IAM Informatics project CASYMS

IAM Institute:

Institut für Informatik und angewandte Mathematik (IAM)
Universität Bern
Neubrückstrasse 10
3012 Bern

Supervisor:

Prof. Dr. Oscar Nierstrasz
Email: oscar.nierstrasz@acm.org

Students:

Baltisar Oswald
Herzogstrasse 22
3014 Bern
Email: baltioswald@yahoo.com

Silvan Auer
Scheibenstrasse 28
3014 Bern
Email: silvan.auer@unic.ch

Space Research & Planetary Sciences:

Physikalisches Institut
Space Research & Planetary Sciences
Siedlerstrasse 5
3012 Bern

Lab address:

Clean Room C40
Tel.: 031 631 44 41

Responsible persons:

Prof. Dr. Kathrin Altwegg
Tel.: 031 631 44 20
Email: kathrin.altwegg@phim.unibe.ch

Adrian Etter
Tel.: 031 631 86 86
Email: adrian.etter@phim.unibe.ch

Table of contents

Goal of the CASYMS application

The CASYMS application is able to, move different hardware pieces of CASYMS, read data from hardware and protect CASYMS and the mass spectrometer from dangerous manipulations of the user. Limits (minimal and maximal values) for the different parts of CASYMS can be defined, to ensure that users deal no damage to CASYMS or the mass spectrometer. Further, the application allows to set software defined zero points of CASYMS's turntable axes. An interface for other software is available to communicate through the application with the CASYMS machine. CASYMS means calibration system for mass spectrometers, and enables to operate mass spectrometers under space-similar conditions.

Our task was to implement an application, which can handle two parts of the CASYMS machine, the turntable and the beam scanner. In addition the application should be expandable and maintainable. Particularly the expandability is very important, because it is planed by our customer to implement software for other parts of the CASYMS machine, based on our application.

Requirements

Installation

The following has to be installed:

- **Java 2 SDK 1_4_0_01 for Linux i586**
  http://java.sun.com

- **Apache Xerces Java 2.0.2 parser (*)**
  http://xml.apache.org/xerces2-j

- **Apache Xerces C++ 1.7 parser (**)**
  http://xml.apache.org/xerces-c

- **AccelePort Xr 920**
  http://www.digi.com

- **Comedi Linux Control and Measurement Device Interface**
  http://stm.lbl.gov/comedi

- **Linux-GPIB**
  http://linux-gpib.sourceforge.net

* Xerces Java 2.0.2 has one bug when including the dtd via transformer object and the setOutputProperty method: attribute lists with more than one attribute declaration have a closing tag too much. To fix this, we implemented a correctSubsetParseError method in the BasicXML class. This may lead to misbehaviors with higher versions of Xerces Java, assumed they fixed this bug.

** Xerces C++ changed a lot of things between version 1.X and 2.X. The 1.X version principally returns objects as copy's and not as reference (call by value instead of call by reference). The 2.X version finally does return references, which would have saved us much code, if it had been so earlier.

The following has to be started:

- **Comedi**
  Type in Console as root to load kernel module:
  modprobe ni_pcidio
  and to assign device module type:
  comedi_config /dev/comedi0 ni_pcidio

- **GPIB**
  see the Linux-GPIB documentation on http://linux-gpib.sourceforge.net

The following has to be done:

- **Source:**
  Make sure the source is available. For details about directory structure of the deployed code see Deploying.

- **Binary:**
  The required binaries are CasymsClient.jar, libcasyms.so and casyms. To build them, type in console in the appropriate directory (in our case /home/casyms/info_projekt/casyms/):
  ```
  ant
  ```

- **LD_LIBRARY_PATH:**
  libcomedi.so is a dynamic library, so you have to add the location of the library to the LD_LIBRARY_PATH. Under Linux you can do it by typing in console:
  ```
  export
  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/casyms/info_project/casyms/lib/
  ```

Running

The following has to be done at least:

- **Start server:**
  Type in console in the directory of the appropriate binary:
  ```
  casyms -l validate/test.config
  ```
  or for more information:
  ```
  casyms -h help
  ```

- **Start client:**
  Type in console in the directory of the appropriate binary:
  ```
  java -jar CasymsClient.jar
  ```
  or specify an ip address different than the default local host and port to connect to:
  ```
  java -jar CasymsClient.jar 192.168.1.2 3501
  ```

Deploying

- **CVS:**
  We decided to use CVS for deploying the whole application. To get the source, change to the desired directory (in our case /home/casyms/info_projekt) and download it from the cvs server by typing in console:
  ```
  cvs -d :pserver:balti@sites-on-linux.unic.ch login
  ```
  After initial login (in our case user balti) type the following in console to make a check out:
  ```
  cvs -d :pserver:balti@sites-on-linux.unic.ch checkout casyms
  ```
  After a first check out is done, you may also use ant to check out the code from the cvs server:
  ```
  ant cvs
  ```

- **Directory structure:**
  - casyms
  - doc
  - lib
  - src
    - c
    - cpp

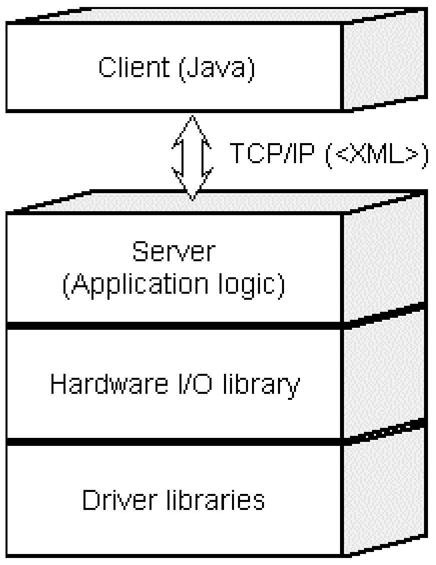
        - action
        - converter
        - hardwareIO
        - model
        - trigger
        - validate
        - xml
    - java
        - classes
        - jar
        - src
            - gui
            - img
            - xml

- **Ant:**
  We use ant in combination with make as build tool for compiling and preparing everything necessary for compiling as well as for other useful purposes. Possible targets to call are 'ant', 'ant java', 'ant make', 'ant cvs', 'ant clean' or 'ant clear'. See the build.xml file for details about the targets.

CASYMS application architecture

Overview

The application is build on a client, server architecture. This concept has been chosen, because an interface that is accessible from other applications is needed (requirements of the customer). Instead of implementing an interface for the graphical user interface GUI and an interface for external applications, we decided to implement only the required one, and to communicate from the GUI through this interface to the server as an external application. The advantages are clear, less programming, external applications and GUI have the same conditions, independence between user interface logic and business logic. The communication protocol is a self-made protocol, written in xml. The client is written in java and the server in c/c++. The client is in fact a graphical user interface, is intuitive in use. The server is a console-based program without graphical elements. The server himself is split up in two layers. A c and a c++ layer. Nearly all application logic is implemented in the c++ layer. The c layer handles the communication between hard- and software. It is responsible for communication with the serial port, the gpib card and the digital IO card.

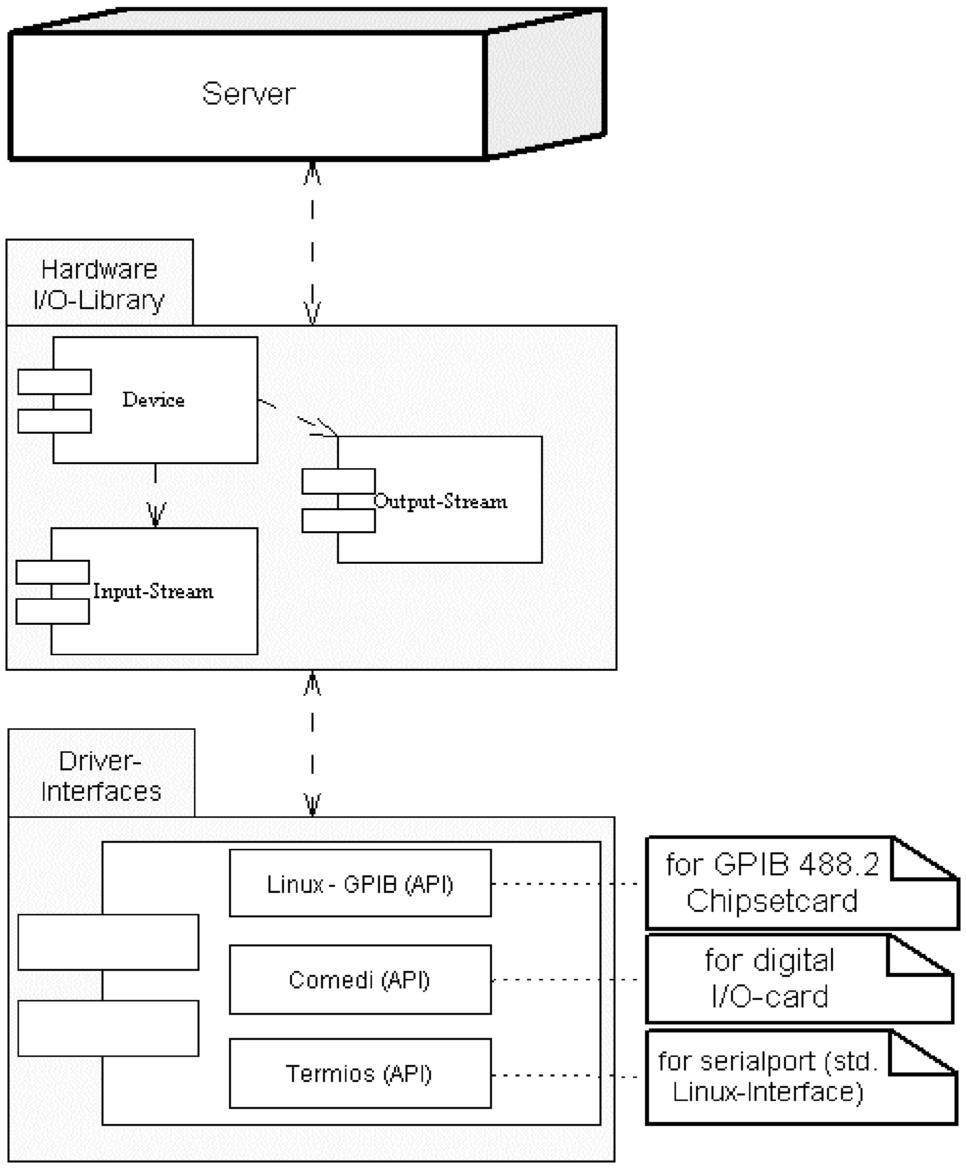


Implementation of the CASYMS server

C layer

Overview

The main scope of the c layer is to specify and define a standard way to communicate with the hardware. Each interface card has its own device, input stream and output stream, but due to an implemented kind of inheritance each of them has the same functions. The picture bellow shows the structure of the c layer. Note the different interfaces for communication with the hardware. There are API's for all of them. Linux-GPIB API for the gpib interface card, comedi API for the digital IO card and termios (standard Linux) for the serial port RJ-232. The c layer defines for each API a device, input and output stream. The gpib device, input and output stream for the Linux-GPIB API, the comedi device, input and output stream for the comedi-API and the serial device, input and output stream for termios, the serial port API.



Device, input and output stream

The following code segment shows, how the “kind of inheritance” is implemented.

```
typedef struct{
  void * constr;
  void (*open_Device)(void*, char*);
  void * (*get_InputStream)(void*, int);
  void * (*get_OutputStream)(void*, int);
  void (*kill_Device)(void*);
}Device;
```

The typedef struct is a kind of class declaration with a void pointer field constr representing the constructor and four fields representing method pointers. This code segment is stored in a file named device.h. The c layer has three devices, the gpib device, the comedi device and the serial device. All three devices are an implementation of the typedef struct Device above. The serial device for example is implemented as follows:

```
#include <device.h>

void open_SerialDevice(void *device, char * device_name)
{
  …….
}

void * get_Serial_InputStream(void * device, int size)
{
  …….
}

void * get_Serial_OutputStream(void * device, int size)
{
  …….
}

void kill_SerialDevice(void *device)
{
  …….
}

Device * new_SerialDevice()
{
  Device *dev = malloc(sizeof(Device));
  dev->open_Device = open_SerialDevice;
  dev->get_InputStream = get_Serial_InputStream;
  dev->get_OutputStream = get_Serial_OutputStream;
  dev->kill_Device = kill_SerialDevice;
  dev->constr = malloc(sizeof(struct SerialDevice));
  return dev;
}
```

First the four methods are implemented, naturally with the same return type and the same number and types of method parameter as the appropriate method pointer defined in typedef struct Device. Otherwise it wouldn't be possible to assign the method pointers to the method pointer fields in the typedef struct Device. The "connection" between device and serial device is done in the new_SerialDevice() method. First of all, memory is being allocated for typedef struct Device.

```
Device *dev = malloc(sizeof(Device));
```

After that all method pointer fields in the struct Device are assigned with the pointer of the implemented methods above.

```
dev->open_Device = open_SerialDevice;
dev->get_InputStream = get_Serial_InputStream;
dev->get_OutputStream = get_Serial_OutputStream;
dev->kill_Device = kill_SerialDevice;
dev->constr = malloc(sizeof(struct SerialDevice));
```

This for example allows the c++ layer to call the method open_SerialDevice like:

```
Device *dev = new_SerialDevice();
dev->open_Device(dev);
```

The line dev->open_Device(dev) could also open a comedi device (digital IO card). Which device is opened depends only on the real time type of the device. It is a kind of polymorphism.

