School of Engineering & Design

Electronic & Computer Engineering

MSc Distributed Computing Systems Engineering

Brunel University

A Modular Framework for Visual Data Exploration Applications

Student:	Andreas Theissler (0531 030 / 1)
Supervisor:	Dr. Ian Dear
Course Director:	Dr. Maozhen Li
Date:	April 2007

A Dissertation submitted in partial fulfilment

of the requirements for the degree of Master of Science

A Modular Framework for Visual Data Exploration Applications

Declaration: I have read and I understand the Dissertation guidelines on plagiarism and cheating, and I certify that this submission fully complies with these guidelines.

Student's name: Andreas Theissler

Student's signature: _____

THANKS TO MY WIFE SANDRA

AND MY KIDS EMILY AND JULIAN

FOR LETTING ME STAY UP LATE EACH NIGHT.

Table of contents

1. Abstract

Data generated by economical or technical processes is typically recorded on computer systems which results in a huge amount of data: databases often contain millions of datasets. Valuable information is often hidden in this data in a way that makes automatic detection impossible. In many cases the data is stored in distributed databases, which makes the uncovering of valuable patterns inside the data even more complex. If no automatic detection of valuable information is possible – this is the case if the knowledge about patterns in the data is vague –, this vast amount of data needs to be explored by humans. An effective way of doing so is applying visualisation techniques, which is referred to as visual data exploration. To be able to do so, users need tools guiding and supporting them during the exploration process. There is a variety of visualisation and data mining tools, but none that enables users to visually explore the data and rapidly extend the exploration process if the supplied capabilities are not sufficient. Typical problems of existing tools are the lack of extensibility or adaptability. So e.g. specific input formats might have to be converted to formats a given tools can process, which sometimes might not be possible. The number of techniques or algorithms integrated in a given tool is limited, which might constrain users in their exploration process.

This work solves the mentioned problems by introducing a design of a framework that can be used to configure scenarios to explore data. An extensible framework is designed and implemented that allows users to develop and add own modules in a rapid way. The data exploration process was abstracted in a way that allowed for general functionality to be integrated into the framework. The visual data exploration process can thereby be defined by the user. The framework will offer common visual data exploration functionality and support users supplying own modules. Typical users of such a

framework are assumed to be data analysts interested in rapid answers to their questions on the data under investigation. This puts the burden on the design of the framework that it shall be simple to use. The development of own modules will not require deep programming skills or software engineering experience as the framework holds the complex parts of the code. Offering flexibility and at the same time simplicity is one key problem that was solved in this work.

2. Introduction

2.1. Background

The storage capacity of hard drives increases exponentially. Data generated by economical or technical processes is recorded on computer systems and stored in databases – often distributed. Examples are web shops storing data about the buying habits of their customers or physicists recording measurements of colliding particles. The amount of data generated and stored is huge: databases often contain millions of datasets. Storing this kind of data in great detail allows for later retrieval of important information. It might be trivial to obtain certain facts like the annual sales in the case of the web shop. Other facts or trends are not explicit. They are hidden somewhere in the millions of datasets. It might not even be obvious what kind of valuable information the databases hold. Gaining knowledge from this raw data can be done by applying *data mining* techniques. The process of extracting information from large databases is also referred to as *knowledge discovery in databases (KDD)* which is defined as the entire non-trivial process of finding information inside data [Fayy96].

In cases where some knowledge about certain patterns of the data exists, it is possible to apply algorithms in order to obtain a result. Yet, data is often recorded automatically with no knowledge about the patterns the data will reveal. In those cases the data needs to be explored manually. As with current database management systems the portion of data that can be viewed at a time is quite small, advanced techniques become necessary in order to efficiently explore the data. *Visual data exploration* helps to present a huge amount of data and integrating each data set into the visualisation. Many visualisation techniques exist, e.g. parallel coordinates or prosection views. Users can choose one visualisation technique based on the characteristics of the data under investigation, e.g. the number of dimensions. The raw or prepared data is presented visually with *no*

algorithms applied to the data. The integration of the user into the exploration process is one key feature of visual data exploration. The typical steps are shown in Figure 1:

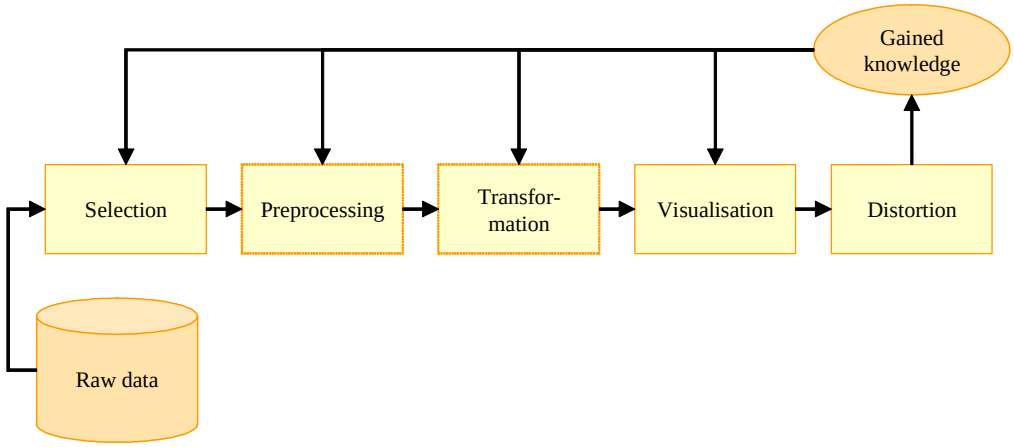

Figure 1 – Visual Data Exploration steps

Raw data is being read as specified in the selection step. This could e.g. be the user's selection of a remote database. The data might be distributed over several remote data bases, which requires additional effort to combine the data. This task could also be handled by the selection step. Optionally the data can be preprocessed, e.g. outliers or missing values can be replaced in this step. The transformation step is optional as well and can be used to e.g. reduce the amount of data by aggregating columns. The data is then visualised and can be manipulated or distorted by the user. This process can be seen as an iterative process: the exploring user might have gained some knowledge that can be used to manipulate the configurations made in one of the previous steps. The result of an exploration process will be a hypothesis about the data that might be valuable. Visual data exploration can supply results users can profit from economically, which justifies the time spent conducting the exploration. Super markets can for example place their products according to the customer's buying habits identified by analysing all customer transactions. Stored data from technical processes can reveal valuable information as well. An example could be the exploration of failures of

electronic control units in automobiles could reveal patterns that allow for the cause to be narrowed down.

With each recorded dataset the chance to gain new knowledge grows, but the complexity to uncover it grows as well. To be able to analyse this kind of data, users need tools guiding and supporting them during the exploration process. This work is concerned with the design of an extensible framework and a visual data exploration tool that can be used to configure scenarios to explore data. A variety of tools conducting data mining exists and a few tools specialised on visual data exploration are available as well. Yet, typical problems of existing tools are the lack of extensibility or adaptability. The added value of this work will be to offer a design and implementation of a framework that enables users to visually explore data and rapidly extend the exploration process if the supplied capabilities are not sufficient.

2.2. Aims and objectives

Although there is a wide variety of tools for visualising data and a not so wide variety for data mining and data acquisition, none of those tools combines these three tasks and in addition allows the user to develop and add own modules in a rapid way. Therefore a framework is to be built that allows users to create own modules in a way that lets the user focus on the data exploration part. Users will be able to extend the framework or to use supplied modules and to interconnect those modules in a flexible manner. The data exploration process can thereby be defined by selection and combination of the appropriate modules. This work will focus on the design of such a modular framework in the context of visual data exploration. This framework shall allow users to discover previously undiscovered, potentially valuable information inside the data under investigation. Following steps will be necessary to reach the goal of designing such a framework:

1. *Survey of the subject visual data exploration to become clear about possible scenarios exploring data and thereby to identify the requirements of the framework*
2. *Survey of modular architectures containing plug-in-, component- and service oriented architectures to come up with an own architecture fitting the needs of possible applications to be built on top of the framework*
3. *Design and implementation of the found framework architecture*
4. *Implementation of a graphical front-end using the developed framework*
5. *Development of sample modules using the developed framework as a proof of concept*

2.3. Project management

This chapter describes what tasks had to be fulfilled in order to come up with this work. Differences of the real progress and the initial project plan are discussed.

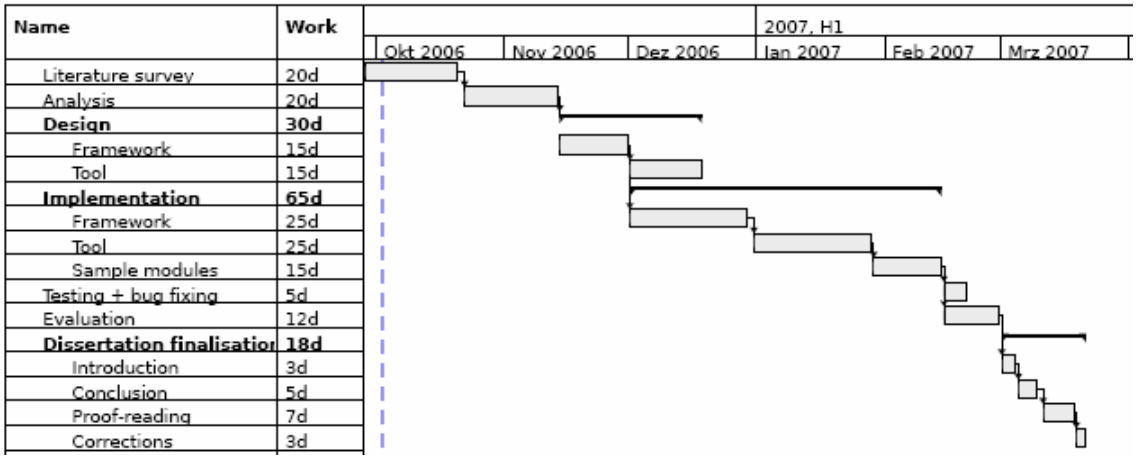

Figure 2 – Project plan

Each task in the project plan is assigned the number of estimated "working days". As this project was conducted part-time the term "working day" varies from what is commonly assumed in project planning and has therefore to be defined:

- *a week consists of seven working days*
- *a working day has approximately three working hours*

The estimated deadline could be met – the project was handed in according to the initial time plan. However, the time taken by the individual tasks varied from what was estimated. The period of proof-reading and corrections took far longer than initially planned – several iterations were necessary there. It turned out that the design and implementation of the framework and the tool could not be done sequentially but rather in parallel. In accordance with a statement found during the literature survey on framework design, the framework should be "used" – or programmed against – during its development phase. Therefore the application and the framework were developed in parallel. This way no major changes to the framework became necessary when using it for the first time by building the application on top of it, which resulted in a shorter development time. As it is the case in real industry projects, the time that could be spent on implementation and bug-fixes is unlimited. There is to be a deadline where the implementation needs to be in a state where it is assumed to be ready for release, no more coding should be done afterwards.

2.4. Contents of dissertation

This chapter describes the content of this document's individual chapters. The document's key chapters will hold a short conclusion.

Literature survey: This work starts with three chapters holding the relevant ideas found in books, papers and further sources. A contemplation of the relevance for this work will be given where appropriate. Initially an introduction into the topic visual data exploration is given (chapter 3), as this subject is assumed to not be commonly known. In order to structure the subject this chapter concludes with a categorization. As the goal of this work is to design a modular framework the subsequent two chapters will be a survey of modular architectures (chapter 4) and the design of frameworks (chapter 5). The chapter on modular architectures will give a general introduction into the topic and

describe component, plug-in and service-oriented architectures. The chapter on framework design starts with the definition of a framework and with the most important aspects in framework design. A chapter on the design of extensible frameworks and the evolution of frameworks will conclude the literature survey.

Analysis: Chapter 6 starts with the identification of the major requirements for the framework to be built. The remaining part of this chapter is subdivided with a section on each of the problems to be solved: the way modules can be combined is discussed, the structure of an individual module is analysed and the way modules are connected based on some module description will be shown. Following this, the need for defining different module types based on the steps required in a visual data exploration process is shown followed by the definition of those module types. A brief discussion on the distribution of modules concludes the chapter.

Design: In chapter 7 the design of the framework will be discussed and conducted based on the results of the analysis. The technology to be used to implement the framework is discussed. Following this, a section on the composition of architectures discusses different design approaches of how existing modules could be connected. A section on the design of one module, one on the overall structure of the framework and one on the way modules make their restrictions about bindable modules available is given.

Implementation: Relevant issues that occurred during implementation will be summed up in this chapter.

Evaluation: Chapter 9 is an evaluation of this work. The design of the framework is evaluated by showing the steps necessary to develop own modules. Following this it is shown how the developed framework can be used to conduct visual data exploration processes. This is done on simulated data and real, measured data.

3. Visual Data Exploration

3.1. Background

Visual data exploration can be seen as one area of visual data mining [Kei01]. It is concerned with **integrating the user into the exploration process**. The term will be defined here and example procedures will be shown. Ways to categorize the subject will also be given. The first step will be an overview of the wider topic data mining.

Ventures often collect a huge amount of technical or economical data in databases. It might be data about e.g. products sold or after-sales service. Those databases hold valuable information that possibly was not even considered when designing the databases. The datasets are usually recorded automatically with each technical or economical process step which can easily lead to some hundreds of thousands of datasets. One dataset typically consists of several parameters so the data is multi-dimensional. The process of extracting this valuable but implicit information is what makes the difference for many ventures – from the economical point of view by e.g. being able to forecast trends or from the technical point of view by e.g. using statistics to calculate probabilities for the failure of electronics. The information is hidden in the data in a way that makes automatic detection impossible.

Database management systems allow for the data to be displayed in a textual manner, which is of no use when the integration of all datasets in the search is required as the data cannot be shown in a clearly arranged manner for the user due to the huge amount of data [Kei02]. The fact that the data contains many dimensions leads to lack of readability as well.

The process of extracting information from large databases is also referred to as *knowledge discovery in databases (KDD)*. This term is sometimes used equally with the

term *data mining* while other sources refer to data mining as being a subset of KDD. **KDD is defined as the entire non-trivial process of finding information inside a huge amount of data** [Fayy96]. The outcome of the process will be some potentially useful and understandable patterns of the data investigated. The usefulness of the gained information can only be determined by the user that triggered the search. The terms used in this definition are hard to quantise.

Presenting the data in some visual way narrows down the area to *visual data mining*. The initially enormous amount of data is concentrated in just a few graphics that might give users a clear understanding of the issue that would have never been possible to get using textual searches only. The possibility of user interaction in order to find information in the visualized data is the process of *visual data exploration* – the area of KDD this work focuses on. Visual data exploration corresponds to visual data mining without mining algorithms to come up with a result.

Keim defines visual data exploration in [Kei01] as a "**hypothesis-generation process**". By visualising data according to the user's interactions, the user gets a better understanding of the data and can come up with a hypothesis. Such a hypothesis could be that there seems to be a connection between certain combinations of electronic control units in a vehicle and malfunction of that vehicle. This found hypothesis can then be verified using visual data exploration techniques as well as other means. It is most useful when the knowledge about the data is initially vague and the outcome of the search is unclear [Kei02]. A major advantage of visual data exploration is that there is no need to fully understand the complex details of the data because the user is given some form of understandable visualisation.

The exploration of data can be further categorised contemplating the knowledge about the data before starting the investigation [Kei02] [Mül05]:

Presentation: Presentation refers to a pure visualisation of information that is known a priori. The choice of an appropriate technique to visualise the data is the focus here.

Confirmative analysis: Confirmative analysis is conducted when a hypothesis about the data exists – possibly the result of an earlier exploration. The data is to be visualised in a manner that allows for the hypothesis to be confirmed or to show its incorrectness. The process is clearly goal-oriented.

Explorative analysis: Explorative analysis refers to an undirected search. The user tries to find some interesting patterns. User interaction is highly important here – interesting patterns can only be seen by the user. The outcome of this process might be a hypothesis that needs confirmation.

In [Shnei96] Shneiderman introduced the term *information-seeking mantra* which could be described as the steps to be taken to gain information from visualised data in general. Those steps are *overview first, zoom and filter and then details on demand.* Further steps to be taken are *relate, history* and *extract*. Those steps – seven altogether – have shown to build the basis for the way data exploration is done and shall therefore be described in more detail. While in [Shnei96] they are described in a very general context – the visualization of information in general – Keim related this principle to visual data exploration in [Kei01].

Overview first refers to the need for the user to get an overview of the data. Interesting patterns can be found and investigated further. This is then done by *zooming in* to analyse the found patterns. It is important to note, that zooming in this case is not just a way to resize the visualisation under investigation. Moreover the capability to show more details at greater zoom levels is important. For example an object that is represented by one pixel in the initial graphics could be replaced by an icon at a higher

zoom level and a detailed sketch labelled with some description at the highest zoom level.

Due to the immense amount of data the user needs to be able to focus on an interesting subset. Reducing the data to a subset is done using *filtering*. Filtering can be subdivided into *browsing* – the direct selection of the desired data – and *querying* which is done by specifying properties and constraints of the data in order to reduce the amount. The formulation of SQL queries could be seen as the most basic form of querying.

Details on demand refers to the possibility to request further details of individual objects – this could be done by pop-up windows, tool-tips or an additional visualisation possibly using a different technique.

One important aspect when implementing tools that support those steps is that the user might want to see the results of each step on one screen: it makes sense to e.g. still offer the overview while the user applies filtering or has drilled-down to some small portion of the data. There is a variety of techniques supporting this.

The step *relate* refers to some way of showing relationships between items while *history* simply indicates the need for some form of undo and redo functionality. Finally the *extract* step allows the user to export interesting portions of the data.

3.2. Classification of visual data mining techniques

Criteria for classification can be the *data type* to be visualised, the *visualisation technique* and the way *interaction and distortion* is supported [Sen] [Kei01].

Classification by the data type: The *data types* can be one-, two-, three- or n-dimensional or temporal data, which refers to data that contains time as one attribute.

Data items with relationships can be visualised as networks or graphs; if the relationships are hierarchical a tree structure will have to be visualised.

