, they use to work with menus that respond to keyboard input and hotkeys. An example for a program that works with a text user interface was the famous Norton Commander.

- Data interface: Allows a component to read data. This interface is used internally only by programs.
• Application programming interface (API): This interface allows programmers to use a component's offered functionality and services by making function calls and passing parameters.

• Voice User Interface (VUI): With a voice user interface, the user communicates with the system by spoken words. The output is made either with voice recorded before or with a speech synthesizer. The input requires speech recognition. A common example for VUI is interactive telephone systems.

• Natural user interface (NUI): Natural user interfaces like, for example, a touch screen are sensitive for contact and react to movements of fingers or the hand. The recognition of gestures tries to be close to natural movements what should allow for a very intuitive usage of interactive devices. An external artificial input device like a mouse or keyboard is no longer necessary. Instead, the superfiace of the device is touch sensitive.

All these different interface types can be put into two main categories: User Interface and programming interfaces. First one offers to the user input and output possibilities. GUI, TUI, VUI and NUI all are a type of user interface. It is important to underline that user interfaces and interface design is not limited to the visual/graphical (GUI) - it extends to any kind of interactions the user has with the system [Ras00].

The second type of interfaces are used by programmers or called by programs internally. API, CLI and data interface fall into this category. Also to this category belongs the important interface concept of object-oriented programming languages and component models. In the following both main types of interfaces are analyzed more detailed.

3.2.2.1 Programming Interfaces

Interfaces, as used by programmers and especially as concept in object-oriented programming languages and as component descriptions in component models in general have two main characteristics:
• An Interface describes the functionality of a component/system/library as a black box. The interface is the only visible part and through which all communication has to be performed. The black box principle \cite{McI68} is commonly accepted for being vital for easy reusability. Therefore a developer who wants to reuse an existing artifact should not have to cope with internal implementation details in order use an artifact properly. With the implementation hiding principle it is tried to separate information which is only relevant for the implementation from that which is necessary for the usage of the artifact. It should be enough to know only the interface of an artifact in order to use it properly. An example of interface usage only limited to this descriptive aspect are header files in C/C++. A further examples for this usage of interfaces, but without excluding the following second aspect of multiplicity are component models that use interface description languages to describe their components.

• Furthermore, the interface concept is especially useful when there are various implementations for a given interface. These implementations may implement the exactly same functionality and only differ in some quality-of-service properties (like speed, precision, etc) or implement different amount of functionality, which however is available through the same function calls. Plug-Ins is an example for the practical usage of latter ones: a program exposes a well defined public interface so that third party developers can write their own plug-ins to extend the functionality of the program. An example is VST plug-ins \cite{Ste}, a standard for audio plug-ins that can be used with nearly any sound processing software and samplers. The same is true for Adobe Photoshop plugin \cite{Ado} in the sector of image processing. The descriptive interface characteristics here also are important as they describe which methods are available to build the plug-in. There are also examples where only this second characteristic of interfaces is important while neglecting the information hiding principle. Abstract classes (in some programming languages they are called virtual) are interface descriptions that implement a part of required functions by themselves. With this, the information hiding principle is already violated, because these ”interfaces” are mixtures of descriptive elements and code. Abstract classes justify their existences because their main usage is not for information hiding purposes but to achieve poly-
morphism, i.e. it allows to work with different implementations (sub-classes) of the same interface (abstract class).

For the first interface aspect of interface description the most important question is which information the interface description contains. This is investigated more detailed in the following subsection. After that the possibilities of how to choose and manage interface implementations are analyzed, which is the most important question regarding the second aspect of interfaces of the multiplicity characteristic of interfaces.

3.2.2.1.1 Interface description: Design by contract

Based on the information hiding (black box) principle the paradigm of "design by contract" was introduced with the programming language Eiffel [Mey92]. The basic idea again is to design without having to care for implementation details. This basic idea is present (in various degrees of strictness) in several paradigms of software development, notably in object orientation, component-based software, and, of course, in service orientation. In fact, SOA implies publishing a "contract" for a service, by means of WSDL definitions, for example, that has to be totally independent from implementation. The client only has to know this definition to use the implementing service. But while the term "interface" only refers to the interface signature, the term "contract" is more general and can enrich the interface signature by more semantic information like pre-, post-conditions and invariants, which also are important information for the proper usage of an component. In fact much information possibly may be part of the contract, e.g. quality of service information (velocity, precision, etc.), or information about valid sequence calls of operations. In the following possible parts of the contract are described.

Interface signature

This contract level is about the most common format of contracts. The basic and minimal information is a formal syntactic description of the interface that can be verified automatically, usually by a compiler. In the case of functions and methods that description normally consists of the name, the parameters, return values and - in case of the existence of and mechanism for exceptions - all possible exceptions that may occur. Many technologies only
use this kind of contract description, i.e. COM, CORBA, Web Services, as well as the most common programming languages (Java, C, C++, etc.) This grants that reusable artifacts can be reused as black boxes on those technologies.

Pre/Post-Conditions and invariants
The pre-, post-conditions and invariants were notably introduced with the programming language Eiffel, introducing the term formal contracts between clients and providers of a function at the same time. Pre-conditions, on the on hand, are used to describe the required state of a program or some other conditions which have to be fulfilled to make use of the function. The post-conditions, on the other hand, specify which predicates must always be true just after the execution of a function. Invariants describe conditions that remain stable and without change throughout the whole interaction process with the application. Some recent languages have adapted this concept like for example AspectJ [CCHW05], a plug-ins for Eclipse or Spec# [BLS05], and an implementation of Design by Contract for C#.

Additional implementation information
One drawback of the previous contract model is its limited expressiveness as already demonstrated by Szyperski [SGM02] or [MW99]. One example for that is the missing capacity to indicate the existence of callbacks. Another problem is the unnoticed violation of a contract that may occur when implementation details of a base class change that has dependency relation with subclasses. In this case, some methods of the subclasses that had worked before probably fail if the contract won’t be adapted. Furthermore it is difficult to express side effects that a function may have with pre-, post-conditions and invariants, i.e. adding changing variables or member class variables to the function, changing input parameters, creating new objects within the function that outlast the working off of the function or to release some objects within the function. Thus, Büchi and Weck [MW99] state that a pure black-box specification is not sufficient and in many cases more information has to be exposed, whereas on the other hand, white-box reuse is confronted with the laborious work to analyze the whole source code as well as with potential copyright problems. For
that reason, Büchi and Weck propose the idea of gray-box reuse that exposes just some important implementation details while others remain hidden.

**Policies and metadata**

Besides the information that refers to internal implementation details, some other, non-functional specifications might be important enough to form part of the contract, e.g. execution speed, precision of mathematical calculations, or behavior in concurrent environments. These are also often referred to as Quality of service information. In SOA contracts containing this information is called policy and handled separated from interface signature information.

**Protocols**

Protocols are information about valid call sequences. Valid call sequences in parts also can be expressed with pre/post-conditions, defining as a precondition of a function call that other functions have been called previously. But protocols are the better choice to do so. The set of valid function calls at each time can for example be represented by deterministic or nondeterministic state machines or with help of Petri-nets.

**Semantic description**

The information provided so far are all technical information and processing requirements. But there is information that is some more difficult to provide like information about the behavior and response under certain conditions of an artifact or what specific tasks it is most suited for and which not. A general problem that all methods and syntactic description methods and languages share is their lower expressiveness compared to a semantic description. Because semantic description in natural language is, due to its ambiguity, very difficult to be read by a machine, formal languages have to be used for this. "Semantic Web" [BLHL01] or "Semantic Web Services" [MSZ01] try to do this. Some existing approaches with better options to express semantic are Kind Description Language (KDL) [Jos99a], Ontologies [Z01] [NR01] like the DAML-S service language (DAML-S) [BHL02], Petri Nets [Wol86], or the usage of agent languages like the Knowledge Interchange Format (KIF) [MRR92] and the Knowledge Query Manipulation Language [LF97]. In the web
service ontology language DAML-S for example the semantic description of a web service is divided into three components: the service profile, the process model and grounding. The service profile includes all non-semantic information described above, i.e. interface signature with input and output types, pre- and postconditions and so on. The Process Model describes how the service works; each service is either an AtomicProcess that is executed directly or a CompositeProcess that is a combination of other sub-processes. The Grounding contains details of how a client can access a service by specifying a communications protocol, parameters to be used in the protocol and the serialization techniques for the communication. Grounding can be understood as mapping from DAML-S to WSDL. The main contribution of DAML-S however is to establish a service description using the concepts defined in semantic web ontologies which provide expressive constructs that are suitable for the automatic discovery and composition of services.

3.2.2.1.2 Choosing interface implementations: IOP & Software traders

Although the usefulness of the separation of interface and its implementations is commonly accepted, in many of the mostly used programming languages (C++, C, Java, VisualBasic.Net) however it has little importance in practice, despite the existence and importance of the interface construct in those languages [Dav07]. The problem is that the interface construct in those languages is more used to archive polymorphism and as a substitution of multiple inheritance instead of separating both. It is even so, that due to the ”implementation bias” problem [Var04] a real and complete separation of interface and implementation is extremely difficult to realize, because each interface requires a hard-coded instantiation with a concrete implementation.

To solve this problem Varney [Var04], proposes a new language design called interface oriented programming (IOP) which changes the interface feature to be usable without explicit instantiation and therefore allowing for programming with interfaces only. This strict separation then also permits further possibilities like the automatic composition of partial implementations and the non-deterministic instantiation of an interface. But IOP requires this way completely new programming languages. To overcome the implementation bias problem within existing languages another idea is to use software traders for the instan-
Separation Approaches

tiation of interfaces. A software trader is according to [Mir97] "a third party object that enables clients to find suitable servers and services in a distributed system". Unlike similar technologies like naming or directory services the search can typically be attribute-driven and a trader can deliver more than one results as response (whereas it’s a 1:1 mapping in the case of naming service). So traders are tools to find, locate and maybe download and execute suitable components by means of interface description or more complex contracts. This way, traders that offer run-time detection of implementations could be a solution to overcome the implementation bias problem in common programming languages.

In the 90th there were a lot of research about traders, but since then, the interest for them has considerably dropped. Traders can be located at different levels in terms of the OSI Model. On a lower level, traders are quite successful: Microsoft’s Universal Plug & Play [JW03], the Service Location Protocol [Gut99] and JINI [FAB+02] are some examples.

On the application level however traders still do not play a very important role. The most important approaches are ANSAWare-Trading [ANS98], ODP-Trading [ISO98] and COS-Trading (CORBA-Trader) [OMG], whereby these three approaches are very closely related. ANSAWare is a preceding version of the ODP-Trading standard, and COS-Trading and ODP-Trading are largely identical and only differ in few details.

A real improvement of ODP-trading however is the UCOM Trader [Sch03]. This trader is able to handle components rather than objects only, whereby it is not limited to a singular component model but supports a number of different component models so that components do not have to be homogenous. It is also possible to extend it by new user-defined component models. A distinction between trading in design time and trading in runtime is made, and several different component description and search languages are allowed which can also be extended by own ones.

UDDI [Cer02], as a central registry for web services, which is frequently used in SOA projects, is commonly not considered to be a trader. But with the support of "TModels", which are like interfaces of the programming languages and the possibility for having various implementations of the same TModel, it can be used like a Trader.
3.2.2.2 User interfaces

With user interfaces we can have the same possibilities to have various implementation of the same interface as with API interfaces. An example is different skins which allow changing the appearance of a program. The different implementations offer the exactly same functionality with the only difference of a different arrangement of the graphical elements and a different appearance. But with user interfaces - in contrast to programming interfaces - there is also the further option of having different interfaces to access the same functionality. Examples for this are different devices which want to access the same program or different users with different rights and privileges, where it might be desirable to define a different interface for each user or device that can be exchanged easily. So also in this field, the clear separation of the interface and the functionality behind is clearly justified. But as with programming interfaces, in praxis the user interface is too often very tightly coupled to the underlying system. This is mainly due to the lack of a built-in mechanism for achieving this separation in most of the commonly used programming languages. There is no comparable interface construct for isolating user interfaces from its "implementations". Such mechanism would be a contract for user interfaces, defining all necessary functionality a user interface has to provide in a black box manner. To make this idea more clearly we first may have a look at the question what exactly is a user interface? As we have seen in the introduction of this chapter, a user interface can be of many different types, conducted by voice, by hotkeys or in most cases it is a graphical surface. In the latter case one can imagine a user interface very plastically - it is what a programmer can build with help of the toolkits that most modern programming environments provide, by just dragging and dropping the elements (buttons, checkboxes, menu-stripes, etc) like the Visual Basic Toolbox - which invented this kind of components - onto an empty panel. In fact, this way of building GUIs with pre-fabricated components can be considered one of most successful and best working component models. This work of designing a GUI can be done completely without writing a single line of code (in fact there is code created behind the scene, but completely automated). The important question is how to connect such a GUI to the rest of the program, which implements functionality. The answer to this is events. There is a possible generalization of all user interfaces with help of events. All user interfaces can
be reduced to produce events with well defined parameters and the system behind has to react to them. So, to completely separate a user interface from the functional part behind it is required to define all events a user interface can produce, and, on the other side a definition of all events a system can understand and process. The connection of both then works by linking the event listeners of the functional side of the program with the events a user interface can produce. A language with these capabilities for a platform independent description of user interfaces however does not exist yet.

So, the interface description or contract for user interfaces in the main has to be a description of its events. The "implementation" on the other side is the concrete arrangement of GUI elements like buttons, textboxes, their colors and further properties, or also the spoken commands a system can understand (which after that is translated to an event). As stated before, user interfaces (GUI interfaces) currently are mostly made with toolboxes and no standardized way for a platform independent user interface description exists. There are however approaches like windows presentation foundation (WPF [Mac10]) (for desktop applications) and Silverlight [Mac12] (for web applications) from Microsoft that bet on a descriptive textual, markup-like description of the GUI, with the advantage that the same user interface description can be used both in desktop applications and web applications due to the fact that the markup language for both WPF and Silverlight is quite identically. This might be a first step towards the independence of the user interface.

At this point there is to underline the importance of user interface for every piece of software and therefore treat is as an integral part of the system design. According to [Ras00] the user interface however is often included too late into the design cycle, not gaining the attention it deserves. We should remember that "As far as the customer is concerned, the interface is the product" (Raskin). Separation technologies are required not to tangle it with the rest of the code and where it is much more difficult to change and optimize it, which normally is an incremental process.
3.2.2.3 Summary

To summarize, the separation of a components description (= contract or interface) of its implementation(s) is very useful as it allows for exchanging implementations, it greatly enhances understandability, and as machine readable contracts allow for automatically discovery of suitable compatible components. Despite the existence of the interface construct in many of the most used programming languages, a clear separation of both cannot be established using this construct, due to the “implementation bias problem” [Var04]. Interfaces are also used in many component models (including SOA), with a variety of possibilities of contract descriptions (e.g. interface description languages), which become increasingly important, together with technologies for discover implementations and technologies to select und switch them.

User interfaces on the other hand currently are often designed to allow things like changing the skin, but rarely can exist independently from the rest of the program nor is there a way to build and define them platform independently, like it would be required to fit perfectly into the SOA paradigm, allowing to treat them just like other services. They use to be designed with platform dependent toolkits that are lacking platform independent description technology like the interface description languages for other components. The development of technologies to support this already begun, but is still immature.

3.2.2.4 Separation of interface an implementation(s) in SOA

Similar to Eiffel’s Design by Contract, in SOA the separation of interface and implementation is conceptually very important. "Interface stable, implementation flexible" [Wij04] or "interface first" (or "WSDL first" in the case of SOA) [Erl06] are common SOA principles and are hold for good design. Components in SOA (services) come with contracts, use policies and rich metadata description. Technologies like UDDI are public repositories to look for suitable services according to service descriptions.

Apart from that there is the problem, which until now only received few attention is by the scientific community, of how to properly integrate the (platform dependent!) user interface
Separation Approaches

into SOA. Executable (and automated) business processes are not suitable to control highly interacting user interfaces [Lie06]. The most common used web service wiring language WS-BPEL does not explicitly support user interaction.