The goal of this structure is to define a "standard" type Device, a kind of classification of the different devices so that the communication with the different cards or ports is standardized. On the same structure also the OutputStream and the InputStream structs are defined. Each output and input stream is an implementation of the struct OutputStream or InputStream. The device contains the methods to get an output or an input stream, open the device and kill (release the allocated memory for the specific device and close the connection) it. To get an input stream for communication with the serial port e.g., the following lines are necessary:

```
Device *dev = new_SerialDevice();
dev->open_Device(dev);
InputStream istream = dev->get_InputStream(dev);
```

The input stream is for reading data from the device (card or ports) and the output stream for writing to the device (card or port). The struct InputStream is defined as follows:

```
typedef struct{

  void *constr;
  void * (*read_Stream)(void*);
  int (*has_more)(void*);
  void (*kill_InputStream)(void*);

}InputStream;
```

The input stream has two methods read_Stream() and kill_Stream(). read_Stream() returns a pointer. The real time type of the pointer depends on the real time type of the input stream. To get an output stream for e.g. the digital IO card, it is necessary to write:

```
Device *dev = new_ComediDevice();
dev->open_Device(dev);
OutputStream ostream = dev->get_OutputStream(dev);
```

The output stream has four methods: write_Stream(OutputStream * stream, void * write) write_buffer_Stream(OutputStream * stream, void * write) append_buffer_Stream(OutputStream * stream, void * write) and kill_OutputStream(OutputStream * stream) which is shown below:

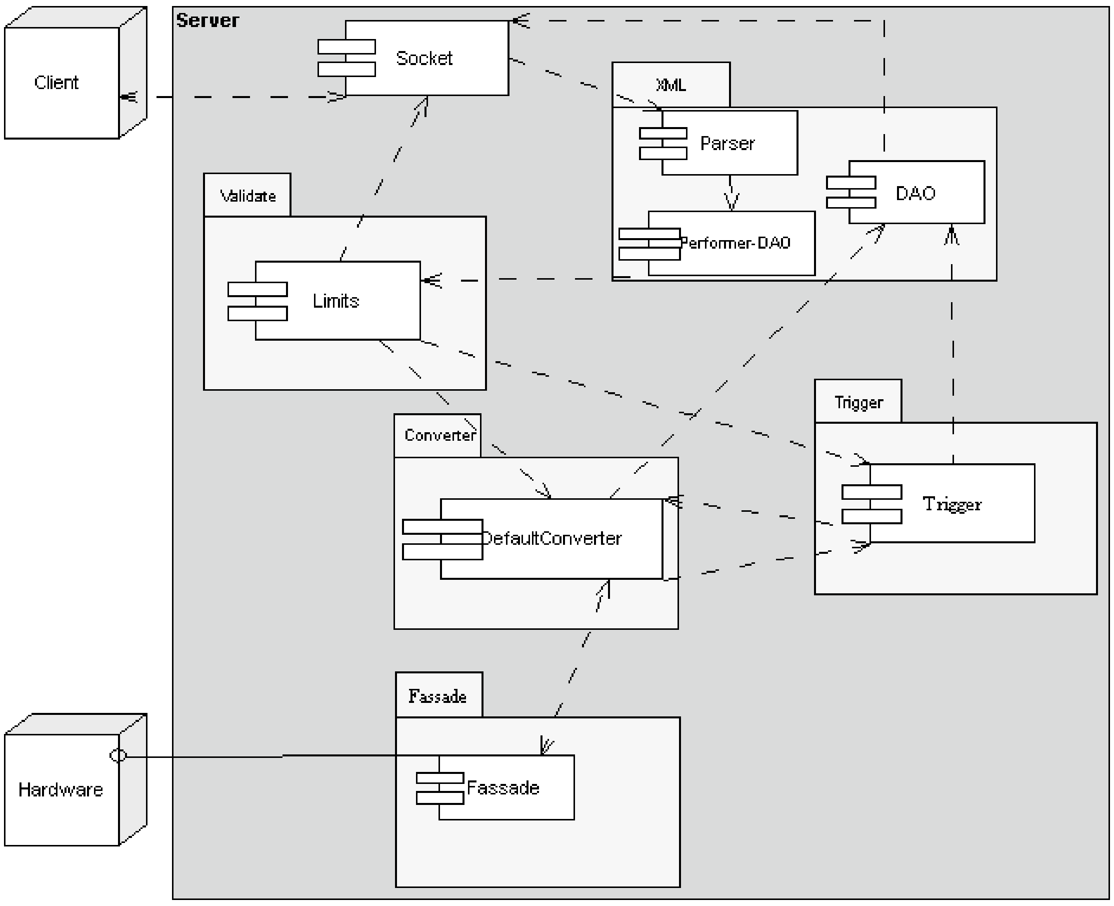
```
typedef struct{

  void * constr;
  void (*write_Stream)(void*, void*);
  void (*write_buffer_Stream)(void*, void*);
  void (*append_buffer_Stream)(void*, void*);
  void (*kill_OutputStream)(void*);

}OutputStream;
```

The method write_Stream writes the values of a pointer “array” to the device. What can really be written depends on the real time type of the output stream. In general there are two possibilities. Bits can be written on the digital I/O card, strings on the serial port and strings with a length of 2 characters on the gpib card. The method write_buffer_Stream does the same as write_Stream and append_buffer_Stream append a pointer to the output stream, without writing it to the card or port. Finally kill_OutputStream releases the allocated memory of the specific output stream.

C++ layer

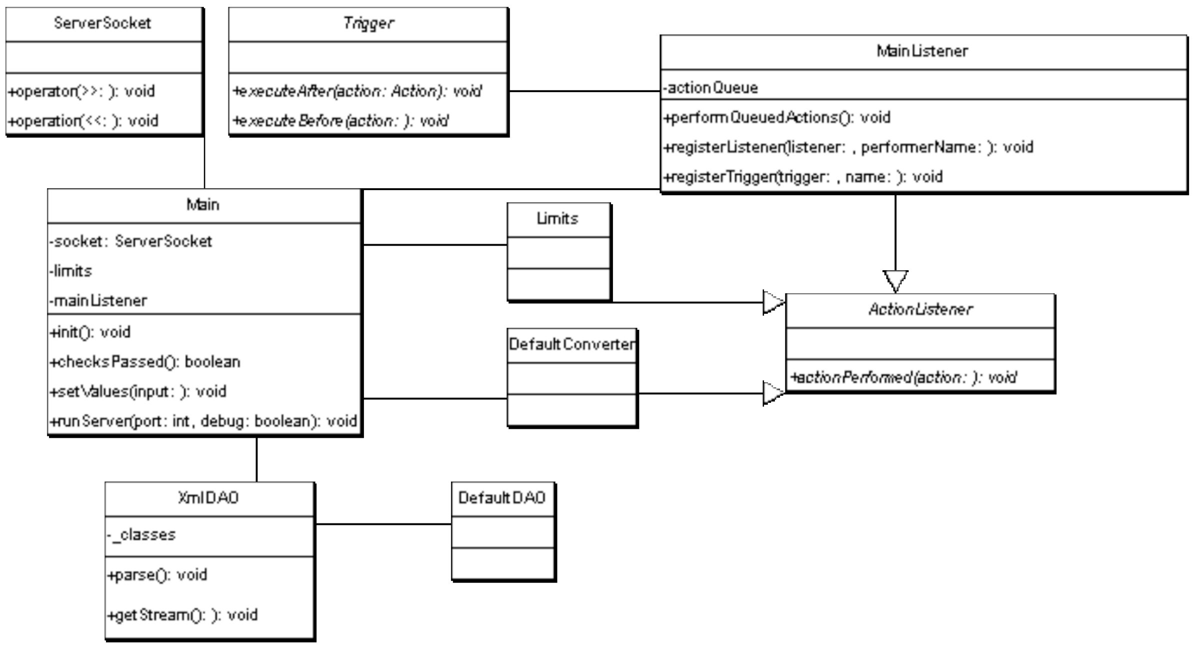
Overview

The c++ layer consist of five packages: XML, Validate, Converter, Trigger and Facade. The c++ layer is build on an, for a server unusual way, event listener architecture. Events or Actions are thrown if the request, the xml stream, contains at least one tag that is registered to throw an action. Actions are thrown by the xml package, more exactly by the DefaultDAO objects. They throw the action to the registered ActionListener objects. Mostly two objects are registered for one action. A Limits object from the package Validate and a DefaultConverter object from the package Converter. Also a Trigger class can be specified for one action, but it is not an ActionListener. The Triggers are also called by the DefaultDAO objects (action performer) but by other methods as the action listeners. The action listeners are called from the actionPerformed(Action& action) methods and the triggers by the executeBefore(Action& action) and executeAfter(Action& action) methods. The Limits object validates the request. This validation rather checks if the requested e.g. table move is allowed at this moment than if a specific field is an integer. Type validation is directly done in the DefaultDAO objects. Trigger-objects do some operations need to be done before a requested operation can be executed. For example if a table move is requested, the beam scanner have to be on a secure position. A Trigger object controls if the beam scanner is on the secure position, otherwise it moves the scanner to this position. The Converter object does the requested operations, e.g. a beam scan, table move, through the Facade and the c layer to the hardware.

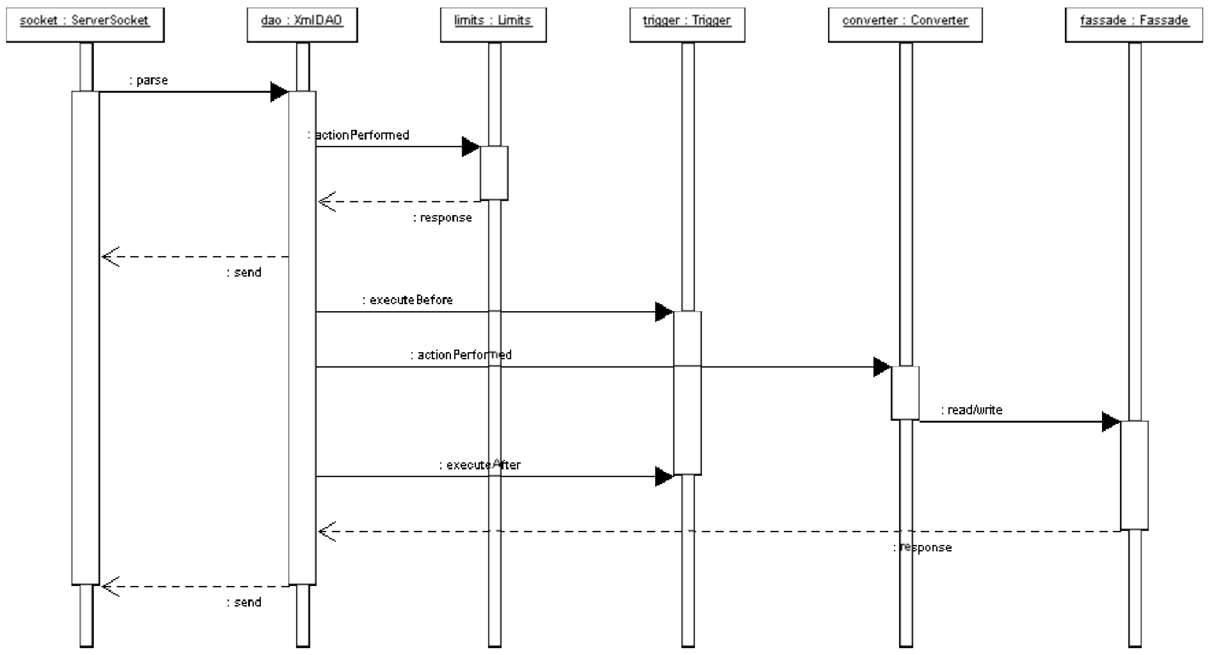


Data flow of the server

The data flow of the application is implemented in two classes, the Main class and the Main Listener class. The Main class is also the container for particularly all objects, as it is shown in the class diagram below. The responsibility of this class is to control and define the data flow. The responsibility of the Main Listener is to queue actions need to be queued and not performed directly.


ServerSocket
+operator(>>:): void
+operation(<<:): void
Trigger
+executeAfter(action: Action): void
+executeBefore(action:): void
MainListener
-actionQueue
+performQueuedActions(): void
+registerListener(listener: , performerName:): void
+registerTrigger(trigger: , name:): void
Main
-socket: ServerSocket
-limits
-mainListener
+init(): void
+checksPassed(): boolean
+setValues(input:): void
+runServer(port: int, debug: boolean): void
Limits
DefaultConverter
ActionListener
+actionPerformed(action:): void
XmlDAO
-_classes
+parse(): void
+getStream():): void
DefaultDAO

In general if a request is incoming, the stream is got from the Main object. It gives the xml stream to the XmlDAO and the dao parses it and performs the actions mentioned before. But the actions have to be in a kind “serialised”, because limits have to be checked before any manipulations on the CASYMS machine can be done. In fact the action listeners which are registered as listener at the performer, get the action directly and listener which are registered at the MainListener object, get the action after the main listener is called to perform the queued actions. The MainListener is not only a listener, but also a performer for queued actions. So other listeners can be registered at the main listener instead of be registered at the performer. The main listener does nothing else than take an action and queue it, until the command comes to forward the actions to the appropriate listeners. The main listener class has also the possibility to register Trigger objects. If the XmlDAO object gets a request, it parses the xml request string in a DOM and calls all action performer who’s names match with a tag name in the xml. The Performer objects (DefaultDAO’s) throw the actions to the registered listeners. First to the Limits objects and then to the MainListener object. When the Limits objects are finished the Main object asks the Limits objects if the input violated one or more Limits. If there is a violation, the Main object gets the Messages and Errors from the Limits objects and put them to the XmlDAO. The XmlDAO converts the DOM in a string and the Main object sends the string back to the client. If there is no violation, the Main object calls the method performQueuedActions() from the MainListener object. In other words the Main object call the Trigger and the Converter objects. First the trigger method executeBefore() then the actionPerformed() method form the Converter and at last the executeAfter() method from the trigger class. Finally the Main object calls the XmlDAO for transforming the DOM in a string and sends the string through the socket to the client.