Classification by the visualisation technique: The second criterion is the *visualisation technique*. Numerous different techniques are known starting with standard 2D- or 3D-plots or charts as known from calculation tools like Excel to geometric techniques or iconic displays. [Kei97] holds a variety of examples:

Geometric techniques are concerned with multidimensional data, data that cannot be visualised using e.g. standard x-y-z plots. The landscape technique for example visualises the data as a perspective landscape. The parallel coordinates technique maps the dimensions of the data to axes in a chart. In order not to overload the diagram, data in the region of 10 dimensions can be shown. Further techniques are scatterplots, prosection views and hyperslices [Kei97].

Icon-based techniques usually map two dimensions to the axes of a chart and the remaining dimensions to the look of an icon. “Chernoff-faces” are well-known in this area. The number of dimensions is limited here.

Pixel-oriented techniques can be applied when visualising a huge amount of data. Each attribute of an item is represented by one pixel – the colour of that pixel is dependent on the value. A variety of ways to do so exist: e.g. the recursive pattern, the spiral and the circle segments technique [Kei97].

Hierarchical techniques partition the data and reveal different levels of detail depending on the level of the hierarchy the user is currently in. Dimensional stacking and treemaps could be named here.

In technical applications the need for *graph-based techniques* arises quite often. Examples are Hygraphs or SeeNet.

Classification by the interaction and distortion technique: The *interaction and distortion technique* is the third item of this classification. Interaction allows the user to manipulate the visualisations in order to reach his goals. Techniques are e.g. interactive ways of zooming and filtering. Distortion is the possibility to show portions of the data with a high level of detail while the remaining data is shown at a lower detail. Techniques in use are e.g. fisheye views and the perspective wall [Kei97].

3.3. Conclusion

Visual data exploration was introduced by putting it into the context of the broader subject data mining. The key property of visual data exploration is that it integrates the user into the exploration process. The justification to spend time and money on the analysis of recorded data was given – ventures can profit from the results economically. The outcome of an exploration process will be some potentially useful and understandable patterns of the data investigated, where the usefulness needs to be determined by the user. The process itself can be described as an undirected search, the result being a hypothesis that needs confirmation. Relevant for this work are Shneiderman's information seeking mantra and the overview of visualisation techniques.

4. Modular Architectures

4.1. Background

The framework developed in this work shall allow users to build *modular* visual data exploration scenarios. Therefore an introduction into modular architectures is given. The term "module" can be defined as follows [Kah98]:

> ***Definition:*** *A module is a logical or physical unit with strict borders in a certain context that consists of:*
>
> - *an export interface – accessible from other modules*
> - *the module's internals – implementation of the module*
> - *an import interface – other modules referenced*

Further on, properties of a module are that it shall not have any side effects and it must be possible to use it only by knowing its export interface. The module's implementation may only access the modules stated in the import interface. A module can be tested without contemplating the context it is used in, just by applying black box tests using the import and export interface.

Two essential properties in defining or evaluating a module are *coupling* and *cohesion*. Coupling is defined as the degree of dependency between modules and cohesion refers to the degree of how well elements within a module "fit together". In order for a system consisting of **modules to be really modular the principle of low coupling and high cohesion is essential** [Kah98].

According to that principle a module shall only receive as much information as required to fulfil a requested task. Passing in more data adds unnecessary dependencies. The

requested task shall be fulfilled by the module itself – using the referenced modules if necessary.

In 1972 David L. Parnas introduced one of the fundamental concepts of today's software engineering practices: modularization based on *information hiding*. In [Par72] he proposed that the effectiveness of a modularization is dependent on the *criteria* used to decompose a system. At that time it was common understanding that the best way to divide a system into modules, was to find the tasks fulfilled by the system and build a module for each of the tasks or process steps. Rather than doing it that way, Parnas proposed that **each module should hide a design decision** – being a detail of the system – that is likely to change independently of the other design decisions [Par]. Design decisions in current systems could be the choice of the persistence layer or DBMS, the graphical user interface library or the choice of a certain external hardware device. By keeping those details secret from the rest of the system, they can be changed without effects on the other modules. So for instance the DBMS (database management system) could be exchanged transparently to the rest of the system. Changes would only have to be done to the module hiding the design decision. This of course postulates that other modules access the module strictly through its interface. The interfaces themselves should hold nothing that is considered likely to change. Using a sample application and introducing possible changes, Parnas showed [Par72], that this way of decomposing a system is indeed superior. Although it could be argued, that the introduced possible changes in that paper seem to be chosen to underline the concept – changes could be thought of where that sample application would suffer changes distributed over several modules – there are no doubts on the advantages of applying Parnas' ideas.

This concept was so fundamental that it led to object-oriented languages at a later point in time. The secrets in that case can be viewed as being the private members of an object only accessible through the object's interface – the methods.

A system that is built in a modular way is flexible by allowing for the exchange of individual modules. If the newly added module's interface suffers no changes, the remaining modules should not be affected by that change. **Likely changes should not require major changes to the modular structure**: interfaces should not have to be changed. Modifying interfaces should only be necessary when very unlikely changes occur.

Modular systems can be modelled using *design structure matrices (DSM)* introduced by Donald Steward in [Ste81]. The rows and columns are labelled by the design parameters – pragmatically spoken by the modules. Interdependencies are then marked with an X where the row and column crosses.

From the software engineering point of view the advantage of the modularization of systems are undoubted. An interesting contemplation of the economical value of modular systems can be found in [Bal01]. Baldwin and Clark state that **modularization creates options**. Plainly spoken, a monolithic application consisting of only one module, gives the architect exactly one option: replace the whole system or leave it as it is. Having a modular system gives the designer a variety of options: modules can be added (augmentation) or replaced (excluding), extended or combined. It is shown how, based on the "real option theory" used in formal theory of finance, modularization can be evaluated from an economical point of view. In other words: the question whether it is worth to do a certain modularization can be answered. When evaluating an individual module the *technical potential*, the *cost* of the modularization and finally the *visibility*

of the module in question need to be determined and calculations can be made using those three factors. The difficulties doing this will probably be to quantise those factors.

4.2. Components

This chapter will contemplate component architectures in general and some specific implementations. Clemens Szyperski defines a component as a *unit of composition* with *specified interfaces* and *explicit context dependencies only*. A system can be built by interconnecting components. They can be **statically bound to a system or dynamically loaded**. In order for components to be reusable they have to obey a certain standard. This standard is technology-specific which means a system can contain components of a specific type. Standards are for example Microsoft's COM and ActiveX, Enterprise Java Beans or .NET assemblies. Component-based software technology became increasingly popular because it was realised that object-orientation didn't lead to code reuse at the extent expected when the object-oriented technology was introduced [Schn]. The granularity of classes was considered to be too fine. It was then hoped that component architectures allowed systems to be built more efficiently by just reconfiguring or adapting existing components and then adding the missing components. In practice, components of different vendors often influence each other in a negative way. Examples might be components referencing conflicting versions of the same third-party component, like common logging frameworks.

Binding components to systems can be categorized by the binding times [Mei01] – the time *when* the component is connected to the system:

Build time: The component is compiled into the system. This will result in one monolithic application and is usually not done nowadays.

Load time: Components are loaded on application start-up. Checking is done at this point in time. The major advantage is that the operating system checks if the loaded component corresponds to the library the program was linked against.

Run time: This is the most flexible way to bind components. The dependency is not known at compile-time. The loading of the component is done during the program's execution. The disadvantage of this solution is that no checking can be done during the development of the system. Errors become obvious only at runtime.

Of interest for this work will be the area of dynamic component systems where components are bound at run time. Therefore the ways to dynamically bind a component to a system will be contemplated. The five basics of how to dynamically bind a component are summarized in [Tor02]:

Runtime table lookup: This is the most common way of how modules are dynamically attached to an application. It uses an indirection: the application uses symbolic references to address elements (e.g. functions) in a component. Those are mapped to the real entry address in the component on loading it. This is for instance used when a native DLL [Chap02] is dynamically loaded. This allows for the implementation of the component to be changed without having to recompile the application as long as the component's interface is not changed.

Load-time code modification: This procedure works without an additional mapping table. The function's entry address is inserted in the referencing application at load-time. Replacing the component at runtime is not possible. The application needs to be reloaded.

Runtime code modification: This case works similar to the one above but the insertion of the entry address is done prior to calling the function. It is also referred to as "lazy

linking". The referenced library could be replaced during runtime. The overhead of linking at runtime will have an impact on the performance of the system.

Load-time code generation: The above procedures referred to binary libraries. Using e.g. Java jar-files, referenced classes are compiled at runtime. The jar-files contain byte code that is compiled into executable code on loading the class.

Full load-time compilation: The source code of the component needs to be delivered and is compiled at load-time. This way of doing it can be viewed as irrelevant at this point in time for two obvious reasons: full compilation from source code to executable binary code is time-consuming and manufacturers of components won't be willing to deliver the source code of their components.

Dynamically added components are by definition not known to the system during its development time. Therefore the system needs to be able to request some meta-information of the component's interface in order to be able to use it. Component interfaces can be specified using an *interface definition language (IDL)*. The system can query the component in order to call specific methods. A component may offer a variety of interfaces, in those cases some way of navigating the interfaces has to be available. The identification of a certain interface can be done using a unique id, the interface's name in conjunction with reflection or other means. Using unique identifications has the disadvantage of having to have some mapping to the real interfaces, while using names of interfaces has the obvious disadvantage of not being transparent to changes of the interface's name. Components can hold information about their interfaces themselves (self-describing components, e.g. .NET assemblies) or in additional IDL files as with COM.

In order for other components already installed in the systems to be able to use newly added components, the architecture has to offer a way for the components to be made available throughout the system. Some form of component registry [POSA2] can be found in many commercial component architectures: e.g. COM components are being registered with the operating system's registry.

4.2.1. COM

Microsoft's Component Object Model (COM) [Greg02] [Mic03] refers to a component framework present on all Windows systems. Physically a COM component consists of a DLL or an exe-file and a type library holding meta-information about the component. It **uses a binary standard which makes it independent of the programming language**. All languages supporting that standard can use existing COM components. The component's interface is described by an interface definition language (IDL) – in this case MIDL. Each component is required to implement the `IUnknown` interface. This is the root interface for all queries for specific interfaces and holds a method to query for specific interfaces. The design pattern in use here is the *extension interface* [POSA2], which enables clients to obtain a reference to an interface and navigate through all interfaces supported by the component. The starting point of this navigation is usually the root interface. The specific interfaces inherit from that root interface, which means that each interface offers the possibility to query for other interfaces. According to the extension interface pattern it should be possible to move from each interface to every other of the component's interfaces. The major advantage of using this pattern is that new interfaces can be added at a later point in time without affecting client code using the component. The idea is to never change an existing interface but to add new ones if the functionality is to be enhanced. This guarantees that clients using the old interfaces will continue to function without recompilation or testing. A disadvantage on the other

hand is the complexity added to the components and the clients due to the variety of interfaces and the code necessary to navigate. Whether an approach like that is to be used in this work will have to be decided.

Interfaces are defined by a globally unique id (GUID) and can thereby be uniquely identified. The memory management is left to the client. Reference-counting is used to determine when to destroy an object, which is an error-prone procedure. Automatic memory management (i.e. garbage collection) is not used here.

Transaction management, resource pooling and object pooling were later added to COM to build COM+ [Mic03.]. The pooling of shared resources – e.g. hardware or database connections – is essential for systems with a variety of clients. Object pooling allows for objects to be reused. They will not be destroyed but rather just deactivated until they are needed again. Both – pooling of resources and of objects – seem to be irrelevant for the framework to be developed in this work as the number of modules operating in parallel will be minimal.

Clients using COM components will hold a huge amount of technical code – something that is generally considered to be problematic. The code representing the business logic becomes hard to read as it contains code that is specific for the chosen technology. Apart from the mentioned reference-counting additional code needs to be written in order to e.g. initialise the COM environment and obtain references to interfaces. The mentioned issues plus the fact that COM components use entirely new data types in order to be usable from various languages makes the client code look unfamiliar. This is something that the architecture to be built here should avoid.

4.2.2. Enterprise Java Beans

Enterprise Java Beans (EJBs) [Sun05] – not to be mixed up with Java Beans – are components running inside a container which in turn runs inside a server (application server). EJBs are therefore server-side components. The container represents the runtime environment for the beans. It offers functionality like transaction management, persistence, possibility of remote access, security and others. The developer is not involved in implementing those issues [Mot02]. The components are referred to as beans and can be subdivided into session beans and entity beans. The session beans hold the application's business logic. They can be viewed as a "single client inside the server" [Sun05] as they perform work on the server on behalf of the client. Session beans can be stateless or stateful. Stateless session beans do not maintain a state for a client. During the execution of a client's method call, the bean can hold a temporary state, after the method's execution is finished that state is being discarded. This type of session beans can be used for generic tasks performed for all clients or if there is no need to remember states for clients. The advantage of stateless session beans is that each instance of the bean is identical and can therefore be returned to a requesting client arbitrarily. Thus instance pooling can be used which allows the server to hold a fixed number of instances with no need for destroying used instances. Stateful session beans hold a state on behalf of a client which means that each client requires one stateful session bean in the server. Entity beans on the other hand represent the application's entity objects. The container offers support to persist those objects.

The current standard is EJB3.0. This version of the standard mainly addressed "ease of development". Earlier EJB standards were often criticised for being too complex and heavy-weighted [Ihn06]. Those earlier versions of EJB are widely used and are – without doubt – very valuable. Yet, the weaknesses identified in those standards can

help to draw important conclusions for this work. Weaknesses of EJB prior to 3.0 are that components **cannot run stand-alone**, they only run inside a container. Unit testing of the contained business logic code is therefore not possible in an isolated environment. Another weakness is that **many conventions exist** that need to be obeyed but cannot be compiler-checked. The **complex configuration procedure** that takes place in external files is error-prone and was therefore changed to configuration inside the source code. Those issues should be avoided in the framework design.

4.2.3. .NET

Components in Microsoft's .NET [Mic03] are referred to as assemblies. Such an assembly consists of the code, metadata and embedded resources [Mei01]. **An assembly is "self-describing"** – i.e. it can be used without additional description files. It holds a so called manifest where following information can be obtained by applications wanting to load the component: identity of the component consisting of a name and a version number, a list of the required assemblies (see "import interface" in chapter 4.1) and the exported types (see "export interface"). The assembly is loaded when first referenced, each method is compiled from an intermediate language (IL) to native code when called for the first time (see "Load-time code generation"). Physically an assembly is represented by an exe-file or a DLL. Unlike with Java not individual classes are loaded here – the granularity is coarser: the entire assembly, which typically consists of many classes, is loaded [Chat05]. Assemblies are usually referenced by other assemblies and can therefore be loaded automatically when needed [Mei01]. If the dependencies are not static, .NET supports the dynamic loading of any assembly at runtime without any initial configuration required.

4.3. Plug-Ins

Plug-ins **are optional modules of a system** [Chat05] [Chat1]. A system should function properly without the plug-ins installed. They are used to extend a working application or as described in [Chat05] to extend other plug-ins. Plug-ins can also be used – as components can – to decompose a system into modules. Those plug-ins will be loaded on demand. An application built with a plug-in architecture allows for third parties to contribute plug-ins and thereby enhancing the functionality of given applications. Examples of such applications are the Eclipse IDE or OpenOffice.

Some framework becomes necessary to handle the addition, the removal and the loading of installed plug-ins. Further tasks of such a framework could be to observe given constraints: When adding plug-ins to a system there might for example be cardinality constraints, e.g. if the plug-in uses a limited resource [Chat05]. A plug-in architecture will typically have a central instance for discovering and loading plug-ins, a plug-in interface that each of the plug-ins will have to implement and the concrete plug-ins.

The plug-in process can be left to the user; installation and definition of the application's extension point for the plug-in to extend will then be done manually. An automatic composition of the system is also possible, yet more complex (self-assembling system). In the case of automatic composition the plug-ins need to be defined in a formal and unambiguous way. The framework itself will need a huge amount of intelligence for combining the plug-ins based on their descriptions. Depending on the plug-in framework, addition of new plug-ins can be done on application start-up or at runtime.

In [May] a major advantage of a system extended with plug-ins is described: Having a plug-in framework **no changes to the code of the existing system have to be made when adding plug-ins**. With traditional architectures the application itself will have to

be enhanced by at least the calls to the newly added component, a plug-in architecture handles the addition of plug-ins without necessary changes. A further point is that the developer of a plug-in does not need a deep understanding of the application. Knowledge of the plug-in interface is sufficient. An important aspect is that by introducing a plug-in interface and developing the plug-ins independently of the main application, the coupling between the plug-ins and the system will be low and coupling between the plug-ins will be minimal.

4.4. Service-Oriented Architectures

A service-oriented architecture (SOA) [Oas06] [Fur04] is a paradigm for organizing and utilizing distributed capabilities. A *service provider* offers a capability (i.e. a service) that a *service consumer* is looking for.

A service is a mechanism to *enable access to capabilities* using a given *interface* and obeying given *constraints* and *policies*, described in a service description. That service description needs to be accessible and understandable for all participants and therefore typically obeys a certain standard. In terms of object-oriented distributed systems a service can be seen as a way to call a method with the implementation being entirely hidden from the service consumer i.e. the client. A major difference is the fact that unlike calling methods of an object a service does not have to be instantiated before the call is made. By definition a service request does not even have to be fulfilled by an IT system but could also be fulfilled manually. This broader definition is irrelevant for the problem discussed in this work and will therefore not be contemplated.