3.2.3 Separation of components and composition

With the separation of components and composition the stable parts of a system (components) are separated from the specification of their composition. As seen in previous varieties of SoC, this intends to increase understandability, maintainability and flexibility of software. Understandability benefits first from the fact that implementation code is not mixed up with wiring code and secondly especial from the possibility of new composition mechanisms, e.g. graphical programming. Flexibility improves most by the opening of the possibility of workflow definitions that directly have effect on the programs behavior and intends to hopefully help solving the so-called ”Business-IT-divide” [BLJM08], which always has been one of the main drives behind SOA. The term ”Business-IT divide” is used to denominate the fact that changes in business processes of a company frequently need a subsequent change of the company’s applications, which is slowly, error-prone and expensive to realize. The solution to overcome this gap would be a system that allows for modeling business processes that later can be just passed to a business process management system to execute, without any further programming needed. Workflow modeling (that should be possible without programming skills) could be done for example with graphical programming environment that does not require any typing of code. At this place, it has to be made a distinction between the general architecture of a program, and the part of the program logic that is modeled by workflows. Although it is obvious that both are closely related [Gan07] this is only a part of the entire architecture of a software system. The whole program provides a framework that constitutes the overall architecture of a program, and within that framework workflow modeling takes place to model different call sequences. The framework is responsible for calling the workflow management engine and loading and executing workflows and for being the mediator between the user and the workflow system.
Historically, the way of composition and the separation of components and composition is closely related to the development of programming paradigms that determine which are the components that are subject to composition. In most traditional imperative languages the overall view of a program can be described as following [JO]:

\[
\text{Application} = \text{Algorithms} + \text{Data}
\]

The reusable parts in these systems are functions that implement algorithms that are used with different data. Algorithms are reusable but not components in the sense that they would be very suitable to build larger units. The composition, meaning all connections between function calls, data passing etc is hard coded. With object oriented programming (OOP) [CN91] the programming paradigm changed to be [JO]:

\[
\text{Application} = \text{Objects} + \text{Messages}
\]

The new approach was that objects are holding data together with the functions (containing for example algorithms) that can be applied to these data. Objects furthermore respond to messages (a function call can also be modeled as message receipt like it is done for example in Smalltalk [GR83]) and emit messages (events) by it selves. The reusable parts in this paradigm are objects. Objects are much more suitable for composition. Objects can have other objects as members and inherit data and methods from other objects. Static composition of objects (to be accurate static composition is rather done with the blueprint of objects, classes) is leading to complex inheritance trees. Dynamic composition with objects can be described using an "Object2Object" [Gan07] composition model, which describes all method calls and messages sent. But also in this approach the component implementation code and component composition code is mixed up and hard to read and separate. Furthermore objects increasingly revealed to be too fine-grained for good reusability in-the-large, especially as they remain platform dependent. Therefore, with Component Based Software Engineering (CBSE) [App01] the paradigm again changed, and an application now is viewed as [JO]:

\[
\text{Application} = \text{Components} + \text{Scripts}
\]
With this paradigm, the idea of separation of both really begins. In CBSE a program consists in the main of coarse-grained, and often with less platform dependencies charged, computational elements (components - the reusable part) and code for wiring these together - scripts. Internally they may be kept on being implemented by OOP, there is no need to dismiss this paradigm. The composition mechanism, especially in the large-scale, however changes. Instead of an Object2Object composition model, the "circuit model" is introduced: The circuit modes means a view of component analogous to electric circuit layout with components represented by "chips" with pins for input and output. These "chips", with a graphically representation by icons, can be connected together like electronic devices to interact with each other in response to external controls [Gan07]. While it is often argued that this composition-in-space model is not appropriate for building large systems because a two-dimensional graph of a "real" application would be too hard to read, there mostly is a mechanism that allows to wrap up a network of components and encapsulate it in a new component, what this way allows also systems of great complexity to be built from two-dimensional diagrams [Gan07].

Trying to separate components and composition, it also turned out that the requirements of a composition language significantly differ from the requirements of an implementation language [NM95]. On the one hand it can be simpler than implementation languages as many constructs of implementation languages are not so important for wiring; on the other hand some additional constructs are helpful. In this way it is suggesting using different and specialized languages for implementation and composition. Although in general every all-purpose language can be used for the composition of components, it might not be the wisest choice. These languages have the disadvantage of tending to make architecture more obscure, as the wiring-code (the part of the code that connects different components) is hidden and tangled within the rest of the necessary code, including implementation code of components. Better for a clear exposition of architecture and the separation of wiring and implementation code is to have a specific composition language. Apart from wiring constructs, such language should also support the possibility to adapt parameters before passing it to another component; while in an ideal component world components simply can be plugged together, in reality a programmer is often forced to work with legacy or third-
part components that need some adaption. In this case glue code is necessary in order to overcome compositional mismatches and to make components compatible which otherwise cannot be plugged together. According to Schneider & Nierstrasz [JO] an ideal composition language for CBSE would allow to express applications as compositions in terms of components, scripts and glue. Existing general purpose languages however do not comply with this demand. As second best choice Schneider claims that some simpler and dynamic typed scripting languages and 4GL languages might be used for wiring components together. Scripting languages [Joh98] are mainly designed for gluing applications that means to make it easy to quickly build small, flexible applications from a set of existing components. They operate on a higher, more abstract level than all purpose system-programming languages and with these higher-level design elements, they are ideal tools for expressing applications as compositions of software components. Scripting languages mostly use a weaker typing methodology and sacrifice execution efficiency to improve speed of development. They typically only support a single, specific architectural style of composing components (e.g. the pipe and filter architectural style supported by the Bourne-Shell [Bou78]) and they are designed with a specific application domain in mind (system administration, GUIs etc). Some examples for scripting languages are TCL [OJ09], Python [Ros95] or Darwin [MDK93].

The next evolution step of programming paradigms was SOA, which promised among other things a better possibility for workflow modeling. While all scripting languages are designed with a specific application domain in mind like system administration, GUIs etc, the application domain really needed for SOA is the possibility for workflow modeling. And that’s what web service wiring languages are made for [WVP06]. The above equation for CBSE can be adapted to SOA, experiencing another change and looking like that:

\[
\text{Application} = \text{Services} + \text{Service Orchestration}
\]

SOA can perfectly be seen as component model, so again there is no need to dismiss the previous programming paradigm. It is a component model that helps overcoming the variety of competing component models with its own standards by offering one relying only on common web standards like HTTP and XML that can be understood by any system and pass through any firewall. Services can be of all granulation, coarse-grained as
components or fine-grained as objects but always are platform independent. Furthermore they are stateless like the functions and procedures of traditional approaches, what makes them more autonomous and self contained than objects. State management of services is delegated to the client or the orchestration engine with the result that services themselves remain without state, but often can handle parameters with state information, which have to be passed again each time a service is called. Also like objects they support messages and complex message exchange patterns.

Web service orchestration now opens the possibility of workflow modeling with web services wiring languages. One approach to categorize existing web service wiring languages is the following schema [BG06]:

![WebService wiring languages classification](image)

In this classification we can match most of the existing web service wiring languages like BPMN [Whi04], XPDL [Cco05], XLANG [Tha01], WSFL[Ley01], YAWL [WA03], SCUFL [OLK+07], WS-BPEL [ACD+03] fall into the orchestration category. Static composition with choreography on the other side is still immature and experimental. WS-CDL [K004] probably is its most important represent, but also other languages like WSCI [W3C02] offer a basic choreography support. The same is true for dynamic composition, where the currently most important approaches like semantic web [BLHL01] and semantic web services [MSZ01] are still in development. The difference between static and dynamic composition are an a-priori composition at design time on the one hand or the location of compatible services at runtime on the other hand. The difference between orchestration and choreography is that orchestration assumes a central coordinator which is responsible for invoking and
combining the single sub-activities, whereas choreography is designed to work without a central coordinator. It defines complex tasks via the definition of the conversation that should be undertaken and understood by each participant.

The completeness or functional range of several web service wiring languages is compared in [PWMA03] according to its support for 20 generic workflow patterns that were defined in [vdATHKB03], as well as 6 communication or message exchange patterns found in [RBM01]. Of all orchestration languages WS-BPEL (also known as BPEL or BPEL4WS) is most important and has developed to be a de facto standard for most workflow applications. Due to this importance in the following we present a short overview.

### 3.2.3.1 WS-BPEL

WS-BPEL [ACD+03] is built on IBM’s WSFL (Web Services Flow Language) and Microsoft’s XLANG (Web Services for Business Process Design). It combines the features of a block structured process language (XLANG) with those of a graph-based process language (WSFL). It is designed from the ground up to work with web services and has developed to be the de facto composing standard for web services. Moreover, to be homogeneous and composable, each WS-BPEL workflow is a web service as well. WS-BPEL has two abstractions of processes: executable and abstract processes. An abstract process is a blueprint of a workflow, specifying the message exchange behavior between different parties without specifying any internal behaviors. The intention is to allow the definition of publicly visible behaviors of a WF, hiding details that may differ between implementations of a blueprint. This is like an interface or a contract in programming languages, but for workflows. An executable process specifies the execution order between a number of constituent activities, the partners involved, the messages exchanged between these partners, and the fault and exception handling mechanisms. They are abstract workflows with all missing details filled in.

WS-BPEL processes define how multiple services interact to achieve a business goal, as well as the state and the logic necessary for this coordination. The interaction is performed through web service interfaces that are encapsulated in partner links. WS-BPEL process
Separation Approaches

definitions are kind of flow-charts. Each element in the process is called an activity. An activity is either primitive or structured. There are several primitive activity types supported, like web service invocation, waiting for a message, assign data values, throwing exceptions or terminate the entire service instance.

WS-BPEL support structured activities for defining an execution order, including conditions, loops, parallel routing, grouping activities and fault handling. Structured activities can also be nested. The execution order can further be controlled through (control) links, which allow the definition of dependencies between two activities: the target activity may only start when the source activity has ended. Activities can be connected through links to form directed acyclic graphs. Besides throwing exceptions it is also possible to define how individual or composite activities within a process are to be compensated in cases of errors. Also support for XPath, ACID properties and parallel execution (using the `<flow>` attribute) should be mentioned.

But there are some limitations of WS-BPEL that are especially relevant for its usage in scientific fields. The most notable limitations are:

- The lack of parallel loop, what is particularly needed in scientific workflows [Slo07].
- If the number of iterations is constant. The `<flow>` attribute does not work if the number of iterations depends on a input to a workflow [Slo07]. Scientific workflows however are often data driven and the number of iterations depends on data input.
- WS-BPEL has a poor support for running a large number of sub-workflows [Slo07], what is very typical for scientific workflows - WS-BPEL is a very complex language which furthermore lacks orthogonality, i.e. many constructs have overlapping functionality.

### 3.2.3.2 Graphical composition with Workflow Management Systems

A big disadvantage of textual workflow languages is its complexity. According to [Sim05] textual formats work well for linear tasks, but not for tasks with lots of branching. It is difficult to get an "overview" of the task as well as to express dependencies within task. In
fact most the textual workflow languages are not easier to understand or create than code of any traditional all-purpose language and its usage is only possible after a steep learning-curve and by professional programmers, but not by researches or people who define business processes in a company, which are usually not programmers [LA08]. Though, the claim of exposing architecture easily or solving the "business-IT divide" problem with this type of language cannot be satisfactorily fulfilled.

A better approach toward that goal are workflow management systems that often are graphical extensions of workflow languages that for example produce WS-BPEL code in the background. In fact today more than hundred business workflow management systems exist, like Fujitsu’s i-Flow [Fuj], SAP’s WebFlow Engine [GRD+09] or IBM Lotus Workflows [IBM05], just to point out some examples. Windows Workflow Foundation (WWF) [Buk08] furthermore is an example for a graphical generic all-purpose workflow system, not limited to the business sector, following the idea of integrating workflows as native part of an application.

### 3.2.3.3 Summary

The separation of components and composition is an idea that was born with CBSE at the evolution of programming paradigms. There are textual component wiring languages like scripting languages or also some service wiring languages but also graphical composition solutions, like WFMSs, which however are only for defining workflows and not the overall architecture of a program. Graphical composition solutions have the big advantage to be much more intuitive and understandable even for persons with no programming skills.

### 3.2.3.4 Separation of components and composition in SOA

The concept of separation of components (in SOA services) and their composition is one of the core concepts in SOA; an orchestration layer is an important part of the SOA stack and is responsible of many of the promised advantages of SOA. At the moment mostly the textual wiring language WS-BPEL is used to orchestrate services. But as WS-BPEL is not a very easy and intuitive language it may be that this is one of the main reasons, why
SOA has to reputation to be difficult to implement, because the composition mechanisms is not very intuitive and easy to perform. To make composition easier, some workflow management systems (that may produce WS-BPEL code) can be used instead. In next section we have a closer look at a special kind of workflow management systems, developed for scientific purposes and specialized for scientific tasks.
4 Scientific Workflow Management Systems

This section deepens the topic of workflow systems and goes on having a closer look at a special variety of those: scientific workflow systems. First it is investigated what differences are between business workflow systems and scientific workflow systems, which are the result from some fundamental different requirements of both domains. Then some existing scientific workflow systems are presented and examined in how far they comply with these requirements. The last part works out what would be needed for future scientific workflow systems and points out the poor SOA capabilities of existing scientific workflow system. This is done in preparation for a later chapter of this thesis, where an own scientific workflow system with better SOA capabilities is being developed.

4.1 Business Workflows vs. Scientific Workflows

First there is to note that business workflows and scientific workflows are not necessarily different in any case. Both share many common characteristics and it depends much on the concrete case. There are examples for business processes that also are very data intensive and highly parallel, which are typical for scientific processes. And there also are scientific processes that have many characteristics which are typical for business processes. Nevertheless the distinction between both makes sense, because the requirements in business and research are different and there are very clear tendencies that justify a general distinction. Also, it is shown that even if a very typical scientific process can also be expressed by
means of business process constructs this easily can lead to unnecessarily complex workflows that are hard to understand, modify, maintain and schedule, what could be avoided using a workflow language made for scientific workflow that offers special constructs for that purposes. In the following the main differences of both workflow types are worked out.

**Difference 1: Dataflow modeling and provenance information**

The center of a business workflow is the fulfillment of a business goal. Business goals obey a set of business rules, may run over a long period of time and involve many different people among various companies or within the hierarchical context of a single company. In contrast, the core of scientific workflows are not rule-driven business goals and business processes, but dataflow modeling. For example, a typical scientific process want to assemble several data retrieval sub processes, filtering processes, computation components and visualization components into a single executable analysis pipeline, which in the main is data flow modeling. Typical questions a scientist asks are: "What are the inputs used to create the final product?" "Were two data products derived from the same raw data?" In short: "scientific workflows are highly data oriented" [WEW+07].

An important concern for scientific workflows therefore is how to address the heterogeneities and inconsistencies that arise when information comes from different sources and communities, a problem that is treated in the data independence section of this thesis. Data flow modeling also generally doesn’t impose any precise execution order of the sub-tasks. In dataflow modeling a node is executed if all input data are available, where different sub tasks also can be executed concurrently. Nevertheless - although control flow modeling this way has less importance and some scientific WFMS even work completely without them, most scientific workflow languages and systems mix data flow and control flow approaches and also offer typical control structures to express conditions etc. Scientists typically also want to run the same process multiple times with different set of data and computing units over a period of time. They furthermore need functions like pause, revise and resume workflows, which are often not foreseen in business WF systems. Next, for scientific workflow systems it’s crucial to record provenance information with the execution of workflows. This
is very important to guarantee for the reproducibility of the results. Scientists need to know not only results but also which input data, filter steps and so on were performed to come to the final result. The provenance records here can be highly fragmented and include many different sources including emails, database queries, journal references, compiler options and other configuration settings, and others [YGN09] [YEM+07] [Shi07] [BG07]

**Difference 2: Large amount of data, number of people involved and flexibility**

In contrast to business processes, very few people are involved in scientific workflows. In fact, most of scientific processes involve only one scientist but several different complex computing units and complex data. Instead of many participants large amounts of data is examined and computed by often thousand of service instances. This is what probably is the most notable difference of both WF types - a difference in scale due to the sheer amount of data that often has to be evaluated in scientific research. To manage this amount of data considerations for stream-based data transfer and concurrent execution of subtasks are required for scientific workflow languages.

Another difference is that scientists - as only experts in their domain - usually have to construct their WFs by themselves, whereas business workflows are often created by experts of IT. Thus, an easy user interface is even more important than in business sector (see page 75 about the business-IT divide). In fact two types of analysis is common in science: Routine analysis based on known common cases, which should be easy to set up and execute just as business workflows and second unique experimental analysis, which is most typical and very important for research and doesn’t occur in business sector very often: As research is exploratory in nature a much more flexible approach of creating and modifying workflows is needed than for business workflows. Research often uses a trial-and-error method when modifying steps. A scientific workflow represents an experiment that is likely to be run only a limited number of times before new ideas and insights need to be incorporated to create novel workflows with previously unseen combinations and configurations of models. Therefore, frequent changes and redeployment need to be supported and made simple.
Finally, as workflows with thousands of service instances are becoming over complex very soon, also an intelligent assistance for such experimental workflow creation is demanded. Mechanisms to hide unnecessary complexities and automation of low-level operational aspects of workflows are key requirements. [YGN09] [BG07]

**Difference 3 Support for scientific workflow patterns**

For business processes security of transactions and persistency are a very important issue. Business workflows typically attempt to express partial order (i.e. control flow) of process tasks. Therefore business workflows languages have to provide explicit control-flow primitives (sequence, branching and choice) for the fine-grained modeling of control-flow. However, they often only have a limited support for repetitive, synchronous and concurrent tasks. In [PWMA03] 20 workflow patterns and 6 communication patterns for common workflow languages are presented. For scientific workflows additional patterns become important. [YGN09] tries to elaborate a set of scientific workflows patterns that are similar important for scientific workflows as business workflow patterns are for business workflows. These scientific workflow patterns include different multiple instance patterns, iterative structures like arbitrary cycles, structured loops and recursion.

