The LUCIFER Control Software

The Core System, Instrument Control and Scientific Applications

Dissertation
zur
Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
in der Fakultät für Physik und Astronomie
der Ruhr-Universität Bochum

vorgelegt von
Diplom-Informatiker
Kai Lars Polsterer
aus Hamm (Westf.)

Bochum, 10.06.2011

PhD Thesis Supervisor: Prof. Dr. Ralf-Jürgen Dettmar

Second Referee: Priv. Doz. Dr. Dominik J. Bomans

Date of Defense: 12.10.2011

to my family

created with Eclipse 3.4.1[1]
+ TEXlipse 1.2.2[2]
in LaTeX 2_ε[3]
→ BibTeX 0.99c[4]
→ MakeIndex 2.12[5]
→ glossaries 1.19[6]
→ listings 1.4[7]
diagrams with VisualParadigm 7.0[8]
+ Gnuplot 4.2.2[9]
+ *APLpy* 0.91[10]
→ Montage 3.3[11]

[1] *Open Source* development platform consisting out of extensible frameworks, tools and runtimes to develop and maintain software. Managed by the Eclipse Foundation `http://www.eclipse.org`.
[2] *Open Source* plugin that adds LaTeX support to the Eclipse *Integrated Development Environment (IDE)* `http://texlipse.sourceforge.net`.
[3] Document markup language for Donald E. Knuth's TeX typesetting system by Leslie Lamport.
[4] LaTeX package by Oren Patashnik that adds bibliography support.
[5] LaTeX package by Pehong Chen that adds indexing.
[6] LaTeX package by Nicola Talbot that adds glossary support.
[7] LaTeX package by Carsten Heinz that adds source code formatting.
[8] *UML* modeling tool
[9] Scientific command line plotting tool
[10] *Python* package for publication-quality plots of FITS files
[11] Astronomical Image Mosaic Engine developed and supported by *NASA*

Contents

Preface

In the early 20th century science was revolutionised by new scientific approaches. For example, ALBERT EINSTEIN presented his theories on special and general relativity and MAX PLANCK introduced the quantum theory. In the 1920's JULIUS LILIENFELD and later OSKAR HEIL independently developed a first idea of field-effect transistors, a device to electrically control current flow that can be used to digitally represent information. In 1947 at the *Bell Laboratories* JOHN BARDEEN, WALTER BRATTAIN and WILLIAM SHOCKLEY built the first transistor of a germanium crystal. This was the dawn of the age of semiconductors. By packing transistors together in integrated circuits and increasing their packing density current microprocessors can easily be composed of several billion *MOSFETs*.

With the new abilities of semiconductors a new kind of sensors and measuring devices could be built. In astronomy objects in new wavelength windows could be observed. E.g., ARNO PENZIAS and ROBERT WILSON unintentionally discovered 1964 the *Cosmic Microwave Background (CMB)* while working on a cryogenic and ultra sensitive microwave receiver at the *Bell Laboratories*. Again at the *Bell Laboratories* WILLARD BOYLE and GEORGE SMITH invented the first *CCD* detectors in 1969. These detectors offer a much higher quantum efficiency with quasi linear photon response characteristics than photographic plates. Today a 30-cm telescope equipped with an off-the-shelf *CCD*-camera generates images that are comparable with photographic exposures taken at the 5-m *Hale telescope* on *Mount Palomar* in the 1950's.

In 1941 KONRAD ZUSE already built the first fully functional and programmable computer called *Z3* out of telephone relays. This masterpiece of engineering combined with the technique of transistors led to our computerised everyday life. Today almost all electrical devices contain semiconductor based microprocessors. Modern cars have more computing power than mainframe systems of the 1970's and even simple energy saving bulbs use integrated circuits.

As computer power increases a new field of scientific research was born: simulations. Besides theoretical and experimental approaches scientific problems can be analysed by using computer simulations. Based on preliminary theoretical considerations a system is modelled with regards to some assumptions. After this system is simulated the results are compared with the data of other experiments. This allows to analyse the relation between a theoretical model and experimentally found results by confining the free parameters of the model.