One essential part of a service-oriented architecture is the **possibility for a client to lookup a service** not only using syntactical but semantic specifications. In other words: lookups will not specify the method to be called in a technical manner (class name,

method name, parameters etc.) but rather some – often human-readable – specification of the form "I need a service that allows me to order a book". Using the service description, a client can *dynamically* bind to that service and is able to use it afterwards. In a SOA there is typically some central instance where service providers register (*publish*) their services and service consumers request a certain service: a service broker. This mechanism is also referred to as *publish-find-bind*:

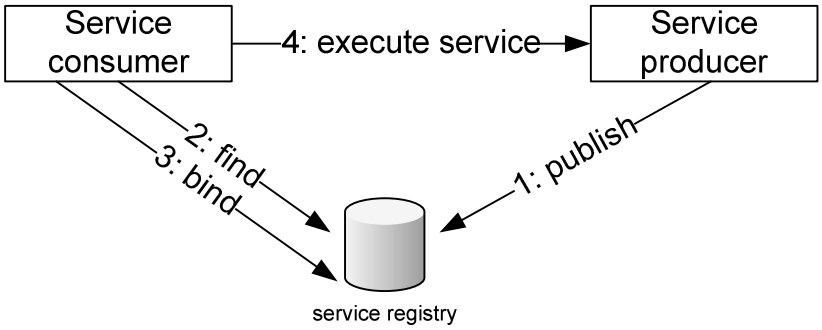

Figure 3 – Typical service scenario

Nowadays SOAs are usually implemented using Web Services [Cou05] with *Web Service Description Language (WSDL)* being the mentioned service description standard and *Universal Description, Discovery and Integration (UDDI)* being the service registry. While the term SOA is often related to huge service environments distributed over several organizations, OSGi (Open Services Gateway Initiative) is one sample architecture of how services can be used within one application. This specific implementation is used in the recent versions of the Eclipse IDE.

4.5. Conclusion

The contemplation of the concepts and technologies in this chapter brought up some issues relevant for the development of the framework. The definition of a module as given at the beginning of this chapter will be reflected in the framework's modules. They will consist of an export and an import interface. It became obvious that

technologies like COM require a huge amount of work in order to get the module running. The environment needs to be initialised, obtaining and administering references is quite complicated. Weaknesses in the EJB standard were identified, like the fact that components cannot run stand alone and that many conventions exist. Those issues should be avoided by the framework to be developed. Contemplating plug-in architectures, the possibility to extend an application without any code changes is the key issue there. The idea to constrain the cardinality of modules to be connected seems relevant as well. Service oriented architectures are very flexible due to the possibility of dynamically binding to a service. The concept of a central service registry is relevant for this work.

5. Framework design

5.1. Background

Nowadays developers use third-party components to build their applications. Those components can often be extended to fit the needs of the application and have therefore to be designed in a way that allows for reuse and extension. This kind of components can be defined as *frameworks*.

Definition: *A framework is a reusable object-oriented library [Cwa06]*.

As a major part of this work will be designing such a framework, some basic rules and principles shall be discussed here:

A framework will be used by a variety of different developers with a varying skill-level. *Ease-of-use* is an important aspect. If that is not the case the framework is unlikely to be used by many developers and the likelihood for using it incorrect is high. [Cwa06] states that "**simple things should be simple and complex things should be possible**". One important aspect that is crucial for the ease-of-use of a framework is *consistency*. If names, idioms or design patterns are consistent throughout a framework, users will be able to transfer their gained knowledge from one part of the framework to other parts. A *low barrier-to-entry* is important as well. New users should be able to experiment with the framework and come up with a working prototype rapidly. This postulates that there is no need for complex initialisation procedures and no unnecessary parameterisation – rather reasonable default values. Structuring the framework using packages or namespaces helps users to become acquainted with a new framework as well. Types commonly used should be easily accessible while types necessary for special cases only should be placed inside sub-namespaces.

Due to the variety of users there are many different, possibly conflicting requirements. Not all of them can be satisfied, a framework will contain many trade-offs. Recapitulating the design of a framework needs to consider the fact that frameworks are usually enhanced by users to fit their application-specific needs.

Central to a framework is the API (application programming interface). As with user interfaces, this is what users – in this case developers – see. Usability tests with developers should therefore be conducted in order to reach an API that follows the issues mentioned above. The design of a framework differs from the design of object-oriented systems. In the latter case the focus lies on the architecture while in the former case the interface is crucial. Once a version of a framework is released it is critical to change the API due to compatibility reasons. As frameworks are used by developers not aware of its internal structure, relationships like the inheritance hierarchy of the framework's classes might not be obvious. As opposed to object-oriented system, where an abstract design is usually conducted in order for the design to be reusable, a very abstract design of a framework will result in lack of usability, so e.g. the interface should contain most specific possible parameters [Blo06].

5.2. Designing for extension

Frameworks usually allow for general solutions. As they are used in different groups of applications the necessity to enhance parts of the framework in order to build an application-specific solution often arises. **Frameworks are typically designed to be extensible** for that reason. An important aspect is, that for parts of the framework not designed to be extended, extension should be made impossible [Blo06]. This is commonly done using programming language specific features such as classes that cannot be extended or methods that cannot be overriden. The parts designed to be

extended should be obvious to the developer. Those are the so called “hot spots” that can be filled in by developers.

5.2.1. Extension by class inheritance

One mechanism to allow for extension of a framework is the supply of base classes that can be extended by the framework’s users which is referred to as *extension by class inheritance* [Dem97]. **By overriding certain methods, functionality can be added or altered**. It is essential that those methods are obvious to the user. Using the template design pattern [GoF95] algorithms can be defined with some specific parts – some methods – left to be implemented by the framework user. The invariant part of the algorithm can be defined in the base class. The base classes in this case will be abstract and the methods to be implemented will be abstract as well. An advantage is that the hot spot therefore becomes obvious to the user and is even compiler-checked, as those classes cannot be used without deriving and adding the missing implementation.

An example that could be used in the framework to be developed is given here: Each module needs some configuration that can be read from e.g. the user. The goal in this example is that the way the configuration is obtained shall be variable and can hence not be defined within the framework. Therefore an abstract base class `AbstractConfigReader` is supplied with the framework. The publicly accessible method `getConfig()` is called by an object of the framework – in this case by an object of the module class – in order to obtain the modules configuration. The abstract base class delegates the reading of the configuration to an unimplemented and therefore abstract method named `readConfig()` that is not accessible from outside the class. Hence this method becomes a hot spot of the framework and in order for the module to be able to get the configuration the application developer is required to supply a subclass implementing the `readConfig()` method. By supplying multiple subclasses

the way the configuration is obtained can be varied by the developer. The objects of the subclasses need to be passed to the module.

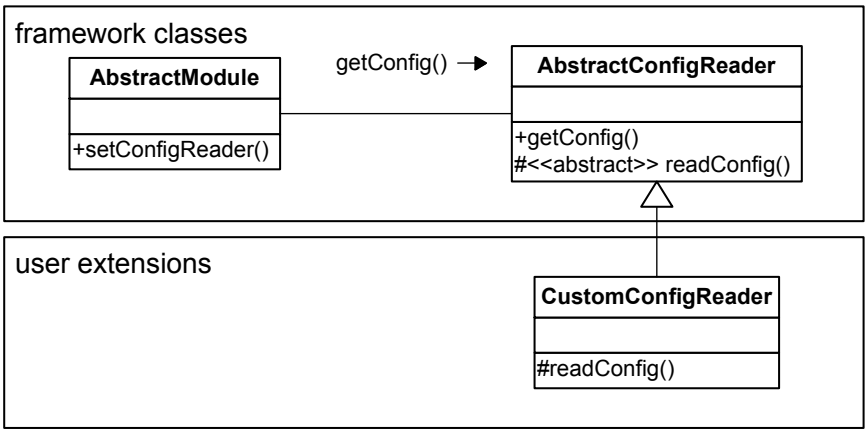

Figure 4 - Framework extension using class inheritance

A disadvantage is that this approach violates encapsulation of the framework classes as the subclasses access the internals of the parent classes and can hence be viewed as being *white box*. Due to that fact each method that can be overridden should be viewed as being equivalent to the class' public interface, even if the specific method is not public. Documentation and testing should therefore be done as it is being conducted for public members [Cwa06].

5.2.2. Extension by object composition

As opposed to have individual methods be the hot spots, with framework *extension through object composition* [Dem97] **entire objects represent the hot spot**. Application developers using the given framework need to configure objects of framework classes. **This allows for the hot spot to be polymorphic**. So for example a framework could use an abstract class for data access (see Figure 5). Users of the framework can use various subclasses – for example one subclass for file data access and one for database data access. Instances of those classes can then be used to fill the

hot spot by passing the objects to the framework classes. This mechanism is more flexible than extension by class inheritance.

The following example shows how varying the reading of a module's configuration could be done using the approach of object composition. As in the previous example a module object needs to be configured by setting the `ConfigReader` to be used. Several `ConfigReader`-classes could be supplied with the framework – the application developer would instantiate the chosen class and inject the object into the module. Equivalent to the case of extension by class inheritance, the developer can supply own subclasses to be used by the module. As shown here the different `ConfigReader`-classes can vary in more than just the abstract method like in the previous example. The hot spot itself can be polymorphic: in this example there is a class hierarchy for reading the configuration from files. Any of those classes could be used and – a major advantage of this approach – even be exchanged at runtime.

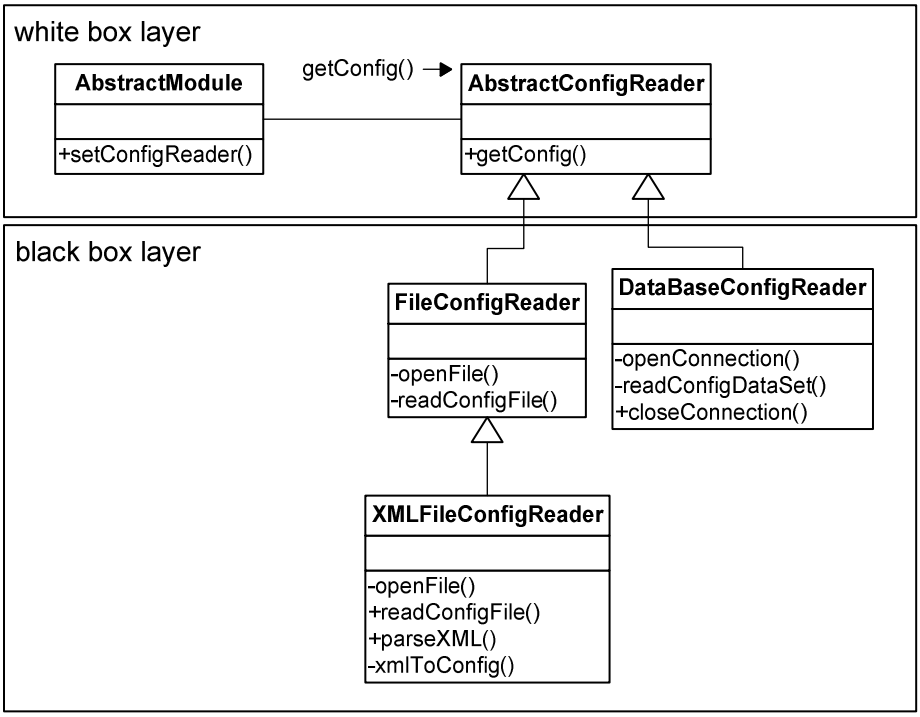

Figure 5 - Framework extension using object composition

Extension using class inheritance is also referred to as *architecture-driven*, while extension by object composition corresponds to a *data-driven* mechanism [Tal94]. **Common frameworks typically include both described mechanisms**. In [Dem97] a way to combine the advantages of both approaches is proposed referred to as class composition. In [Tal94] a combination of both approaches is proposed: the framework shall be extensible by class inheritance and object composition. Supplying default implementations is important for the ease of use. This way an application developer can start by using the supplied default implementations and extend parts of the framework only when those supplied implementations are not sufficient. Hence a layered framework is proposed: one "white box layer" corresponding to the abstract base classes that can be extended and one "black box layer" corresponding to the default implementations. Developers can decide whether to build their applications on the white box layer by implementing own classes and filling in the hot spots or by using the supplied classes of the black box layer.

5.3. Designing for evolution

As opposed to the extension of an existing framework by developers, which is something the framework was designed for, the evolution of a framework due to new requirements, technical changes or bad design decisions is somewhat harder to handle. Once a version of the framework has been shipped, **changes introduced to the framework should not affect existing code** built by users. To ensure that applications using the framework still work with an enhanced version of the framework some guidelines can be followed:

Better to leave out functionality: The introduction of new parts is always possible without affecting client code, but the removal of existing functionality, classes, methods or parameters results in clients not running with the new version of a framework.

Therefore in [Blo06] it is proposed to better leave out functionality if not clearly specified, well-designed and tested and provide it in a future release. As opposed to the development of applications, where functionality can be added for an upcoming release and refactored [Fow99] afterwards, this is not possible with frameworks unless the interface is not affected.

No change of interfaces: No interfaces that have been shipped with a framework should be changed as this would result in classes developed by framework users implementing those interfaces not compiling with the new version. If changes become necessary one approach is to add a new interface beside the existing one. The design pattern *extension interface* could be used for navigating interfaces (see 4.2.1), which adds additional complexity to the client code.

Advantage of abstract base classes over interfaces: [Cwa06] discusses the advantage of abstract base classes over interfaces when designing frameworks. Programming against interfaces, not against classes, is commonly done to decouple a class' contract (the interface) from its implementation. In the case of framework design this puts the burden on the designer that shipped interfaces cannot be enhanced without affecting client code. Using abstract base classes to specify the contract allows for later enhancement of that contract. As long as only concrete – no abstract – methods are added, existing client code will still compile with the new version of the framework. One major disadvantage of this approach is that due to the fact that most modern programming languages do not have multiple inheritance it is not possible to inherit from additional classes. In [Gur01] the use of roles in a framework is proposed. Each role is represented by an interface which means that one class could implement several interfaces if it is capable to offer functionality required by different roles. This is not possible if only abstract base classes are used. Additionally it can be said, that the case

of extending a framework by implementing a supplied interface in a custom class is referred to as “reuse of design” while extension by inheriting from a supplied class and adding functionality is referred to as “reuse of implementation”.

It might not always be possible to stay compatible with prior versions of the framework. In those cases it is recommendable to add the new parts and mark the old ones as being obsolete [Tal94]. In future releases those obsolete parts may be dropped. If major changes of the framework should become necessary the migration should be supported by tools.

5.4. Conclusion

The discussion about framework design was based on the definition that a framework is a reusable object-oriented library. Central to each framework is its application programming interface as this is what users encounter. The importance of usability tests was stated in this context. As frameworks are typically not developed for one application, they tend to be general. To fit an individual application’s needs, frameworks have to be extended. The two key concepts to do so were discussed being extension by class inheritance and by object composition. The former one is based on the so called white box layer while the latter on is based on the black box layer. The conclusion can be drawn, that a framework should offer both layers to framework users. One major difference between framework design and the design of a common system is that changes introduced to the design of a new version shall not affect code based on earlier versions of the framework.

6. Analysis

Amongst the issues to be solved will be the design of a basis the user can start building own modules on – e.g. a framework of abstract base classes. Architectural questions to be discussed and solved will be whether there will be one central instance administrating all modules of the system – some kind of module registry following the component registry design pattern [POSA2] – or will each module be responsible for holding the information about possible connections to other modules, which would be a decentralized approach.

The connection (binding) of modules has to obey certain rules – e.g. not each two modules can be interconnected. Those rules will have to be defined. They will be based on meta-information about the modules. Therefore a way of making each module's meta-information available to the rest of the system needs to be found. This will lead to the question whether a module itself holds this information or there is a need for an additional form of module descriptions – possibly description files.

Having configured a set of modules to work together, the question arises of how to deal with updates of individual modules. Is there a need for dynamic updating, i.e. the exchange of components at runtime [Pla], or is it sufficient to just replace the module and all affected configurations. If updates shall be supported in a way that is transparent to the remaining modules, there will be the need for restoring the module's state after the update process. This would lead to the necessity to store a module's state – referred to as externalization. Some way of persisting parts of a module's state will also be necessary in order to remember the user's configurations after restart. Issues like distribution of modules could also be addressed. It would have to be defined if there is a necessity to do so or if the possibility of later distribution exists.

6.1. System overview

The system will consist of a framework and an application built on top of that – the *Visual Data Explorer*. Users can enhance the system by supplying own modules, adapting parts of the framework's administration code or building an entirely new application on top of the framework:

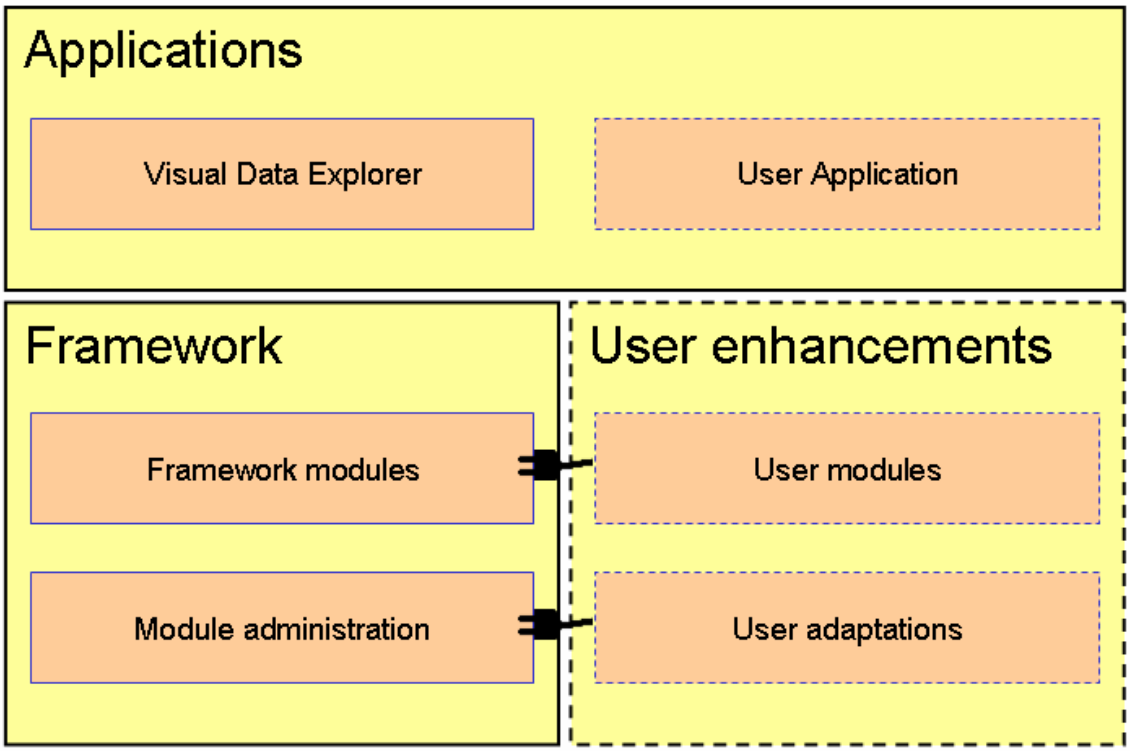

Figure 6 – System overview

6.2. Requirements

The first step towards design and implementation of the framework will be to identify the requirements. A major difference between the design of object oriented systems and framework design regarding requirements engineering is, that in the latter case requirements are to be found for two parties: developers using the framework and users of tools built on top of those frameworks. For that reason some of the requirements tend to be quite technical as they address the needs of developers. The identified major requirements are listed below followed by a brief discussion on each requirement:

R0: Top level requirement: The system shall enable users to explore data and deduce hypotheses they can profit from economically.