To summarize the differences between both workflow types, on the one hand there are business workflows that implement business processes, work with business objects that are either task-centric (e.g. payment) or entity centric (e.g. invoice), rely and are conducted by business rules, which possibly are evaluated by an additional business rule evaluation engine (e.g. in WWF). The main focus of business workflows lays on the right order of process tasks, guarantee of ACID properties of a transaction as well as long running processes are very important [TDGS07]. Business workflows are typically little dynamic and evolving in nature. A business workflow captures a set of activities and their relationships in order to describe a business process. The overall aim is to be able to automate these processes and execute it repeatedly over possibly long periods of time. This way in business workflows business people are concerned about a particular order of sub tasks to be kept and want to know which parts of their processes can be optimized. Scientists on the other hand care
Existing scientific workflow systems

much more about data - input data provenance and reliability, the intermediate steps, and not only the result of a process. They therefore have other requirements to observe the processing of data. Each step may need to be watched carefully and possibly wanted to be modified before forwarding it to a successor component. Scientists tend to a much more experimental usage of workflows whereas optimization or end-user robustness requirements are less important. Iterations, recursion, loops and cycles in all flavors are very common. The (comparative) application of different algorithms to the same data set is as important as multiple application of a process to the different sets of data. Concurrent execution of sub tasks help to master the often huge amount of data to be processed. Of course also the requirements of different domains of science are different. While in life science generally spoken an easy way of rapidly pulling together third party services into prototypically in-silicio experiments is wanted most, in physics and astronomy domain for example, carefully designed work flows aim for the best exploiting of grid resources.

4.2 Existing scientific workflow systems

In this chapter an overview over some important scientific workflow management systems is given, with the identification of their main features, strengths and limitations. Workflow management systems that are specialized for bioinformatics are out of scope, as they are very specialized for this domain (applications like gen decoding etc) and often are not designed as general purpose scientific workflow systems, as well as some other large initiatives (Cactus Framework [Goo07], ICENI [MGLC+07], Askalon [FPD+07]) as they are often built on top of one of the described systems.

4.2.1 Kepler

Kepler [LAB+06] is one of the major initiatives of open source scientific workflow systems. It is a cross-project collaboration that includes several organisations that are developing modules to be used in scientific workflow applications. Originally it was developed for ecologists, geologists and biologists. Kepler is a stand-alone system based on Java and the
Existing scientific workflow systems

Ptolemy II framework (2005). Kepler/Ptolemy II use the notion of ‘actors’ as an abstraction for the modules that are plugged into a workflow. In general it is very complex and has a huge library of actors together with flexible control strategies to compose these. In Kepler distributed data access is provided through a set of EcoGrid interfaces. EcoGrid allows data and computation nodes to incorporate through a standard API. Resources are added through a distributed registry. Sub-workflows are used in Kepler to wrap the functionality of multiple components that form logical groupings. Actor/Sub-workflows can be replaced easily as needed. According to [LA08] the main limitation of Kepler is that it does not scale well to distributed environments or to functionality provided by third-party modules.

4.2.2 Taverna / MyGrid

Taverna [OLK+07] is an environment that merged with the myGrid project and allows researchers to access and link together a significant range of bioinformatics web services and grid services hosted by the European Bioinformatics Institute. Currently over 3000 services are available for a Taverna/myGrid user. Taverna hereby is MyGrids workflow execution and development environment. The workflow language of Taverna is called Scufl. Taverna is designed to work data flow centric and to support the whole range of the eScience life cycle of in-silico experiments (i.e. data management and the management of provenance information). This is for example done by flexible metadata generated, what helps to manage and share results. One of the key requirements for Taverna was the ease of use and the possibility of easy and rapid user-driven and ad-hoc workflow creation. The execution of workflow works with two failure recovery strategies: Retry a number of times and trying of alternative service (identical or not)

MyGrid is a project to build middleware to support workflow-based in-silico experiments in biology. MyGrid is designed having three layers: The App-data flow layer (User-level workflow object model). Its purpose is to present the WF from a problem-oriented view, hiding the complexity of the interoperation of the services. Users think in terms of the data consumed and produced by logical services and connecting them together. They are not interested in the implementation styles of the services. The execution flow layer relieves the user of most of the details of the execution flow of the WF and expands on control-flow
Existing scientific workflow systems

assumptions that tend to be made by users, manages data structures, iterates, implements fault recovery strategies. Avoids mixing the mechanic of a WF with its conceptual purpose. Last, the processor invocation layer is interacting with and invoking concrete services. Taverna/My Grid has components for: Service directory, ontology driven search tool, data repository, semantically driven metadata stores for recording the provenance of a WF and the experimental life cycle, distributed query processing and event modification.

4.2.3 Triana

Triana [TSWH07] was originally developed as data analysis environment for gravitational wave detection project. It offers an attractive user interface, provides a means for categorizing resources hierarchically and composing them into WFs. It allows scientists to compose their local applications and distribute their computations. Over 500 Java tools are available for Triana. Triana is completely dataflow-centric with no explicit support for control constructs. Loops and branching are handled by specific components. In essence Triana is a data-flow system for executing temporal WFs where cable connecting the units represent the flow of data during execution Triana components are simply units of execution with defined interactions which don’t imply any notion of state of defined format for communication Triana components represent a number of local and distributed primitives. E.g. Java objects, legacy code, WFs, WS-RF, P2P, web services, Grid Jobs, Files. Recently Triana components have evolved into flexible proxies that can represent a number of local and distributed primitives. For example a Triana unit can represent a Java object, a legacy code, a WF, a WS-RF, P2P or web service, a grid job, or a local or distributed file. Grid-oriented components and service oriented components. Each component has a definition encoded in XML that specifies the name, input/output specifications and parameters. Triana supports virtual data language (VDL) and directed acyclic graphs DAG workflows as WF language plus a proprietary simple XML-representation. The external representation of Triana WF is a simple XML document consisting of the individual participating component specifications and a list of parent/child relationships representing the connections. WF composition is somewhat independent of WF language constraints and currently we have
Existing scientific workflow systems

implementations for VDL and DAG WFs. Triana can be used as a translator between such representations.

Web service resource framework support: (WS-RF) Triana allows an additional context to be associated with these WS-RP operations

4.2.4 Condor / DAGMan

Condor [CKR+07] is a high throughput batch job scheduler. Instead of concentration on highly tuned, high-performance computing for short periods of time or a small number of applications, the focus lays more on reliable access to computing over long periods of time. Condor consists of different independent program parts:

Condor-G: Condors ability to interact with grid-systems

Stark: Used for data placement. Unlike in other scientific workflow management systems, data placement is treated as first-class citizens. Stark is responsible for allocating space, transfer the data and finally release space that is no longer needed. It also is able to continue incomplete files, or trying alternative transfer protocols. It is designed to work together with DAGMan.

DAGMan: DAGMan allows users to submit large workflows to Condor. DAGMan resides as a layer above the batch system in the software stack and reads the logs of the batch system to follow the status of submitted jobs rather than invoking interactive tools or service APIs. It has no persistent state. The runtime state is determined from input files and logs. As the name already implies it used workflows represented as directed acyclic graphs. The execution of workflows is done parallel whenever possible, and otherwise sequential. DAGMan is able to retry the execution of single node if they fail. As some nodes also need some setup before and cleaning-up afterwards, pre/post scripts can be defined for each node. A DAG node can submit any valid jobs, including submitting another DAG. This allows the creation of DAGs with conditional branches in them. The DAG node can make a choice and then submit an independent DAG based on the result of that choice.

VDS: VDS resides on the top of DAGMan and Condor-G. The user provides a description of what data are available and how the data can be transformed, then request the data
Existing scientific workflow systems

they need. VDS now creates a DAG that fetches and transforms data as needed. That is another way to deal with the data independency problem.

4.2.5 Pegasus

Pegasus (= Planning for execution in grids) [DMS+07] is a workflow system for mapping large-scale workflows to distributed resources. Pegasus is currently distributed as part of the Griph N VDS package that contains: Pegasus, Abstract Planer, Data transfer tool, WFvisualization and Kickstarts. Pegasus allows for designing workflows on application level without worrying about the execution environment, be it a grid, a set of condor pools or a local machine. Users provide a workflow template and artificial intelligence techniques are used to coordinate the execution of apps on a heterogeneous and changing set of resources. Pegasus generates an appropriate executable workflow from a workflow instance plus the information about available resources. Pegasus uses three different workflow abstractions:

- Templates: the skeleton of a computation. Describes participating components and dependencies between them.

- Instances: In order to fully specify the analysis data need to be provided to the workflow. Template + input data = workflow instance (abstract workflow). Instance workflows are portable and can be mapped to a variety of execution environments.

- Executable workflows: are workflows that includes all the necessary resource information (concrete workflows)

A variety of different methods is available to create a workflow instance

- Choose to design the WF instance directly according to a predefined schema

- Using Chimera to build the WF, based on the user-provided partial logical WF description specified in VDL

- Using Triana with GUI
• Using intelligent WF editors such as the Composition Analysis Tool (CAT). Uses formal planning techniques and ontology’s to support flexible mixed-initiative workflow composition that can critique partial workflows composed by users and offer suggestions to fix composition errors and to complete the workflow template.

Since the execution environment can be very dynamic and the resources are shared among many users, it is impossible to optimize the workflow from the point of view of execution ahead of time. It is assumed that the environment Pegasus run on is a set of heterogeneous hosts connected via a network, often a wide-area network, with one visible head node. The mapping process not only involves finding the appropriate resources but may also include some workflow restructuring to improve performance. In order to be able to schedule jobs remotely the resource needs to have appropriate software deployed that can provide information about the resources, stage data and accept job submissions. Pegasus interfaces with various catalogs to discover the data locations, executable locations and the available resources and their characteristics.

4.2.6 Java CoG Kit / Karajan

Java CoG Kit [vLHK07] is a tool for process management for grid and non-grid resource environments. It supports different WF solutions. The Karajan WF framework is the most important WF solution supported by Java CoG. Karajan supports two workflow languages (both can be converted into each other), one XML based, which derives from GridAnt and another workflow language with a simplified syntax. The philosophy of Karajan is based on the definition of hierarchical WF components. Outstanding features include among others, supports primitives for generic sequential and parallel execution, sequential and parallel iterations, conditional execution, functional abstraction and common data types like lists, maps. One of the important differences to other WFMS is that Karajan can be extended both through parameterized user-defined workflow elements (functions) and/or by implementing new workflow elements in Java. Also the Karajan feature of modifying workflows at runtime is usually not supported by other WFMSs.
4.3 Strength and weak points of existing scientific workflow systems: A look into the future

Existing scientific workflow management systems are close related to grid computing. The mapping of a workflow to a grid to process large amount of data and a high number of concurrent tasks have been the main drive behind their development and still is their main operational area. Many of the existing scientific workflow systems have originally been developed for areas that require many resources like earthquake simulation, weather forecast, etc. Scalability and performance have been the top priorities for scientific workflow systems. The concurrent execution of sub-tasks might become even more important in the future, as single CPU core performance growth begun to stagnate recently and multi-core processors emerged on the market instead [ZRF08]. However, scalability and performance is just one requirement that many scientific applications have.

Stating further requirements require a more global vision of the problem and taking into consideration the general differences between business and scientific workflow as in the first section of this chapter. Scientific workflow management systems might be the right approach to address the “data rich, information poor syndrome” [HHU+97] from which suffers not only the domain of water management. Also in [YEM+07] the question is discussed about why the growth of scientific data analysis and understanding is not going proportional to the exponential growth in computing, data storage, network and other performance elements. They come to the conclusion that, given that the number of scientists is roughly constant at the same time, one of the main reasons is the lack of more effective tools that aid scientist to not to be inundated in data and associated tasks in their research work. [WMGM08] as well as [Pen07] therefore demand to establishing a fundamental “science” of scientific workflows, the scope of scientific workflow tools should be expanded to support the entire scientific research cycle and scientifically working and data analyzing methods which includes data flow, design flow and knowledge flow tasks, considering the life cycle of in-silicio experiments, the knowledge supply chain [Liu07] to help analyzing massive amounts of data generated by eScience applications in successive steps as well as to better support the exploratory nature of research.
Strength and weak points of existing scientific workflow systems: A look into the future

Concretely the following are the most frequent mentioned issues that current scientific workflow management systems lack: Not being prepared for SOA [YYH09][CKRJ05][TCBnAE07], interoperability problems [YYH09][TCBnAE07], few possibilities for interactive steering [Liu07][Dee07], to little support for data provenance recording [BEO+07][Liu07], poor fault tolerance [BEO+07][Dee07] and difficulties for debugging, no support for WS-* extensions [PG], remaining platform and enterprise dependencies [YYH09] and poor understandability and readability of workflows [GG09] [MPAD+05]. Future scientific workflow management systems therefore should aim to address the following issues at its design:

**Workflows for capturing methods**

An example where the substantial usage of workflows and future workflow management systems could change the publishing of scientific results is the "method section" of scientific papers. Until now, an explicit documentation of methods is recorded in free-text "method"-section of publications. Typically only the conceptual steps are recorded. The multitude of computational details imposed on the data to enable execution is typically not recorded, although these may have significant effects on the result of the analysis. As workflows provide a formal specification of the scientific analysis process from the data collection, through analysis to the data publication, it would catch the different steps much more detailed and reproducible for everyone. Of course, a kind of standard workflow or meta model would be extremely useful for this, as well as workflows that better describe what they do. [GG09] for example observes that there is little coincidence between the single workflow execution steps and the natural language description of the same process. They aim to identify key constructs that intelligent workflow systems could support to allow for more natural workflow representation. Into the same direction goes [MPAD+05] who uses annotations to workflows and semantic web standards for the sake of more descriptive workflows.
Workflow reuse

So if workflows become the standard mechanism to capture and communicate methodology knowledge, reusing existing workflows created by other scientists needs to become much more common and easy. Workflow reuse involves two major aspects: retrieval and adaptation. These activities may become a natural way to conduct experiments and share scientific methodology within and across scientific communities.

Retrieval means finding appropriate workflows what requires organized publically accessible repositories. Discovery may work at multiple levels and depend much on the guarded metadata and used description language (see subsection "Design by contract" page 67). For helping discovery a unified description model like developed in [WGG+07] for example could be very helpful. Also approaches based on web ontology languages like found in [DNYC08] could also be an option. An easy way to deploy workflows, to store it in publically accessible repositories and to support easy discovery is urgently required. Regarding adaptation there is the rule that the less sophisticated a user is, the more he is likely to reuse entire workflow structures. More sophisticated users however might want to create their own variants of the workflow. Possibilities of adaptation include substituting input data, substitute components, adding new steps or remove some nodes. After that another consistency check might be necessary. A workflow management system must allow performing those issues easily.

An example towards this goal stated here of sharing workflows might be www.myExperiment.org, which is a virtual research environment that facilitates collaboration and sharing of workflows through a social web approach. Users can upload and retrieve workflows, tag them with annotations, connect to their associates and message other users. It also supports the packaging of experimental results and other data with workflows. Currently however only Taverna workflows are used in this platform.
Support for experimental usage

One of the most important questions for the design scientific workflow systems is "How can workflows support the exploratory nature of science and the dynamic processes involved in the scientific analysis?" According to [Slo07] one way to enhance the experimental flexibility is using a "scientific laboratory notebook" paradigm. This mainly requires support for rapid prototyping, testing and evolution of partial solutions. Several features need to be supported like tools for developing a workflow incrementally, and for being able to add and remove steps in a running workflow; to modify existing workflow activities and its structure during execution, to allow repeat execution of workflow parts, and branching by cloning its state.

User interaction

With the grid computing background of most of the scientifically workflow systems, most of them only support a "one shot"-user interaction [Dee07]: Once started workflow execution must continue to completion, be aborted or cancel due to error states. Providing support for more interactive workflows poses great challenges for the future that may include debugging and monitoring the execution of workflows, stopping execution to examine intermediate result and to possibly change them before going on, repeating steps with another set of data or in general navigating back and forward as desired, altering the workflow at runtime, and much more. WS-BPEL for example, still the mostly used web service wiring languages does not offer support for user interactions [AMA06].

Enhanced SOA capabilities

Last, the issue that is most important for the work of this thesis is that most of current scientific workflow management systems do not fit very well into the SOA paradigm. Many authors claim that they are not prepared for dynamic changes and the interoperability capacities demanded by SOA. Dynamic changes for example may affect reliability, so that there is a need for dynamic verification mechanisms, which are not featured by most of the scientific workflow management system [BEO+07]. Further they do not support WS-*
Strength and weak points of existing scientific workflow systems: A look into the future

extensions [PG]. Also performance issues are mentioned [LA08]. Current workflow management systems are mostly monolithic architectures and impose platforms and enterprises dependencies [YYH09]. For a true SOA approach extern service repositories should be used instead of the internal repositories used by those. The biggest advantage of SOA however still is its interoperability capability that helps to overcome incompatibility problems between different components of different systems and platforms. But while many scientific workflow systems allow for the integration of SOA services, they do not offer their own functionality as services that can be integrated into external software. For this reason [YYH09] claim the next generation of workflow management system architecture, that should consist in a set of generic and platform independent services. They develop a distributed, adaptive, platform independent workflow management system architecture based on standards and a common reference model.