Besides the new kind of detectors in astronomy the available computer power led to many technical improvements. Today azimuthal mountings are the most common design used for large telescopes. These kind of mountings permit much stiffer and low weight construction designs with the disadvantage of real time requirements to continuously control the individual axes. With the availability of fast control electronics this disadvantage has vanished. Early approaches to build large optical reflective telescopes failed to produce large, stiff and thermally invariant mirrors. The active compensation of gravitational distortion of the optics by control electronics allows to reduce the thickness and consequentially the mass of the mirrors. This simplifies the thermal stabilisation of the mirror with an optical quality undistorted by gravity. The next big improvement in computer controlled optics was the invention of adaptively compensating mirrors. These *AO* systems are able to minimise the negative effect of the Earth's atmosphere on spatial resolution and to reach the theoretical resolution of an optical system.

The newest evolution on telescope techniques is to synthesise a virtual radio telescope in a supercomputer by interfering the data of single, wide spread radio antennas. With the constantly increasing computing power more and more scientific experiments will migrate from fixed to software based setups. These flexible setups allow faster and easier changes of the configuration or even of the whole experiment. Because it is faster, easier and more cost efficient to change software in comparison to hardware these multipurpose research structures will become a typical approach.

By using microprocessors the number of possibilities to carry out an experiment increases together with its complexity. With further advances in science new scientific questions can only be solved in co-operations that combine the knowledge of individual experts. E.g., the world's largest and complex machine the *Large Hadron Collider (LHC)* is built by over 10,000 scientists and engineers. To manage and organise this huge workforce is another major task. Even though a person builds and understands only a part of the whole machinery each contribution is important for the project to be successful in the end. Thus every scientist and engineer is participating accordingly in the scientific outcome.

The *Large Binocular Telescope (LBT)* is another example of a large cooperation to build a unique research facility. Scientists from *USA*, Italy and Germany are working together to realise one of the biggest optical telescopes in the world. As part of the German collaboration the *LUCIFER* instrument is built by five institutes. This thesis describes the control software of the *LUCIFER* instrument as a key contribution to a successful realisation of a reliably and efficiently working *near-infrared* instrument. Without computer science to handle the complexity of the instrument the astronomer would not be able to carry out his observations.

Outline

This thesis is divided into 10 chapters. The first two chapters are designed to give an introduction. In Chapter 1 the *LBT* and its instrumentation are described. Especially the hardware, optics and electronics of both *LUCIFER* imagers and spectrographs are covered here. Additionally a short introduction to *infrared (IR)* astronomy is given. Chapter 2 is specialised on software development basics. The development environment of the *LUCIFER* project as well as the chosen development approach are presented. Chapter 3 contains the overall design of the *LUCIFER Control Software Package (LCSP)*. The service deployment and start strategies as well as the used external packages are covered. The next four chapters give a more detailed description on the individual tiers of the control software. Chapter 4 contains all frameworks and services needed to run the control software system. The remote service framework as a fundamental part of all services of the control software is presented. The service description includes time synchronisation and message generation as well as persistent storage of data. To implement this storage an *XML* file access package and a special database interaction framework with its own database client had to be developed. Chapter 5 describes the hardware-software interaction tier. The communication framework that enables serial access to the electronics is presented. As the representative services of this tier the services of the motion control electronics and of the switch box electronics are presented. In addition other environment controlling services and their electronics are discussed as well as the central service responsible for tracking the state of the instrument. In Chapter 6 the tier that contains the instrument motion control logics is presented. The complete sequencing framework and its error analysing capabilities are shown. The *Multi-Object Spectroscopy (MOS) Unit Service* as the most complex part of this tier and the specially developed sequencing framework are depicted. Chapter 7 contains a short description of the instrument operation software, the interaction of the embedded services with the other tiers and the *GUIs* which provide access to the observers and engineers. In Chapter 8 the virtual instrument that is used to simulate hardware access and its visualisation interface are presented. First results of *LUCIFER* observations are presented in Chapter 9. Finally in Chapter 10 a new approach in selecting high redshift *QSOs* candidates is presented.

Explanation of Typographic Conventions

This thesis uses a couple of typographic conventions to emphasise special textual content. The following example demonstrate the character formats used and their purpose.

▷ Donald E. Knuth ◁
Persons are shown in small capitals.

▷ *Mount Graham* ◁
Italic font is used for names of places, objects and institutions.

▷ *Gregorian* ◁
Special terms are written in slanted characters.