R1: The framework shall enable users to build modular visual data exploration applications: "Modular" in this case refers to the fact that users shall be able to configure visual data exploration scenarios with respect to their needs. One visual data exploration scenario shall not be a monolithic construct but rather be adaptable to specific needs by e.g. adding own modules. Existing modules should be usable in various setups.

R2: The contribution and integration of own modules must be possible: The framework should enable users to develop own modules based on the framework and integrate those modules. For the framework it should be irrelevant if a certain module was supplied to together with the framework or was added by framework users at a later point of time.

R3: It must be possible to connect different modules: In order to come up with a requested scenario, users must be able to connect different modules. The output of one module shall be the input of the connected module.

R4: The framework must support the entire visual data exploration process: Typical steps or tasks to be conducted in an exploration process were described in chapter 2.1. It should be possible to conduct all of those steps using this framework. Following the definition of visual data exploration, the framework and the application on top of it must allow for an interactive exploration process.

R5: The framework shall be simple to use and it shall have a low-barrier to entry: The typical users of such a framework are assumed to be people interested in rapid answers to their questions on the data under investigation, not people with the focus on

the development process of new modules. Therefore it is essential to have an easy to use API and a framework that does not require complex setup scenarios of additional runtime environments or tools.

R6: It shall be possible to use data from arbitrary sources to conduct data exploration: Visual data exploration typically uses mass data which is stored in common databases or data warehouses. Other sources like plain text files or export files of other tools could be used as well. For that reason the framework should not restrict the data source to one specific type of source but rather support several data sources and – in accordance with requirement R2 – allow for any data source to be used by offering the possibility to contribute own modules that access a specific data source.

R7: User configurations shall be persistent: Visual data exploration setups configured by users can become quite extensive depending on the modules in use. Therefore those conducted configurations should not be lost on system shutdown and on module updates.

6.3. Architecture composition

Specific architectures composed of given modules will have some hierarchical form due to the fact that the **modules are categorized into module types** meaning not any two modules can be connected.

> **Definition:** *In this work the term "module type" is used to categorize different modules according to their visual data exploration task they fulfil. The framework defines base module types that can be specialised.*

For example one type of module will be responsible for acquiring data and offer that data to other types of modules. Therefore modules able to process this data will be connected to the former kind of module as its child modules.

Definition: *In this work the term "parent module" will be used for the module contained in the module's import interface while the "child module" is defined to be the module contained in the export interface. The parent module is defined to be the module closer to the data source.*

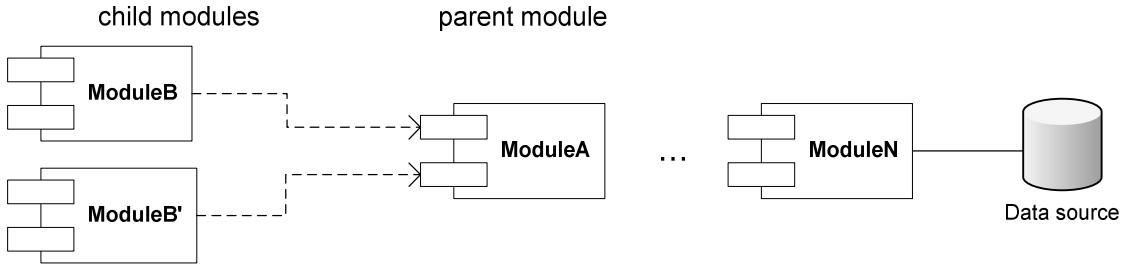

Figure 7 – Parent and child modules

The connection of several modules will lead to a hierarchy or – in the case of the limitation of the cardinality of import modules (parent modules) to one module – even to a tree. For general frameworks that allow for the composition of modules, this kind of prediction cannot be made. In general frameworks with no restrictions on the binding of modules quite complex architectures are possible (see Figure 8, architecture on left side). Having restrictions on the possible connections will result in better structured architectures.

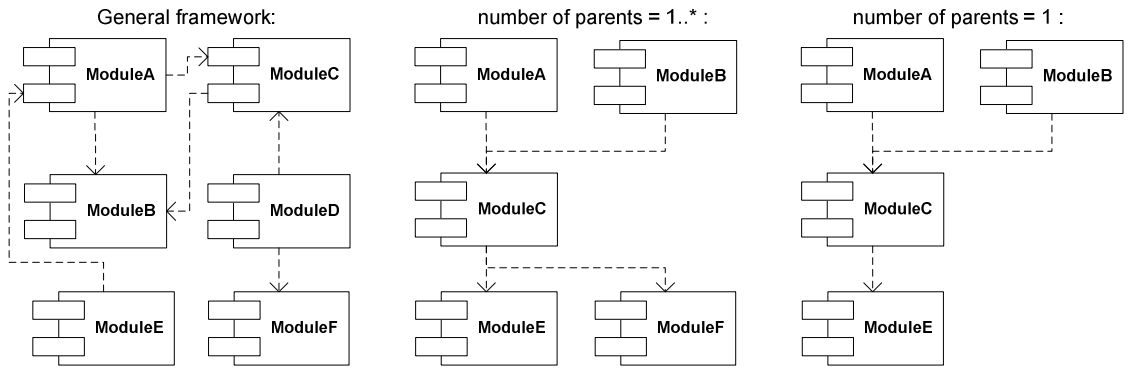

Figure 8 – Possible architectures depending on number of parents

Allowing more than one parent module could lead to complex scenarios that would have to be handled by the framework. So for example the data type supplied by each of the parent modules has to be in some way compatible in order to be combined by the child module. Module developers would have to specify this which would add extra complexity to the configuration procedure. Advantages of having multiple parents would e.g. be the possibility to acquire data from multiple data sources.

The fact of knowing the module types and the form of the resulting architecture can be used to come up with a design that is simpler. It is essential to define the module types and the constraints on binding two modules: not each two modules can be connected due to technical aspects like differing data formats or logical aspects. Such a logical aspect could be that a module supplying data cannot be a child of a visualisation module.

6.4. Structure of an individual module

A module should be able to function properly without the existence of specific modules given that the import and export interface are somehow satisfied. Modules shall be configurable by users individually, which means that each configurable module needs to offer a user interface. Combined configuration of several modules could be thought of but influences the modularity in a negative way. As each combination of modules is

possible and new modules can be added as well as existing ones be removed the solution to configure each module individually is preferable. This postulates that developers of new modules offer a user interface allowing users to configure the module's parameters. In order to have a consistent way of configuring modules, the framework should offer functionality to create such configuration dialogs easily. Configurations that could be thought of could be the specification of the data source in the case of data acquisition modules. Modules could offer monitoring functionality by allowing the user to view certain details of the module's task in its user interface. Offering an easy to use way to create such user interfaces will be essential for the ease of development of modules.

Modules need to be able to communicate with new modules, regardless of the module's supplier. According to requirement R2 users shall be able to add own modules. **The possibility to add own modules means that the modules are not necessarily compiled together**. To make sure each two modules can communicate the modules' interfaces need to be specified and it needs to be checked that each module obeys that specification. Each module needs to at least offer the information necessary to bind to it through that interface – the "binding properties". Apart from that part of the interface that will be common to all modules, there can be another part of the interface being common to all modules of a specific module type.

> **Definition:** *A module's "binding properties" hold information about the way this module can be bound to other modules. Based on these properties the framework will decide whether two specific modules can be connected.*

Modules need some form of deployment unit. Developers of modules need to be able to package and deploy their modules. Those deployment units need to be defined. Examples of deployment units are libraries, archives or individual files.

6.5. Flow of information

As described in the previous chapter, composed architectures will have a hierarchical form. Flow of information in general can be categorised as proposed by Structured Analysis [Kah98] into *data flow* and *control flow*.

By analysing the flow of information that will take place in such an architecture, it becomes obvious that that the *data flow* will be directed strictly downstream (from parent to child module) while the *control flow* will take place upstream. So e.g. data acquisition modules will offer the acquired data to requesting modules connected to them which is data flow. Those connected modules process and forward the data to their children. Users on the other hand will request information and thereby trigger a module that in turn will trigger its father module. This can be seen as a control flow. This categorisation will help when designing the modules' interfaces because it can be said that each modules import interface is responsible for triggering or controlling other modules in order to obtain data while a module's export interface offers data.

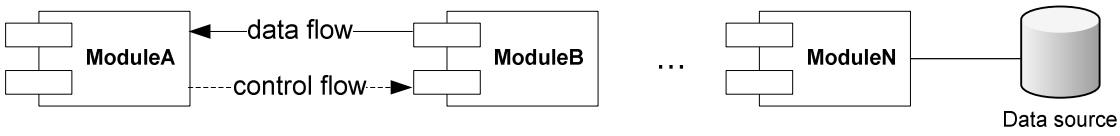

Figure 9 – Data and control flow

6.6. Binding properties

In order to dynamically interconnect modules, each module needs a description of its binding properties. Those **binding properties are the rules used to decide whether a certain constellation is acceptable**, i.e. if two specific modules can be connected. The binding properties will also be used to offer the list of bindable modules to the user. Therefore those properties have to be defined:

Module type: As opposed to a general framework the modules in this work can be categorised. The interconnection between different modules will depend on the type of the modules. Therefore each module has to hold the information about its type.

Cardinality: As the composition of the system will be done by the user there is a need for certain constraints. Modules can have 0 .. n child modules connected to them. It might be necessary to limit the number of connected modules due to e.g. performance issues. In order for the system to be able to check this constraint, each module needs to hold this information.

Import interface: By categorization into module types it will be possible to formulate statements about types of modules a specific module can be connected to. Certain combinations do not make sense and have therefore to be ruled out. Such a scenario could be trying to connect a module responsible for data acquisition as a child of another module responsible for visualisation. Those kinds of connections can be ruled out by specifying the types of modules a specific module can be connected to.

Export interface: Each module will deliver some kind of specific data. That data needs to be processed by modules connected via the export interface (child modules). Connecting two modules corresponds to setting up a bi-directional relationship. This can be done based on the information specified about one direction. Therefore it is sufficient to specify the import interface – the export interface will not be included in the binding properties. In other words: Modules have restrictions on their potential parent modules, but no restrictions on their child modules.

Chaining allowed: Connecting modules of the same type – referred to as “chaining” in this work – allows for flexible constructs to be built. A module type responsible for preparing data might remove outliers or do filtering. Combining tasks like that in a

chain would allow developing many fine grained modules with very specific tasks that could be chained in a specific scenario. This binding property could be explicitly specified or depending on the import interface implicitly specified. So for instance listing the type of the current module in the import interface corresponds to allowing chaining of that module.

Users or developers of modules should have to be concerned with specifying those binding properties as less as possible. Therefore the question arises who is to do those specifications. If those properties can be abstracted to a level that each module type has identical properties, they could be specified once for each module type and be integrated in the framework. This solution would simplify the development of new modules but would result in a static solution. Each module would have the properties specified for its module type. Having the module's developer doing the specification would add more flexibility to the system, because each module could specify its own values. A solution that supplies valid default values of each module type's binding properties while allowing module developers to configure those values would be ideal and will therefore be chosen.

6.7. Definition of the module types

The framework will be constrained in a way that **modules added to the system need to be of a pre-defined module type**. The addition of new module types will not be possible because it is considered to be unnecessary. Putting certain constraints on a framework helps to reduce complexity as not any possible constellation needs to be considered – this way the framework will be simpler from the user's point of view (see Requirement R5). The module types will be given by the framework, developers of modules are free to add any modules of a pre-defined module type. The definition of the

module types will be connected with the steps involved in visual data exploration processes which for that reason are shown in Figure 10:

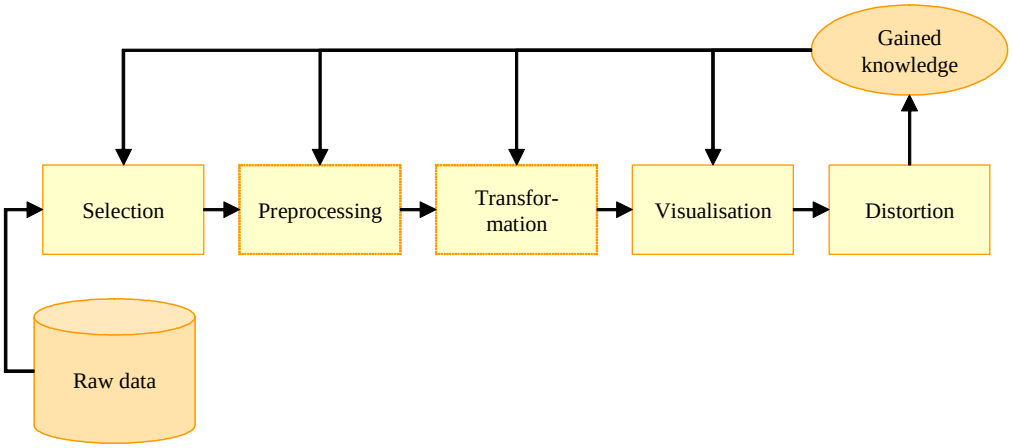

Figure 10 – Visual data exploration steps

Finding the module types will be done by contemplating typical visual data exploration scenarios, which follows a scenario-driven design approach as proposed in [Cwa06]. The tasks that need to be fulfilled (based on the figure above) will be highlighted and assigned to the different module types. According to requirement R4 this will result in all visual data exploration steps being mapped to the framework:

In a visual data exploration application there is some source for the data to be analysed. Users need a way to *select that data source*. Input data of more than one data source could be combined here. *Specifying the subset of data* to be used from the data source and the *acquisition of that data* from the selected data source is to be done as well as *offering that data* in a format processible by other parts of the system. The acquired data will be raw data that might need *preparation* or preprocessing before analysing it. Preparation could be removing outliers or filling in missing values. There are different approaches for doing so – e.g. using the mean value or the neighbour value to fill in a missing value – which means these tasks need to be configurable. The data resulting from that step could be visualised using known visualisation techniques, but the amount

of data is often too huge. That data can be reduced by applying a *transformation* [Han01]. This could be done by reducing the dimensions of the data – e.g. by summing up attributes. In the transformation step the data can as well be mathematically transformed into a form reasonable for visualisation.

The data resulting from the prior steps – prepared, transformed or raw data – needs to be presented visually. This step forms a major part of visual data exploration applications. The variety of *visualisation* techniques is huge as described in 3.2. The choice for one form of visualisation can only be done by the user and depends on the data under investigation. Different users might have entirely different requirements regarding the visualisation. Therefore it is self-evident that users might wish to add their visualisation techniques to the system. Visualising identical input data with different visualisation techniques can help users during their investigations. In addition to pure graphical representation of data, visual data exploration allows users to *interact* e.g. by zooming in, highlighting interesting parts of the data or by distorting irrelevant parts.

The tasks highlighted (written *italic*) in this scenario can be combined and assigned to module types. Resulting from this scenario one can come up with the following module types. Some of those module types are mandatory in order to have a reasonable visual data exploration scenario while others are optional:

Module type:	**Tasks of module type:**
Data acquisition *(mandatory)*	• selection of data source • selection of subset of data • acquisition of data • offering data in a processible format
Preparation *(optional)*	• preparation of data
Transformation *(optional)*	• reducing the data to be analysed • transforming the data into an appropriate form
Visualisation *(mandatory)*	• presentation of data in a visual form • user interactions to manipulate the presentation

Having identified the module types a scenario containing the module types could look as shown in Figure 11. A constellation like that will result in three entirely different visualisations of the initially same raw data. While on the left side a scenario with all steps involved once is shown, modules of specific types could as well be chained as shown in the middle. On the right side it can be seen how modules defined as being optional could be skipped which in this example would lead to the visualisation of the unprocessed raw data:

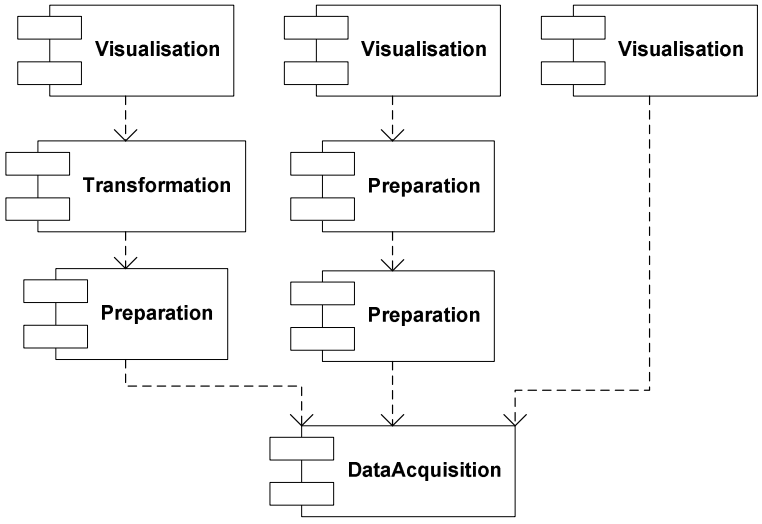

Figure 11 - Example scenario of visual data exploration

6.8. Distribution of modules

The data to be explored is typically not stored in a local database but rather **in some distributed database system** accessible through the company's network. Having the data acquisition module running on a local system while the data is located on some remote system, **the queries conducted by that module would be executed over the network and result in high network load**. For that reason it would make sense to be able to deploy the data acquisition module on a remote system and only transmit the selected and filtered data over the network. Lack of performance of desktop PCs might also be a reason to have some preparation or transformation steps being done on more

powerful, possibly remotely located systems. Therefore the need to deploy those kinds of modules on remote system might arise. Apart from performance issues a valid reason to distribute the modules could be to have the data acquisition and the preparation step done once and to have visualisation modules running on several systems to have the data presented to several users. Therefore the possibility to distribute modules is something that could be built into the framework or considered during the design to be added later on.