In Part IV of this work we are going on with scientific workflow systems, designing and developing an own solution that tries to consider as much as possible from the enlisted issues, especially the limited user interaction and SOA capabilities of existing scientific workflow systems. This way NORTIFlow will bases on a set of services similar as proposed by [YYH09]. But before, we will have a look at environmental modeling and water quality assessment in the next chapter.
Part II

Water quality assessment models
5 Environmental modeling: Water quality assessment models & SOA

This chapter explains the biological background of this project, the domain of environmental modeling and empirical water quality assessment. Furthermore it treats the possibilities, potential advantages and existing approaches of using the SOA programming paradigm for implementing environmental models.

5.1 Environmental modeling for freshwater quality management

Environmental modeling tries to model nature and natural phenomena with mathematical and physical methods. Weather forecast, simulation of climate change, earthquake simulation, population growth simulation or hydrologic models are some widespread examples. All these phenomena in reality are very complex and often include a high number of variables with sometimes thousands of parameters. In spite of this complexity, often very good approximations and predictions can be achieved.

In general the application of models is limited by two main factors: the quality of input data and the limited knowledge about the processes. Due to the second factor, regarding a limited knowledge about the processes, it is a common practice to refine and improve models always further even while using them and with the presence of new data. In water quality assessment this may for example include the development of multi-metrics espe-
cially adapted to the water bodies of a region. Thus, the development of environmental models in general is frequently an experimental, iterative and often very complex process [CWF15] [GDG10]. This fact should be considered when offering software to implement environmental models, for example by using techniques like workflow systems described in the last chapter that allow easy prototyping, modifications and refinements of models.

In the sector of freshwater management models are used in the different areas, mainly: characterization of water bodies, identification of stressors, planning, control and the estimation of efficiency of measures and decisions. Three different types of models are in use: hydrological models, material models and ecological models. The main purpose for hydrological models are for balancing partial systems, to model the interaction of partial systems or to model fluid dynamics. Material models are used to simulate the remaining of nutrients and toxics within a medium. They consider currents, transport and transformation of material. Both hydrological and material models use different modeling approaches depending on the type of water (groundwater, lakes, rivers etc). Ecological models, on the other hand, do not make such a distinction and are better classified according to their aim. Some examples are habitat suitability models, meta population models, food web models or succession models.[KK02]

All models furthermore are either mathematical deterministic or empirical (statistical) models. While deterministic models often base on physics and try to simulate physical processes, empirical models base on statistical techniques and try to draw conclusions from measures and samples taken, being therefore more "black-box" biased than deterministic ones. A clear separation of both however is not always possible when it is for examples not very clear how much weight the factors need to have. Models can also base on different mathematical methods than statistics like fuzzy logic, rule-based knowledge or include spatial information [KK02]. Anyway, in this work we will only tackle empirical models, that we will describe more in detail in the next section.

Special attention also needs to be paid to the coupling of models. Single models in water management are all either micro-scale models (for single sites, wetlands or river sections) or meso-scale models (for bigger regions, whole river basins etc). Problems can occur when it is necessary to consolidate information of both. Similarly, in many countries the responsibi-
ties for regions are shared and each responsible part may use different models for evaluation. But as ecosystems are not restricted to boundaries defined by humans such as between local governments or countries, this may prevent an integrated approach [HHU+97]. Also here the same problems may occur when trying to enhance the quality of the information obtained by connecting the models to cover bigger spatial units or when trying to compare results directly. While some models only consider some factors for evaluating water quality while neglecting others at the same time, other models base on other factors so that results are not always comparable directly very well. Last, many models also already are compositions of independent partial models that are used to model sub-problems. To keep it short, the coupling of modes is a very important and actual research area. Many examples can be found in research works to illustrate the importance of coupling of models in the sector of water quality assessment. Kronvang [KHS+99], for example pointed out that “The implementation of the water framework directive however requires an integrated view of all elements within a basin that are important for water and material emission as well as for the composition of the water biocoenosis. For this models need to be coupled by adequate interfaces.”. Also Kluge [KK02] states that “For the future a coupling of hydrological, material and ecological models is required urgently. The coupling of models in general should be realized through open and predefined interfaces, so that partial models also can be applied independently. Before the development of the partial modules it is necessary to define definitely the project space and the spacial resolution. For the application of the models in environment administration it is necessary to explain the models in a transparent manner, especially the errors and the inaccuracies at the presentation of the results.”, to give another example. The coupling of models hereby takes place on two levels: At the modeling level and at the implementation level. At the modeling level it is a problem of logic and that partial models treat complementary aspects of the problem. At the implementation level it is first necessary to choose an environment to implement the model. Many models for example exist as mathematical description so that Computer Algebra Systems (CAS) like Matlab, statistical software and so on can be most suitable for implementation. Others are implemented using ordinary programming languages, scientific workflow systems, environmental information systems, or with the programming possibilities of a database management system. The problems of later connecting these models
Empirical/statistical models for water quality assessment

implemented on different environments are those treated in the first part of this thesis regarding compatibility and interoperability, platform and data independence, the lack of clear exposed interfaces and so on. To more easily connect models this thesis argues to use SOA based approaches and to introduce them into environmental modeling, what would be an easy way to overcome such compatibility problems.

5.2 Empirical/statistical models for water quality assessment

One type of models used in water quality assessment are empirical models. These models base on statistical techniques and try to draw conclusions about water quality from samples taken. Empirical models are the most employed in water quality assessment field as they allow to obtain useful and realistic results without needing to know and quantify with detail all of the very complex interactions between the elements involved in water ecological status. Those models often base on physicochemical characteristics on the water, and the presence of various makrozoobentos, but also may use other indicators like makrophytos, phytobenthos, phytoplankton, fish or samples of chemical substances dissolved in the water. Some of the modeling methods are considering only one single indicator, while others combine various different factors to hopefully obtain a more reliable result or to better draw conclusions about the most responsible stressor. Many of the models are following a reference-based approach, where the evaluation is obtained comparing the testing site to an ideally unaffected reference spot of the same water typology, which can also be a sample of real sites previously selected by their good condition. An example is the ecological quality ratio approach (EQR) [vdBS06], which is the proportion between an index of the testing site and the same index of a reference site. With the growing attention paid to an holistic management of ecological status of freshwaters, those approaches have gained increasing relevance, largely due to governmental decisions that encourage this type of assessment, like the European Water Framework Directive [Dir00]. A classification of the different statistical water quality assessment methods can be done by differentiating single metric approaches, multi-metric approaches and multivariate analysis techniques that are
Empirical/statistical models for water quality assessment described in the following. In last decades, also soft computing and artificial-intelligence methods (based on heuristic search methods, artificial neural networks [HHZ08], inductive logic programming, etc.) are gaining importance [Goe05], but their usage is still very limited and more experimental than applied. They are out of the scope of this work and not described here any further.

5.2.1 Single Metrics and Bimetrics

5.2.1.1 Saprobic System

The saprobic approach was the first river assessment system used, already developed at the beginning of the 20th century by Kolkwitz and Marsson [KM02]. The determination of the oxygen supply and saprobity by makrozoobenthos is still considered a scientifically very reliable method, which helped a lot in the last century to assess and enhance water quality. Saprobity is related with all processes that are consuming oxygen in the water. Pioneer work to this topic dates back to the publications of Streeter and Phelps in 1925 [SP25]. They describe the bacterial decomposition of organic carbon characterized by biochemical oxygen demand (BOD) and its impact on dissolved oxygen conditions. Later on this work were improved several times always considering more factors and distinguishing more details like introduction the settling rate in addition to the decay rate, the sediment oxygen demand as an additional parameter or distinguish between carbonaceous BOD (CBOD) and nitrogenous BOD (NBOD). More advanced models also include the effect of photosynthesis, differentiate between immediate and delayed oxygen depletion or represent daily oxygen fluctuations. Present water quality models include as factors also biodegradation, sediment oxygen demand, respiration and nitrification [CUWU96][Jan89][SWF+00][Goe05]. Therefore, the main idea behind the saprobic approach is to evaluate water quality based on the pollution tolerance of the indicator species present. It is known that every species has a specific tolerance against pollution as they depend on organic substances and the dissolved oxygen content. This tolerance can be expressed as a value. Together with different weights for different indicator species, these values are used to calculate the saprobic index for a river section. This index is compared to a standardized list of indices to evaluate water
Empirical/statistical models for water quality assessment

quality. By using a high number of indicator species this measure is normally very reliable. The river section can now be assigned to a water quality class. The classical saprobic approach of Kolkwitz and Marsson uses four different water quality classes. Over time, these have been extended to seven by introducing some intermediate stages to allow for a finer differentiation. The main advantage of the saprobic approach is a quick classification of the investigated community by means of a saprobic index. The main drawbacks are the required identification of the organisms up to species level and that the saprobic index calculation also requires measuring abundances, which is a time-consuming and error-prone task. Furthermore there is the problem that the pollution tolerance is a value that is very difficult to determine. The values in use are based on empirically ecological observations and are rarely confirmed by experimental studies. The saprobic approach furthermore cannot deliver information about the type and amount of specific noxious substances and pollutants and further stressors. This is why it is not sufficient to assess water quality only with the saprobic approach if we expect to have a comprehensive view of the situation. Therefore, more sophisticated approaches have been proposed, as we describe below.

5.2.1.2 Diversity approach

The diversity indices [Was84] [Goe05] was developed with help of information theory methods. It uses three parameters: richness, evenness and abundance. The richness is the number of observed species at a site; the abundance is the number of individuals; and the evenness measures the variation in the abundance of individuals per species within a community (or in other words, the uniformity in the distribution of individuals among the species). Communities with less variation in the relative abundance of species are considered to be more "even" than a community with more variation in relative abundance. The diversity index now relates richness to abundance, i.e. it evaluates the community with respect to the occurrence of species. The basic idea is that disturbance of the water ecosystem or communities under stress leads to a reduction in diversity. Therefore, the most diversity indices that are most frequently applied in stream studies are: the species richness, the total diversity and the evenness index [CUWU96][SWF100]
Advantages of this approach include that it is easy to use and calculate, its applicability to all kinds of water courses with no geographical limitations and its suitability for comparative purposes. The main disadvantage is that it has no clear endpoint or reference level. The diversity index value in natural undisturbed waters can vary a lot. They depend as well on the sampling method as on the nature of the study site. So endemic-rich zones, for example, can have indices very different from other zones even if they have similar geomorphologic characteristics, making comparisons very difficult. Moreover, diversity index values are unable to indicate if the community consists of pollution-tolerant or pollution-intolerant species. Furthermore, all species in the diversity approach have equal weight, independently of their inherent characteristics, their sensibility to stressors, and then they interest to measure water quality. This is probably the reason why not one country in Europe has adopted a diversity index as a national standard for biological water quality assessment.

5.2.1.3 Biotic approach

The biotic approach is a combination of the diversity approach with the pollution tolerance indication of individual species or higher taxa, as used in the saprobic approach, into a single index or score. It therefore can be considered a biometric approach, as it is combination of two metrics to obtain one only. The most sensitive taxon present in a surveyed site, together with the number of relevant taxonomic groups, is translated into a synthetical single numerical value. The basic principle behind the biotic approach is that the number of taxonomic groups reduce as pollution increases and macroinvertebrate groups disappear at the same time. The disappearance of the groups happens in a determined order from the most sensitive groups to the more tolerant ones as described by Mackenthun [MU69].

Woodiwiss [Woo80] distinguishes between biotic indices and biotic scores: In the biotic index approach the biotic index can be directly obtained from a table, which combines the taxa richness with the most sensitive taxon present . An example is the original Trent Biotic Index [Woo64]). Biotic indices obtained this way do not consider abundance per taxon for their calculation, except for threshold values that are sometimes considered for inclusion
of taxa. In the biotic score system, on the other hand, a score is assigned to each taxon. The score for the site is then obtained by summing the individual scores assigned to each taxon. Some biotic scores also require an abundance measure of the organisms. The main advantages of the biotic approach are that only qualitative sampling is required and that identification is mostly at family or genus level without the need to count abundances per taxon (even if early biotic indices still required identification up to the species level). This also helps for example to make the biotic approach widely accepted in East Asia where the lack of taxonomic knowledge remains the biggest constraint in applying bioassessments.

One remaining problem however is how to determine representative reference communities to which the investigated stations can be compared to.

[CUWU96][Jan89][SWF+00][Goe05]

### 5.2.2 Multimetrics

Multimetrics are the combination of various single metrics into one single value to enhance the expressiveness of a single number and to consider more factors that influence water quality. [VM06] distinguishes the eight classes "richness measures", "enumerations or composition measures", "diversity measures", "similarity/loss measures", "tolerance/intolerance measures", "functional and trophic", "strategy metrics" and "condition metrics" of single metrics which are used to build multi metrics. The first explicitly called multimetric systems were developed in the US by Karr [Jam81] for assessments based on fish. It originally included 12 metrics that include number of species, presence of intolerant species, the richness, composition and proportion of some indicator species as well as some ecological factors including number of individuals in sample, proportion of some other indicator species and the proportion with disease, tumors, fin damage and other anomalies.

In further works [Kar91]Karr proposed a methodology to obtain an index of biotic integrity (IBI) that is obtained by comparing metrics derived from fish samples from the site to evaluate (test site) to metrics from a reference site that was previously identified as a high quality site (reference site) for the same type of water body. This comparison delivers similarity scores which are summed up to obtain the IBI value. Karr’s approach was later
Empirical/statistical models for water quality assessment

widely used in the USA [Bio96] and in Europe [HMSV04]. The IBI nowadays is the most widely used multi-metric index to assess the biological health of fish communities.

In general, for multi-metric systems, different metrics represent different characteristics of the macroinvertebrate/fish/diatoms community or whatever species are used for the evaluation. It is expected that working with more descriptors will result in an index being representative for a specific aquatic environment. The selection of the metrics that take part of the multi-metric calculation should be based on how complementary (orthogonal or uncorrelated) they are. It is to avoid that correlated metrics dominate the overall assessment. The choice of metrics can also base on how explanatory they are to get insight in the causes of deterioration. Although most of the multi-metric systems are not capable to separate the impact of different stressors [ADCP04], some examples of such stressor-specific multi-metric systems can be found in [BZN+04] and [BECK04].

5.2.3 Multivariate Analysis

Multivariate analysis (in contrast to univariate approaches) means analyzing various statistical variables at the same time. Several multivariate techniques have been applied in environmental modelling in general and particularly in water quality assessment. Since the nineties they are also commonly applied for the development of multi-metric systems. All multivariate approaches base on a similarity index that indicates how similar a sample of a test site is to samples of a reference site or a set of reference sites. Some commonly used similarity indices are the Jaccard’s index that expresses the percentage of species shared between two sites, the percentage similarity index, the Bray-Curtis dissimilarity index, the Sorensen index or the Euclidean of ecological distance [CUWU96][Goe05].

As the knowledge of macro invertebrate communities in rivers within geographical regions has now reached a sufficiently advanced level, the multivariate approach is also used for developing predictive models to evaluate the ecological status of aquatic ecosystems. Two of the more influential ones of these predictive models are described in the following:
5.2.3.1 **RIVPACS**

One major example of assessment systems using multivariate approaches is River Invertebrate Prediction and Classification System (RIVPACS) [CWF15]. RIVPACS tries to find statistical relationships between the fauna and the environmental characteristics.

To do this, first largely "unaffected" high quality sites are selected that can be used as references. Here "unaffected" means in the absence of pollution or other environmental stress. Those reference sites are classified in groups according to their faunal composition, and statistical relationships between macroinvertebrate fauna and environmental variables of those sites are determined. Then, to assess other sites, they are firstly assigned to one group basing on their environmental characteristics. Next, their faunal composition is predicted using statistical relationships with environmental characteristics determined for reference sites of this group. This prediction is compared to fauna from reference sites, which is the expected fauna in good ecological conditions, to obtain indices of ecological quality and status of the site. The RIVPACS approach was also used in other models like AUSRIVAS [SN00], which is the adaption of RIVPACS to Australian conditions, Medpacs [PATP+09], an adaption for the Mediterranean rivers, or the Mondego model [FRFVG07].

5.2.3.2 **Reference Condition Approach**

Reynoldson et al. [RNR+97] proposed another approach based on reference sites called the Reference Condition Approach (RCA) and developed the BEAST predictive model [RDP]. RCA shares several common aspects with RIVPACS. It defines the Reference Condition as the state of sites minimally exposed to stressors, measured by means of operational criteria previously set. Differing from other approaches, RCA specifically rejects use of criteria based on biota diversity and interactions, because they are often subjective and immeasurable. Once Reference Condition is established, other sites can be assessed comparing their biota with the corresponding reference biological community. If significant deviation is detected, further study should be considered to confirm if this is really due to existence of some stressors.
The EU - water quality framework directive

The water quality assessment models used in the NORTI project on which bases this thesis are first of all multimetric and multivariate analysis models (an adaption of the reference condition approach described last). These are the currently most used models, whereas single metric models are losing importance.