Responsibility of the xml package

The xml package consists of one class XmlDAO, helper classes and an abstract class DefaultDAO with their implementations. The XmlDAO has the task to parse xml stream into an object representation, called DOM. The idea is that the XmlDAO does nothing else than parse and calls the DefaultDAO objects. The DefaultDAO objects are responsible for creating model objects. They are stored in a kind of hashmap. A map (ClassContainer) containing a key (type string) and a value (type DefaultDAO object). Containers for all need dao's. After the XmlDAO parsed the xml, the dom is iterated through the nodes a checked if a name of a dom element corresponds with a key of the container. If there are corresponding keys, the XmlDAO calls the method **void * transformToModel(DOM_Element& root)** as follows:

```
DefaultDAO myClass = (DefaultDAO*)_container->forName(name));
void * model = myClass->transformToModel((DOM_Element&)current);
```

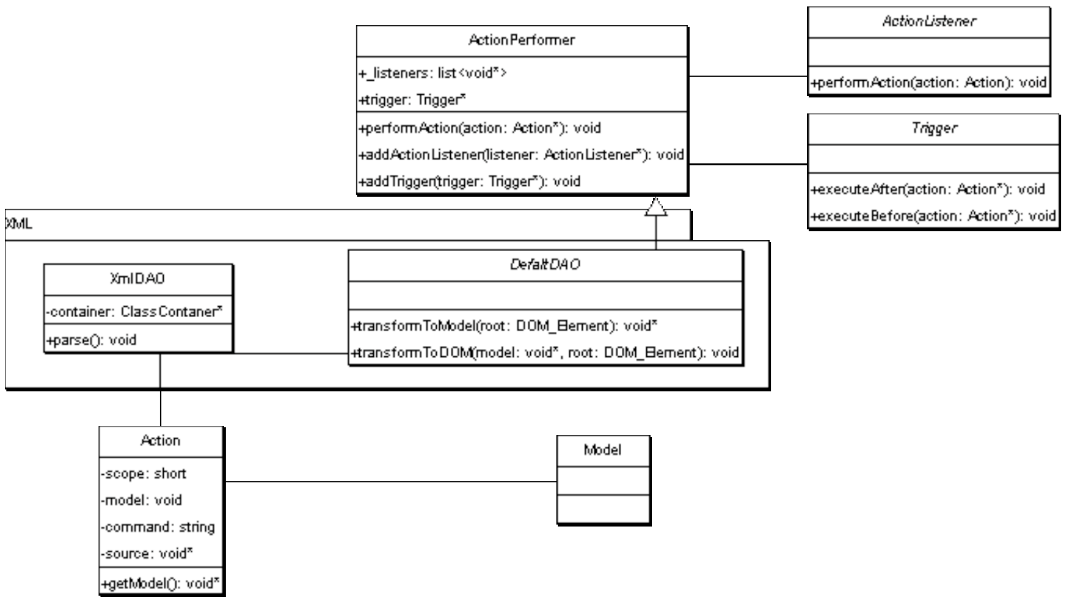
This method is declared abstract in the DefaultDAO class, so that all implementing classes of DefaultDAO have to implement it. The implementation of that method should be able to transform a parsed xml sub tree and a dom sub tree, into a model object. Model objects are simple "data containing objects" without any functionality. After that the XmlDAO object calls the **performAction(Action *action)** method as follows;

```
Action * action = new Action(myClass, QUEUED_ACTION, model);
myClass->performAction(action);
```

Implemented in the super class of the DefaultDAO class, the ActionPerformer class. This method iterates through the actionListener list and calls the **actionPerformed(Action& action)** method of the listeners as follows;

```
void performAction(Action* action){
  ActionListener* _listener;
  list<ActionListener*>::iterator iter;
  for(iter = _listeners.begin();iter!=_listeners.end();iter++) {
    _listener = *iter;
    _listener->actionPerformed(*action);
  }
  performedActions.push_back(action);
};
```

The method **actionPerformed(Action& action)** is defined abstract in the ActionListener class. After the listeners ended, the actions are coming back to the XmlDAO. The XmlDAO now calls the method **void transformToDOM(void * model, DOM_Node& root)** in the DefaultDAO class. The implementation of this method should be able to transform a Model into a dom sub tree. Bellow a class diagram with the mentioned methods and classes:


ActionPerformer
+_listeners: list<void*>
+trigger: Trigger*
+performAction(action: Action*): void
+addActionListener(listener: ActionListener*): void
+addTrigger(trigger: Trigger*): void
ActionListener
+performAction(action: Action): void
Trigger
+executeAfter(action: Action*): void
+executeBefore(action: Action*): void
XML
XmlDAO
-container: ClassContaner*
+parse(): void
DefaltDAO
+transformToModel(root: DOM_Element): void*
+transformToDOM(model: void*, root: DOM_Element): void
Action
-scope: short
-model: void
-command: string
-source: void*
+getModel(): void*
Model

An example implementation of the DefaultDAO class which transforms a power element

```
<!ELEMENT power (#PCDATA) >
<power>10000</power>
```

is shown bellow.

```
class PowerDAO : public DefaultDAO{

public:

  PowerDAO(){};
  PowerDAO(string name) : DefaultDAO(name){
  };

  void * transformToModel(DOM_Element& root) {

    int value;
    Power* power = new Power();
    DOM_Node current = root.getFirstChild();

    if(current != NULL){
      char * v = root.getFirstChild().getNodeValue().transcode();
      value = atoi(v);
      delete[] v;
    }else{
      value = 50000;
      power->addMessage(new Message("Start tag without value. Set it to 50000", MSG_WARNING));
    }
    power->setPower(value);
    return power;
  };
  void transformToDom(void * model, DOM_Node& root) {

    Power *power = (Power*)model;
    DOM_Element powerElem = doc.createElement(POWER_XML);
    this->appendFloatTextChild(powerElem, power->getPower());
    bool inserted = false;

    if(!root.hasChildNodes()){
      root.appendChild(powerElem);
    }else{
      DOM_NodeList children = root.getChildNodes();
      for(unsigned int i=0;i<children.getLength();i++){
        DOM_Node current = children.item(i);
        if(current.getNodeType() == DOM_Node::ELEMENT_NODE){
          char * tname = ((DOM_Element&)current).getTagName().transcode();
          string tagname = tname;
          delete[] tname;
          if(tagname == MESSAGE_XML){
            root.insertBefore(powerElem, children.item(i));
            inserted = true;
          }
        }
      }
      if(!inserted){
        root.appendChild(powerElem);
      }
    }
  };

};
```

Responsibility of the converter package

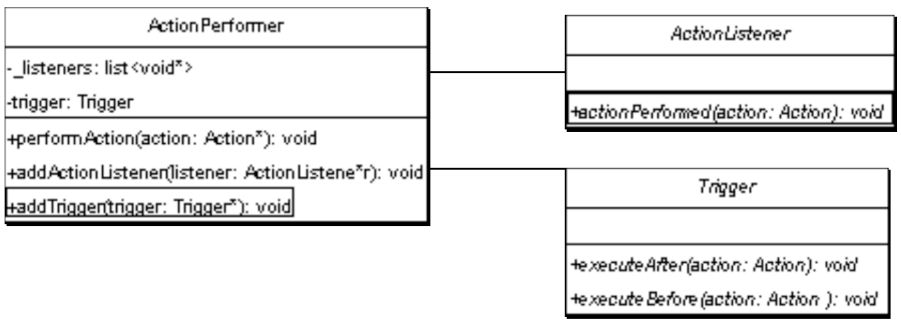
The main responsibility of the Converter objects is, to read or write data from and to the hardware and to convert the different data types coming from the hardware to mostly numeric types. As we have seen before, Converter(s) are ActionListener(s). In fact, Converter objects do not access the I/O cards and ports directly but through the Facade and the c layer to the I/O cards and ports. Converter objects are also to convert mostly bit arrays or strings in numeric types. For example, reading the position of the table, the Facade returns a bit array of 14 bits. This array has to be interpreted as a number, the position of the table. The transformation is done with linear interpolation. But as the requirements told us that the linear interpolation has to be changeable, the parameters of the linear function are stored in a file. Although is to mention that there exists a super class of all Converter(s), called DefaultConverter. This class extend the ActionListener class, but does not implement the abstract method **actionPerformed(Action *action)**. The DefaultConverter class implements the linear approximation of the convert function, so that this logic does not have to be implemented from the subclasses of the DefaultConverter class.

Responsibility of the limits package

The responsibility of the Limits objects is to check whether a requested operation is allowed at a specific state of the CASYMS machine. Limit objects are implementation of ActionListener. Never less there exists a super class for all Limits objects, called Limits. This class extends from the ActionListener class, but does not implement the ActionListener abstract method **actionPerformed(Action& action)**. This method has to be implemented by the subclasses of the DefaultLimits class. Reading the limits file is implemented in the abstract Limits class, so that subclasses of the Limits class, must not implement it again.

Responsibility of the trigger package

Sometimes it is useful or needed to do some operations or manipulations on the CASYMS machine, before and after a converter class is called (before and after a requested operation is done). For example before the turntable can be moved, the beam scanner have to be set to the position y=-450 and z=0. Otherwise it is possible that the turntable touches the scanner and damages it. As it is shown in the class diagram bellow, not only ActionListener objects but also Trigger objects can be registered to ActionPerformer objects.


ActionPerformer
-_listeners: list<void*>
-trigger: Trigger
+performAction(action: Action*): void
+addActionListener(listener: ActionListene*r): void
+addTrigger(trigger: Trigger*): void
ActionListener
+actionPerformed(action: Action): void
Trigger
+executeAfter(action: Action): void
+executeBefore(action: Action): void

If an object is registered as Trigger, this object has to extend from the Trigger class. This class contains two abstract methods **void executeBefore(Action& action)** and **void executeAfter(Action& action)**. These two methods are called as follows:

```
void performAction(Action* action){
  ActionListener* _listener;
  list<ActionListener*>::iterator iter;
  for(iter = _listeners.begin();iter!=_listeners.end();iter++) {
    _listener = *iter;
    if(_trigger != NULL)
      _trigger->executeBefore(*action);
    _listener->actionPerformed(*action);
    if(_trigger != NULL)
      _trigger->executeAfter(*action);
  }
  performedActions.push_back(action);
};
```

The code segment above shows that the Trigger class method **executeBefore(Action& action)** is called before all registered action listeners are called and **executeAfter(Action& action)** after the call of all registered listeners. Unfortunately in some cases it is necessary to call the Trigger methods only for a specific listener. It wouldn't be difficult to implement this feature but for other reasons we decided to implement another class where a trigger object can be registered for each action listener, called MainListener. The main responsibility of the MainListener class is to queue the actions. As we have seen before all ActionListeners receive the actions. But for our purpose we need a kind of serialised action. All Limits have to be called before any Converter is called. To achieve that, the MainListener class has been defined. The MainListener object does nothing else than put all incoming actions on a queue and performs them after the non-queued actions are performed. Gone back to the Trigger objects registered at the MainListener object, Trigger objects can be registered for a specific action listener. For example the request contains a table move. There are two action listener registered for this action. The TableLimits object registered at the ActionPerformer, in this case the TableDAO object and the TableConverter object registered at the main listener. First of all the Action is directly thrown to the Limits object without being queued, but no trigger has to be executed. So the Trigger object cannot be registered at the ActionPerformer object. The Trigger object instead is registered at the MainListener object. After the TableLimits object validates the requested move, the MainListener object perform the queued action and before performing the action at the TableConverter object executes the executeBefore method and after the executeAfter method of the TableTrigger.

Responsibility of the facade package

The Facade package is a Facade for the c layer. The Facade is not a big thing, because the "standardisation" to access the hardware is implemented in the c layer. Nevertheless does it make sense to have a Facade. Facade contains also the simulation of the hardware access. We couldn't be every time in the clean room, where the computer and the CASYMS machine are. For working at home, we implemented a very basic simulation. The simulation is implemented in the Facade package.

Device, Input and Output stream

Serial port Device, Input and Output stream

Communication with the serial port is character or byte based. Means that it is usual to send and receive strings. And that's how the application communicates with the turntable controller. Accordingly the device, input and output stream are not very complicated. The only thing the device has to do, is to open the port for communication. Exactly this is done by the set-up of the serial device (baud rate and so on) implemented in the serial device and is called by the following:

```
Device* dev = new_SerialDevice();
dev->open_Device(dev);
```

After that it is possible to send commands to the turntable controller with the following lines:

```
OutputStream* ostream = dev->get_OutputStream(dev);
ostream->write_Stream(ostream, “G2\n”);
```

For the commands documentation of the turntable controller please refer to the CASYMS documentation.
The following lines do Reading from the port:

```
InputStream* istream = dev->get_InputStream(dev);
char* input = istream->read_Stream(istream);
```

To mention is that the serial InputStream reads from the serial port until a carriage return incomes. It is configured as a so-called canonical input. But there is no problem with it, because the turntable controller who is the only part of the CASYMS machine, who communicates through the serial port, is configured equal.