6.9. Conclusion

The requirements of the system were identified. The key requirement is that **users shall be able to explore data and deduce hypotheses they can profit from economically**. The framework will hold modules that can be connected, where the module closer to the data source is defined to be the parent module, the connected one is defined to be the child module. Modules were categorized into module types with dedicated tasks assigned to those module types. It was shown that it is essential to define those module types and the constraints on binding two modules. The connection or binding of modules has to obey rules that can be defined by each module's binding properties. **The introduction of these binding properties will allow developers of modules to formally specify possible module hierarchies**. Users of the framework should be able to add own modules fitting their needs, which means that the modules are not necessarily compiled together. This is an issue that has to be addressed in the design. The deployment of new modules needs to be simple for the user, the framework has to handle the integration into the system.

7. Design

7.1. Choice of technology

As the underlying technology will have an effect on the design of the framework a discussion about the technology to be used for later implementation shall be given here. The identified major requirements for the technology will be listed and the technologies introduced in chapter 4.2 will be evaluated based on those requirements:

Creation of extensible framework: The ability to create a framework that can be extended is a crucial requirement. Those kinds of frameworks can be built with all current technologies that support some form of packaging and extension. This is valid for all object oriented technologies.

Binding of modules at runtime: The ability to dynamically bind modules at runtime is essential in order to be capable of enhancing the framework: developers shall be able to develop own modules and deploy them. The framework itself should be responsible for loading and instantiating those modules. COM, EJB and .NET allow for modules to be loaded dynamically.

Easy to use technology: The main focus of the potential users of the framework is not software development but rather the exploration of data. For that reason the technology used should be easy to use. No highly specialised skills should be required. For that reason the COM technology will not be used, as it requires special skills to develop COM components. EJB components are as well difficult to implement and deploy for users with no experience in that area.

Stand-alone technology: Having to setup a heavy-weight runtime environment in order to get the framework and the application on top of it running would be something that

will discourage potential users. The EJB technology requires an application server and service-oriented approaches typically require a web server to be running. COM and .NET on the other hand require no additional systems to be installed – the runtime environment is assumed to be installed on current computer systems.

None of the existing component technologies (e.g. COM, EJB) was chosen to be used as it requires special skills which should be avoided in order to build an easy to use framework. The need for those kinds of technologies is not given as issues like concurrency or persistency are not relevant for the framework to be developed in this work. It was decided to use an off-the-shelf programming language.

7.2. The framework

7.2.1. Connecting modules

As described in 6.3 **there need to be rules for the connection of modules**. The framework should do as much checking as possible in order to disburden the user. Otherwise connecting two modules will be error-prone. The constraints regarding the connection of modules will be specified by a module's binding properties (6.6). The decision if two modules can be connected will be made based on the binding properties of the individual modules by a central instance of the framework – the module registry. Each module will specify which parent modules are acceptable. A module lookup can then be conducted according to the specifications made. Following design approaches could be used for this:

Specification of the accepted module instances: Each module could be identified by a unique module identification. This identification could be held inside each module being a property, accessible through the module's interface for lookup purposes. Modules could specify all modules that are acceptable parents by listing those modules'

identifications, which in the case of no restrictions on the accepted parents corresponds to a list of all modules of the framework. A major disadvantage of this approach is that users enhancing and thereby specialising certain modules will need to add the new module to the list. There is no relationship of the module types even there is a relationship in terms of implementation reuse by inheriting from a certain module. This approach would therefore lead to an inflexible framework and running systems that require change with every new module introduced to the system.

Specification of the accepted module types: Modules could specify the accepted parents by specifying those modules' types which corresponds to their classes. This way a property of object-orientated design, namely polymorphism, can be used to constrain the acceptable parents to an entire hierarchy of modules. For instance all modules of the class `DataAcquisitionModule` could be specified as accepted parents. If framework users extend this module type and add their own implementation to the framework, the newly added module can be connected without changing the modules' binding properties. This is due to the fact that inheritance corresponds to an "is-a" relationship and therefore e.g. a custom module named `NetworkData-AcquisitionModule` *is a* `DataAcquisitionModule`. Constraining the accepted modules to specific modules can be done by specifying more specific classes. Entirely different modules need to be listed separately, e.g. `DataAcquisitionModule, PreparationModule.` If all module types are accepted, it is sufficient to list the modules' base class. A disadvantage of this approach is that modules can be specified by inheritance but not the data types they supply. The specialisation of the data types is technically possible by extending a data type base class, but this will not be obvious in the specialised module's interface. So for example a module of type `DataAcquisitionModule` can supply data of the type `Table`

and the specialised module `NetworkDataAcquisitionModule` can supply `NetworkTable` – a subclass of `Table` – but in the interface of the class `NetworkDataAcquisitionModule` the method to get the `Table` must still return `Table` otherwise the contract of the base class will be broken which results in the class not compiling. If the subclass `NetworkTable` adds new methods a downcast will become necessary in the using module. This can lead to runtime errors if the module does not return the assumed type. Another disadvantage is that there is no way to specify the acceptable parents by means of the data they supply. Specifications like: "all modules returning `NetworkTable` are acceptable", are not possible.

Specification of the accepted data: Specifying the acceptable parent modules by listing the data types they may return is a more flexible approach. In order to make use of polymorphism an additional hierarchy of data types becomes necessary. Constraining the accepted parents would work equivalently to the approach mentioned above regarding the inheritance hierarchy. **This way the specification of the accepted parents is independent of the module types** which has the advantage of being able to specify the binding properties in the form of: "all modules supplying data of a certain type are acceptable". Hence it is not necessary to list module types, which has two advantages: Firstly if several module types return the same data type only one parameter needs to be specified and secondly if due to the evolution of the framework new module types should be added later on, there would be no need to change existing binding properties. Since many contemporary programming languages do not support the concept of multiple inheritance, those data types may not be base classes that can be extended because each module extends one of the framework's module base classes and hence cannot inherit from an additional class. The principle of an inheritance corresponding to an "is-a"-relationship would be broken as well. Therefore interfaces

need to be used that can be implemented by the individual modules which has the disadvantage that each module would have to implement the code necessary to satisfy the interface: this would be reuse of design but not of implementation. Those interfaces mark a certain module as being capable of supplying a specific data type, such an interface could be e.g. `NetworkTableSupplier` with a single method `getNetworkTable` (). This approach would add the idea of roles to the framework. A module would take the "role" of a certain data supplier. As the implementation of multiple interfaces is possible, one module could take more than one role and thereby offer its data in various forms.

Specification based on services: Each module could describe the service it offers to other modules – which would then be termed as service consumers. Those service descriptions would have to be of some standardized form and need to be supplied with each module type. Framework users would be free to describe the services their modules offer. Some form of service lookup would have to be developed that matches service requests to the modules offering requested services. As described in chapter 4.4 an advantage of using services is that it can be seen to be one level above object orientation and therefore would not rely on implementation details such as class or interface names that could be used for lookup purposes applying the other approaches. A major disadvantage of this approach is that the description of services would put a significant amount of extra work load on developers of modules and the service lookup would make the development of the framework itself more complex.

7.2.2. Design of individual modules

Framework users willing to develop new modules **should be able to do so in a rapid way and without the need to understand all implementation details** of the underlying framework. In chapter 5.2 the design of frameworks regarding the extension

of those frameworks was discussed. Obviously some tasks are identical for all modules in the framework, tasks such as the specification of the binding properties, the creation of connections to other modules and the acquisition of data from other modules. Two ways of supplying those generic tasks to module developers could be thought of: offering a collection of classes or offering a generic implementation that can be enhanced to fit the user's needs:

Collection of classes: Those classes could be referred to as helper classes and could be used by developers inside their developed modules. The developers would have to write the so called "glue code" in order to use those helper classes. This approach would require significant amount of code to be written by module developers and would require a deep understanding of the way the framework works in order to use the helper classes correctly. Developers would not only have to know how to use certain classes but be aware of the classes' existence in the first place. Supplying sample code and a good documentation could help here.

Generic implementation: Offering a base implementation that holds the workflows that are identical in all modules would allow developers of modules to start by extending a framework class and filling in hot spots as described in chapter 5.2 (extension using class inheritance). This approach was chosen as it results in a better ease-of-use. It shall therefore be discussed in the remaining parts of this chapter:

An abstract base class `AbstractModule` will be supplied to function as the base class for all modules. It holds members that *cannot*, members that *can* and members that *must* be overridden: Member functions that have fixed functionality – like getters and setters or the setup of module-connections – are not reasonable to be overridden and thereby be altered in subclasses. Programming language constructs are used to prevent those methods from being overridden. This restriction is essential for the ease-of-use of

this specific base class: developers cannot override methods that are not designed to be overridden. The second type of methods are those that *can* be overridden – their behaviour could be altered in subclasses. Those methods supply a valid default implementation. Finally a small portion of the methods *have to be overridden*. They are not implemented by the abstract base class – only abstract methods are supplied. Methods of this kind are e.g. the methods used for the initialization and the destruction of modules – those tasks are specific for each module. During initialization, data from files could be read or network connections could be set up. As those methods are abstract they are obvious to the developers as subclasses will not compile without those methods being implemented.

Resulting from the discussion in chapter 6.7 the types of modules will be restricted to five pre-defined types: `DataAcquisitionModule`, `PreparationModule`, `TransformationModule`, `VisualisationModule` and `Interactive-VisualisationModule` Due to that reason one layer of subclasses of `AbstractModule` can be supplied with the framework: there is one abstract subclass for each module type. So for example one of those subclasses is `DataAcquisition-Module` which can be used as the base class for all modules responsible for data acquisition.

It was decided that **a module can have one parent module and n child modules**. References to those modules are kept by the module itself. The code to administer those references is located in the base class. The class `AbstractModule` and thereby each of its subclasses holds an object of the class `ModuleType` which holds the information about the specific module type. Binding two modules is done based on the data held in the `BindingProperties` class which will be described in greater detail in chapter 7.2.4. In order to be able to uniquely identify a certain instance of a module, a unique id

is kept by each module instance. The following class diagram shows the described classes:

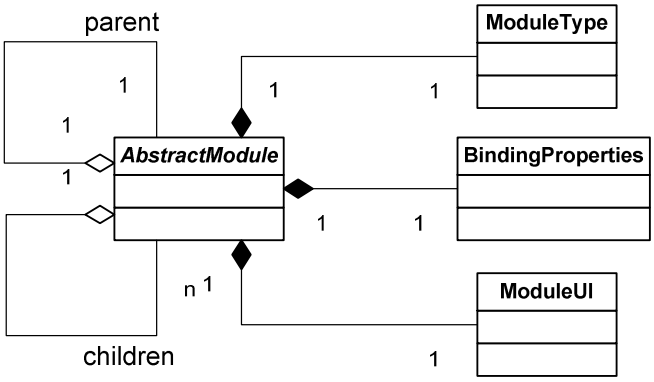

Figure 12 – Classes of a module

7.2.3. Overall structure

The modules described in the previous chapter can be connected which is technically done by assigning module B as the *parent* of module A and adding module A to B's list of *child* modules respectively. It should be mentioned, that in the case of chaining modules of the same module type, cyclic dependencies must be prevented: It must be prevented that a module may be bound to the same instance of that module or in other words a module may not be the child module of itself and none of the parent modules may be connected as a child module – this needs to be checked recursively:

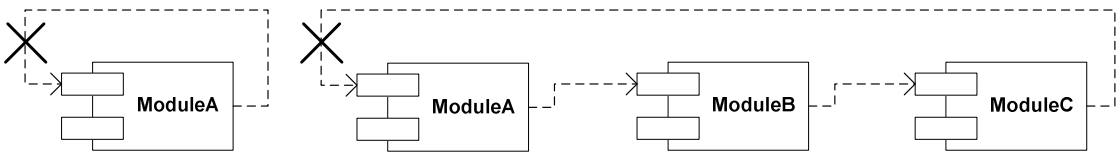

Figure 13 – Cyclic dependencies of modules

After successfully binding two modules the child module can process (e.g. visualise or prepare) the parent module's data and offer the result to its connected child modules. Together with the framework some modules are supplied and it is assumed that users

will add own modules as well which leads to a variety of different modules. To be able to structure setups created by users, the entity *project* (represented by the class `Project`) will be introduced. An application built on top of the framework can contain various projects and each project can contain various modules. To reduce the start up time, the modules of a project will only be loaded on activation of that project which is to be done by the user. A project's modules will be categorized by the basic module types (see chapter 6.7) that will be held by each module as objects of the class `ModuleType`. Modules to be added to a project will be added to the appropriate base module type. The configurations made within one project will be stored which means the framework will remember the modules of a project and their connections. The data to be analysed and the results will not be stored due to the vast amount of data. It will however be possible to store the resulting visualisation. Due to the user configurations being persistent it is essential to not only be able to add new projects and modules but also be able to remove them.

The relationship of the framework's most significant entity classes is as follows, where `VisualDataExplorer` is an application built on top of the framework and not an integral part of the framework:

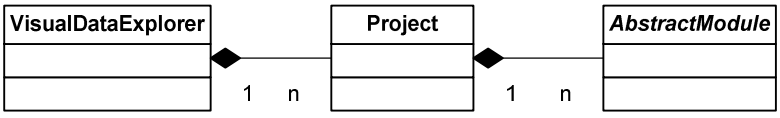

Figure 14 – Overview class diagram

A framework is typically used by various applications and runs in the "context" of those applications. Therefore the framework context is to be configured prior to using it which is done using the `FrameworkContext` singleton [GoF94] class. This class holds application options and a reference to the using application for callback reasons,

e.g. signalling that changes have occurred. The loading of all modules is controlled by an object of the class `ModuleAdministrator`, which registers all loaded modules with the `ModuleRegistry`. The configurations of the individual projects are administered by the class `ProjectAdministrator`. The steps necessary to start up the framework are shown in the collaboration diagram in Figure 15:

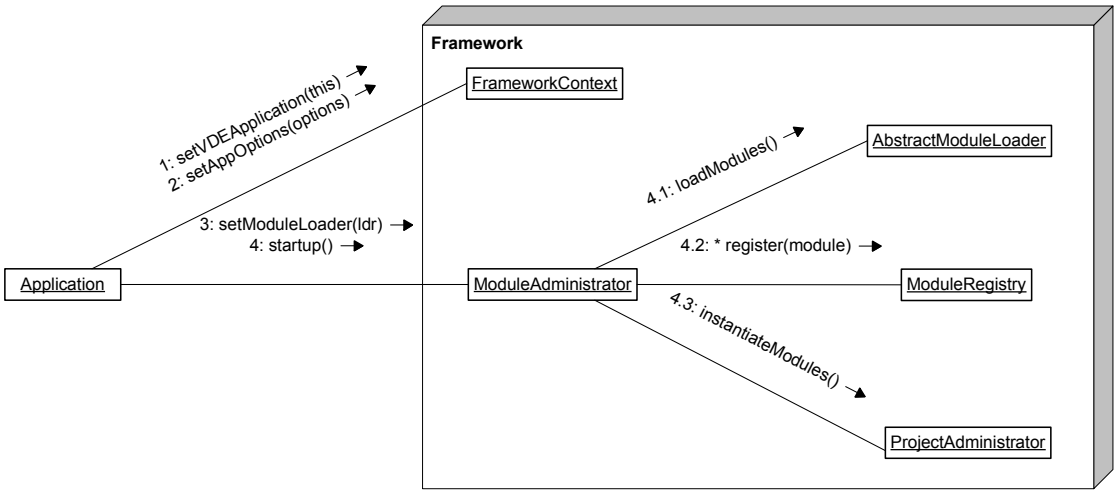

Figure 15 – Collaboration diagram of framework start up

The first step is to configure the framework context: a reference to the using application is passed for callback purposes (1) together with the stored application options (2). Then the module loader to be used is injected into the framework (3) and `startup()` is called on the `ModuleAdministrator` object (4), which is a singleton class [GoF94]. Following this call, all modules will be loaded using the passed module loader (4.1) and all loaded modules will be registered with the module registry (4.2). Afterwards the `ProjectAdministrator` object instantiates all modules of all projects (4.3).

7.2.4. Making the binding properties available

After having specified the binding properties in the analysis chapter (chapter 6.6), a way needs to be found to make those properties available technically. Some possible approaches will be discussed below:

Description files: Modules could specify their properties in files. Those files would have to obey a certain specification in order for other modules to be able to read them. An advantage of this solution is that the description files could be exchanged at runtime without recompilation of the module, which means the binding properties could be influenced by the user, not just the developer of a module. Disadvantages are that those files cannot be checked for syntactical correctness by some compiler. This would lead to errors occurring during the composition of the system. Additional checking tools could be developed and added to the framework but this would add complexity to the development phase of a module as the developer would need to specify his/her module in a file and afterwards use an additional tool for checking. Another disadvantage is that a module's code and its description are not directly linked – the module and its binding properties will be specified by two different means: code and some description file.

Meta-information: Many modern programming languages allow for meta-information to be added to the code referred to as "attributes" in the case of .NET or "annotations" in the case of Java [Kru05]. The information about the module can be directly added to the module's code. This makes the specification of the binding properties comfortable.