5.3 The EU - water quality framework directive

When on 22.12.2000 the EU water framework directive (in short WFD)[Dir00], which aims to achieve at medium-term a good ecological status of all European waters, became applicable, a new chapter in water management and water quality assessment begun. Until then, official water quality assessment mainly based on the saprobic approach described in the last section. The disadvantages of this approach however include its limited expressiveness regarding the important identification of specific stressors that are most responsible for declining water quality. From now on, an integrated assessment of the whole ecological state is required. Beside hydrological and hydrochemical ones, also biological and structural (morphological) parameters should be covered, like for example the impact of water course development measures, which in some European counties already became the main impact factor [SdSFdB08]. The EU water quality framework directive strictly requires the use of four Biological Quality Elements (BQEs) in freshwater assessment: fishes, benthic invertebrates, benthic algae and macrophytes. Furthermore it also demands that used models have to base on a reference approach (see last section). As an acknowledgement to the fact that real undisturbed waters like required for reference approaches are rarely available in praxis, the directive also allows "best available" or "least impacted" sites [SdSFdB08].

The used assessment models should reflect different impact factors and allow for the identification of main stressors. For the realization of the directive therefore some completely new assessment models and methods are necessary. The targeted aim of the directive is to establish good ecological and chemical water quality in all European water bodies by 2015. The models therefore need to help at decision making, i.e. to allow drawing conclusions about concrete measures to put into practice, which again is closely related to the identifi-
Environmental software implementation using SOA techniques

Realization of the main stressors. This for example can be realized with a modular approach that is capable to capture saprobity, acidosis, morphological degradation, etc, separately.

The water framework directive also requires the use of Ecological Quality Ratios (EQRs). The resulting values are scaled between zero and one, what allows for an easier comparison between member states as, for instance, required for intercalibration. These values then are translated into five ecological quality classes from high (= reference), good, moderate, poor to bad [vdBS06].

5.4 Environmental software implementation using SOA techniques

As pointed out before, for water quality assessment, as for environmental modeling in general, the ability to connecting different models it is very important. The demanded integrated approach of the EU framework directive means connecting different models as well as multi metrics connect different metrics. Software like AQUEM [AJS+01] (only for macro invertebrates), or OMNIDIA [CLC] (only for diatoms) support this by a modular design. The problem however is that they are closed systems, each with its own data base that has to be created and populated and with few possibilities to adapt the models to own needs or to access them from outside. The first step towards a SOA based approach would be to access data by using web services. Research work has been done in the domain of environmental modeling to achieve this. Examples are: [GHW+08], who creates a machine accessible interface for the National Water Information System (NWIS), which is an online repository of historical and real-time stream flow, water quality and ground water level observation maintained by the United Staes Geological Survel (USGS). [UJSW10] created a set of sensor specific services, primarily based upon standards of the Sensor Web Enablement initiative of the Open Geospatial Consortium (OGS). [Pap05] also deals with the same problem of receiving weather data from different sources. Faced with the reality of always changing requirements and looking for an architecture that is designed to minimize the impact of future changes, he proposes a ”plug & play-approach” as a possible solution,
Environmental software implementation using SOA techniques

which works with well defined standards for interfaces, what reminds of the components of component based software engineering.

The second step would be to implement models such way, that they can be reused and connected. A work investigating this problem area is [HHdV10], who analyses the problems of connecting different environmental models implemented in different systems. Mismatch between data semantics (data independence problem, see page 30) is pointed out as a big problem and he comes to the conclusion that extensibility must be built into the design for example by a strict "design by interfaces" (see also page 67). He also observes how the tight coupling of the user interface to a model implementation hinders the reusability (as pointed out in "separation of the user interface", page 71 in this thesis) and claims a separated implementation in order to increase reuse possibilities. Surprisingly he does not mention or think of SOA as a possible solution for these problems.

But to overcome these problems with SOA, we need in addition to data gathering services also web service with processing capabilities. Projects also doing this in contrast to creating data services are still rare. The AWARE project [GDG10] however is an example for this. In this project geospatial services are created. This work therefore has a similar focus as this thesis. Important differences of both projects are however, beside treating another sub-domain of environmental modeling, that their workflows are fixed built into the portal, which is a less flexible solution. On the other hand, its design bases on established standards (released by OCG and used in the INSPIRE project), while in the sector of water quality assessment no standards are released yet to keep on.

The MEDiterranean Prediction And Classification System - MEDPACS [PATP+09], also in the same sector of water quality assessment, is a system to evaluate the ecological status of the Spanish Mediterranean streams (the predictive models can be used for the spanish mediterranean basins - except the Ebro basin while it is planned to increase their application area in future updates to the rest of spain). MEDPACS allows to apply some water quality assessment models (also based on multi metrics multivariate analysis models) through a web portal as well as downloading the results and reports. The biggest difference in the design of this system is however that there is no possibility for developing own models and therefore, although available online, it is more related to close systems like AQUEM. They do not explicitly use a SOA approach. It is also necessary to populate their database and it
Environmental software implementation using SOA techniques

is not foreseen to adapt the models to own needs. This is actually the reason, why we could not just use this approach for the Norti project. Collaboration support, flexible extension possibilities, model and workflow sharing as well as support for coupling of models are the main requirements of our system.
Part III

Methodological Approach
6 Research aims

In this section the principal and secondary research objectives of this thesis are described briefly. The primary aim was to conduct research on reuse in environmental modeling software design and development, and particularly on how SOA techniques can facilitate software reuse in this domain. From this main aim, and according to the context of our research, the following secondary aims have been established:

- Investigating the design and implementation of a SOA based solution for the calculation and application of metrics, multi metrics and numeric multivariate water quality assessment modes based on biological parameters. SOA based implies to make it data base independent, platform independent and more flexible with the possibility to easily add new future models. This includes the design of data structures and components sufficiently generic and adaptable to cope with the given requirements for water quality monitoring, where different organizations are in charge of water quality assessment in different regions that may have to work together and possibly want to share data and models. Data may be introduced into the system coming from a variety of other software and data sources used by the different organizations. This should to be accomplished easily. Different users with different privileges need to be foreseen. Also the other way round needs to be supported: the service infrastructure to be designed must allow access from other applications to data and functions offered by the system so that external entities can develop their own applications, define their own processes and adapt the components to their own needs and allow for example the subsequently processing of the generated data by other tools and software (e.g. spreadsheets, statistical software or OLAP packages, etc.).
6. Research aims

- Analyzing the implementation process. Determine remaining difficulties for using SOA in this kind of projects and in this domain. Extracting constructs, models, methods and/or design patterns that are most suitable for solving the given problems and that could be applicable also in similar projects, gaining a high reusability this way.

  Studying the viability of the inclusion of several COTS that should be integrated. These are first of all a calculus engine, which is needed due to the fact that many of the models are doing complicated calculations that often are already implemented in some COTS packages, GIS packages: as also GIS capabilities are needed frequently in this sector, as well as scientific workflow systems that allow for a fine grained control over the modeling process (see next point).

  Evaluation of the achieved flexibility, data and platform independence in comparison to traditional implemented systems

- Designing a system that allows fine grained control over the service composition. The legacy software as well as most other environmental software does not allow such a fine grained control over the service composition process as would be needed due to the experimental nature of model development. Refinements of the models need to be done easily. The reliability and limitations of models need to be evaluated by for example comparison of results with other modes. It also should be considered that most of the users are not programmers but nevertheless may want to develop and test new assessment models. Scientific workflow systems allow much of this flexibility but it remains to analyze how to connect them with SOA, which allows for the other part of the requirements of interoperability and compatibility, as well as integration of the created workflows into own projects.

- Designing a system that allows fine grained control over the process execution. Also the process execution needs support for a fine grained control over it. As stated above, in the scientific domain it is crucial to be able to observe each intermediate execution step. Tools for visualization the data need to be at hand and making modifications of these should be allowed. The user wants to have possibilities to interfere into the execution process, to stop execution at any time to continue it at another time, to repeat steps, going back, sending the result to other nodes, and so on.
6. Research aims

This research work was conducted as a case-study. The context where it was developed was the necessity to improve an existing first limited single-user prototype version of the Norti software for an easier integration of new assessment models and further requirements. The required redesign of this system was used as the frame to conduct this research work as a case-study with the previously described aims.
7 Design Research

This chapter reflects the methodology used to perform this work and to achieve the research aims, something which, unlike to most other engineering domains where the importance of following a strict defined methodology is more commonly accepted to be crucial, often seems to be neglected in the domain of software engineering. Although most designers in fact are applying and following some methodology - the methodology of design research - they are doing it unaware of it and therefore do not strictly following a guideline and highlighting the relevant questions clearly enough.

7.1 Design vs. Design Research

First there is to hedge design research off against ordinary routine design. The main difference between routine design and design research is the clear identification of a contribution to the knowledge base of foundations and methodologies. Design in design research always has a new, original, experimental or innovative part or solved problems in more effective or efficient ways and therefore adds some knowledge to the knowledge base and contributes something to further research, whereas ordinary design or system building only applies existing best-practice knowledge to already solved problems. The key differentiator between routine design and design research therefore is the clear identification of a contribution to the knowledge base of foundations and methodologies. Another issue is the type of problems, to which design research can contribute. Design-science research in software engineering mainly addresses what are considered to be wicked problems [Bro96]. That is, those problems characterized by [HMPR04]:

[118]
- Complex interactions among subcomponents of the problem and its solution
- Inherent flexibility to change design processes as well as design artifacts
- Unstable requirements and constraints based upon ill-defined environmental contexts
- A critical dependence upon human cognitive abilities (e.g., creativity) to produce effective solution
- A critical dependence upon human social abilities (e.g., teamwork) to produce effective solutions

We notice that the type of problems treated in this work that mainly are adding an enhanced reusability, flexibility openness, extensibility and so on to a system fits into this definition of "wicked problem". Thus, design research might be the appropriate methodology to treat this type of problem. According to the design research paradigm, knowledge and understanding of a complex problem is achieved by building and applying the designed artifact. The process of building artifacts and analyzing their usefulness contributes to better understand the real problems depth. Creating an artifact is to apply known solutions and extend them by the experience, creativity intuition, and problem solving capabilities of the researcher. This process helps to get closer to the core of the wicked problems that otherwise are very difficult to grasp. This process helps developing always better artifacts and methods with the potential feasibility to solve it. [NCP90][HMPR04]

### 7.2 Design Research Framework

The next question remaining is how to conduct, evaluate, and present design research. To accomplish this, this work follows the framework and the guidelines presented in [HMPR04], which is summarized briefly here:

1. The environment defines the problem space in which reside the phenomena of interest [Sim96]. For IS research, it is composed of people, organizations and their existing
or planned technologies [SMB95]. They define their business needs or "problems" as perceived by the researcher.

2. Given such an articulated business need, IS research is conducted in two complementary phases: Development (build) and Justify (evaluate). The design process is a sequence of expert activities that produces an innovative product (i.e., the design artifact). The evaluation of the artifact then provides feedback information and a better understanding of the problem in order to improve both the quality of the product and the design process. This build-and-evaluate loop is typically iterated a number of times before the final design artifact is generated [MAL02]. During this creative process, the design-science researcher must be cognizant of evolving both the design process and the design artifact as part of the research.

3. The knowledge base provides the raw materials from and through which IS research is accomplished. It is composed of methodologies and existing design artifacts. The design artifacts are broadly defined as constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices) and instantiations (implemented and prototype systems).

- Constructs provide the language in which problems and solutions are defined and communicated.

- Models use constructs to represent a real world situation - the design problem and its solution space. Models aid problem and solution understanding and frequently represent the connection between problem and solution components enabling exploration of the effects of design decisions and changes in the real world.

- Methods define processes. They provide guidance on how to solve problems, that is, how to search the solution space.

- Instantiations show that constructs, models, or methods can be implemented in a working system. They demonstrate feasibility, enabling concrete assessment of an artifact’s suitability to its intended purpose.
7.2.1 Evaluation of the seven points of the design research framework

According to the design research framework these seven points have to be evaluated in order to correctly perform a design research work what we will do in the following:

1. Design as an artifact: As stated above to be the main difference between ordinary routine design and design research, it is most important to clearly identify artifacts being constructed, which can be constructs, models, methods and instantiations. This work contributes to the design science knowledge base with two types of artifacts: constructs and an instantiation. Constructs provide the vocabulary and symbols used to define problems and solutions and have a significant impact on the way in which tasks and problems are conceived [Sch83]. The extraction of key factors and basic principles for SOA solutions is an attempt to elaborate such a vocabulary as we try to approximate the problem in terms of independence, technology agnostic design and separation approaches. The constructs then should enable the constructions of models or instantiations of the problem domain. The instantiations again have a profound impact on design work, as they demonstrate feasibility of both of the design process and of the designed product. This way of proof is called "Proof by construction". The implementation of the Norti-Online web portal that meets as many as possible of the primer stated requirements should then prove of the truth of the assumptions made in the theoretical part. Models and methodologies are not created in this work.

2. Problem relevance: The problem that is treated to be solved needs to be of interest for science. The relevance of the problems treated in this work is obviously. To prove this, it is enough to refer to the amount of literature regarding incapability, data mismatch and similar problems that are very basic and general problems in software engineering, as well as the number of different approaches trying to overcome them.

3. Design Evaluation: The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well executed evaluation methods. Evaluation of a designed IT artifact required the definition of appropriate metrics and possibly the gathering
and analysis of appropriate data. IT artifacts can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant quality attributes. When analytic metrics are appropriate, designed artifacts may be mathematically evaluated. The evaluation of designed artifacts typically uses methodologies available in the knowledge base. Regarding this project and possible evaluation methods, only a descriptive way of evaluation is being performed. According the framework this only should be done for especially innovative artifacts or for which other forms of evaluation may not be feasible. The reason is, that it might be very complicated to find mathematically accurate metrics for measuring reusability, flexibility, etc.

4. Research Contributions: The ultimate assessment for any research is to give answer to the question "What are the new and interesting contributions?" Most often, the contribution in design research is the artifact itself. It may extend the knowledge base or apply existing knowledge in new and innovative ways. Also in this case, the main contribution will be the constructed artifacts and constructs. Especially NortiFlow as a scientifically workflow construction and execution environment can also be used for implementation of other SOA projects in other domains and contribute to connect scientific workflow modeling with the SOA world. But also the Norti-Online web portal with its collection of web services is specially designed to be reused which may makes it easier to implement own solutions in this domain.

5. Research rigor: "In particular with respect to the construction activity, rigor must be assessed with respect to the applicability and generalizability of the artifact" [HMPR04]. It is however admitted that rigorous evaluation methods are extremely difficult to apply in design-science research and that the principle aim still is to determine how well an artifact works and not to theorize about or prove anything about why the artifact works. Regarding the ability to generalize the results of this work, we think that the SOA key factors and principles that have been worked out also have the same importance and relevance in most other similar project. Especially NortiFlow architecture and design is not limited to the sector of water quality assessment, as it can be reused by other domains.
6. Design as a Search Process: According to the design research paradigm, design is essentially a search process to discover an effective solution to a problem and to reduce or eliminate the differences between a goal state and the current state. [HMPR04] [Sim96]. This search process starts simplifying a problem by for example decomposing a problem into simpler sub-problems and limiting itself to solve them and should iteratively expand the scope. As the scope expands so far to make assumptions more realistic, the design artifact becomes more and more relevant and valuable. In this case it is impossible to determine how to add a software artifact the definitely best reuse capacity and flexibility, meaning that it definitely will perfectly fit in future component models (as these still do not exist in present and therefore have unknown requirements) - we have to keep to the principle of a "satisfactory solutions", i.e. a solution which is satisfying without explicitly specifying all possible solution while the search process for even better solution goes on. That our solution definitely is not the last word spoken can easily be seen in the fact that we cannot solve every open remaining question like how to achieve real data independence or user interface separation like shown in the first part of this work.

7. Communication of Research: This last aspect of design research is to reason about who will read this work and if the content is presented to them in an adequate way? Design research works usually must be presented both, a technology-oriented as well as a management-oriented audience. In fact this thesis is more dedicated to technology-oriented audiences, who have to solve similar problems of implementing flexible SOA systems maybe in the domain of environmental modeling. The biological part of this thesis is reduced to the minimum needed to understand that the water quality assessment models do. Also implementation details are not explained very detailed and the focus lays more to explain the general design and design problems.
Part IV

NORTIFlow
The main goal when designing NORTIFlow was to create a tool that helps introducing scientific workflows into a SOA environment. The orchestration layer of SOA stacks is normally designed to process automated business workflows with little user interaction. Processing data centric scientific workflows however has different requirements, as worked out in a previous chapter.

Existing scientific workflow management systems, although several of them are capable of accessing and using web services, are largely made for creating, modifying and executing workflows within the workflow management system (unless they produce reusable WS-BPEL code). This not only forces the user to purchase a copy and install it on the local system but also makes it difficult to build scientific SOA-based applications, web sites or portals that just want to include these workflows, but execute them within their own application/web site or in other words to allow a workflow to be treated as a native part of an application. For SOA solutions it is crucial that functionality of all participating components is offered as platform independent (web) services as it is one of the main important ideas behind SOA to be independent of platforms and concrete systems. SOA standards furthermore are extremely useful for sharing workflows. Metadata and service descriptions for example can be added to those with common SOA technologies, and UDDI can be used as workflow repository to easily store and find workflows.