Comedi Device, Input and Output stream

Communication with the digital I/O card is bit based. A digital I/O card has normally several channels and a sub device groups eight channels. The digital I/O card installed in the CASYMS computer has four sub devices. Reading and writing to the digital I/O card is much more complex than the serial port. First we have to define on which sub devices and channels what has to be written and read. There are two files containing all digital I/O card configurations. named comedi_config.c and comedi_config.h. To understand these files, another source file is needed named axis.h. The file axis.h contains only constants. Each axis has a number, which is used in the whole application as an identifier for this axis. Also the configuration of the axis is stored with the identifier as key. Most axes have two configurations, one for writing and another one for reading.

Normally the writing configuration only has to tell the multiplexer which axis value should be read. For example, before reading the position of the x axis, the first, second and third channel on sub device A have to be set 0 0 1. After that the position can be read on the channels 16-24 on sub device B and 1-8 on sub device C. comedi_config.c stores the write and read configuration of one axis in a LINKEDLIST containing structs of the type ComediConfigItem.

```
typedef struct {

int subdevice;
int channel;
int bit;

}ComediConfigItem;
```

and all LINKEDLISTs in an array. For example the write configuration of the turntable x axis is stored at the array position 0:

```
#define X 0
```

as you see, the array position correspond with the #define X pre-processor directive defined in the file axis.h. All axes are stored at the same position as it is defined in axis.h.
Example:

The x turntable axis has a write and a read configuration LINKEDLIST.
The write configuration is filled up as follows:

ComediConfigItem	Sub device	Channel	Bit
0	0	0	0
1	0	1	0
2	0	2	1

And the read configuration LINKEDLIST as follows:

ComediConfigItem	Sub device	Channel	Bit
0	1	16	-1
1	1	17	-1
2	1	18	-1
3	1	19	-1
4	1	20	-1
5	1	21	-1
6	1	22	-1
7	1	23	-1
8	2	0	-1
9	2	1	-1
10	2	2	-1
11	2	3	-1
12	2	4	-1
13	2	5	-1
14	2	6	-1
15	2	7	-1

There exists such a configuration for each axis, which can be read and written to. To write something to the digital I/O card the following lines are necessary:

```
Device* dev = new_ComediDevice();
dev->open_Device(dev);
OutputStream* ostream->get_OutputStream(dev, X);
ostream->write_Stream(ostream, NULL);
```

The second parameter of the method config->get_OutputStream(config, X) returns the write configuration for the x-axis of the turntable. It is one of the #define X 0 pre-processor directives defined in the axis.h file mentioned before. Finally the ostream->write_Stream(ostream, NULL) method iterates through the linked list containing ComediConfigItems and writes the bits on the sub devices and channels stored in the ComediConfigItems. The same could also be done as follows:

```
Config* conf = new_ComediConfig();
OutputStream* ostream->get_OutputStream(dev, 0);
ostream->write_Stream(ostream, conf->get_AxisWriteConfig(conf, X));
```

The following lines read the position of the x-axis:

```
unsigned int* in;
istream = (dev->get_InputStream)(dev, Z);
in = (unsigned int *)(istream->read_Stream)(input);
```

Converter

As mentioned in the CASYMS application architecture chapter, it is the responsibility of the Converter objects to read and write data to the hardware and to convert it from a bit array or string to a decimal number. To do the conversion between the different types we have chosen a linear approximation. This approximation is implemented in the abstract DefaultConverter class. Access to the hardware should be implemented by the subclasses of the DefaultConverter. A subclass has at least to implement the **actionPerformed(Action& action)** method. The parameters for the linear approximation are stored in a file for all functions, it doesn't matter if it is a turntable or a beam scanner function. Functions are mostly used for computing the decimal value of a binary value. Unfortunately the binary values receiving from the digital I/O card don’t match the real decimal value of e.g. the turntable alpha axis.
That is because of a function has been defined. First an integer value is built from the binary value and is set equals to a variable x (e.g. binary value = 101 = 1*2_+0*2_+1*2° = 5 = x = integer value). After that, the function is used to compute the decimal value (x = 5; f(x) = -.5*x + 1.5 = -1 = decimal value = real alpha axis position). Now function parameters in this example are a = -0.5 and b = (+)1.5.
Let us resume this example:

Axis	**Alpha**
binary value	101
integer value	5
function parameter a	-0.5
function parameter b	1.5
general function form	a*x + b
function for alpha	-0.5*x + 1.5
decimal value	-0.5*5 + 1.5 = -1

Back to the function parameter file:

```
#function y=ax + b
#name a b
#----------------------------------------

x -0.03026738 990.84
y -0.03026738 990.84
z -0.0091463415 290
alpha 0.0054945055 -180.68
beta -0.0013157895 43.69
theta 0.0054945055 -180.68
bs_y -0.12204052 1001
bs_z -0.12204052 1000
bm_cem -0.12204052 1000
bm_factor -0.001 8.194
bs_cem -0.12204052 1000
bs_factor -0.001 8.194
fc_10 -0.0012204 10
fc_12 -0.0012204 10
fc_factor -0.001 8.194
```

The function parameter string is a line witch composed of axis, the parameter a and the parameter b, separated by one or more empty spaces.

A default function parameter file is defined but it is possibly to specify a new one and to load it on server start up with the option -f filename. The comment out character is #
Allowed axes are:

x	*turntable x axis*
y	*turntable y-axis*
z	*turntable z-axis*
alpha	*turntable alpha axis*
beta	*turntable beta axis*
theta	*turntable theta axis*
bs_y	*beamscanner y-axis*
bs_z	*beamscanner z-axis*
bm_cem	*beammonitor cem*
bm_factor	*beammonitor factor*
bs_cem	*beamscanner cem*
bs_factor	*beamscanner factor*
fc_10	*faraday cup factor 10 function*
fc_12	*faraday cup factor 12 function*
fc_factor	*faraday cup factor*

All axes and their functions have to be specified only once in the function parameter file. Otherwise the application will not start. The main reason, why the function parameter are stored in a file and not hard coded in the source is that specially the 0 position of the turntable axis change from time to time! There should be a possibility to adjust the application without recompiling, if for example the alpha turntable axis 0 position changed.
When we move the alpha axis (real) to 0, we read the position of alpha from the application = 2. To adjust this we have to change the **b** parameter value in the function parameter file in the matter of: **value = old value - 2**.

Implemented is the conversion in five methods. If the constructor is called, the functions file is opened read and stored in a hashmap.

```
ifile.open(_fileName.c_str());
if(!ifile){
  cout << "Can't open file: " + _fileName << endl;
}
while( ifile ){
  ifile >> *this;
}
ifile.close();
```

The overwritten operator istream& operator >> (istream& in, DefaultConverter& converter) reads line per line from the function file and calls the setFunktion(axis, new FunctionParameter(a, b)) method. This method stores one line of the function file in a hashmap. At last the

```
checkAllFunctions()
```

method is called. This method checks if every axis is defined and if they are defined only once in the function file.

Limits

The responsibility of a limit class is to check if a limit is violated. Limits are maximum and minimum positions or values of the different axis of the turntable, beam scanner and ionizing power. The limits depend on the mass spectrometer. So every mass spectrometer has its own limits. That’s why limits are stored in a file, which is loaded on start up of the server.

```
#axis min max
#-------------------
z 20 600
alpha -500 500
```

```
beta -50 50
#theta -301 30
#x -40 350
beam_y -455 65
beam_z -65 65
power 0 100000
```

A custom limits file can be loaded while starting the server with the option -l filename and must have the same format as shown below. A limit is a line witch consists of the axis, the minimal value and the maximal value, separated by one or more empty spaces.
The # character comments out the whole line and must be at the beginning of the line. The minimal value must be less or equals than the maximal value otherwise the application will not start.
Allowed axes are:

z	*turntable z-axis*
alpha	*turntable alpha axis*
beta	*turntable beta axis*
theta	*turntable theta axis*
x	*turntable x axis*
y	*turntable y-axis*
beam_y	*beamscanner y-axis*
beam_z	*beamscanner z-axis*
power	*power*

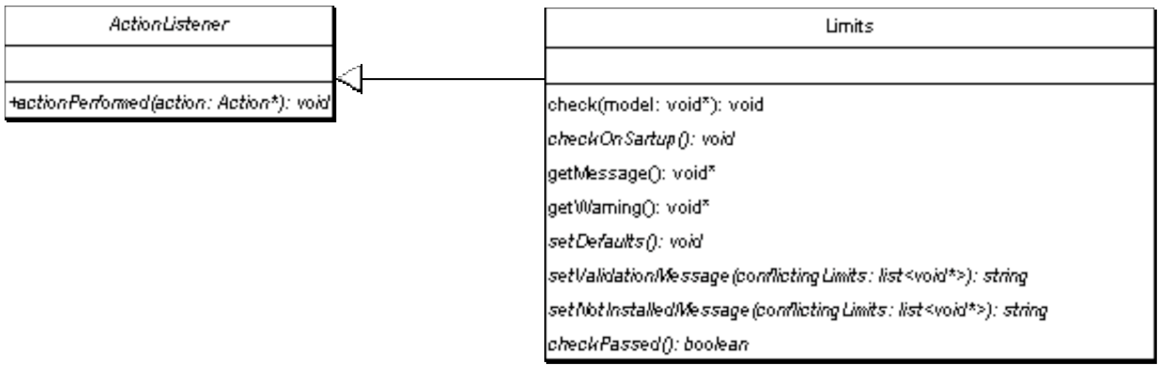
All specified axis are suspected being installed and all not specified axis are suspected not being installed. This means that every installed axis should be specified in the limits file. Otherwise the application does not allow setting or moving the not specified axis.
If a not installed axis is specified in the limits file, the behavior of the application is not defined.
Reading and checking the limits file is implemented in the abstract limits file. Calling the constructor the limits file is opened and the limits are stored in a hashmap.

```
ifile.open(fileName.c_str());
if(!ifile) {
  cout << "Can't open file: " + fileName << endl;
  exit(1);
}
while(ifile){
  ifile >> *this;
}
ifile.close();
```

The overwritten operator istream& operator >> (istream& in, Limits& lim) reads line per line from the limit file and calls the addLimit(string name, float min, float max) method. Additionally the operator checks whether the minimal value is less or equal the maximal value. If not the application will stop with the message “min value of " << *name* << "is bigger than max value” typed in console. The method addLimit stores one line of the limit file, the limit for each installed axis, in a hashmap and check if the axis is allowed, means if the axis is one of the axis CASYMS have.

If a software limit is necessary for an operation, which can be requested, it is done in a subclass of the abstract Limits class. As a matter of course it is possible to check if the request doesn't violate a limit in the converter class, but there is one big problem. The limits should be checked, before all operations are computed. Assuming we have two tags in the request. The turntable should be moved and the power should be reset. If the request violates the power limit, the turntable should not move. But if we check the limits in the converter classes, the turntable will move and after that we try to set the power and would see that a limit is violated. So we have moved the turntable, even if the power limit is violated. But as defined in the requirements, this should not happen. The limit classes are called before the first converter class is called.
For example: The client sends a request with a power and a turntable tag. The PowerDAO transforms the tag into a power model object and throws an action.
The power limits class receives the action and checks if a limit is violated. Now the TableDAO transforms the turntable tag into a table model object and throws an action. The table limits class receives the action. This class checks the limits and returns. So the converters could be called, but again we check the power limits. Then we call the power converter. We check the table limits and call the table converter.

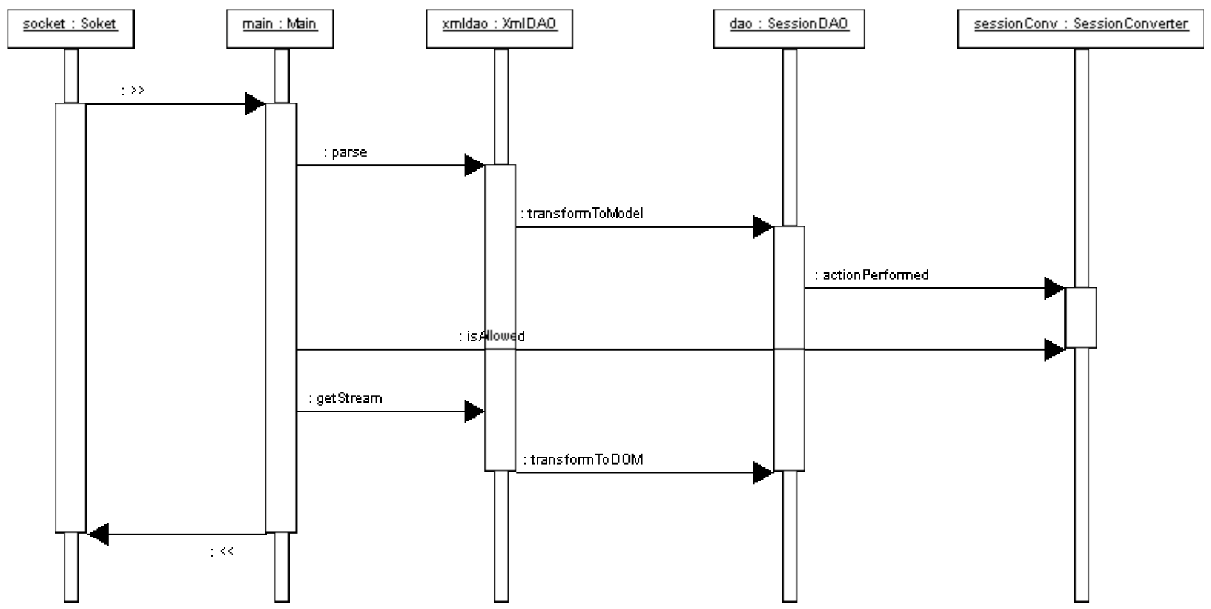
There are six abstract methods in the abstract Limits class: **void checkLimits(void* model), void * checkOnStartup(), void setDefaults(), string setValidationMessage(list <void*> conflictingLimits), string setNotInstalledMessage(list<void*> conflictingLimits) and bool checkPassed().** They have to be implemented in the subclass. Limits classes are ActionListener. Also the void **actionPerformed(Action& action)** method has to be implemented.