Architectural description languages: The use of architectural description languages (ADL) [Tor02] might also be used to describe the preferences and restrictions of each module as described in [Chat05] for a plug-in architecture. Code generators exist for that purpose, but a major disadvantage would be the additional complexity of that approach.

Binding properties class: The properties to be specified could be abstracted for all modules of the framework and described in a class. This would allow for compilers to check the description syntactically. Logical checking could easily be done before adding a module to the framework. The list of properties could easily be enhanced using inheritance and therefore defining a hierarchy of binding property classes. **This approach was chosen due to the ease-of-use and the possibility to supply default implementations**. Objects of the class `BindingProperties` will be held by each module. If those objects' values should not meet the user's needs they can be configured appropriately.

7.2.5. Extension points – hot spots

In this chapter the most relevant *extension points* or *hot spots* will be contemplated being the parts of the framework that can be extended or altered. As described in chapter 5.2 hot spots can take the form of methods in the case of extension by class inheritance or entire objects or classes respectively in the case of extension by object composition. Examples of how these two mechanisms are used in the design of the framework are shown in this section:

Extension by class inheritance:

Framework users wanting to contribute own modules have to decide on the module type based on the identified base module types (chapter 6.7). For each of these module types an abstract base class exists in the framework. This class is to be used as the base class for the implementation of own modules. As shown by the class diagram in Figure 16 each of those classes extends the framework's base class for all modules: `AbstractModule`. The first level of subclasses offers methods specific for one module type. Developers extend one of those base types and have to implement the hot spots. Additionally a visualisation module can either allow the user to directly

manipulate the visualisation or be restricted to pure presentation. To distinguish between those two cases, a subclass – `InteractiveVisualisationModule` – is supplied with the framework, which e.g. holds methods required for mouse interaction:

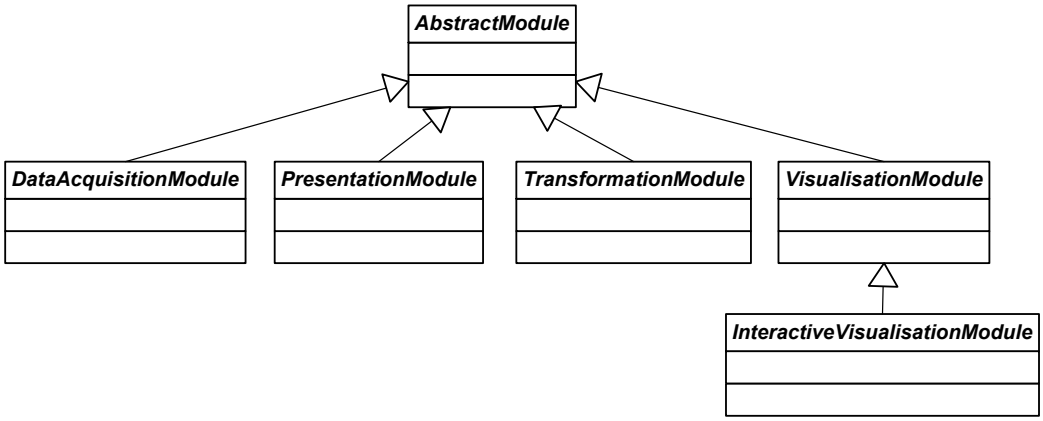

Figure 16 – Module base classes

Apart from distinguishing between interactive and non-interactive visualisation modules, a visualisation module can optionally offer textual output to e.g. print tables or statistical values. Representing the capability to print text in form of subtypes would lead to four different base classes of visualisation modules which was considered to be too complex from the user's point of view. A method indicating that capability was instead added to the visualisation module class (`hasTextualOutput()`) which can be overridden to indicate a subclass' ability to print text. A disadvantage of having a class hierarchy of visualisation modules is that it is not possible to implement a non-interactive visualisation module and later on add an interactive module based on the former one. This is due to the fact that multiple inheritance is not supported by many current programming languages and the latter module would have to inherit from `InteractiveVisualisationModule`. The shared code has to be factored out in a helper class by the module developer. The module classes mentioned above correspond to the *white box layer* of the framework.

Extension by object composition:

Modules can either be supplied with the framework – some default implementations and commonly used modules will – or be contributed by application developers using the framework. The latter point means that the integration of modules that are not known at development time of the framework must be possible. Therefore a way to dynamically load modules needs to be offered. The default implementation supplied with the framework will take the name of a folder on the hard drive and recursively load all modules located there or in any subfolders. This behaviour will most likely be sufficient for most scenarios: all developed modules need to be copied into a specific folder and the framework will start using them. Yet, cases could be thought of, where the loading of user modules should act differently: e.g. if modules are developed by more than one developer in a distributed environment, searching for modules across network resources might become necessary. Therefore the loading of modules was chosen to be a hot spot – a part of the framework that can be altered by application developers. Extension by object composition was chosen to be used here:

The abstract base class `AbstractModuleLoader` is used as the base class for all module loaders. The algorithm for loading modules will be encapsulated in subclasses according to the template design pattern [GoF94]. One of those subclasses is supplied with the framework being the above mentioned class loading modules from a certain directory: `LocalModuleLoader`. An object of the module loader to be used needs to be configured (e.g. setting the root directory or specifying security settings) and afterwards passed to the framework class loading the modules: the `ModuleAdministrator` class. This class and hence the framework itself works with the type `AbstractModuleLoader` only, the way the modules are really loaded is therefore transparent to the framework and depends purely on the passed object.

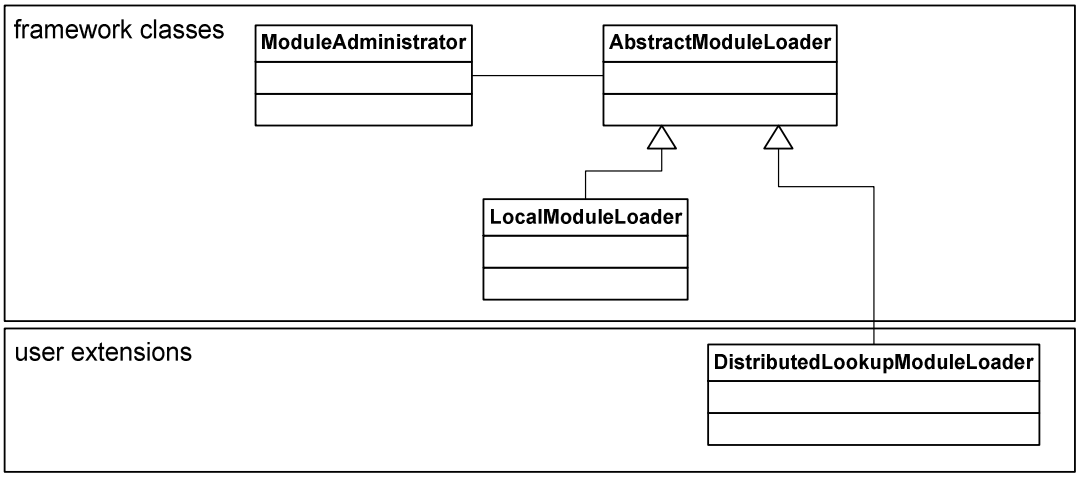

Figure 17 – Setting the module loader using object composition

This technique is also referred to as *inversion of control* or *dependency injection* [Fow04]. If the behaviour of the default implementation should not be sufficient, an additional subclass needs to be developed and passed to the framework class.

7.3. The application Visual Data Explorer

A framework itself cannot be instantiated or run without an application built on top of it, therefore an application needs to be developed using the designed framework to allow users to explore data. Offering the framework without an application would discourage potential users, as an enormous amount of work was necessary before data can be explored. If this application should not fit the users' needs, own applications can be built on top of the framework. The design of this default application named V*isual Data Explorer* will be briefly described here:

A graphical user interface is required to comfortably administer the projects and modules. As described in chapter 7.2.3 "projects" exist to structure visual data exploration scenarios. The user interface will be subdivided into a part used for administration purposes and one showing the modules of the exploration process. There will be one view showing a project's modules based on the modules' base types and one visualising the interconnections of those modules. It could be argued that this graphical

front-end is an integral part of the framework as it is shipped together. For this work the application Visual Data Explorer is defined not be a part of the framework.

7.4. Conclusion

Prior to starting the design, existing component technologies were evaluated based on identified requirements for the framework to be built. Essential is extensibility of the framework, the ability to dynamically integrate modules into the running system, the ease of use and the ability to run stand alone. The conclusion was drawn, that none of those component technologies will be used but rather an off-the-shelf programming language. Following this, the design of the architecture was discussed. It was decided to have a central instance responsible for module registration and administration – the module registry. The decision if two specific modules can be connected will be made by this instance based on each module's binding properties. This way the information which modules can act as parent modules is placed inside each specific module and can be specified by developers adding new modules. It was decided that a module can have one parent module and n child modules.

Some of the modules' tasks were identified to be identical for all modules in the framework. Those tasks need to be integrated into the framework. Two ways of doing so were identified: offering a collection of classes or offering generic implementations that can be enhanced to fit the user's needs. The latter approach was chosen, as developers would have to write so called "glue code" using the former approach. With generic implementation – or base classes in the object-oriented sense – developers are guided what code needs to be supplied. Developers need to fill in the so called hot spots or extension points of the base classes. Extending the framework can either be done by class inheritance or object composition.

8. Implementation

As discussed in chapter 7.1 the framework was developed using no existing component technology, but rather using an off-the-shelf programming language. As the technology should be well-known and widely used, pure Java or .NET were contemplated in greater detail as the technology to be used for implementation. Due to the fact that Java is currently more widely used and is considered to be *really* platform independent – .NET *officially* runs on Microsoft operating systems only – pure Java was chosen. Porting the framework to another technology if required, e.g. .NET, should be straight-forward. The framework was implemented using the Java programming language in its version 5.0. The application Visual Data Explorer and some relevant implementation details will be shown in this chapter.

8.1. Implementation of the Visual Data Explorer

The application Visual Data Explorer is a graphical front-end on top of the developed framework. It mainly consists of the code required to build the user interface which was implemented with Java's Swing Class Library. Each module's user interface is part of the module itself and thereby part of the framework code. The look of the Visual Data Explorer is shown in Figure 18. The user interface is subdivided into one part used for administration purposes like adding projects and adding modules to existing projects which can be seen on the left side and one part showing the data exploration process shown on the right side:

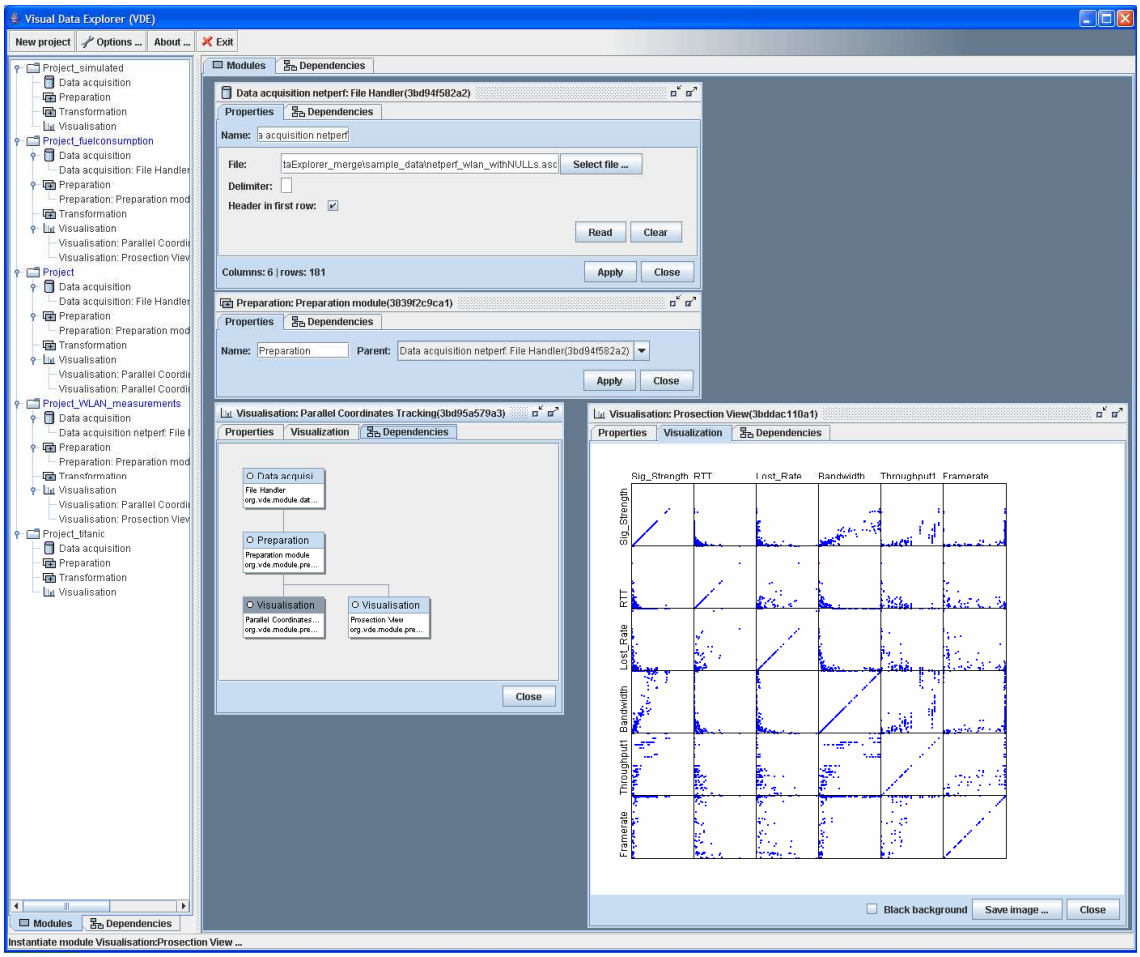

Figure 18 – Screenshot of the Visual Data Explorer Application

Tree views are used for the administration of modules: one holds a list of all projects together with the modules of this project assigned to their base module types, e.g. one node represents the base module type `DataAcquisitionModule` which in turn has child nodes that represent all added data acquisition modules. This tree view can be switched to another view showing the modules' dependencies. The right part of the user interface shows all modules of the selected project, where each module is represented by its own user interface. In this part of the GUI users connect the modules by selecting the parents, configure the data exploration process by configuring the appropriate modules or directly interact with the resulting visualisation in the case of visualisation modules supporting user interaction.

In Figure 18 the top module represents a data acquisition module reading data from an input file. Below that module a preparation module, filling in missing values into the data supplied by the data acquisition module, is placed. The two remaining modules are visualisation modules, where the left one does not show the visualisation of data but rather the dependency tree of the currently connected modules. The visualisation module on the right side visualises data supplied by the preparation module.

8.2. Dynamic loading of modules

Modules in this framework correspond to Java classes and therefore to Java `.class` files. To be able to dynamically load modules a way to dynamically load those `.class` files had to be found. As described in the previous chapter, the loading is done by a class named `LocalModuleLoader`. The way this class functions shall be briefly discussed here:

Starting from a root directory on the hard drive the module loader will recursively search for `.class` files. As in Java the directory structure on the hard drive corresponds to the package structure, the module classes need to be copied into the search directory together with the existing directory structure. Having found a Java class, the current directory is added to the Java classpath [Kru05]. This step is necessary to be able to load that specific class. Afterwards the class is loaded using Java's `URLClassLoader`. Before instantiating the class using the reflection mechanism `Class.newInstance()` two things need to be checked: firstly the class needs to inherit from `AbstractModule` and secondly it may not be abstract. If both conditions hold, the class will be instantiated and added to the list of loaded modules. Instantiating the classes this way puts the limitations on the module classes that they have to have a default constructor, otherwise the `newInstance()`-call will fail.

If other technologies were used, e.g. .NET, COM or standard C++, loading modules would function slightly different: in those cases the module classes would not be stored as individual files on the hard drive but would rather have to be put in some library (DLL in the case of Windows operating systems). This would be a heavy-weight approach if several modules were to be added to the framework. Copying the Java `.class` files is much simpler and therefore meets the goal to create a framework that is *simple* to use.

8.3. Programming language features useful for framework development

Features of the programming language Java were used in order to reduce the risk of using the framework incorrectly and to indicate where developers need to contribute own code:

Classes not designed to be extensible were marked to be non-extensible using the keyword `final` which leads to compiler errors if developers try to inherit from such a class. In the case of classes designed for extension, those methods not designed to be altered were marked `final` as well.

It is common practise in object oriented design to keep internal parts non-accessible to developers which is typically done by applying the appropriate access modifiers: `protected` or `private`. For the design of frameworks this is a highly important aspect: while in the case of methods or fields the restriction of the access is usually straight-forward, with framework APIs the developers should not even "see" *classes* not meant to be used from outside the framework – their existence should be completely hidden. This was technically achieved by applying the access modifiers on the classes. For code required to be implemented by the user in order to have a running module,

`abstract` methods in the base classes were used. Not implementing those methods leads to compiler errors.

Additionally it should be mentioned that the programming language features used and described above exist in other programming languages as well, e.g. in .NET `sealed` would have to be used to prohibit class inheritance.

9. Evaluation

9.1. Evaluation of the design

In order to evaluate the usability of the framework a typical visual data exploration scenario that requires to supply own modules will be set up. This evaluation is conducted based on measurements taken in a wireless network that are to be analyzed. The data is in the form of a formatted ASCII text file with some values being undefined. All steps necessary to come up with a working solution based on the framework will be discussed. Those steps are:

1. *Identifying which existing modules can be used*

2. *Development of the missing modules*
    a. *identifying if the new modules could be based on existing modules*
    b. *implementation of modules based on existing modules or on the framework's base types*

3. *Deployment of the developed modules*

4. *Setting up the visual data exploration scenario using the application Visual Data Explorer*
    a. *Creation of a project*
    b. *Addition of the necessary modules*
    c. *Connection of the modules according to the scenario*

1. Identifying existing modules: Following the list, the process starts by identifying which existing modules can be integrated into the exploration process. The data is to be visualised using the parallel coordinates and prosection view visualisation techniques. Modules for both visualisation techniques exist.