NortiFlow now is a scientific workflow tool that fits into the SOA paradigm by offering its functionality as SOA (web) services that can be easily integrated into own projects. A
special focus is laid on supporting interactive workflows, instead of the limited "one-shot" approach. The user can watch, save and modify every intermediate step, select only a part of the data from a result list, going back and try again with different data, send deep-copies of data to various successor nodes and more.

NortiFlow has a data centric design: loops for example are data conducted as they accept lists as input and counter. The execution of a node is performed when all input data is present. Rich provenance information is recorded that provides information about all performed actions that finally lead to the data output.

According to SOA principles workflows in NortiFlow are based on contracts only. Concrete service implementations are searched and instantiated in runtime. NortiFlow uses different levels of abstractions and supports hierarchical workflow creation. Mappings between the different levels can be defined.

Regarding the main limitations of this prototype, there are no grid capabilities implemented and performance issues as well as security issues have had little priority. Those concerns were not addressed because performance and security did not play an important role for the water quality assessment models NORTIFlow was originally designed for. In fact the goal at the design of NORTIFlow was less to create a full-fledged scientific workflow management system, but a lightweight tool to easily assembling web services to scientific workflows, where support for user interaction and SOA capabilities were the main requirements.

### 8.1 Ingredients of scientific workflow systems

Before explaining the design of NORTIFlow, we first have a look at the main issues of what needs to be taken into consideration when designing a workflow system. This can be roughly divided into first workflow creation and reengineering possibilities, and second, workflow execution issues. Workflow creation is about choosing an appropriate workflow representation - meaning the workflow language and allowed primitives and building blocks. Furthermore, workflow creation contains the possibilities of assistance at building, making extension and modifications, at design time or at runtime. Workflow execution deals with
the functional range and services of the execution and enactment engine. We will have a closer look at each one in the following sections, and next we describe how they were designed in NORTIFlow.

8.1.1 Workflow creation / reengineering

The first important design decision is to choose an appropriate workflow description model. Should the workflow system use a script based workflow language and produce that code in the background or does it use a graph based workflow language that directly is created graphically?

8.1.1.1 Workflow description models

Existing workflow descriptions can be grouped roughly into two classes: Script like (including rule based) and graph based [HA07]: Script and rules based have in common that they do not have a graphical representation and are built and described textually. Example for script based workflow languages are WS-BPEL [ACD+03], GridAnt [AvLH+04] or Karajan [vLHK07]. Rule-based workflows can be found for example in the Cactus framework [Goo07]. Other example for script like workflow description languages are the PI-calculus [WPW07], which is an algebra for describing and analyzing the behavior of concurrent systems in terms of processes, channels and names, or the Virtual data language [ZWF07] for describing both, data sets and workflow procedures.

Within graph based workflow representations there are in the main three important approaches: directed acyclic graphs (DAG), directed cyclic graphs (DCG) and Petri Nets. Directed graphs consist of nodes that are linked by some connecters (edges) that have a direction. Nodes in a DAG represent programs or processes while edges represent data dependencies. An edge from node A to node B for example indicates that data produced by A is used by B. They are in general very easy to read and to understand. The simplest form of them are directed acyclic graphs. The limitation of being acyclic makes it however hard to describe complex workflows. DCGs work as DAGs, but allow cycles. Cycles are
Ingredients of scientific workflow systems

important for loops and iterations but also add complexity, especially when arbitrary cycles are permitted (loops that do have multiple entrances).

An alternative to directed graphs are Petri Nets [Wol10] [HA07], which is a formalism for describing distributed processes by extending state machines with a notion of concurrency. Elements of Petri Nets are place nodes, transition nodes and directed arcs (connecting places with transitions). The main application for Petri Nets is in loosely coupled systems that exhibit a certain granularity of components. A common classification distinguishes three levels of Petri Nets:

- Level1: Boolean Places (e.g. State Machines)
- Level2: Integer Value Places (e.g. Ordinary Petri Nets)
- Level3: High-Level value Places (e.g. High Level Petri Nets; Colored Petri Nets)

With the Petri Nets markup language (PNML) a XML-based interchange format for exchanging Petri Nets between different Petri Net tools is available. A big advantage of Petri-Nets based workflow engines is that they can process almost every workflow pattern without any further extensions. Petri Nets are very expressive and simple at the same time and there is no need to implement special functionality for workflow constructs such as loops, if-then and synchronization points. All these constructs are supported implicitly by the Petri Net approach. Beside their ability for comprehensible visualization of processes, another advantage is the availability of formal analysis and verification techniques for Petri Nets. The formal foundations of Petri Nets allow the automated proving of criteria for workflow nets, for instance absence of deadlock [vdAvH04] [Dee07]. Comparing Petri Nets with WS-BPEL, latter one has two main disadvantages: it processes complex and rather informal semantics, which makes it more difficult to use formal analysis methods and to model workflows, especially for the unskilled end-user; furthermore WS-BPEL has less expressiveness and it does not directly support some workflow patterns, such as arbitrary cycles. One drawback of the Petri Nets approach is the fact that the graph may become very huge for complex and fine-grained systems. One solution for this is the use of hierarchical Petri Nets where one transition represents a whole sub Petri Net.
8.1.1.2 Design primitives, assistance and further functionality

After choosing a workflow description model is important to reason of how to actually create the workflow, which design primitives are available, which assistance and wizards is given and what additional functionality like verification, optimization and mining the workflow management system offers. In the end it is important to make workflow creating as easy and intuitive as possible, an aspect that we think is important to underline here: Many authors (e.g. [Shi07]) warn not to make workflow languages over complex because many of the advantages of workflows systems are getting lost this way. It should not be tried to add to a workflow language so much functionality to make it a Turing complete language, because if we do so we have to ask ourselves why not to use any of the many already existing all-purpose languages instead to connect our services and sub-tasks. Many script based workflow languages tend to be very complex. WS-BPEL for example is very complex and not much easier to learn than any all-purpose programming language that alternatively could be used to connect the services. Also graphical extensions to script-like workflow languages that are mapping the language elements 1:1 to graphical representations suffer from the same problem. From a SOA perspective the most important aspect of using workflow languages is the separation from implementation code, but also an intuitive usage of the composition languages, including for non-programmers, is an important goal. For sharing and adapting workflows it is even more important to keep them easy understandable, and that the composition of services is not hidden in complicated source code. There obviously is a conflict between offering constructs for fine-grained control flow modeling and simplicity. Workflow languages should find a way to allow simple creation but nevertheless offer full needed functionality. To solve this conflict it is necessary to hide the real complexity of workflows. Various approaches can be used to do so:

Different abstraction levels A good strategies for hiding complexity are using different abstraction levels for the description of workflows that support varying degrees of reuse and adaption. A workflow composition system should always present workflows to the user at an appropriated level of abstraction. For example three steps of workflow creation are very common: (1) Template, (2) Instance, and (3) Executable. A flexible workflow composition
system should accept partial workflow specifications from users and automatically complete
them into the next abstraction level by automatically adding these steps where the format
requirements for experiment critical components are declaratively specified and when the
component library includes appropriate components.

**Contract based** Also here contract based solutions help to reduce complexity. Implemen-
tation details like the concrete call order of services are hidden what opens the possibility
to just concentrate on the required input and the obtained output. Workflow management
systems are perfect for composing with such contract based components. A compact and
meaningful visualization for those however still is an important challenge: components that
for example are internally composed of hundreds of sub-tasks can be represented as a single
component hiding all the internal complexity. But when the component then also is wanted
to be enlarged to view and modify the details of the inner process, the workflow manage-
ment system must allow this and switch to another view. Contract-based approaches also
makes it easier to modify workflows later, and is especially useful to allow for reengineering
workflows at runtime, exchange implementation of partial sub-workflows, etc.

**High level language constructs** Traditional programming languages experimented an
evolution in generations, from coding directly binary in machine language, over assembler,
up to the 4th generation languages nowadays. With each new generation achieving the same
goal required less code than before, which is in the main due to introducing new language
constructs on a higher abstraction level. These language constructs hide the complexity
behind the scene. Similar to this, also workflow systems can introduce language constructs
that help to hide and manage complexity. One example of this is implicit parallelism
"Scientific workflow systems aim to provide a simple concise notation that allows easy
parallelization and supports the composition of large numbers of parallel computations,
therefore they may not need all the constructs and features in a full-fledged conventional
language, and implicit parallelism is preferred to explicit parallelism specification, as the
latter requires expertise and attention to the details of parallel programming, which may
be difficult for end users." [ZRF08]. Another example is the loop feature of NORTIFlow,
where loops are created by simply switching on the loop mode of node design primitives.
8.1.2 Workflow execution

Workflow execution is about automating the execution of processes by scheduling, controlling and monitoring the tasks. The workflow management system is responsible for inter-component communication, logging, monitoring, failure recovery, checkpointing, persistence and more. The main concerns a workflow execution engine has to address are:

8.1.2.1 Enactment

Enactment means instantiation of the workflow onto a grid. This includes mapping, scheduling and managing parallel execution. Executing workflows using different enactment services is given less emphasis in business workflows, which will typically be carefully negotiated and agreed by the business involved and executed in a fixed known context. In contrast, a scientific workflow will be shared and evolved by a community and executed by many individual scientists using their favorite enactment service.

8.1.2.2 Data Placement

Data placement is managing the data and metadata that are input and output by the tasks in the workflow. Currently workflows mostly use file-oriented inputs and outputs. Data placement includes allocating space, transferring data, releasing no longer used resources, to continue incomplete files or to try alternative transfer protocols. In most workflow management systems this is handled internally and only few use an external separated data placement tool for this job. An argument for latter approach is that modern scientific workflow systems need to set large scale data management as one of its primary objectives to for example ensure that data movement is minimized by intelligent data-aware scheduling both among distributed computing sites.
8.1.2.3 Persistency

Persistency is important especially for long running workflows, but also for the case that the computer crashes or to resume a workflow at a later date for some reason. Checkpointing is a technique for preserving the state of a process in order to reconstruct it at a later date. This also helps for fault handling.

8.1.2.4 Provenance recording

The importance of provenance recording for scientific workflows has already been treated in the first part of this chapter. Reproducibility of results is one of the key requirements of scientific work. Together with the more dynamic nature of workflows in science due to experimental usage of those, it is important to record all conditions that are important to get to a certain result. Ensuring reproducibility enables the re-execution of analyses, and the replication of results.

8.1.2.5 Fault handling

In the case of error at the execution of a workflow there are different fault handling strategies a workflow management system can support or let the user chose from. The most important strategies are:

1. Retry/replication: Retry the invocation of the component that failed a number of times.

2. Checkpointing: Go back to the last saved state of workflow execution and try again

3. Alternative tasks: Try an alternative component or also just skip the node that produced the error. For this strategy it is necessary to allow the user at the definition of workflows the specification of optional nodes or alternative paths and components. For a contract based component it is also possible for the workflow management system to look for an alternative component that fulfills the same requirements automatically.
8.1.2.6 User Interaction

Last there is the question about which possibilities of monitoring and interaction is offered to the user. Some possibilities are:

- Introduce data (strings, numbers, yes/no, others)
- Making selections
- Watch and/or save intermediate step data
- Modify intermediate step data before going on
- Go back and repeat execution with different data
- Pause/Stop/Repeat/Cancel workflow execution
- Skip a node
- Reengineer the workflow graph at runtime

8.2 Design of NORTIFlow

NORTIFlow consists of web services and a graphical editor. Until now, the web services are only for executing workflows. Workflow creation has to be accomplished with the graphical editor belonging to NORTIFlow. The basic idea of a SOA workflow management system however is to offer its whole functionality as services that include building and modifying workflows at runtime. Only this way the whole functionality can be integrated in extern software and web portals. However the solution for now is to create the workflows with help of graphical extensions and only to execute them with help of an execution service. In the following it is described how the above enlisted ingredients of a workflow systems are realized in NORTIFlow and which design choices for each of those aspects were taken.
8.2.1 Workflow Creation with NORTIFlow

For workflow creation NORTIFlow has adapted the circuit model explained on page 77. Components in NORTIFlow are represented as nodes with a number of input and output pins that have to be connected.

8.2.1.1 Workflow abstractions

NORTIFlow uses two different types of workflow abstractions, both using a slightly different workflow description model:

1. Interface based DCG with additional notations for expressing conditions (IProcs)
2. DAGs that directly connect concrete web services (WProcs)

IProcs consist of regular interface nodes (INodes), ForkNodes to express conditions and control flow, as well as connection primitives between them. Loops are not represented by own primitives, but any INode can be executed in a loop mode. WProcs are directed acyclic graphs that represent the execution order and parameter passing of concrete web services. Each execution step needs to be associated with a data viewer that beside viewing data also should offer the possibility to modify data. Because not every step necessarily need to be supervised, it is also necessary to select one of the following modes for each execution step: Show (only show intermediate result), Edit (shows the intermediate result with the possibility of altering it) or Bypass (don’t show, just pass the result to the successor node). When building WProcs the user is assisted at finding appropriate services that match. This is accomplished only by the analysis of data types, further metadata isn’t included yet.

8.2.1.2 Design Primitives

The following are the basic design primitives to create workflows with NORTIFlow:

8.2.1.2.1 INodes

An INode represents a logical operation to create output data from input data. Input and
output pins are representing these data. Internally it can take various execution steps to produce output data from the input data, which however are not visible at this level of abstraction.

Figure 8.1: The left figure shows the INode definition window. For the input and output pins it is possible to choose from a list of known data types or to create new ones. The right figure shows how the above defined INode appears in the process building window. A unique color is automatically assigned to each data type, which makes it easy to see matching types.

8.2.1.2.2 ForkNodes

ForkNodes are used to express conditions. Like INodes, ForkNodes have a number of input pins that represent the input data needed for the evaluation of conditions. Unlike INodes they do not have output pins. Instead they have a number of branches at their place that can be connected to successor nodes. Basically, ForkNodes evaluate the input values and return a single number, indicating the branch to choose.
Fork nodes can be used to model if-else (true/false) decisions or to model cases with more possibilities (comparable to the switch/case construct of many programming languages). They can also have more than one input pin.

8.2.1.2.3 Connectors
Connectors are used to model data flow. Output pins are simply connected with input pins of the same data type. All input pins must be satisfied in order to execute a node, while it is not necessary that all output pins have a connection to process them any further. If they remain vacant this just means that this output data is not needed for further calculations, or already is the final result. Output pins can be used as input data for the input pins of various successor nodes (fan-out).
8.2.1.2.4 Control flow lines

Creating workflows in NORTIFlow is not pure data modeling, because it is also possible to define control flow with control flow lines (Figure 8.4). Control flow lines always start at the output branches of fork nodes, and are going to INodes (unlike data connectors which always end at input pins).

With these four basic design primitives it is already possible to create IPoc process definitions. What however still is left is to define implementations of the interfaces.

8.2.1.2.5 WProcs

IProcs, with INodes and ForkNodes are purely interface based specifications and represent a high level view to the process. The internal steps of how to generate the output data from the input data of each INode can involve many steps and is hidden at this level of abstraction. Nevertheless the INodes need to be implemented, and that's what WProcs are for. WProcs are DAGs who represent a specific call order of concrete web service methods.

An example of a WProc process is shown in Figure 8.5:
Figure 8.4: This figure is an example that includes branches and control flow lines. The station of the first component (SelectStation) here is checked to be valid. If an invalid (empty) selection was made, the control flow repeats the station selection (note black control flow lines). If the choice is valid, the successor component "EvaluationModel-Simple" is activated, which causes the red point turning green, which means that it can be executed as soon as all input data is available.

Figure 8.5: Example of an WProc proces. Also here, the same output can be used as input for various successor nodes, as well as nodes can gather input data from various predecessor components (fan-in and fan-out)
To later enable tracing the complete chain of data production, it is necessary to associate each step a data viewer (Figure 8.6). The viewers can be specialized to view (multidimensional) matrices, lists, trees, or, in case that no special viewer exists a general viewer for all kind of objects can be used. Latter one then shows all public properties of the object. Also a specialized chart viewer is available, which is capable to show some results graphically, as well as a viewer for downloading files, that can be the result of some components.

Figure 8.6: This figures show the selection dialogs to assign a viewer and a result treatment mode to each processing step

8.2.1.3 Implementing contracts

We explained that WProcs are the design primitives that abstract concrete web service call sequences that implement the interface based contracts of IProc elements. To define these implementations the WProc processes now need to be mapped to input and output pins of the INodes they implement. The mapping procedure consists of assigning each input and output of the INode a node of the WProc. A wizard to help accomplishing this is at hand in the graphical editor. Figure 8.7 shows a screenshot of the wizard.
The mapping of at least one implementation to all INodes of the graph makes an abstract IProc workflow definition a complete executable workflow. When mapping interface based (IProc) sub-workflows to bigger interface based workflows (also IProc), also a hierarchical creation of workflows becomes possible. If various implementations are defined to implement a given INode, the system has more possibilities to choose an implementation and is more error tolerant.