ActionListener
+actionPerformed(action: Action*): void
Limits
check(model: void*): void
checkOnSartup(): void
getMessage(): void*
getWarning(): void*
setDefaults(): void
setValidationMessage (conflicting Limits : list<void*>): string
setNotInstalledMessage (conflicting Limits : list<void*>): string
checkPassed(): boolean

Limits objects are not only used as ActionListener, but also called on server start up, to check whether a position of an axis is moved outside the limit manually. If something has to be checked on application start up, it should be implemented in the void * checkOnStartup() method. The void setDefaults() method sets all default values. Have in mind that a limit object is only initialized at application start up. If some fields should be reset between two requests, it can be done in the setDefaults() method. The actionPerformed() method is the start point. This method has at least to call the check() method, which is a method of the super class. The check() method calls the void checkLimits(void* model) method. This method checks the limits. If one or more limits are violated, this method should throw an exception. If any exceptions are thrown, the check() method catch them and call, depending on the exception type, the string setValidationMessage(list <void*> conflictingLimits) or the string setNotInstalledMessage(list<void*> conflictingLimits) method, implemented in the subclass. These methods should return a string declaring which limit is violated and so on. This string is send to the client. Finally the bool checkPassed() method returns true if the checks passed, otherwise false.

Session

For security of the CASYMS machine, the application allows only one client at one time logged in. It isn't a login with username and password but only a request for a session. If a session is active no one else can login. So the session module has to activate, deactivate sessions and control if the request contains the appropriate session. The sequence diagram bellow shows the cycle from the request to the response.



First the main controller got the xml stream. Then it calls the method parse() from the XmlDAO object. It parses the xml and calls the SessionDAO object to transform the DOM sub tree in a model object, if there is a session tag in the request. Referred to the dtd, a session tag is required in every request. Otherwise the XmlDAO will break up and return a parse error. The session element in the xml stream is defined as follows:

```
<!ELEMENT session (#PCDATA) >
<!ATTLIST session scope (login | key | logout) #REQUIRED >
```

A login the element looks like

```
<session scope=”login” />
```

If the login is already done, the session tag must look like:

```
<session scope=”key”>akkdaiei22451</session>
```

The body string is the session key received from the server on login. The SessionDAO transform the DOM sub tree in a SessionModel object, adds the object to an action and throws the action to the registered ActionListener, the SessionConverter. The SessionConverter controls the following:

State	Required
Request for login	No active sessions
Request for logout	Active session key correspond transmitted session key
Request	Active session key correspond transmitted session key

Depending if the incoming session is ok or not, the SessionConverter object sets a flag = true or false. When the SessionConverter finished, the main controller calls the method isAllowed() from the SessionConverter. If the method returns true the operation will proceed, otherwise the operation will stop with an appropriate message. Which message is shown in the section Error Handling.

Turntable

The sequence diagram below shows which objects are used to move or to read the position of the turntable. First the main controller receives the xml stream and hands it over to the XmlDAO. The XmlDAO parses the xml stream and if a turntable tag is in the request, it calls the transformToModel() method. The turntable tag is defined in the dtd:

```
<!ELEMENT turntable (alpha?, beta?, z?, theta?, x?, y?, speed?) >
<!ELEMENT alpha (#PCDATA) >
<!ELEMENT beta (#PCDATA) >
<!ELEMENT z (#PCDATA) >
<!ELEMENT theta (#PCDATA) >
<!ELEMENT x (#PCDATA) >
<!ELEMENT y (#PCDATA) >
<!ELEMENT speed (#PCDATA) >
```

It is a body tag with six possible inner tags. The first five tags represent five axes and the last tag “speed” the speed to set the positions of the table. The effect of speed is the precision of setting the new positions (the more slowly the more exactly). If one inner tag of the turntable has value 0, this axis will not be moved but the actual position is read from the table and returned. For example if the following tag is send:

```
<turntable>
  <alpha>5</alpha>
  <beta>-4</beta>
  <x>0</x>
</turntable>
```

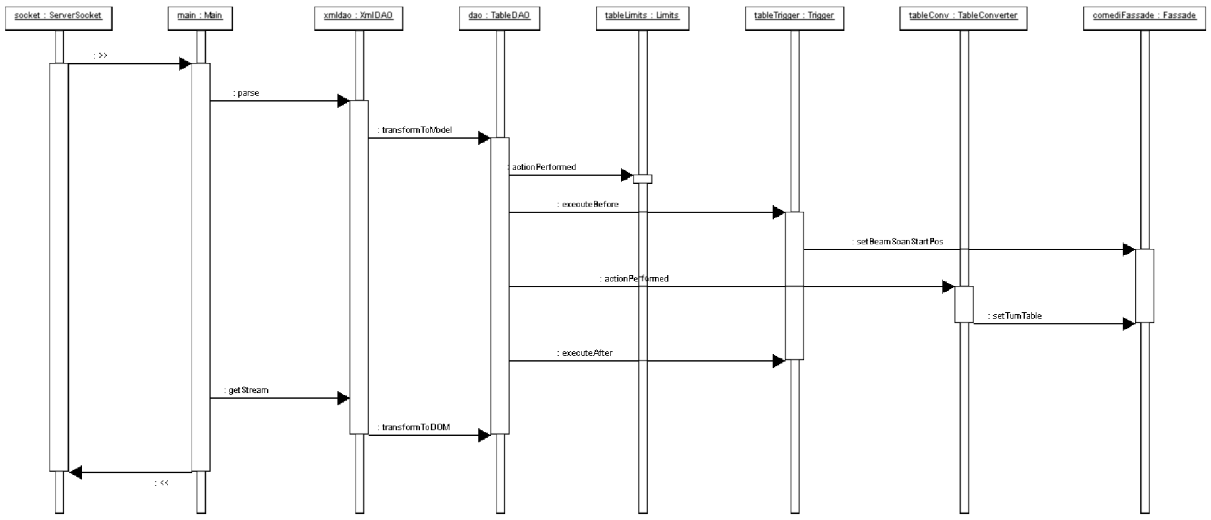
and the actual position of the installed axis is:

x	100
z	220
alpha	2
beta	15

The alpha axis will be set to 7 and the beta to 11. With the z-axis nothing have to be done and the position of the x-axis is read. So that the response contains the following:

```
<turntable>
  <alpha>7</alpha>
  <beta>11</beta>
  <x>100</x>
</turntable>
```

The TableDAO object transforms the DOM sub tree to a TurnTable object, sets it to a new action instance and throws the action to the registered listeners. Registered listeners for the TableDAO are: a TableLimits object and a TableConverter object. Additionally a trigger object is registered, the TableTrigger object. First the TableLimits object gets the action, validates the table model, to make sure the requested positions of the turntable don’t violate one or more limits. If everything is ok, the application proceeds. Otherwise the request breaks up and returns an error as response. Further details on messages please refer to the Error section. Assuming no limit is violated, the trigger is activated and the executeBefore() method is called from the TableLimits object. The executeBefore() method calls the Facade object and sets the BeamScanner in a secure position. The secure position is (0, -240). The action proceeds to the TableConverter object. The object gets the table model object from the action and sets the position of the turntable with help of the ComediFacade object and the SerialFacade object. When it is done, the executeAfter() method is called. But in this case this method does nothing, because nothing have to be done.



For moving the turntable, the comedi device and the serial device are needed. The positions of the table can be read from the digital I/O card, but new position can only be set from the serial port. So for setting new positions to the turntable, the actual positions of the table have to be read, the difference between the positions have to be computed and new positions have to be set. Finally the position is read and set to the model object.
After the trigger method executeAfter() is called, the main controller calls the getStream() method from the XmlDAO. This object calls the transformToDOM() method from the TableDAO. TableDAO transforms the, from the TableConverter new set model in a DOM sub tree and adds it to the response document. Finally the XmlDAO builds a string from the DOM and returns it to the main controller. The controller puts it into the output stream of the socket to the client.

Beam scanner

The sequence diagram bellow shows which objects are used for moving or reading the position of the turntable. First the main controller receives the xml stream and hands it over to the XmlDAO. The XmlDAO parses the xml stream and if a turntable tag is in the request, calls the transformToModel() method. The turntable tag is defined in the dtd:

```
<!ELEMENT scan-pair ( scan?, scan?, info?) >
<!ELEMENT scan (pos, start, end, data*) >
<!ATTLIST scan direction (horizontal | vertical) #REQUIRED >
<!ELEMENT pos (#PCDATA) >
<!ELEMENT start (#PCDATA) >
<!ELEMENT end (#PCDATA) >
<!ELEMENT data EMPTY >
<!ATTLIST data axis CDATA #REQUIRED
          intensity CDATA #REQUIRED >
<!ELEMENT info (eff?, bs-cem-hv?, bs-cem-area?, bs-fc?, bs-fc-area?, bm-y?, bm-z?, bm-cem?, bm-
cem-hv?, bm-cem-area?, sis?, ses?) >
<!ELEMENT eff (#PCDATA) >
<!ELEMENT bs-cem-hv (#PCDATA) >
<!ELEMENT bs-cem-area (#PCDATA) >
<!ELEMENT bs-fc (#PCDATA) >
<!ELEMENT bs-fc-area (#PCDATA) >
<!ELEMENT bm-y (#PCDATA) >
<!ELEMENT bm-z (#PCDATA) >
<!ELEMENT bm-cem (#PCDATA) >
<!ELEMENT bm-cem-hv (#PCDATA) >
<!ELEMENT bm-cem-area (#PCDATA) >
<!ELEMENT sis (#PCDATA) >
<!ELEMENT ses (#PCDATA) >
```

For the request only the scan tag with their inner tags does make sense. The inner tag pos describe the position of the beam scanner. It depends on the direction of the scan which position is meant. The pos tag refers to the direction defined as value of the scan tag attribute’s direction and the start and end tags refer to the orthogonal direction of the defined direction of the scan tag attribute’s direction. If a scan is required the start tag defines the start position where the scanner should begin to scan and the end tag the terminal position of the scan. Scans can only be done in two ways, horizontally or vertically. So there are three commands for the beam scanner. Moving the scanner at a new position, scan the beam horizontally or scan the beam vertically. In a request the three possibilities look like:

```
<scan-pair>
  <scan direction=”horizontal”>
    <pos>20</pos>
    <start>10</start>
    <end>-10</end>
  </scan>
</scan-pair>
```

This request does a scan from the position (20, 10) to the position (20, -10).

```
<scan-pair>
  <scan direction=”vertical”>
  <pos>10</pos>
  <start>-50</start>
  <end>50</end>
  </scan>
</scan-pair>
```

This request does a scan from the position (0, 10) to the position (50, 10).

```
<scan-pair>
  <scan direction=”vertical”>
    <pos>20</pos>
    <start>30</start>
    <end>30</end>
  </scan>
</scan-pair>
```

This request moves the beam scanner at the position (30,20). The only difference between the first three requests is, that the last one has an equal value for the start and the end scan. If there is so, the application will not do a scan but only move at the appropriate position. A different request can have the same result. For example the requests

```
<scan-pair>
  <scan direction=”horizontal”>
    <pos>10</pos>
    <start>20</start>
    <end>20</end>
  </scan>
</scan-pair>

<scan-pair>
  <scan direction=”vertical”>
    <pos>20</pos>
    <start>10</start>
    <end>10</end>
  </scan>
</scan-pair>
```

moves the beam scanner at the position (10, 20). The response of a beam scan could look like.

```
<scan-pair>
  <scan direction=”vertical”>
```

```
    <pos>20</pos>
    <start>10</start>
    <end>-10</end>
    <data axis="10" intensity="5.3"/>
    <data axis="9.5" intensity="15.3"/>
    …
    <data axis="-10" intensity="5.3"/>
  </scan>
  <info>…</info>
</scan-pair>
```

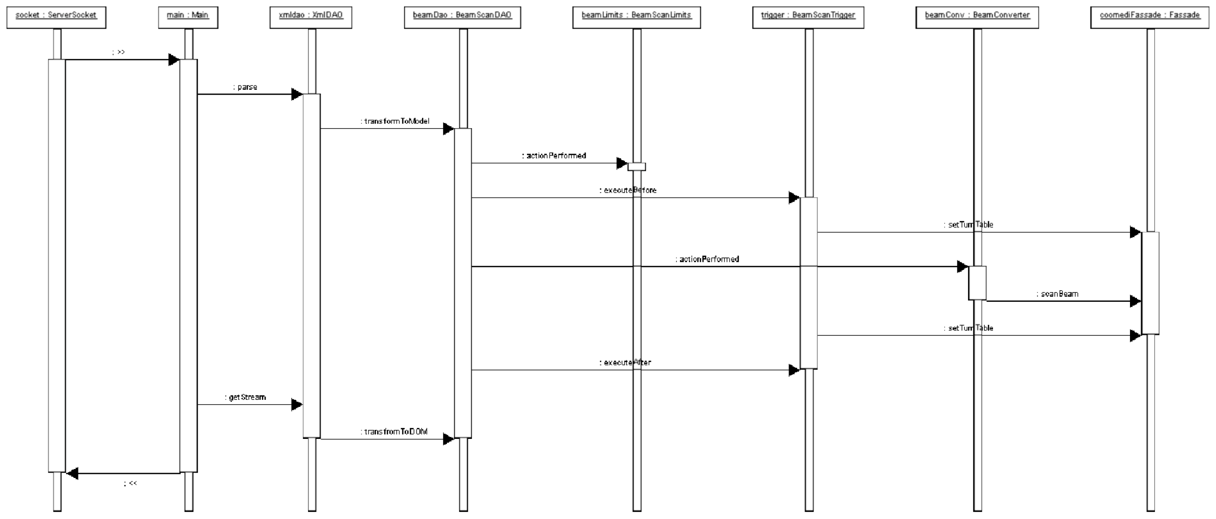
The data tag within the scan tag has two attributes. The value of the axis attribute is the position, the scanners measure the intensity of the beam and the value of the intensity attribute, is the measured intensity of the beam. The example response above, have the following data for creating a beam intensity diagram, done from the GUI client.