2. Developing missing modules: In order to parse the specific file format, a data acquisition module will be developed. This module offers the parsed data, which

contains columns without values and therefore needs further preparation. An additional module will be developed that substitutes the missing values with the column's mean value. Both modules will be based on the framework's base types.

The first step is the choice for the appropriate base module type. If directly extending one of the framework's base classes, the hot spots need to be filled in. Those are methods for initialization and destruction of the module: `internalInit()` and `internalDestroy()`. The `paint()`-method has to be implemented – it allows modules to present any kind of graphics to the user. The data type to offer needs to be specified, e.g. a table, a cube, etc. For common data types the framework offers interfaces, e.g. `TableSupplier` with the method `getTable()`. Other modules will retrieve the data via this method. In order for the *Visual Data Explorer* to determine whether a module is "active" the `isActive()`-method is part of the `AbstractModule` class and needs to be implemented by the subclasses.

The module's binding properties need to be configured as discussed in chapter 6.6. The `getBindingProperties()`-method of each module returns an object of the class `BindingProperties` that holds the module's information about possible connections to other modules.

Developing the module reading the input file:

As this module's task is to acquire data from an input file, it is of the type "data acquisition module" and therefore uses the framework's class `DataAcquisition-Module` as its base class. The module's user interface has to offer user interface controls that allow to select the input file and to specify the separator used to distinguish the columns. This user interface is set up inside the `internalInit()` method. A button allows the user to trigger the parsing of the specified file, the parsed data is

stored in the module's data field. The data is structured in a tabular way containing rows and columns. Therefore the framework's data type `Table` is used and the module implements the `TableSupplier`-interface. Calling that interface's method `getTable()` supplies the parsed data. For administration purposes the `getDescription()`-method is overridden offering a brief description of the module's task. Additionally the `isActive()`-method needs to be implemented. This module is defined to be active when the data has been parsed. As no special visualisation is necessary the `paint()`-method is left empty. No further code needs to be written for this module. Default binding properties for data acquisition modules are specified in the framework's base class – amongst other properties, the restriction that a data acquisition module may not have any parent modules, is specified there. Those binding properties are sufficient for this specific module and therefore the `getBindingProperties()`-method does not have to be overridden.

Developing the module filling in the missing values:

Filling in missing values into the data to be analysed is assigned to the "preparation-step" of the visual data exploration process. Therefore the framework's `PreparationModule` class is the appropriate base class for this module. The prepared data will be of tabular form and therefore the module implements the `TableSupplier`-interface. Inside the `getTable()`-method the data of the connected parent module is read using that module's `getTable()`-method. Each column's mean value is calculated and filled in where values are missing in the original data. Additional code is not required to be written which makes this module quite simple to implement. The number of lines of user code is in the region of 20.

3. Deploying the new modules: The deployment of the developed modules takes place by copying the compiled modules into the framework's search path. This search path

might have to be configured using the application Visual Data Explorer if the default setting is not appropriate.

4. Setting up the visual data exploration scenario: In order to set up one exploration scenario, a project is to be created in the Visual Data Explorer. A project has subnodes for each module type of the framework:

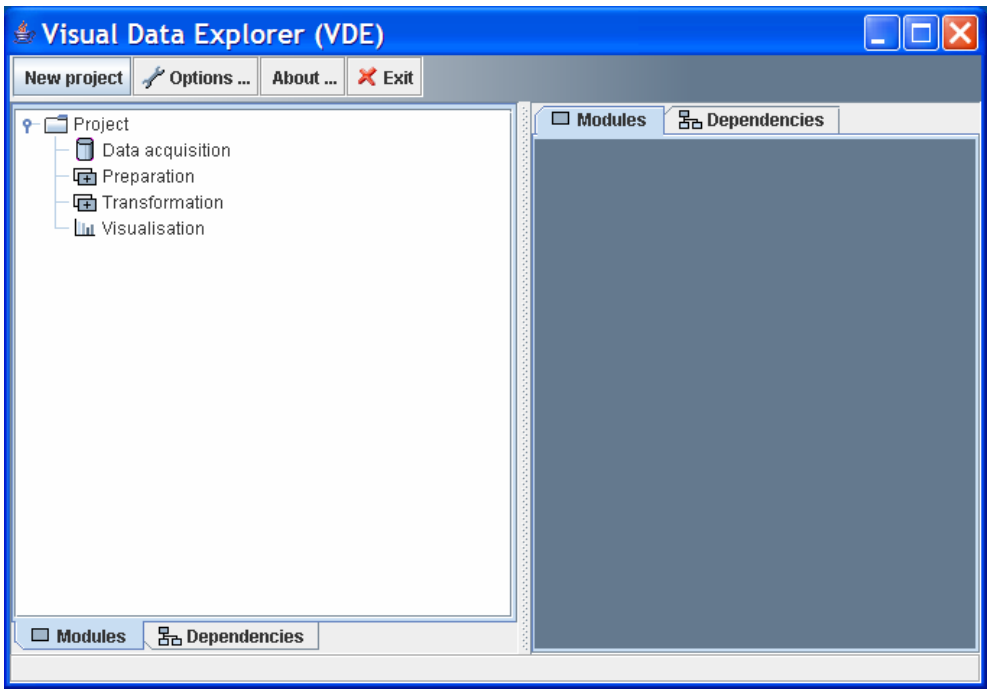

Figure 19 – Creation of project

The required modules have to be assigned to the project. The application will offer all available modules of the module type the context menu was opened for (Figure 20). The list of offered modules contains all modules that are supplied together with the framework as well as all modules found in the search path.

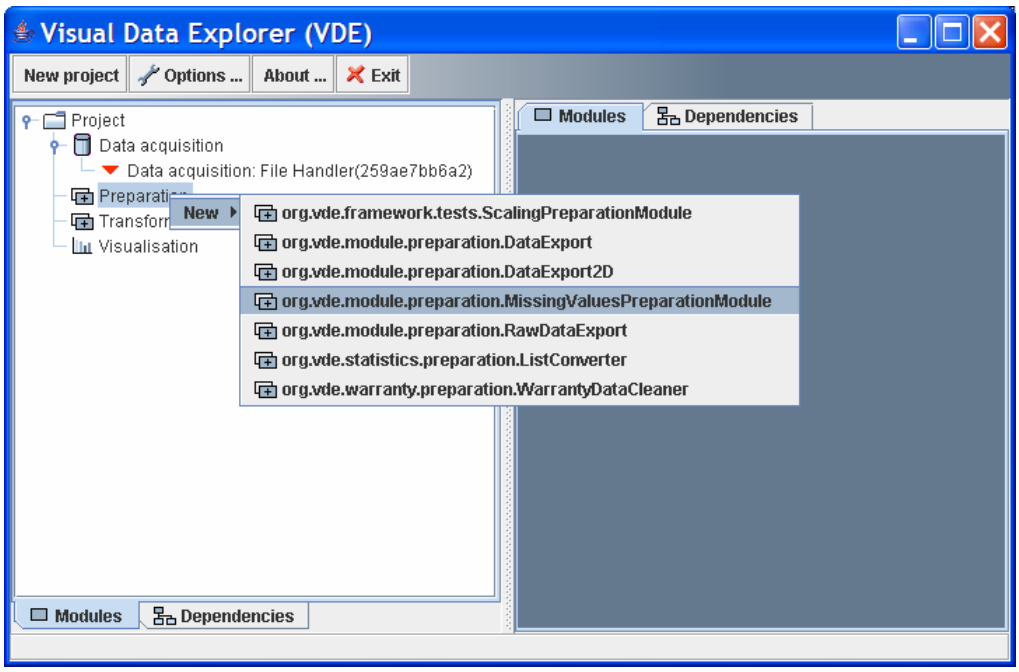

Figure 20 – Adding modules to a project

Modules within a project need to be connected to determine the flow of data. This is done using the modules' user interfaces. The "parent" module can be selected there. The list of offered modules is determined based on the binding properties of the project's modules. After connecting all modules, the data will be visualised. In order to come up with a hypothesis, several iterations might be necessary. Users might change module configurations or even adapt the implementation of individual modules:

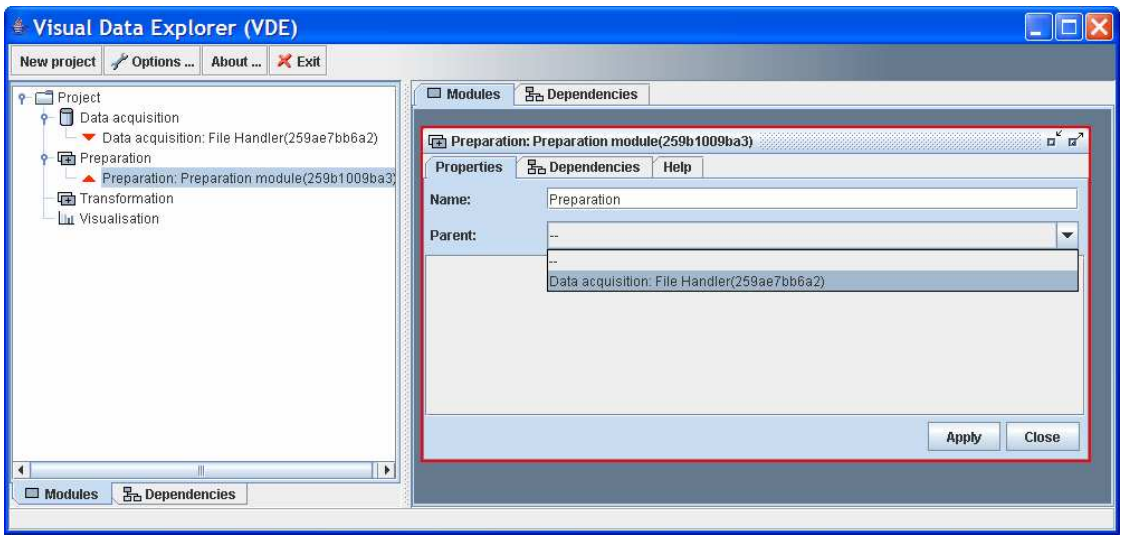

Figure 21 – Connecting modules of a project

The described development of the two modules showed that rapid development of new modules is indeed possible. Especially the development of the preparation module was fast and simple. One challenge a module developer faces is the need to choose the module's base class and the choice for a data supplier interface. Typically one of the framework's five abstract base classes will be chosen to be extended. If an existing module can function as the base class, this can be done as well. As it seems the data type of the data to be analysed in data exploration processes cannot be abstracted reasonably: e.g. a table, a cube and key-value lists differ significantly. A way to come up with a hierarchy that would allow the framework's modules to e.g. contain a `getter()`-method for the data with the data type being specialised as the module class itself is specialised by supplying subclasses (see chapter 7.2.1) could therefore not be found. The framework design rather contains data supplier interfaces, supplying specific data types that need to be implemented by the module. This is one fact that might lead to confusions when supplying own modules and could lead to runtime errors when the binding properties are not set properly allowing users to connect two modules with differing data types.

9.2. Exploring simulated data

The following discussion will be based on data to be explored that is stored in a table. The columns of that table correspond to the columns of database tables or in other words each column represents one parameter. Each row corresponds to one dataset.

Finding dependencies between different parameters is a non-trivial process if the datasets are multi-dimensional and the number of datasets is huge. The following discussion shows how the developed framework can be used to identify such dependencies or correlations. For evaluation purposes the data to be analysed will reveal information which is obviously correct in order to show the correctness of the

framework behaviour. As described in chapter 3.1 the outcome of a data exploration process will typically be some hypothesis that needs further confirmation.

To be able to conduct contemplations, a special module was developed that creates simulated data based on users' configurations. The values are generated randomly within a specified range. Two of the parameters of this simulated data are dependent: the values of column 1 are dependent on the values of column 0. Simple dependencies like this are trivial to be identified in the case of two or three-dimensional data by visualising the data as a mathematical function of one or two variables respectively, i.e. `y=f(x)` or `y=f(x,z)`. In this example 10-dimensional data was chosen to be analysed.

In order to find dependencies in multi-dimensional data the k dimensions need to be mapped onto the two dimensions of the screen. This can be done using *prosection views*, which are a composition of sections and projections, where projections are simple mathematical functions of one variable (`y=f(x)`) [Fur93]. Each two parameters are related and the result is presented in the form of an x-y plot, which leads to a number different relations. The relations where a parameter is related to itself can be ignored; half of the remaining x-y plots could be neglected as well, as they represent inverse functions. This leads to a number `n` of relevant relations to be contemplated:

$$n = \frac{k^2 - k}{2} \qquad \text{k: number of dimension}$$

A prosection visualisation module was connected to the mentioned simulated data module in order to visualise the generated data. The result was exported using the built-in functionality to save the visualisation as a `png`-file. This export-functionality could

be seen as the implementation of Shneiderman's "extract"-step in his information seeking mantra (see chapter 3.1):

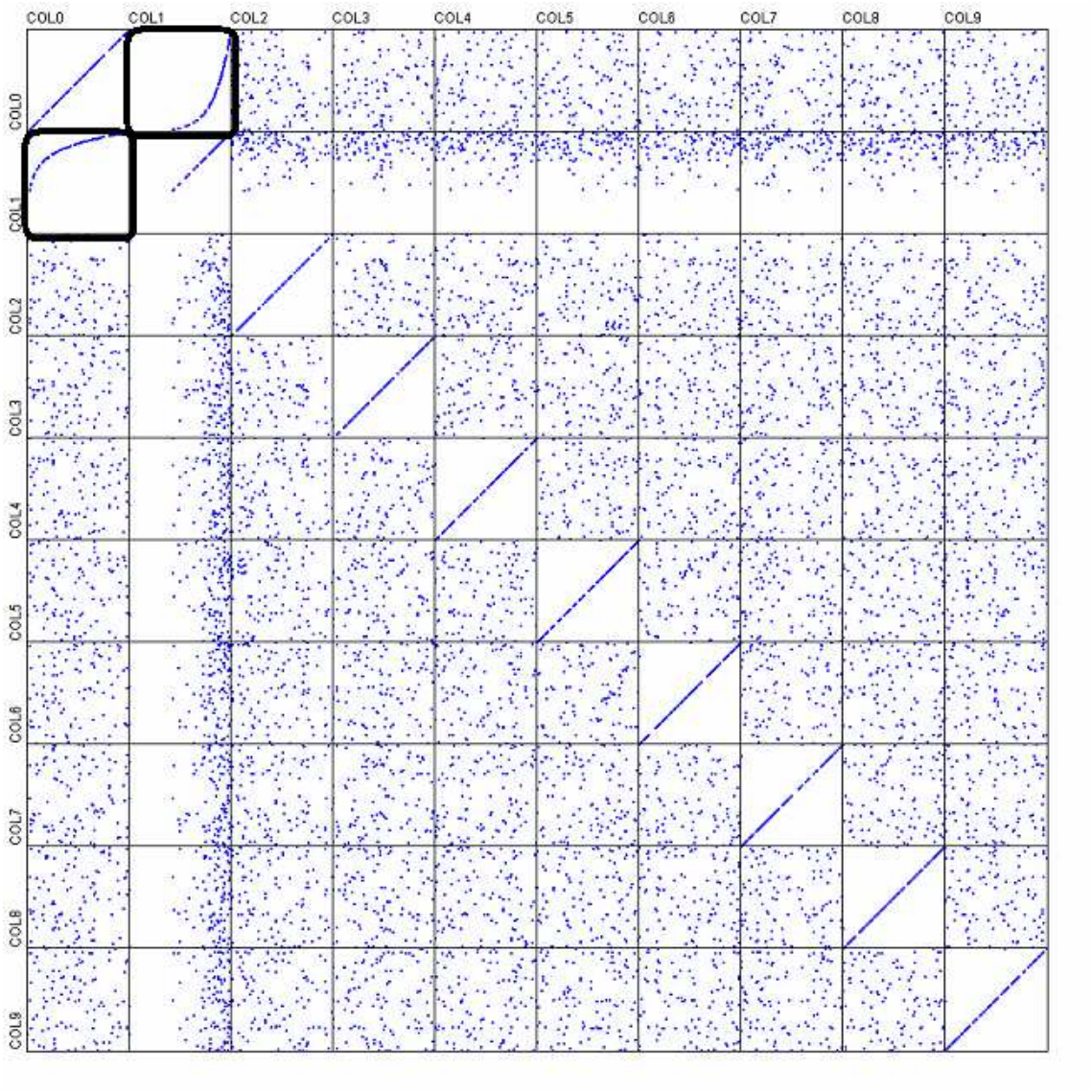

Figure 22 – Prosection view of simulated data

The diagonal, where each parameter is related to itself is irrelevant. It results in a linear function, as in the case of `y=f(x)`, x equals y. The x-y plots where column 0 (col0) is related to column 1 (col1) and vice versa identify the dependencies between those two parameters. In this example the dependency is of the type:

$$col1 = f(col0) = \log col0$$

This leads to the inverse function:

$$col0 = f(col1) = e^{col1}$$

It should be added, that in this contemplation the input data can be viewed as ideal, as the dependency is based on a mathematical function mapping the values of column 0 to column 1. Conducting this sort of investigation on real data will most likely not reveal a function as clear as shown here due to e.g. outliers. In this case the raw data of the data acquisition module might need some preparation which can be done by an appropriate preparation module.

9.3. Exploring real, measured data

A second evaluation was done based on measurements taken by the Worchester Polytechnic Institute while streaming video on a wireless network. The data, which was taken from `davis.wpi.edu/~xmdv`, has six dimensions: signal strength, round trip time, lost rate, bandwidth, throughput and the frame rate.