8.2.1.4 Service registration, assistance and further language constructs

8.2.1.4.1 Loops

Every INode can be executed in a loop mode. Loops in general always need an execution counter or canceling condition. The counter in most programming languages is a fixed number. As NORTIFlow has a data centric design, loops in NORTIFlow however are conducted by lists of data arrays, where the number of elements determines how often the loop is repeated (comparable to the foreach-language construct in some programming languages). The array that is used as counter needs to be selected when switching the loop mode on. Figure 8.8 shows how the appearance of an INode changes when it is put into the loop mode.
It is also possible selecting various input pins as counter. In this case they however need to have the same number of elements. In NORTIFlow, an "[ ]" after a data type indicates an array or data type. In consequence of loop activation, all output data types of the INode switch to the array version of themselves. This easily is explained because after executing the INode repeatedly, also each output data is present various times. Note in figure 8.8, how Station from the input pins changes to Station[ ] (the input pin that was selected as counter) and with it its color, as well as both output pins change to their array version.

8.2.1.4.2 Registering WebServices

Before starting to create WProcs it is necessary to acquaint the WFMS with available web services. Normally the idea is to maintain the repository of available services by reading in WSDL descriptions of web services. A further possibility would be to establish a connection to external repositories, based for example on UDDI. These functions are however not implemented yet in the current version. At the moment the only possibility to register a web service is passing instantiations of wrapper classes (for example those automatically created by Microsoft Visual Studio when adding a web reference) to the corresponding registering methods. After that they appear in the web service choosing form, from where it is possible to select the public methods they expose.
8.2.1.4.3 Assistance for creating workflows

For creating IProcs, very little assistance is needed. The dialogs to create these are very intuitive and easy to use. Colors indicating data types that match, loops can be created by simply setting an INode to a loop mode. The mapping assistant was already presented. Also creating WProcs works graphically is very intuitive to perform. Figures 8.9 and 8.10 show some screenshots of the WProc construction process.

![Web Service Selection Form]

Figure 8.9: This figure shows the assistant for choosing web service methods. After choosing a method and closing the form, the method appears as a new node (in blue), with all required input parameters as new nodes in red, which indicates that it is necessary to choose a implementation method for them.

The construction process consists in assigning all incomplete (indicated by colors) specifications a web service method that deliver the required data, a viewer and a result treatment mode until the definition is complete.
8.2.2 Workflow Execution with NORTIFlow

Regarding design decisions for execution, here a brief explication of how the above enlisted concern regarding workflow execution are handled in NORTIFlow:

8.2.2.1 Enactment

As NORTIFlow do no implement any grid capabilities, the question about an enactment mechanism becomes obsolete.

Execution of NORTIFlow workflows is possible in three ways: If it is desired to include workflows into own applications the NORTIFlow dll (only available for the DotNet platform), needs to be linked with the project, or, if working with another platform, the execution web services have to be used that can be used from any application that is capable to invoke web services, be it desktop application, web applications or mobile devices. It is however further required to implement all required data viewers in the target platform according to the implicit viewer contracts.
A third possibility is to execute the workflows is directly from the graphical editor, which however has the drawback of many other workflow management system, that it is necessary to have a copy of NORTIFlow installed on the local system, and there is no way to integrate the workflow execution into third party programs.

### 8.2.2.2 Data placement

Unless most workflow systems the input and output data of NORTIFlow components are not file based. The results are passed directly from one component (i.e. web service) to another. Internally data types are converted if necessary. If for example a data type as result of one web service is compatible as input for another web service, but will not be accepted for having another name or being defined in another namespace, the system converts the data type to the required one by cloning all publically accessible properties.

### 8.2.2.3 Persistency

Persistency is supported by NORTIFlow. Workflows can be saved at any time to continue execution another time. The saved workflows contain both, all so far produced data as well as the workflow description. So if the workflow used to produce the data of a saved workflow have been edited in the meantime before continuing execution, the original version is saved, which is important to guarantee reproducibility.

### 8.2.2.4 Provenance recording

Rich provenance information is recorded. The system logs every action, from choosing INodes to execute, selecting implementations and associates to every input and output pin a snapshot of the current log file. This way, for every set of data its way of constructing can be backtracked. There are occasions, where the system passes control over to the data viewers and cannot continue its observation. Here, the data viewers are responsible to log every action, modification and selection made to the data set and to report it back to the workflow system before continuing with the next execution step.
8.2.2.5 Fault Handling

Two of the three mentioned fault handling strategies are implemented. Repeat execution n times, and if this won’t help, try alternative implementations. Automatic check pointing, as further strategy, is not implemented. It is only possible to make check points manually, by saving execution and loading the file to repeat at the saved check point in the case of error.

Figure 8.11: This figure shows the dialog to configure the fault handling strategies. In the same dialog it is also possible to select an implementation selection strategy.

8.2.2.6 User interaction

The user has complete control over the execution process. In general every single step is observable thanks to data viewers associated to each node. Because this is much more control than usually desired (if not for looking for specific errors in a debugging mode), and a partly automated execution often is desired, the scientist chooses nodes where he wants to check data, setting all the rest of the steps to ”bypass”-mode. If results do not conform to expectations, he can go steps back, repeat execution while having a closer look at the intermediate steps or trying alternative implementations. It is even possible to alter the execution graph in runtime, to make some experiments.
Figure 8.12: The execution window is partitioned in five sectors, instruction/information, data viewers, process graph, logging window, and forward/backwards-buttons. With the forward/backwards buttons it is always possible to go steps back and repeat execution with modified data or different selections. The logging window will record every action for later being able to reproduce the obtained results.

Figure 8.13: The process graph window has two tabs. The first shows the whole process (see above) with the INode that is currently executed marked in green. The second tab (see this picture) shows the selected WProc process (which includes several execution steps), that delivers the implementation for the selected INode.
Figure 8.14: Execution stops at predefined nodes, and intermediate results are shown.

Figure 8.15: Some viewers are capable to show results as graphics.
Figure 8.16: In the last example we switched in a running process (note SelectStation and GetRefTable are painted in grey, which indicate that they already have been executed) back to the edit mode, and let us show the data of the "Station" input pin of the EvaluationModelSimple component. There we can view the selected stations and if we so desire switch to provenance information log to these data. It would also be possible to add new nodes or make modifications with the connectors and then continue execution.

### 8.2.2.7 Execution order

The last mentioned possibility of editing workflows at runtime becomes possible by the fact that the system does not create a determined execution order when executing a workflow. In fact, WProcs implementations do have a strict linear execution order, but in-the-large (IProcs) after every execution step the next node to executed is searched in run-time. Two conditions need to me met: (1) data for all input pins are present (2) the node needs to be set to the active state (fork nodes have influence on this) If various nodes meet these
conditions, the system could also execute these parallel in different threads to increase performance. This is however not supported yet and left for the future.

8.2.3 Executing workflows with help of the execution web service

For executing workflows with help of the execution web service, it is necessary to perform the following steps: First we need to call the load methods, passing the name and location of the workflow file. Then we need to set the execution parameters, the same we did with the execution dialog in the graphical editor what is the error treatment strategy and implementation selection strategy, as well as preferred implementations in the case of ”user select”. After that, we can enter the simple basic execution loop: ”ExecuteNext” - ”ShowResult” - ”SetModifiedResult”, which is repeated until reaching the end of process execution. ”ExecuteNext” returns data from the next execution step, where the treatment mode is set to either ”Show” or ”Edit”. It also contains a string with provenance information belonging to the data. This data now needs to be visualized by passing it to ”ShowResult”. This method needs to be implemented by the client system, which is responsible to implement the data viewers. The data viewers also should allow modifying data and implementing a method for obtaining the modified data, which subsequently is passed to ”SetModifiedResult” of the execution web service. After every ”ExecuteNext” call it is also possible to request bitmaps showing the current state of the process graphically, like in the graph window in the graphical editor. Of course at every time it is possible to leave the basic execution loop and doing steps backwards before going on again. Also methods are available to save the current execution state, and everything else that can be done from within the graphical editor.
Part V

Norti-Online
The aim of the Norti-Online web portal is to establish a web portal that offers workflows and services around the topic of empirical water quality assessment. The initial idea was to offer a web portal for the full development cycle of implementation, refinement, further development and execution of workflows related to the assessment of water quality as well as the (comparative) visualization of results. By its close connection to the SOA based scientific workflow management system NORTIFlow the portal should become a kind of online workflow system for scientific workflow modeling. The current state of the development of NORTIFlow however only includes execution web services (see last chapter - the functions for creating and modifying the workflow are only accessible through the graphical editor), therefore the portal is currently limited to those functions the NORTIFlow services offer. The portal works as a web user interface to NORTIFlow as the intrinsic core of the system and executes workflows created with NORTIFlow.

The other important part of the portal are the web services implemented for this system that deliver required functionality for water quality assessment models. These include some mathematical-statistical services, data acquiring services to straightforwardly retrieve data from heterogeneous sources (i.e. geomorphic, biologic, or land uses data, among others), as well as some all-purpose services. The web services are designed to be reusable by anyone who deals with similar issues and many of the offered services may also be useful for other domains.
9.1 The Norti legacy software

The aim of NORTh Spanish Indicators system (in short NORTI) project, which began in 2004, is to develop models that provide quantitative and qualitative biological assessment of water quality, in order to implement the respective aspects required by the WFD. The north Spanish basin is currently administratively divided in three sectors: the Miño-Sil basin, with a surface of 17,757 km², which principally includes Miño and Sil river basins, two of the most important of the north Spanish region. It is managed by the Hydrographic Confederation of Miño-Sil. The Cantabric basin, with a surface of 22,452 km², is managed by the Cantabric Hydrographic Confederation, including a more heterogeneous set of river basins, whose commonality is to flow into the Cantabric Sea (north coast of Spain). The third sector, which is managed by the Galician regional government, includes all river basins of this northwestern region except for the Miño-Sil basin (the reason of this is that this last one is transregional and international). This sector was not currently included into the project. A map of the area, with the two involved sectors grayed, is shown in figure 9.1.

Figure 9.1: Map of the area covered by NORTI project (in gray). Sampling stations are represented by points.
It is worth pointing out the heterogeneity of the freshwater bodies in the North Spanish basin. From a geomorphologic viewpoint, 21 different types from a total of 36 possible types of river stretches are present in this basin, according to "System A" classification established by the WFD. This is also outstanding from an ecological viewpoint, with relatively high endemism with regard to the benthic fauna [VM06].

Due to the heterogeneity of the north Spanish water bodies the design, implementation and calibration of several models are necessary. Two models were designed following the Reference Condition Approach respectively using macroinvertebrates and diatoms biological quality elements or indicators [PD05]. One multi metric model based on macroinvertebrates [DPG10] was also developed, with both aims: providing complementary assessment information of water quality and allowing intercalibration and development of water quality measures comparable at European level, as encouraged by WFD. Those models are currently stable and operative, but their evolution and the addition of new models are already planned to answer improvement requirements, as well as to test and eventually incorporate new models and approaches regularly provided by research in this field. These three models currently available were built using a large collection of different datasets resulting from seasonal data collection campaigns carried out from the year 2000 until now, in 683 different sampling sites distributed across the entire basin. Collected data included physicochemical, geomorphological, and hydromorphological status. It also contained data resulting from biological samplings, consisting of benthic invertebrates, abundance at family level, and diatoms abundance by species. Land use and cover data per site were gathered from CORINE Land Cover database [BFO00] elaborated by EEA (European Environment Agency). Those different dataset have to be retrieved from heterogeneous sources. Data sources multiplicity and variability are, in fact, a constant aspect of this project, as since the beginning its development is coinciding with important changes in water information management policies, due as much to WFD implementation at national and European levels, as well as to organizational and technological changes at regional agencies level. Currently, some dataset are being integrated into regional GIS systems, while other ones will probably be migrated to centralized national repositories. NORTI is implemented as Microsoft Access
9.2 Legacy software reformation steps

With the arising of new requirements that more models should be added to the system, more data sources should be included and a future integration of a GIS was foreseen, the decision was taken to re-write the whole application using a SOA-based approach, in order for the new version to be better prepared for such modifications and more future extensions. The problem of allowance of flexible composition finally lead to the development of NORTIFlow, which was already described in the last chapter. Large parts of the old NORTI system could be reused to implement the new system, which is described in the following:

9.2.1 Data Base reformation

The NORTI data base had been designed for supporting two implemented water quality assessment models, one based on invertebrates, another based on diatoms. It contained several tables to store information related to those models. In order to be more generic and allow for the inclusion of a not specified number of new models the tables for the different models have been generalized to meet the requirements of both. Furthermore some attributes in generic tables like "Measuring station" were taken out as they referred to concrete models. An example is "is reference station for diatoms". Instead, a new table "Model", "Station", "Group", "IsReference" was introduced to support all models that work with the reference approach. The reformed data based finally looks like this (Figure 9.2).

After that the data access layer of the new system could be implemented. Data access services to obtain all relevant data for the implementation of water quality assessment models have been implemented and stored in the "Norti PortalDataBaseServices" web service.
Legacy software reformation steps

Figure 9.2: NORTI data base
9.2.2 Inclusion of CAS Engine

One of the requirements for the Norti-Online web portal have been to offer a web service that permits access to a CAS engine, which can be used for many of the complex mathematical calculations needed for the implementation of water quality assessment models. For this, the web services from the IMO Matlab project [GR10] have been used, which encapsulated Matlab and offers web services for access to Matlab. With help of the IMO Matlab library, a collection of frequently needed functions have been implemented and stored in the StatisticalUtilitiesService web service. These services include similarity algorithms (i.e. Bray Curtis dissimilarity, Euclidean, or Jaccard distances), matrix operations, trigonometric and logarithmic functions, calculation of probability ellipses, principal component analysis, logistical regression algorithms and more.

9.2.3 Encapsulating code into services

Last less generic code, specialized for the different water quality assessment models, was extracted and put together with some new implemented services into two further web services, NortiOnlineMetricsServices and LogisticalModelService. NortiOnlineMetricsService contains some single well-known biological metrics and indexes (i.e. richness and frequency measures, Margalef metric, evenness, Shannon diversity index, Bray Curtis index, etc) as well as methods for the multimetric evaluation model, specifically developed for the NORTI project. The Logistical model service contains methods for assigning measuring stations to previous defined reference groups by means of station attributes and samples taken in this station.

9.3 NORTI-Online web portal system design

After completing the first step toward a SOA-based solution that was to encapsulate all code of the legacy system into services, there is still left to manage the coordination of service invocations and to connect the system with a user interface. These tasks are preformed in
the orchestration and user interface layers of the SOA stack of the system. The resulting overall architecture of the developed system is shown in Figure 9.3. The system is divided into four main layers: Data Layer, Service Layer, Orchestration Layer, and User Interface. Some of them are further divided into sub-layers. In the next subsections each layer is described.

9.3.1 Data Layer

The lower layer of Norti-Online web portal design is the Data layer, which consists of the data backend, that is, physical databases or other types of data sources, and, on top of these, the data sub-layers devoted to data selection and data preparation, whose services are explained further on. All the services of the layers above have to access data via this layer. This way, the physical implementation of the databases is completely hidden. If in the future the usage of another data base or the consolidation of various data sources is needed, the data layer services can be modified, or even substituted as a whole, while the rest of the program remains unaffected. In the Data layer, services are grouped into two sub-layers: data selection sub-layer and data preparation sub-layer. The design decision that leaded to this division was the different specialization level of each type of service. In data selection sub-layer we have services that do simple queries like retrieving sites or samplings depending on some conditions. The services in this sub-layer are more generic than those in the next sub-layer, as they are used by many services of the layers above. In data preparation sub-layer there are more specialized services, oriented to a certain type of task. Those services usually accept results from the data selection services as input, and they typically prepare those data for some specific task (i.e. to build a matrix of samples able to serve as input for distance matrix calculation algorithms, for example). While data selection services are designed with a minimalistic approach, retrieving only the strictly necessary information to identify entities (typically key data fields), data preparation services may need to query some additional data directly from data sources, which justifies their location into the Data layer.
Figure 9.3: The Norti-Online web portal architecture
9.3.2 Processing service layer

The next layer is the Processing services layer, which contains a collection of reusable and platform independent services that implement the core functionalities required to further build higher level and more complex functionalities. The services contained in the Processing services layer are certainly the most relevant services in terms of functionality with regard to water quality assessment, which is the main aim of the Norti-Online web portal. In contrast to the services of the other layers, the services of this layer are mainly data computation services implementing transformation, complex algorithms, or data conversions that can be common to different water quality assessment models. Granularity of those services is an important issue, as coarser services will be simpler but less reusable, but finer ones can penalize performance and can make composition excessively complex. The services of this layer can be grouped in the following categories:

- Ordination services: those services perform data analysis, dimension reduction, classification and ordination tasks, using algorithms like principal components analysis, multidimensional scaling, k-means, logistic regression, etc. They are typically used to determine relevant variables to distinguish sites, and to define groups of sites based on the similarities of some variables.

- Group assignation services: those services allow assigning sites to previously defined groups, usually according to a particular model. They include for example discriminant or logistic functions. For example, to assign new test sites to one of the predefined site groups, the macroinvertebrate and diatoms reference condition models use a service that implements a logistic function whose coefficients have been previously obtained by means of a logistic regression, and are stored in the data backend.