Position	Intensity
(-10,20)	5.3
(-9.5,20)	15.3
(10,20)	5.3

For additional information about the other not mentioned inner tags of the tag info, please refer to the documentation of the CASYMS machine.
The BeamScanDAO object transforms the DOM sub tree to a BeamScan model object, sets it to a new action instance and throws the action to the registered listeners. Registered listeners for the BeamScanDAO are: a BeamScanLimits object and a BeamScanConverter object. Additionally a BeamScanTrigger is registered. First the BeamScanLimits object gets the action and validates the request according to limit violation. If everything is ok, the application proceeds. Otherwise the request is interrupted and returns an error as response. For further details on messages please refer to the Error section. Assuming no limit is violated, the trigger is activated and the executeBefore() method is called from the BeamScanLimits object. The executeBefore() method calls the Facade object and sets the turntable in a secure position. The secure position of the turntable is:

• Alpha	=	0
• Beta	=	0
• Theta	=	0
• Z	=	40
• X	=	0
• Y	=	0

The action proceeds to the BeamScanConverter object. The converter does the scan or sets the position with help of the ComediFacade object. When it is done the executeAfter() method is called. This method sets the table on the position it was before the executeBefore() method was called.



Implementation of the CASYMS client

Goals of the Java Client

One DTD for all cases of communication between client and server.
All communication between client and server is validated by a DTD.
All xml is generated dynamically without any print statements.
A Swing GUI that implements the default standards a GUI should have.
No latching of data on the client side in a superfluous data model.
A beam scan graph optimized for the representation on a screen.
Simple and global usable listeners to handle actions.

DTD and cases of communication

One of the goals of the server and client application was to share one common DTD for all cases of communication. The following figure shows in which way the possibilities of the DTD are used in the communication between the client and the server. The left side shows an abstract of the xml a client sends to the server, while the right side shows an abstract of the xml a server returns to the client:

Clients request	Servers response
Establish a valid connection	
An empty session with scope login and one empty request.	A session with scope login data and a response with nested turntable and power data. Or an empty session with scope login and a response with nested message with type error data if another client is already connected or a nested message with type warning data if some table axis are not installed.
Change turntable position	
A session with scope key data and one request with nested turntable (with nested alpha, beta, z, theta, x, y and speed) data.	A session with scope key data and a response with nested turntable and power data. Or a session with scope key data and a response with nested turntable, power and message with type error data if the client is not logged in or tries to move an uninstalled table axis or the move exceeds an axis limit, or a nested message with type warning data if some table values are not decimal or missing.
Make beam scan	
A session with scope key data and one request with nested scan-pair (with one or two nested scan with direction horizontal or vertical with nested pos, start and end) and one info data.	A session of scope key data and a response with nested turntable, scan-pair and power data. Or a session of scope key data and a response with nested turntable, scan-pair, power and message with type error data if the client is not logged in or the move exceeds the scan limit, or a nested message with type warning data if some beam scan values are not decimal or missing.
Change beam power	
A session with scope key data and one request	A session of scope key data and a response with

with nested power data.	nested turntable and power data. Or a session of scope key data and a response with nested turntable, power and message with type error data if the client is not logged in or the change exceeds the power limit, or a nested message with type warning data if some power values are not decimal or missing.
Process a file	
A session with scope key data and one or more request with either a nested turntable, scan-pair, power or wait data.	A session of scope key data and a response with nested turntable, scan-pair and power data. Or a session of scope key data and a response with nested turntable, scan-pair, power and message with type error data if the client is not logged in or tries to change an uninstalled table axis or the change exceeds an axis, scan or power limit, or a nested message with type warning data if some table, beam scan or power values are not decimal or missing or if the wait value is not integer or missing.
Abort a valid connection	
A session with scope logout data and an empty request.	

The following figure shows in which way the possibilities of the DTD are used by the client only. Again, the left side shows an abstract of xml with a request, while the right side shows an abstract of xml with a response:

Request	Response
Save a file	
A session with scope key data and one or more request with either a nested turntable, scan-pair, power or wait data.	
Open a file	
A session with scope key data and one or more request with either a nested turntable, scan-pair, power or wait data.	
Export a beam scan	
	A session of scope key data and a response with nested turntable, scan-pair and power data.
Save a beam scan	
	A session of scope key data and a response with nested turntable, scan-pair and power data.
Open a beam scan	
	A session of scope key data and a response with nested turntable, scan-pair and power data.

DTD and validation

Beside the fact that all data is validated on the server before being submitted to the hardware, the client should prevent that any corrupt data is submitted to the server at all. To be on the secure side, documents are not only validated before submitting. This means that xml is also validated when being received. The validation of data to submit includes a stream to send, a beam scan or task file to save or a beam scan to export. The validation of received data includes a received stream or an opened saved beam scan or saved task file. The following lines of code are from the both classes BasicDOM and BasicXML, which are responsible for sending and receiving, and therefore need to validate data:

For each xml file to validate a temporary file casyms.dtd is created in the same directory as the file we want to validate, by copying the file casyms.dtd from the lib directory in the jar, via the method createDTDFile.

```
File dtd = createDTDFile(file);
```

Whenever the document builder from the package javax.xml.parsers is used, the flag for validating on the document builders corresponding document builder factory is set to true:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
```

Each new created document uses a document builder based on a validating factory:

```
Document document = builder.newDocument();
```

Each opened xml file is parsed to a validated document with a document builder based on a validating factory:

```
Document document = builder.parse(file);
```

Each received xml stream is parsed to a validated document with a document builder based on a validating factory:

```
BufferedReader in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
Document document = builder.parse(new InputSource(in));
```

Each existing document is first transformed to a temporary xml file with the DTD set in the doctype tag, and then parsed back to a validated document with a document builder based on a validating factory, before the validated document it is sent as xml stream, saved as xml file or exported as csv file:

```
File tmp = new File("tmp.xml");
DOMSource sourceDOM = new DOMSource(document);
StreamResult resultStream = new StreamResult(tmp);
Transformer transformer = factory.newTransformer();
transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, dtd.getName());
transformer.transform(sourceDOM, resultStream);
Document document = builder.parse(tmp);
```

The generation of xml

All xml code is generated dynamically on the built DOM by the use of transformer and transformer factory classes from the package javax.xml.transform. Any print or println statement to generate xml is omitted. The following parts of code are all taken from the class BasicXML:

The DOM is built in a classic way by reading out values from the GUI and creating appropriate elements. The values desired depend upon the users interaction, in the following lines it would be the case of a turntable move. The document is initialized by the use of a document builder factory as described in the chapter about the DTD and validation:

```
Element root = (Element) document.createElement(Finals.CASYMS);
document.appendChild(root);
Element session = (Element) document.createElement(Finals.SESSION);
root.appendChild(session);
Attr attr = document.createAttribute(Finals.SCOPE);
attr.setValue(Finals.KEY);
session.setAttributeNode(attr);
Text text = document.createTextNode(socket.getSessionKey());
session.appendChild(text);
Element request = (Element) document.createElement(Finals.REQUEST);
root.appendChild(request);
Element turntable = (Element) document.createElement(Finals.TURNTABLE);
request.appendChild(turntable);
Element alpha = (Element) document.createElement(Finals.ALPHA);
turntable.appendChild(alpha);
text = document.createTextNode(panel.getAlpha());
alpha.appendChild(text);
…
```

A transformer factory is prepared to transform the DOM to a temporary xml file, which is ready to save. Ready to save means that beside the doctype is set as system attribute, the xml is well formatted so that it can be easily read and edited in any text editor:

```
File tmp = new File("tmp.xml");
Transformer Factory factory = TransformerFactory.newInstance();
DOMSource sourceDOM = new DOMSource(this.document);
StreamResult resultStream = new StreamResult(tmp);
Transformer saveTransformer = factory.newTransformer();
saveTransformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, dtd.getName());
saveTransformer.setOutputProperty(OutputKeys.INDENT, "yes");
saveTransformer.setOutputProperty("{http://xml.apache.org/xslt}indentamount", "2");
saveTransformer.transform(sourceDOM, resultStream);
```

The temporary xml file ready to save is written as a persistent file with a buffered Reader and a buffered Writer:

```
BufferedWriter out = new BufferedWriter(new FileWriter(file));
BufferedReader in = new BufferedReader(new InputStreamReader(new FileInputStream(tmp)));
while (in.ready()) out.write(in.readLine()+"\n");
in.close();
out.close();
```

The temporary xml file ready to save is exported as cvs file via transformation from the DOM. The method createXSLFile returns an xsl file csv.xsl by copying the one from the lib directory in the jar:

```
File xsl = createXSLFile(file);
sourceDOM = new DOMSource(this.document);
resultStream = new StreamResult(file);
Transformer exportTransformer = factory.newTransformer(new StreamSource(xsl.getName()));
exportTransformer.transform(sourceDOM, resultStream);
xsl.delete();
```

A transformer factory is prepared to transform the temporary xml file, which is ready to save to a temporary xml String that is ready to send over the socket. Ready to send over socket means that beside the doctype is set as internal subset, no formatting is done to keep the stream compact. The method correctSubsetParseError corrects the error that occurs when a dtd is included as subset via transformer object and the setOutputProperty method: attribute lists with more than one attribute declaration have a closing tag too much!

```
StreamSource source = new StreamSource(tmp.getName());
StringWriter writer = new StringWriter();
StringBuffer buffer = writer.getBuffer();
```

```
StreamResult result = new StreamResult(new BufferedWriter(writer));
Transformer socketTransformer = factory.newTransformer();
socketTransformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, dtd.getName());
socketTransformer.transform(source, result);
correctSubsetParseError(buffer);
String string =  buffer.toString();
```

The temporary xml String ready to submit is sent over the socket with a buffered Reader and a buffered Writer:

```
String str;
BufferedWriter out = new BufferedWriter(new PrintWriter(this.socket.getOutputStream(), true));
BufferedReader in = new BufferedReader(new StringReader(string));
while (in.ready() && (str = in.readLine())!=null) out.write(str.trim());
out.flush();
in.close();
```

Default standards the GUI have

The window

On top of the windows frame are the standard minimize, maximize and close icons. The window is resizable and has an own icon.

The menu bar

On top of the window there is a menu bar, containing from left to right the following menu items: File, Edit, Tools, and Help. The File menu contains Open, Save, Export and Print for the Beam Scan tab, and New, Open, Close and Save for the File tab. The File menu always contains an exit. The Edit menu contains Add Task, Edit Task and Remove Task for the File tab. The Tools menu contains Connection and Verbose Console Output, the first one for changing the servers URL or IP and port, the other one to enable console output if the application is started from a console. The Help menu contains About. The menu items are only enabled, if the appropriate functions are available, for example the menu item Remove Task is only enabled if we are on the File tab and a task node is selected. All menu Items are reachable via Alt key, for example Alt + F for the File Menu, and provide a shortcut, for example Ctrl + X for Exit.

The tool bar

Under the menu bar there is a tool bar that can be placed on the top, left, right or bottom side of the window, or even can be detached from the window. The tool bar contains icons for common shortcuts to the often-used menu items. From left to right there are: New, Open, Close, Save, Print, Export, Add Task, Edit Task, Remove Task, About and Exit. The icons are only enabled, if the appropriate functions are available. When the mouse pointer stays over the icons, a description appears in a tool tip.

The tabs

The main windows main area is divided in five parts by tabs: Overview, Turntable, Beam Intensity, Beam Scan and File. Except the Overview tab all tabs have an OK button on the bottom to activate changes. All changed values are validated when entered. While Overview only shows the current state, Turntable, Beam Intensity and Beam Scan show their current state in the upper half of the tab, while they allow setting a new state in the lower half of the tab. The File tab provides a tree structure.

The Overview tab

The Overview tab shows the Turntable’s position, the Beam Intensity’s power and the Beam Scan’s additional information.

The Turntable tab

The Turntable tab shows the Turntable's position and allows setting a new position by either entering a new absolute position or a relative change. If an absolute (or relative) field is changed, beside the obligate validation, the appropriate relative (or absolute) field is actualized automatically. Further, there is the possibility to enable the manual speed setting by activating a checkbox. If done so, the user can set the speed in percentage, supported by a slider.

The Beam Intensity tab

The Beam Intensity tab shows the Beam Intensity's power and allows setting a new value supported by a slider.

The Beam Scan tab

The Beam Scan tab shows the Beam Scan's vertical and horizontal graph using as much space as possible as well as the scans positions and allows setting the values needed for a new vertical or horizontal scan. It is possible to make a vertical and horizontal scan as well as a single vertical or horizontal scan only by selecting the appropriate checkboxes. A done vertical respectively horizontal scan remain exist until a new vertical respectively horizontal scan is done. By clicking the button Scan Information a new window with additional scan data appears and allows the user to complete those values the server can't evaluate. The Beam Scan's vertical and horizontal graph and the scan information can be saved, opened and printed. Additionally the data can be exported as comma separated values. Export, open and save are realized with standard file choose dialog interfaces as printing is done with a standard print dialog interface.

The File tab

The File tab shows the File's tasks in a tree structure. Possible tasks are: Turntable Task, Beam Intensity Task, Beam Scan Task and Wait Task. The tasks appear in their sequential order as a node under the root Casyms. The parameters of a task appear as leafs of the corresponding task node. New tasks can be added at an arbitrary place under the root, existing tasks can be deleted. The parameters of existing tasks can be edited. To add one or several new tasks, a dialog window to choose the type of task appears, and upon the users selection of type of task from the drop down again a dialog window to edit the chosen type of task appears. This edit task dialog window is closed after the user clicked his OK or Cancel button, while the choose task dialog stays open for reuse until the user clicked his OK or Cancel button. Created files can be saved, closed and opened. Open and save are realized with standard file choose dialog interfaces.

The status bar

On the bottom of the main window is a modestly status bar showing current process state or other information during five seconds.

The error dialog

Whenever an error occurs, there appears an appropriate message in a pop up dialog, giving information about the cause. If the error occurred on the server side, the core of the message is received with the response from the server and a warning message with it is displayed:

Cause:	Warning: Casyms Error Message
The client is not logged in.	Casyms DOM: You are not logged in. Do login first!
The login is not valid or another client is logged in.	Casyms DOM: Wrong session key. Someone else is already logged in or you sent a wrong session key.
Move an uninstalled table axis.	Casyms DOM: Set axis *name* failed. Axis is not installed.
Exceed the table axis move limit.	Casyms DOM: Actual position of *name* does not allow moving the axis more or less than *value*.

Exceed the beam scanners move limit.	Casyms DOM: Actual position of *name* does not allow moving the Beamscanner more or less than *value*.
Exceed the power limit.	Casyms DOM: Power cannot be set more or less than *value*.
General violation of the dtd.	Casyms DOM: Line *number* character *number*: tag *name* missing.

To inform about missing hardware, together with the first response from the server the core of the message is received and a warning message with it is displayed:

Cause:	Warning: Casyms Warning Message
Some table axes are not installed.	Casyms DOM: Axis *names* are not installed!

To inform about parsing errors on the server side by reading values from the request, the following warning messages are displayed:

Cause:	Warning: Casyms Warning Message
The table value cannot be converted to decimal type.	Casyms DOM: You specified tag *name* with the non-decimal value *value*. Set it to 0
The table value is missing.	Casyms DOM: You specified tag *name* without a value. Set it to 0
The beam scan position value cannot be converted to decimal type.	Casyms DOM: Pos tag with the non-decimal value *value*. Set it to 0.
The beam scan position value is missing.	Casyms DOM: Pos tag without value. Set it to 0.
The beam scan start value cannot be converted to decimal type.	Casyms DOM: Start tag with the non-decimal value *value*. Set it to 0.
The beam scan start value is missing.	Casyms DOM: Start tag without value. Set it to 0.
The beam scan end value cannot be converted to decimal type.	Casyms DOM: End tag with the non-decimal value *value*. Set it to 0.
The beam scan end value is missing.	Casyms DOM: End tag without value. Set it to 0.
The power value cannot be converted to decimal type.	Casyms DOM: Power tag with the non-decimal value *value*. Set it to 50000.
The power value is missing.	Casyms DOM: Power tag without value. Set it to 50000.
The wait value cannot be converted to integer type.	Casyms DOM: Wait tag with the non-integer value *value*. Set it to 0.
The wait value is missing.	Casyms DOM: Wait tag without value. Set it to 0.

The following warning messages are displayed if an error occurs on the client side by entering values into the GUI trough the users interaction:

Cause:	Warning: Casyms Number Format Exception Message
Entered port value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid Port number
Entered table value cannot be converted to numeric type.	Casyms GUI: *value* is not a number

Entered table speed value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid Speed value
Entered power value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid Power value
Entered beam scan value cannot be converted to numeric type.	Casyms GUI: *value* is not a number
Entered wait value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid Wait value

The following warning messages are displayed if an error occurs on the client side by setting values into the GUI from the response:

Cause:	Warning: Casyms Parse Exception Message
The alpha value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid alpha value
The beta value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid beta value
The z value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid z value
The theta value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid theta value
The x value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid x value
The y value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid y value
The speed value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid speed value
The power value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid power value
The beam scan position value cannot be converted to numeric type.	Casyms GUI: *value* is not a valid position value
The beam scan information value cannot be converted to numeric type.	Casyms GUI: *value* is not a number

The following error messages are displayed if an error occurs on the client side in one of the classes from the xml package:

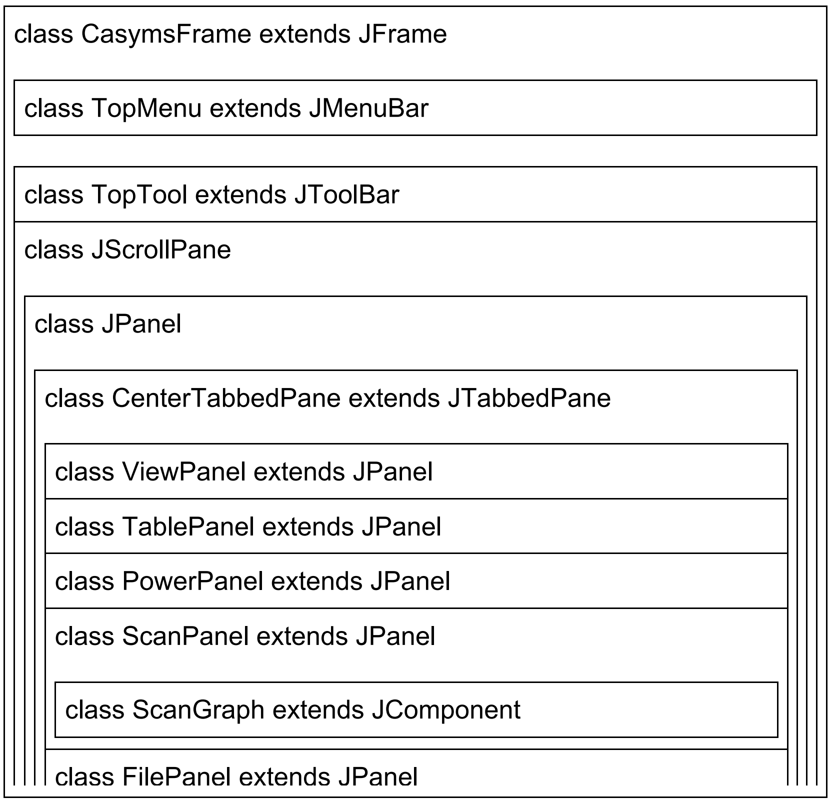
Cause:	Error:
Session key is not valid.	Casyms Exception Message Casyms Socket: Session key not valid
Value does not have the appropriate format to convert to the numeric type.	Casyms Number Format Exception Message Casyms Socket: *value* is not a valid port number
IP address of host could not be determined.	Casyms Unknown Host Exception Message Casyms Socket: *message*
Failed or interrupted I/O operation like server not running.	Casyms IO Exception Message Casyms Socket: *message*

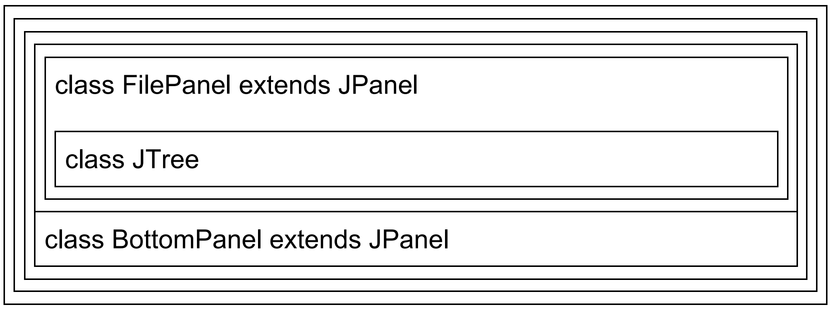
General SAX error or warning.	Casyms SAX Exception Message Casyms DOM: *message*
Failed or interrupted I/O operation.	Casyms IO Exception Message Casyms DOM: *message*
General SAX error or warning.	Casyms SAX Exception Message Casyms XML: *message*
Failed or interrupted I/O operation.	Casyms IO Exception Message Casyms XML: *message*
Exceptional condition occurred during the transformation process.	Casyms Transformer Exception Message Casyms XML: *message*
Indicates a serious configuration error.	Casyms SAX Transformer Configuration Exception Message Casyms XML: *message*

Data model on the client side

The basic idea was to parse the data out of the xml stream directly into the text fields of the GUI when receiving a response, and equally directly parsing the data out of the text fields into an xml stream when sending a request. The received data is only stored in GUI elements and taken from there to generate a new stream to send. For saving, opening or exporting the stream gets replaced with a file. This idea could be realized except for the scan. The scan information data appears non persistent in a new window and therefore is stored in a LinkedHashMap in the scans panel for persistency. The whole response is kept as DOM document when containing scan data, to guarantee the beam scan data is available for printing and saving after another not scan related request is done and a response not containing a scan is received.

Because the GUI represents the data, not only visual but also in the sense of a model, it is important to understand how the GUI components are nested into each other. The following figure shows how classes of the gui package are nested in the main window:


class CasymsFrame extends JFrame
class TopMenu extends JMenuBar
class TopTool extends JToolBar
class JScrollPane
class JPanel
class CenterTabbedPane extends JTabbedPane
class ViewPanel extends JPanel
class TablePanel extends JPanel
class PowerPanel extends JPanel
class ScanPanel extends JPanel
class ScanGraph extends JComponent
class FilePanel extends JPanel


class FilePanel extends JPanel
class JTree
class BottomPanel extends JPanel

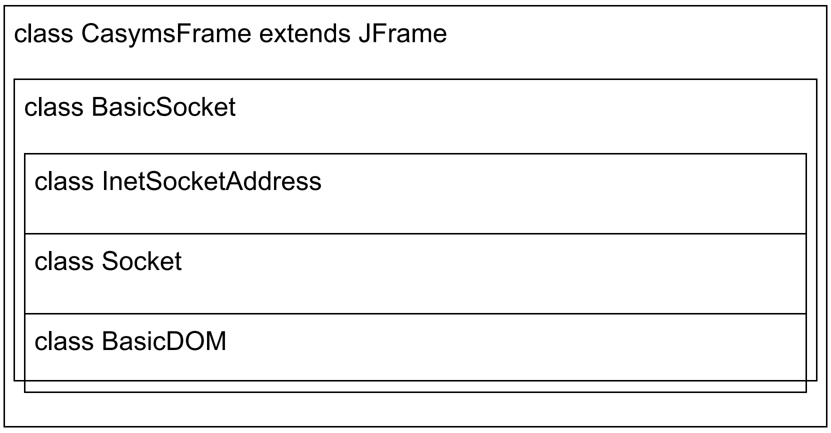
The CasymsFrame class has the following get methods to reach the desired sub components:

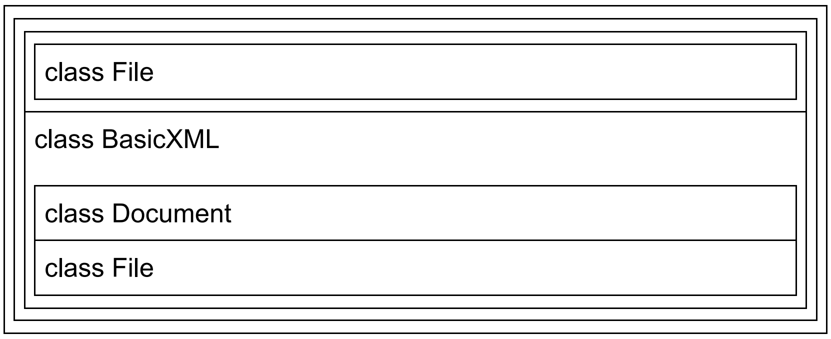
getMenu()
getTool()
getTabs()
getView()
getTable()
getBeam()
getScan()
getGraph()
getFile()
getTree()
getBottom()
getSocket()

The sub component classes have get and set methods for their text fields or similar components if necessary. So if any class knows the CasymsFrame class, it can get or set values in any CasymsFrame sub component class. For this reason most of the constructors need the CasymsFrame class as parameter. In some cases, the following method returns the CasymsFrame, and the constructor can be done without CasymsFrame as parameter:

```
private CasymsFrame getFrame(JPanel panel) {
   Container frame = panel.getParent();
   while (!(frame instanceof CasymsFrame)) {
      frame = frame.getParent();
   }
   return (CasymsFrame) frame;
}
```

The classes in the package xml do the parsing between text fields and xml stream. The class BasicSocket instances a Socket connection as well as the BasicDOM and BasicXML class. The BasicDOM class has the goal to parse the data from the xml stream or xml file and set it to the GUI. The goal of the BasicXML class is to get the data from the appropriate part of the GUI and transform it to a valid xml stream or file. The following figure shows how classes of the xml package are nested in the main window:





The CasymsFrame class has, as mentioned above, a method getSocket(), returning the BasicSocket class. This class itself has a getDom() and a getXml() method, returning the BasicDOM respectively the BasicXML class. The core method from the BasicDOM class is called toGui() and reads the data from the xml stream or an xml file in case of file open by traversing the DOM tree and fills them in the appropriate part of the GUI. The core method from the BasicXML class is called fromGui() and reads the data either from the appropriate part of the GUI, or from the buffered beam scan response in case of scan save and scan export, and transforms it to a valid xml stream or a xml file in case of scan save and file save respectively a csv file in case of scan export.

How the beam scan graph works

The raised data for each beam scan is displayed as a two dimensional graph with the vertical axle representing the beams intensity as Hertz and the horizontal axle representing the scanned range in millimetres. It is possible to draw a vertical and a horizontal beam scan graph, the vertical scan graph is draw in blue color while the horizontal is red. To draw the graphs, as much place as available in the component is used. This means that the inscription of the axles is not only adapted to the range of the raised data (for one scan or for two scans together), but also to the windows size and the font metrics. The adjustment seizes beside the range of the inscription the size of the used interval for each axle, so it is possible the interval gets smaller when the window size increases. In the following part, the scenario that is explained is how a vertical beam scan graph is painted:

In the xml representation, the data the server did measure for the graph is built on several data tags embedded in a scan tag:

```
<scan direction="vertical">
…
<data axis="value" intensity="value"></data>
…
</scan>
```

In the BasicXML class we create a LinkedHashMap to fill in this information, while traversing the response. The LinkedHashMap hash afterwards is passed to the ScanGraph via the ScanPanel class who holds and displays the JComponent ScanGraph:

```
LinkedHashMap hash = new LinkedHashMap();
…
hash.put(((Element) childs.item(i)).getAttribute(Finals.AXIS), ((Element)
childs.item(i)).getAttribute(Finals.INTENSITY));
…
frame.getScan().setVerticalScan(hash);
```

In the ScanGraph class, the LinkedHashMap hash first gets converted into a two dimensional array of type double (the intensity values are multiplied by 1000 because we want hertz and not kilo hertz), and after storing the array simply calls repaint:

```
Set set = hash.entrySet();
double[][] tmp = new double[2][set.size()];
Iterator it = set.iterator();
…
```

```
Map.Entry entry = (Map.Entry) it.next();
tmp[0][i] = new Double((String) entry.getKey()).doubleValue();
tmp[1][i] = 1000.0*(new Double((String) entry.getValue()).doubleValue());
…
dataY = tmp;
repaint();
```

The call for repaint calls paint which calculates the values width and height based on the components width and height and calls the ScanGraph's methods paintCoordinates to paint the axles and their inscription, paintIndex to paint the description on top of the graph and paintLine to finally paint the graph:

```
width = getSize().getWidth();
height = getSize().getHeight();
paintCoordinates(g2);
paintIndex(g2);
paintLine(g2);
```

ScanGraph's method paintCoordinates calls getStepSizeI respectively getStepSizeR needed for the inscription on intensity and range and calculates the values top, left, bottom and right to paint the axes upon them:

```
final double BORDER  = 10.0;
stepI = getStepSizeI(g2.getFont());
stepR = getStepSizeR(g2.getFont());
top = BORDER+getFontMetrics(g2.getFont()).getAscent()+BORDER;
left = BORDER+getFontMetrics(g2.getFont()).stringWidth(""+(int)tillI)+BORDER;
bottom = BORDER+getFontMetrics(g2.getFont()).getAscent()+BORDER;
right = BORDER+getFontMetrics(g2.getFont()).stringWidth(Finals.MM)+BORDER;
…
g2.drawLine((int)left, (int)(height-bottom), (int)left, (int)top); // then draw vertical line
g2.drawLine((int)left, (int)(height-bottom), (int)(width-right), (int)(height-bottom)); // draw horizontal line
```

By calling the methods getStepSizeI respectively getStepSizeR, besides returning the step size, the values fromI, tillI, fromR and tillR are evaluated. They contain the values the inscription of the intensity axle and the range axle starts and ends with. Typically, the start of the intensity axle inscription (fromI) is lower than the smallest intensity measured (minI) and the end of the intensity axle inscription (tillI) is greater than the highest intensity measured (maxI). The same applies to the range axle and their inscription:

```
if (maxI>=0.0) tillI = maxI + (maxI % step);
else tillI = maxI - (maxI % step);
…
if (minI>=0.0) fromI = minI - (minI % step);
else fromI = minI - (step + (minI % step));
…
if (maxR>=0.0) tillR = maxR + (maxR % step);
else tillR = maxR - (maxR % step);
…
if (minR>=0.0) fromR = minR - (minR % step);
else fromR = minR - (step + (minR % step));
```

ScanGraph's method paintLine paints the actual graph by using a transformation to map the machine data to the screen. In this transformation, the values width, height, left, top, right, bottom, fromI, tillI, fromR and tillR are used for scaling, shifting and reflecting the graph:

```
double m00 = (width-left-right)/(tillR-fromR);
double m10 = 0;
double m01 = 0;
double m11 = -(height-top-bottom)/(tillI-fromI);
double m02 = left;
double m12 = height-bottom;
```

```
AffineTransform aT = new AffineTransform(m00, m10, m01, m11, m02, m12);
…
x = -(fromR-dataY[0][i]);
y = -(fromI-dataY[1][i]);
a = -(fromR-dataY[0][i+1]);
b = -(fromI-dataY[1][i+1]);
Line2D.Double l = new Line2D.Double((int)x, (int)y, (int)a, (int)b);
Shape s = aT.createTransformedShape(l);
g2.draw(s);
…
```

The transformation x' = m00 * x + m01 * y + m02; y' = m10 * x + m11 * y + m12 in single steps would mean first to scale the graph to fit the available space, then shift so that the zero point from the data maps the zero point from the axles, and finally reflect the graph at the x axle, because java has the y axis the way that zero is on top, while the graph is desired vice versa:

Scale	(width-left-right)/(tillR-fromR)	0	0
	0	(height-top-bottom)/(tillI-fromI)	0
Shift	1	0	left
	0	1	height-bottom
Reflect	1	0	width
	0	-1	0
All together	(width-left-right)/(tillR-fromR)	0	left
	0	-(height-top-bottom)/(tillI-fromI)	height-bottom

The concept of the listeners

Each OK button from the panels in the tabbed pane as well as all menu items and all too bar items use the same action listener class CommandListener. Upon the actionPerformed method of the CommandListener class, xml data is submitted and received, values in the main window are set or new dialog windows are opened. Whenever a own new dialog window is opened, which means not in cases of file choose or printer dialog windows, the corresponding panels OK button is listened by the class DialogListener. Upon the actionPerformed method of the DialogListener class, xml data is submitted and received and values in the main window and the dialog window are set.

If one of the tabbed pane panels or one of the own dialog window panels component need an action, change or focus listener to enable or disable components, to get values from components or set values to components or to validate values, the panel class implements the listener itself. For example the class TablePanel extending JPanel implements the interfaces ActionListener, ChangeListener and FocusListener for listening on its own check box, slider and text fields. If the check box is selected, the ActionListener enables the slider, if the slider is moved, the ChangeListener sets the sliders value to the appropriate text field, if a text field loses his focus, the FocusListener validates the value, and so on.

Further there is a class FileTreeSelectionListener implementing the interface TreeSelectionListener listening for selection changes on the file tabs tree. Upon the selection, the class lets the menu items and tool bar icons be enabled or disabled. And finally there is a class PrintListener implementing the interface PrintJobListener listening on the print process. This class lets information like if data has been successfully transferred to the print service or not been displayed in the status bar of the window.

All four mentioned classes implementing listeners need the CasymsFrame as parameter in their constructor for operations on nested sub elements from the CasymsFrame. The following lines of code are taken from the class CommandListener:

```
public CommandListener(CasymsFrame f){
    this.frame = (CasymsFrame) f;
}

public void actionPerformed(ActionEvent ae){
    command = ae.getActionCommand();
    …
    else if (command.equals("Add Task...")){
        frame.getFile().addTask();
    }
    else if (command.equals("Beam O.K.")){
        frame.getSocket().getXml().fromGui();
    }
    …
}
```

Use cases

Reading the position of the turn table

The data concerning the position of the turntable is read in by assistance of COMEDI over the digital input/output card. The position of the turntable is given by the six variables alpha, beta and theta as well as x, y, z.

Movement of the turn table in all directions

The situation of the turntable can be determined through the setting of the six variables. The data is sent over the serial port.

Movement of the turn table with absolute and relative values

Sometimes during the definition of a position with a relative value the absolute values of all six variables change, even if the table is changed in only one dimension.

Movement of the beam scanner

The beam scanner could touch the turntable during moving. If this happens, it would damage the delicate beam scanner. So for security purpose, the turntable must be moved in a certain secure position, before the beam scanner can be moved.

Prevent movement by the software during manual use

The beam scanner can be in the way of the movement of the table. If the turntable is operated manually, the table cannot be moved by software for safety reasons.

Software limit and hardware limit of the movement

The hardware limit is the minimal and the maximal position of the turntable or the beam scanner, given by the CASYMS machine. The software limit is to regard as limit configuration of the individual experimental assembly.

Change of the CASYMS configuration

The basic configuration of the software can be changed after an adjustment of the turntable with the assistance of a high precision spirit level.

Only one client can work at the same time

For safety reasons it must be ensured that in all possible cases only one client moves the equipment.

Status report

	What
2	**Erste Version der Requiremenets komplett** Eine erste Version der Requirements liegt vor.
3	**Ansteuern vom Serial Port** Der Serialport kann mit read und write angesprochen w
3	**Verschidene "Hilfsklassen" fertiggestellt 1.Version** Verschiedene Hilfsklassen wie z.B. LinkedList in C erste
	Client und Server können kommunizieren Client und Server Klassen können kommunizieren. Test Nullmodemkabel erfolgreich!
[illegible]	**Deployment Struktur (ant und make)** Erste ant und make files sind bereits vorhanden.
1	**Basic java Client** Erste GUI Version.
1	**Ansteuern von der Digital I/O Karte** Erfolgreicher Test von Termios und Comedi.
1	**Erster Prototyp** Vorführung des ersten Prototypen in der Vakuumkamm Drehscheibe ansprechen und Werte auslesen nach eine
1	**Zweite Version der Requiremenets und Use Cases u** **Dokumentation erarbeiten**
2	**Bewegen des Drehtisches** Der Tisch kann um eine bestimmt Achse und einen bes Betrag bewegt werden.
2	**Auslesen der Position des Drehtisches** Die Positionen aller Drehtischachsen können gelesen w
	Der Drehtisch ist vom Client aus bedienbar
2	**Zweiter Prototyp** Vorführung des zweiten Prototyps in der Vaakumkamm Drehscheibe kann nach vier Monaten mit absoluten unc Werten eingestellt und bedient werden.
5	**Position des Beamscanners lesen**
1	**Beamscanner auf eine bestimmte Position setzen**
2	**CEM, Faraday, usw. lesen** Alle Werte, die über die DigitalI/O Karte gelesen werder lesen
5	**Beamscans funktionieren** Mit einer Positition, einer Start- und Endposition einen E machen
2	**Beamscanner fährt weg, wenn der Tisch bewegt wir**
2	**Tisch auf Position 0, wenn ein Scan gemacht wird**

	Der Drehtisch soll auf 0 fahren, wenn ein Scan gemach der Scan gemacht, geht der Tisch wieder auf die ursprü Position
6	**Speichern der einzelnen Schritte bei einer Ablaufste**
1	**Dritter Prototyp** Benutzeroberfläche ist nach sechs Monaten vollumfäng entwickelt.
1	**Kleine Anderungen vornehmen und das Programm übergeben**
	Dokumentation schreiben und abgeben
	Änderungen der Dokumentation schreiben und abg

* Order of importance: Step >> Milestone >> Prototype >> Delivery

Conclusion

The old application ran on a MicroVAX 3500, a computer first introduced in 1987 that used a KA650 CPU with 22 MHz, 1KB on chip cache, 64KB external cache and maximal 32MB ECC memory. Because the warranty of the MicroVAX computer ran off, in case of a defect it will be the longer the more difficult to replace the old hardware parts. The new computer has a Pentium II CPU with 450 MHz and 256 MB SIMM memory, the operating system is Linux. The old solution was however over decades in use, and we hope so will ours. Creating an application that will be as successful as its predecessor was the main motivation. We are confidently that our solution will be in use for years, because stability, maintainability and expandability were always high weighted during this project.

While requirements collection, analysis, design, testing and writing the documentation was done together, some parts of the implementation were split up in a server and a client part. So one reached a deeper know-how in hardware related programming, while the other learned more about the development of a state of the art GUI. Up on the work we did separately, we realized how important the standards we defined for the project had been. By the work we did together, both of the authors could deepen the before learned knowledge about software development. This includes a responsibility driven design, building frameworks, using development tools as well as testing with tools.

We did analysis, design, implementation, and testing successfully in small iterative steps, except for the requirement analysis and the documentation. We waited to long with enforcing the customer to specify his desires more specifically after the first and second prototype. Beside the only weak represented use cases in our iteration circles, there was a lack of discipline in writing the documentation in iterative steps. This led us to problems with keeping the dates of the documentation delivery. And sometimes it would have been useful to have a simulation with more functions than the one we implemented with our facade. A good experience, beside the software development in iterative circles, was the risk analysis, because no worst case happened. The choose of technologies and the architecture our solution depends on, was validated several times during developing.

The expandability that was, as mentioned in the chapter “Goal of the CASYMS application”, an important task of the project, is on demand already now. The institute of Space Research & Planetary Sciences expands our application for the ion beam power setting of the CASYMS machine. It seems, that our solution is prepared for the future, except for some minor details like a missing abort mechanism. So beside some details the targets could be reached, and both of the authors are satisfied with their work.

Literature

- U. Badertscher und B. Kaspar
  Prototyp einer Steuerung des neuen Drehtisches in der SPAMAS-Vakuumkammer des Physikalischen Instituts der Universität Bern
  1989 (available at the lab)

- Gerhard Willms
  C - Das Grundlagen Buch
  Data Becker, 1997, ISBN 3-8158-1381-6

- Helmut Erlenkötter
  Java-Applikationen
  rororo, 1998, ISBN 3-499-19898-3

- Nicolai M. Josuttis
  Object-Oriented Programming in C++
  Wiley, 2002, ISBN 0-470-84399-3