The visual data exploration techniques applied on that data were parallel coordinates and prosection view. In order to visualise k dimensions, the *parallel coordinates technique* [Few06] [Hau02] uses k equidistant axes. Each of them is linearly scaled from the minimum to the maximum value with data items intersecting the axes at the point that corresponds to the value of the parameter. To highlight parts of the data *brushing* is used, which allows to select subsets of the data. The possibility to select a subset of data items was proposed by Shneiderman [Shnei96] who termed it “relate” (see chapter 3.1):

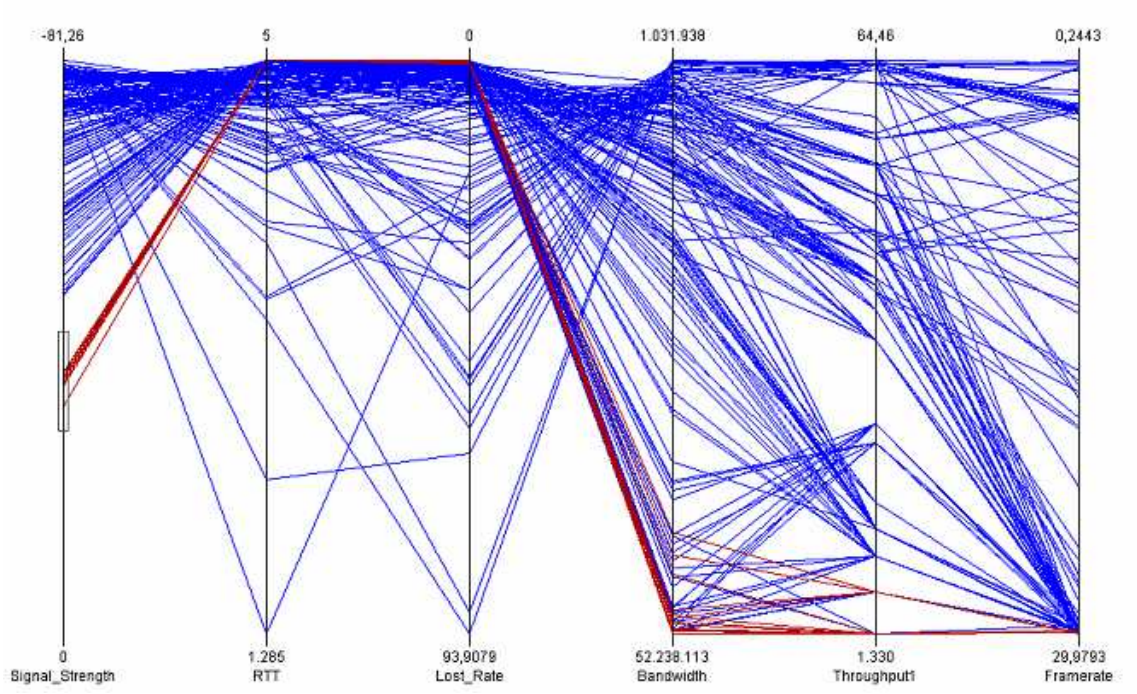

Figure 23 – Parallel coordinates showing WLAN measurements

The measurements with the highest signal strength were selected to be highlighted. A correlation becomes obvious between all six parameters: if the signal strength is high, the round trip time as well as the lost rate of packets is at its minimum. The bandwidth together with the throughput and the frame rate is at its maximum. Applying a prosection view on the same data reveals the dependency between the bandwidth and the throughput: high bandwidth typically, but not necessarily, results in high throughput:

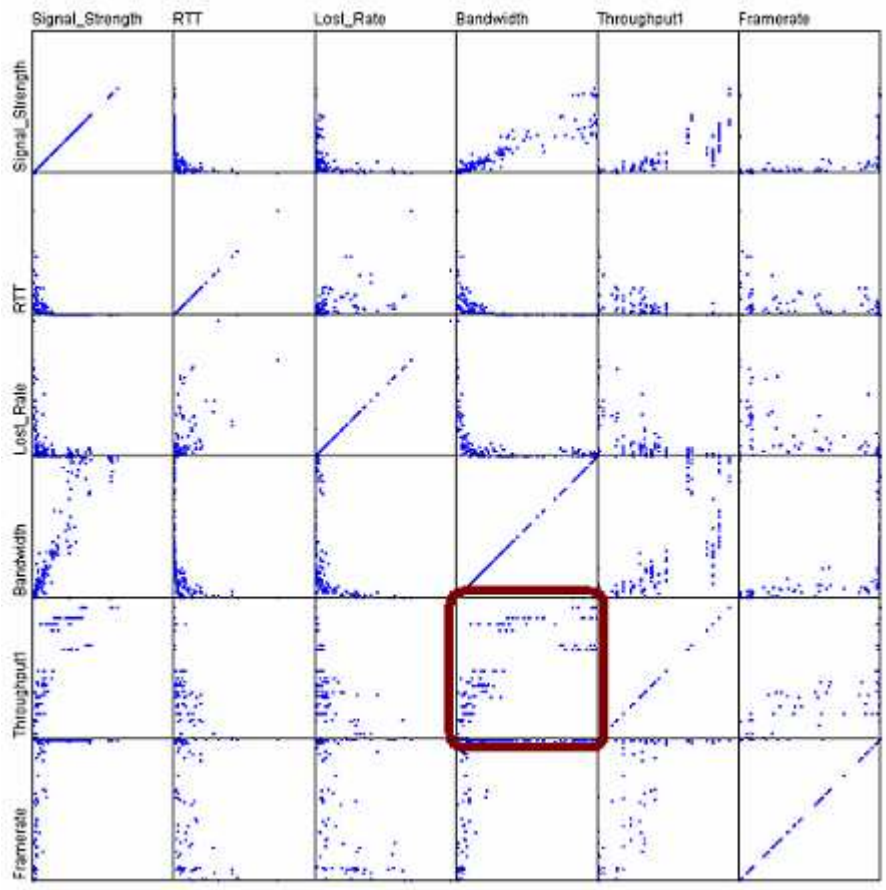

Figure 24 – Prosection view showing WLAN measurements

9.4. Final evaluation

In order for the framework to be used commercially, the usability of the API would have to be evaluated conducting usability tests. In the case of a commercial product one would come up with an alpha and a beta version of the framework, have developers using it and improve the API along the way. Naming is a highly important aspect here: it cannot be said, that all developers have the same understanding of the framework's classes or methods. Terms introduced in this work, e.g. *binding properties*, the idea of *supplier-interfaces* etc. need to be made clear to the users.

A limitation is that the framework's modules currently work on numbers only. Enhancing the framework in order to work on strings, timestamps and further data types found in databases should be possible without major changes.

Exchanging modules at runtime was thought of but not implemented. It should however be possible to update the list of currently available modules triggered by events signalled by the specific module loader classes. Such an event could be the removal of files from the module search path.

The *Visual Data Explorer* application built on top of the framework would require a help system and would have to be made more stable.

10. Conclusion and Outlook

In this work, a framework together with an application (the *Visual Data Explorer*) was designed that allows for data to be explored applying visual data exploration techniques. Common exploration techniques have been introduced and their need been discussed. The key requirements were identified and the design was conducted based on those requirements. One crucial achievement of this work is the mapping of the visual data exploration theory to a framework covering or building the basis for all visual data exploration scenarios. The outcome of this work has been evaluated and **it could be shown that users can indeed profit from the developed framework by being able to come up with valuable results** from data exploration processes. The relevance and the mightiness of data exploration processes became clear by exploring measured data.

Guidelines for framework design were shown and obeyed where reasonable. Two different approaches for building extensible frameworks have been discussed: class inheritance and object composition. Both of them were integrated in the design. The difference between white box layer and black box layer were discussed and the usefulness of both could be shown: the white box layer offers base classes with so called hot spots that need to be extended and the hot spots filled in. The black box layer on the other hand offers valid default implementations that allow users to start building applications without extending framework classes. The developed framework contains both layers. A conclusion that can be drawn is that **frameworks should generally contain those both layers** in order to allow developers a rapid entry.

It was shown that by building the framework in a modular way, the aim to have a *low-barrier-to-entry* [Cwa06] could be reached, as the supplied modules can be used to start with. Only if those do not fit the user's needs new modules need to be added based on existing modules or on the framework's base types. To extend modules it is not

necessary to understand the inner workings of the framework which results in a framework that is *simple* to use – a goal identified in chapter 6.2. In the evaluation of the design it could be shown that **the development of new modules is simple and does not require software engineering skills**. The modules were categorized into module types that correspond to steps of the visual data exploration process which helps users with a visual data exploration background to get acquainted with the framework. Generally it can be said that mapping keywords of the application domain – of the theory of visual data exploration in this case – to names inside the framework code, increases the usability of the framework, as developers find "their terms" inside the framework.

The development of sample modules in the evaluation phase of this work showed that the aim to build a framework that allows users to rapidly add own modules could indeed be reached. **The usefulness of the framework could be proven by analysing real-world data** and coming up with a result that is obviously correct: measurements taken in a wireless network were analysed for this purpose. The number of modules supplied with the framework is limited but due to the extensibility **this work can function as the basis for a mighty visual data exploration system**. Several users could contribute modules and offer them for exchange.

More globally, contemplating the future of visual data exploration one can come up with the conclusion that this area will gain importance in the field of data analysis. The growth of storage capacity is exponential, it is estimated to be somewhere in the region of 40% per year [Sea06]. Due to that growth the amount of stored data will increase tremendously as well. Identifying interesting patterns in data will therefore become increasingly harder and will thereby require more intelligent exploration tools.

If various users should start using this framework, export and import functionality would be very helpful: data exploration scenarios can become complex if several modules are being used and configured. It should be possible to export entire projects in order to make them available to other users.

The framework can be used to search for certain patterns in data. If those patterns are identified, the results could be integrated into other existing applications. For that reason the necessity to share the module code might arise. Some way to make the business logic part of the modules (e.g. code to prepare or to visualise data) available to other applications could be thought of.

The optional aggregation of multiple visualisation modules into one view would allow seamless interaction with the visualisations. The manipulation – e.g. zooming or selection of small partitions of the data – of one visualisation should result in appropriate changes in the remaining visualisations.

The distribution of modules has been discussed in chapter 6.8 but not been implemented. To set up the communication between the modules would be possible using Java's RMI technology [Hei05]. Minor code changes to the framework and to the existing modules would become necessary. A challenge would be the distributed look-up of modules and the setup of a central instance to administer all modules.

11. Acknowledgements

This work was conducted in connection with the PhD thesis of Frank Müller-Hofmann. Thanks to him for helping me to get into the topic of visual data exploration and for interesting discussions.

Thank you to Dr. Ian Dear of Brunel University for supervising my dissertation.

Thank you to Marco Hentschel for discussions about design decisions and Java implementation details.

Thanks to Kalin Kotzev for proof-reading my dissertation and his valuable feedback.

12. Bibliography

12.1. Visual Data Exploration

[Fayy96] Fayyad U. M., Piatetsky-Shapiro G., Smyth P. (1996), *From Data Mining to Knowledge Discovery: An Overview*, American Association for Artificial Intelligence Press, 1996.

[Few06] Stephen Few (2006), *Multivariate Analysis Using Parallel Coordinates*, Perceptual Edge – Visual Business Inteligence

[Fur93] George W. Furnas, Andreas. Buja (1993), *Prosections views: Dimensional inference through sections and projections*, Journal of Computational and Graphical Statistics, vol. 3, no. 4, pp. 323–353, 1994.

[Han01] Jiawei Han, Micheline Kamber (2001), *Data Mining – Concepts and Techniques*, Morgan Kaufmann Publishers

[Hau02] Helwig Hauser, Florian Ledermann, Helmut Doleisch (2002), *Angular Brushing of Extended Parallel Coordinates*, In Proc. of the IEEE Symposium on Information Visualization 2002.

[Kei01] Daniel A. Keim (2001), *Visual Exploration of Large Data Sets.* Communications of the ACM, August 2001/Vol.44

[Kei02] Daniel A. Keim (2002), *Information Visualization and Data Mining.* IEEE Transaction on Visualization and Computer Graphics. Vol.7, Jan-Mar 2002.

[Kei97] Daniel A. Keim (1997), *Visual Techniques for Exploring Databases*. University of Halle-Wittenberg

[Mül05] Frank Müller-Hofmann (2005), *Visual Data Exploration*.

[Sea06] *Seagate Outlines the Future of Storage*, Published by Vijay Anand on Friday, 27th January, 2006.
http://www.hardwarezone.com/articles/view.php?cid=1&id=1805&pg=2
(Taken 26.01.2007)

[Sen] Tam Weng Seng, *Visual Data Exploration Techniques for System Administration*

[Shnei96] Ben Shneiderman (1996), *The Eyes Have It: A Task By Data Type Taxonomy for Information Visualisation.* In Proceedings of Visual Languages. IEEE Computer Science Press.

12.2. Software engineering

[Bal01] C. Baldwin, K.Clark (2001), *Modularity after the Crash*. Harvard Business School Working.

[Blo06] Joshua Bloch (2006), *How to design a good API and why it matters.* In Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 2006.

[Can] Carlos Canal, *On the dynamic adaptation of component behaviour*. Universidad de Malaga, Spain

[Chat05] Robert Chatley (2005), *Predictable Dynamic Plugin Architectures*. University of London and Imperial College London, PhD Thesis.

[Chat1] Robert Chatley, Susan Eisenbach, Jeff Magee, *Modelling a Framework for Plugins*. Imperial College of London

[Chat2] Robert Chatley, Susan Eisenbach, Jeff Magee, *Magic Beans: a Platform for Deploying Plugin Components*

[Chap02] Davis Chapman (2002), *Teach Yourself Visual C++.NET in 21 Days*. Sams Publishing

[Cou05] George Coulouris, Jean Dollimore, Tim Kindberg (2005). *Distributed Systems – Concepts and Design.* Fourth Edition. Addison Wesley

[Cwa06] Krzysztof Cwalina, Brad Adams (2006), *Framework Design Guidelines – Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley.

[Dem97] Demeyer, S., Rieger, M., Meijler, T. D., Gelsema, E. (1997), *Class Composition for Specifying Framework Design*. In Proceedings of ESEC/FSE 1997

[Dow] Jim Dowling, Vinny Cahill, *Dynamic Software Evolution and the K-Component Model*. Workshop on Software Evolution, OOPSLA, Tampa, Florida, USA

[Fow04] Martin Fowler (2004), *Analysis of Inversion of Control Containers and the Dependency Injection Pattern*.

[Fow99] Martin Fowler (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional; 1st edition

[Fur04] N.Furmento, J.Hau, W.Lee, S.Newhouse, J.Darlington (2004), *Implementations of a service-oriented architecture on top of Jini, JXTA, and OGSi*. In Proceedings of Second Across Grids Conference, 2004.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995), *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[Greg02] Kate Gregory (2002), *Visual C++.NET*. Markt+Technik Verlag.

[Gur01] J. van Gurp, J.Bosch (2001), *Design, implementation and evolution of object oriented frameworks: concepts and guidelines*. John Wiley & Sons, Inc.

[Hei05] Cornelia Heinisch, Frank Müller, Joachim Goll (2005). *Java als erste Programmiersprache – Vom Einsteiger zum Profi (Book on java programming)*. 4. Auflage. Teubner Verlag.

[Hun05] John Hunt, John McGregor (2005), *A Model for Software Libraries*. Workshop at Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 2005.

[Ihn06] Oliver Ihns (2006), *It is a POJO*. Conference Paper, Java Forum Stuttgart 2006.

[Kah98] Bernd Kahlbrandt (1998), *Software Engineering – Object-Oriented Software Development*. Springer Verlag.

[Kru05] Guido Krüger (2005). *Handbuch der Java-Programmierung (Java developer book)*. Addison-Wesley.

[May] Johannes Mayer, *Graphical User Interfaces Composed of Plug-Ins*. Department of Applied Information Processing and Department of Stochastics, University of Ulm.

[Mei01] Erik Meijer, Clemens Szyperski (2001), *What's In a Name? .NET as a Component Framework*. In Online proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components.

[Mic03] Microsoft (2003), *Developing XML Web Services and Server Components. Microsoft Press*

[Mot02] Belaramani Nalini Moti (2002), *Component-based Technology*. University of Hong Kong. MSc Thesis.

[Oas06] Oasis (2006), *Reference Model for Service Oriented Architecture 1.0*

[Par72] David L. Parnas (1972), *On the Criteria To Be Used in Decomposing Systems into Modules*. Carnegie-Mellon University, Pittsburgh, USA.

[Par] David L. Parnas, P.C.Clements, D.M.Weiss, *The Modular Structure of Complex Systems*. University of Victoria. Victoria, BC, Canada.

[Pla] Frantisek Plasil, Dusan Balek, Radovan Janecek, *DCUP: Dynamic Component Updating in Java/CORBA*. Technical Report No. 97/10, Dep. of SW Engineering, Charles University, Prague.

[POSA2] Douglas C. Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann (2002), *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley and Sons Publishing.

[Ric02] C.Rich, N.Lesh, J.Rickel, A.Garland (2002), *A Plug-In Architecture for Generating Collaborative Agent Responses*. In Proceedings of 1st International Conference on Autonomous Agents and Multiagent Systems, Bologna, Italy, 2002.

[Schn] Jean-Guy Schneider, Jun Han, *Components – the Past, the Present, and the Future*. Swinburne University of Technology, Victoria, Australia.

[Ste81] D.V. Steward (1981), *The design Structure System: A Method for Managing the Design of Complex Systems*, IEEE Transaction on Engineering Management, vol. EM-28, no. 3 pp. 71-74, August, 1981.

[Sul01] K. Sullivan, Y.Cai, B.Hallen, W.Griswold (2001), *The Structure and Value of Modularity in Software Design*. ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 2001, Vienna.

[Sun05] Sun Microsystems (2005), *The Java EE 5 Tutorial*

[Tal94] Taligent, IBM (1994), *Building Object-Oriented Frameworks*.

[Tor02] Richard Torkar (2002), *Dynamic Software Architecture*.
Högskolan Trollhättan/Uddevalla, Sweden

[Tu98] M.T. Tu, F.Griffel, M.Merz, W.Lamersdorf (1998), *A Plug-In Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents*. Proceedings 2. Intl. Workshop on Mobile Agents, MA'98, Stuttgart, Sept. 1998.

[Völ99] Markus Völter (1999), *PluggableComponent – A Pattern for Interactive System Configuration*. In Proceedings of the 4th European Conference on Pattern Languages of Programming.