- Metrics and indicators calculation services: they provide single well-known biological metrics and indexes (i.e. richness and frequency measures, Margalef metric, evenness, Shannon diversity index, Bray Curtis index, etc). Besides those services, which are rather generic, there are more specific services that implement algorithms designed for particular conditions. Examples are the EQR indicators or the multimetric evalu-
ation model, specifically developed for the NORTI project, which reuse some generic services of the same layer, like biological metrics calculation services.

- Common calculation and utility services: they offer generic calculation functionality that can be required in environmental models, like similarity algorithms (i.e. Bray Curtis dissimilarity, Euclidean, or Jaccard distances), matrix operations, trigonometric and logarithmic functions, etc. Also in this category are services more oriented to formatting and preparing data presentation. Some of them, for example, facilitate putting and managing data and graphs in worksheets (i.e. Excel), or perform data conversations or simple transformations (for example to calculate coordinates in order to build probability ellipse graphs).

Those services provide the basic building blocks to subsequently define complete models by composing them, but they don’t implement application-level flow control, data access or user-oriented visualization functions, as those functionalities are implemented by services of the other layers. Models, for example, can be built by combining the required services of this layer and of Data access layer in a workflow. This latter task is supported by the next layer of the system, which is the Orchestration layer.

### 9.3.3 Orchestration layer

In the Orchestration layer, the calculation and processing services from the Processing services layer, and the data access services from the Data layer, can be combined into workflows that model application-level functionalities. This layer relies on NORTIFlow and its execution and workflow creation and reengineering services. Defined workflows are stored in a file and passed to the execution service which executes the service until user interaction is foreseen. In this case the user interface layer has to take over control to present the data to the user and return the modified data back to the execution service. How to define processes with NORTIFlow has already been treated in the last chapter of this thesis. Next the user interface layer is described more detailed.
9.3.4 User interface

The top layer of the system is the user interface layer. User interfaces for different target systems can be implemented on this layer that can be user interfaces for desktop solutions, web solutions or mobile devices. The user interface components are typically platform-dependent, as they have to directly interact with client system presentation capabilities. Therefore they have to be implemented for each desired target platform. In order to execute workflows they have to implement two things: the application controller and data viewer components. The purpose of the application controller is to manage the execution and modifications of workflows, as well as to react to other user actions like stop execution and save the current state for continuing its execution on a later time. It also for example can ask for pictures from the execution service showing the current state of the workflow graphically. But first of all it is responsible of loading a workflow upon user request and to trigger its execution. Then, if during execution some data have to be shown to the user, the application controller is responsible for directing results to suitable data viewers. In the case that data viewers allow to modify, extend or select a subset of the result, the application controller also have to return the modified result back to the workflow execution service before going on with process execution. The data viewer components on the other hand are responsible to show intermediate data and final results to the user, as well as to allow the user to modify it to make selections of a subset of the data. Until now, a set of five generic viewer components were designed for the Norti-Online web portal, that we identified as commonly useful, which are:

- ListView: To view a list of objects with the possibility to select multiple elements, to modify elements or to extend the list by new elements. A variation of it ListViewSingleSelect forces the user to select only one element from the list to go on.

- MatrixView: To show a data matrix, with the possibility to edit the cells and to extend the matrix. The data type of the cells can be either numeric or string. Each matrix furthermore may have a column header and a row header to store further information.
NORTI-Online web portal system design

- **ObjectView**: A generic viewer to view a single object. All public properties are shown as a new line in the style `PropertyName: Value`.

- **ChartView**: To visualize diagrams and graphics. The component is most suitable to show charts, but also the probability ellipses which are the result of some of the implemented water quality assessment modes are visualized using this component.

- **FileView**: To download files. If the result of a web service is a file (for example a worksheet file), the File View allows downloading it to the client.

- Furthermore there is a container called `ViewerContainer` that can be used if the array version of a viewer is needed. When for instance instead of a matrix an array of matrices need to be visualized, `MatrixView` in combination with `ViewerContainer` can be used.

All viewers at least have to implement at least the following methods of the viewer’s contract:

```java
public interface IViewer {
    bool ReadOnly
    object GetResult();
    string GetLog();
}
```

The property `ReadOnly` indicates, if it should be allowed to modify data or not. `GetResult()` is used to ask for the data after potential modification. `GetLog()` records all changes made to the data. The application controller is responsible to call this method and adds it to the provenance information of the data set.
9.4 Defining water quality assessment processes

Now with all required services added to the repository of the Norti-Online web portal, the water quality assessment model processes have been defined using the graphical process editor of NORTIFlow.

9.4.1 Multivariate analysis based on invertebrates and diatoms

The basic sequence of processing steps for both diatoms and invertebrates models are the same. Both rely however on two sub-models (ordination model for forming groups and the group assignation model) that need to be executed previously in order to work correctly.

The basic process looks like follows:

![EvaluationModelSimple.png](attachment:image.png)

Figure 9.4: basic water evaluation process for the invertebrates and diatom model

First the select station module is executed to select the station that should be evaluated. Various implementations for the select station module exist that allow for selecting a station by different criteria. Further output nodes of the module are the selected campaign and the evaluation model (invertebrates or diatoms). The GetRefTable module gets from the data access layer a table, which contains information about which stations of which models are
considered to be reference stations as explained in "data base reformation". After that the
main component "EvaluationModelSimple" is applied and as last step follows a module for
visualizing the resulting confidence ellipses graphically. The main component was defined
in two versions:

Figure 9.5: Version1: Nodes in Orange correspond to the four input nodes. The
processing steps then are the following: Create sample matrix - Log10 Transformation
- Transpose - BrayCurtis - MDScaling - CalculateEllipses

Figure 9.6: Version2: Alternative definition

The alternative definition hereby allows a more fine grained control over especially the first
step of the creation of the sample matrix. It gets first the group of the station. With this
Defining water quality assessment processes

information it gets all stations of the group. After that the SampleID and SampleIDs of reference stations are requested. With this information the data matrix is created, allowing additionally to choosing a sample format for this process. Then the process continues with the same steps as above, except for allowing choosing the MDScaling method before applying this module.

The same process was defined for evaluating various stations at the same time and subsequently calculating EQR values (figure 9.7).

![Figure 9.7: Evaluation of various stations with calculation of EQR](image)

### 9.4.2 Metrics/Multimetrics

To also give another example from the metrics/multimetrics area, here exemplary the defined multi-metrics process (figure 9.8):
As with different implementation of the main module of the first process, also the multi-metrics module can be defined allowing a more fine-grained control over the process, which allows for example choosing exactly which metrics are used and combined how to calculate the multi-metric value. The shown processes in this chapter were exemplary. More processes have been defined, e.g. for single metrics, logistical model, for calculating abundance/frequency and presence matrix as well as converting one to another.
Using the system for own programs

9.5 Using the system for own programs

Services and workflows of the Norti-Online portal, which includes NORTIFlow can be used for own programs for any platform, for web based and for desktop solutions. It is easy to add more evaluation models to the portal by defining own services and combine them to own workflows. To execute workflows of the Norti-Online web portal from within own projects it is only necessary to implement the application controller for the target platform as explained above, to load the process and then entering the basic execution loop, as well as implementing the data viewers. Latter ones need to implement certain functionality to be capable to show and to let modify the data. Currently two versions of those components have been implemented and are available, for client desktop applications on the .Net platform, and for ASP.Net web applications. Figure 9.10 shows a prototype version of a web application using the Norti-Online web portal. The application uses the ASP.Net version of the viewers and application controller that work within a browser.

Figure 9.10: Norti-Online accessed via ASP.Net components
10 Conclusion

One big challenge for the future of software engineering is to offer support for easier connecting different systems and permitting collaboration of different programs, domains or people and organizations of the same domain. This for example may include the definition of workflows across the systems and program boundaries and, especially for science, to include all steps of the scientific research chain into it. A technology that may help realizing that challenge is the service paradigm of software engineering called SOA, which not only is useful and promising for the business sector where it derives from, but also for a large range of different other domains as well.

Our contribution towards that goal is a case study in the environmental modeling software domain, where we analyze how to offer water quality assessment models in a service oriented manner. With the lessons learned in this case-study we contribute to the research that is still needed in this area to determine the best practices, rules and patterns to effectively implement the SOA paradigm in scientific domains like environmental modeling. A general problem still is the complexity of the SOA world and the sheer number of technologies one has to make oneself familiar with in order to design a SOA based software system or just to get an overview. Many authors (e.g. [AZE+07] or [LS] ) complain about that. Here, exemplary, a statement of Dave Thomas [Dav07]:

"In the rush to create a middleware platform dependency larger than your current legacy platform dependency, vendors and their well-paid industry analysts push a plethora of complex technologies on organizations that just want to run their businesses. Some of the most talented technical experts I know find the complexity overwhelming when there are at least 5 different ways to do the same thing." Finally he concludes in the summary: "There is just
too much stuff in the SOA for most developers to absorb. It is difficult to understand what is real and what is hype. Beyond that, it is difficult to understand the performance and inter-operability of different implementations approaches. Developers need guidelines for when and where to use XML, objects and BPEL and for how to make them play together.

Apart from getting an overview over the confusing amount of competing technologies in the SOA domain, one furthermore has to find out the limits of what is supported by the current state of development in this area. For instance, the first problem that arose in our case was how to compose services into flexible workflows in an easy and natural way. We seemed to be able to choose from a variety of service composition languages and composing systems for that aim, thus, as a very suggesting choice for such composing system, we wanted to use scientific workflow management systems for that purpose. We noticed however, that existing ones for various reasons don’t fit well into the SOA paradigm yet as for example they cannot be accessed as services from outside to integrate their workflows into own programs. Although most of them indeed are capable to integrate services, they nevertheless are not designed to fit well into a SOA.

Trying to write services that are independent of concrete data sources we found out that despite the existence of numerous technologies in this problem field, none of them permits including new data sources by just defining new mappings as it was the initial idea.

A third difficult question was the problem of incorporating the user into the service paradigm. How to do this properly? We need to create highly interactive workflows, while WS-BPEL, the most frequently used language for service orchestration, is most suitable for creating automated workflows with little user interaction like needed in the business sector.

To now facilitate the understanding of how different SOA technologies and problem areas are related and where to put in mentioned problems and where to look for a solution, this thesis elaborates a theoretical fundament of SOA where the basic principles of SOA are put into the following two categories: technology agnostic design, and SoC techniques.

Technology agnostic design means using the outstanding capabilities to bridge differences between different systems and platforms to overcome platform dependencies, which actually in most cases is the main reason for the decision for a SOA based approach.
work we make clear that because every design decision may introduce some unwanted new platform dependencies, a design unaware of the dependency problem easily unmakes this main advantage of SOA, and introduces new dependencies on workflow management systems, databases, concrete linked services and more. For a true SOA, it is necessary to carefully minimize dependencies at all levels.

Technology agnostic design also includes databases and data structures and the dependencies deriving from them. Many of the current unsolved problems are a consequence from the data independence problem, and a solution for this maybe would be the biggest step forward towards an easier future for service cooperation: an uniform and in large parts automated forward and backward transformation of data to the required form coming from any kind of data sources. Even if in this work we cannot significantly contribute to solve this problem, our analyze comes to the conclusion that for true data independence two transformation steps are necessary to compensate for different database models and second for data structured in different ways. None of existing data independence technologies however fully supports both transformation steps. Existing data independence technologies therefore only allow a certain grade of data independence like for instance design patterns like data access objects (DAO), what we finally used for our implementation. Here, all data access is performed via data access objects and encapsulated into a data access layer (DAL). A change of the data source requires rewriting the DAL, but does not affect the rest of the system.

The second category of SOA principles are various SoC techniques that are compelling to be used in a service oriented architecture: A layered design, i.e. the SOA stack, the separation of interface and implementation(s), and the separation of components and composition.

When trying to design a SOA stack, one will notice that there is no consensus of which layers a SOA stack has to consist of. Many different designs and approaches are reflecting different views and philosophies to the SOA idea as well as different conditions found in different companies and projects. This for example includes the role of the ESB. It is only clear that SOA extends the traditional two or three layered architecture by some additional layers and a more fine-grained layer design. It can be expected that the ongoing development of WS-*
standards may change the view onto this topic as they may establish standard solutions for problems that today still need to be solved in an ad-hoc way for every SOA implementation.

In the SOA stack we implemented, we concentrated in the main on the novel orchestration layer where service composition takes place, as well as the often neglected user interface layer, which with current technologies cannot be integrated seamlessly into the SOA stack in the same way than other services. In contrast to traditional programming, the user interface as all other parts of the SOA should be loose coupled to the system. The problem here is that the user interface is different from services by its platform dependent nature. To solve this problem, in this work we proposed the development and usage of some kind of service contracts for the user interface. A basic example was provided by the data viewers developed for NORTIFlow.

The question of separating interfaces and implementations is about how pure interface based programming can and should be realized. Only interface based programming makes it possible to switch components and program parts later and this way contribute to gain the desired enhanced flexibility, adaptability and failure tolerance. This problem area of interface based programming furthermore includes finding a standard way of describing services with all metadata and semantic information necessary, to publish it in an easy way in public available distributed repositories from where it is easily possible to locate and use a suitable service when looking for it. The same is applicable for complete or partial workflow definitions.

In NORTIFlow we adapt the interface based approach by describing workflows on the top level in a pure interface based way. Later, concrete services that implement the needed functionality are mapped to the interface nodes, which this way can easily be exchanged if alternative mappings are available. The second problem of finding and publishing services and workflows however is not tackled in this work any further, we only argue for using established SOA technologies like UDDI for that purpose.

Finally the problem area of separating components and composition treats the already mentioned search for an appropriate composition mechanism for the orchestration layer. Here, this thesis still argues for graphical composition mechanisms as used by (scientifically) work-
Further works

Flow management systems we wanted to use, allowing for defining workflows for everyone and not only expert programmers. For this issue, NORTIFlow was designed as a lightweight scientifically workflow management system that can be integrated into a SOA due to its service oriented design. NORTIFlow was subsequently used to launch the Norti-Online web portal, being well prepared for future extensions, the addition of new assessment models, workflows, viewers and data sources.

10.1 Further works

To proceed with our research, the immediate future is to continue the development of NORTIFlow, to improve the mechanism to register services to accept WSDL descriptions and to work together with UDDI repositories, as well as to improve the trader to handle more sophisticated contracts that also include quality of service aspects or other metadata and semantic information. Especially the last issue needs some more investigation and we can refer here to the research about Semantic Web and Semantic Web Services. Also we think of adding some more possible viewers to the system for handling more complex graphics, more data structures or viewers specialized for visualizing GIS information. At long term we hope to develop the system step by step further towards a full-fledged workflow system with grid capabilities, parallel and optimized execution and scheduling, etc.

Investigation work could be done of how to extend the workflow model so that INodes can also react to and emit events. That should for example allow better incorporating the user interface. NORTIFlow already incorporates a part of the user interface by its viewer contracts, but this approach is limited to linear and stepwise execution order that is typical for many workflows, but might not be flexible enough to model the whole complex interaction with a sophisticated user interface.

Furthermore an interesting investigation work would be to use the system for some more complex case studies to study the hierarchical workflow construction options. A mechanism for meaningful visualization on all hierarchical levels must be developed, that is capable to switch different views from different abstraction- and hierarchical levels as needed. Also it
Further works

would be interesting to evaluate the NORTIFlow workflow model against the scientific and business workflow patterns that are possible to model and possibly develop some further constructs for easily modeling the remaining ones.

Last the workflow construction services need to be designed and implemented. Only those allow to access this functionality from the outside and to include also dynamic workflow construction and editing capabilities into own projects that uses the system, independent from the graphical editor that comes with NORTIFlow.

Practical extensions for the web portal may include adding more models, more services and more workflow definitions. The services and workflows to be added may also include some that not only use the offered data for evaluation models, but also allow managing data base information, to read in data from flat files or spreadsheets etc. Furthermore we want to investigate how to specially include GIS services into the portal. This is important for two reasons: First, GIS functionality is widely needed and an increasing amount of it is publically available through various projects and web pages, and second because standardization there is more advanced than in the sector of water quality assessment, what may help to develop similar standards for this sector.
Bibliography


[AJS+01] Andrea BUFFAGNI, Joanna KEMP, Stefania ERBA, Carlo BELFIORI, Daniel Hering, and Otto MOOG. A europe-wide system for assessing the quality of rivers using macroinvertebrates: the aqem project*) and its importance for southern europe (with special emphasis on italy). *Journal of Limnology, 60(1s), 2001.


Bibliography


Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, New York, 1995.


Bibliography


Bibliography


[IB] IBM and BEA. Next-generation data programming: Service data objects: A joint whitepaper with ibm and bea.


Bibliography


[TPD+] Tom Oinn, Peter Li, Douglas B. Kell, Carole Goble, Antoon Goderis, Mark Greenwood, Duncan Hull, Robert Stevens, Daniele Turi, and Jun Zhao. Taverna/ my Grid: Aligning a Workflow System with the Life Sciences Community.


Bibliography


Bibliography