▷ *Large Binocular Telescope (LBT)* ◁
Abbreviations and acronyms are presented in italic characters.

▷ **de.rub.astro.util.time** ◁
Names of software packages are written in bold typewriter font and use the source code colour.

▷ **TimeClient** ◁
Classes are written in a smaller version of the font used to symbolise packages.

▷ **RMITimeService** ◁
Interfaces are written as classes.

▷ *.getTime()* ◁
A method name is written in italic characters and may be appended to the fully specified class name.

▷ `isSynchronized` ◁
Attributes are presented in small typewriter characters.

▷ **@author** ◁
JavaDoc tags are written in bold face typewriter font in dark green.

▷ **-name** ◁

Program arguments use the same font as *JavaDoc* tags in dark red colour.

▷ `<abcd12345>` ◁

Character strings that are used in the context of software use typewriter font and are framed in "<>". This may be file path strings, name strings, parameter strings and so on.

▷ ISO 9000 ◁

Any kind of standardisation e.g., *ISO* standards or *DIN* norms are printed as sans serif font.

Part I

Introduction

Chapter 1

The LUCIFER Instrument

As an introduction to the embedding project, the Large Binocular Telescope (LBT) and its instrumentation are briefly described. In order to understand the requirements for the control software particularly the mechanics, the optics, the electronics and the detectors of the LUCIFER instrument are presented in detail. Further information on the control computer hardware is given. Finally a short overview of infrared (IR) astronomy provides the scientific background to understand the complexities of operating a near-infrared (NIR) instrument and the consequences for the LUCIFER Control Software Package (LCSP).

Based on the technological progress, new designs of telescopes and instruments become possible. Lighter mirror designs that rely on actively controlled support structures can be used to build larger telescopes. This growth in light collecting area increases the sensitivity of the telescopes and therefore allows to observe fainter objects. Combined with new techniques to cancel the atmospheric distortion, larger telescopes provide higher spatial resolution than classical ground based telescopes. The *Large Binocular Telescope (LBT)* is representative for this new class of technology based telescopes.

1.1 The Large Binocular Telescope

The *LBT* is built by a cooperation of different countries, funded by half by European partners from Italy and Germany. On the Italian side there are the *Osservatorio Astrofisico di Arcetri* (Florence), the *Osservatorio Astronomico di Bologna*, the *Osservatorio Astronomico di Roma*, the *Osservatorio Astronomico di Padova* and the *Osservatorio Astronomico di Brera* (Milan) managed by the *Istituto Nazionale di Astrofisica (INAF)*. The German *LBT Beteiligungsgesellschaft (LBTB)* consists of the *Max-Planck-Institut für Astronomie (MPIA)* in Heidelberg, the *Landessternwarte (LSW)* in Heidelberg, the *Astrophysikalisches Institut Potsdam (AIP)*, the *Max-Planck-Institut für Extraterrestrische Physik (MPE)* in Munich and *Max-Planck-Institut für Radioastronomie (MPIfR)* in Bonn. Both European partners contribute 25 % each to the *LBT* project. The other half is funded by partners in the *USA*. The state of Arizona, paying 25 % of the costs, is represented by the *University of Arizona* (Tucson), the *Arizona State University* (Tempe) and the *Northern Arizona University* (Flagstaff). The remaining 25 % are distributed between the *Ohio State University*, the *University of Notre Dame*, the *University of Minnesota* and the *University of Virginia*. This international cooperation is necessary to build one of the largest optical telescopes on earth. To make such a project affordable for the contributing universities, the telescope has been designed, built and is maintained as a very cost effective project. The overall expenses for building the telescope was estimated in 1989 at approximately 800,000 *US*\$ per square metre of light collecting area. With 110 m^2 this sums up to a total of approximately 88 million *US*\$ of 1989. This figure is two to four times lower than for other ground based telescopes in this class. On the basis of a 10-year telescope life time

Figure 1.1: Enclosure image by DAVID HARVEY from rooftop of the *VATT* (Aug. 2007). The dead trees in the background resulted from forest fires, that almost destroyed the telescope building. (Image taken from http://medusa.as.arizona.edu/lbto/, 2009)

and a construction cost amortisation within that period and including the maintenance and personnel expenses a one-night observation will cost approximately 80,000 *US*$.

At an elevation of 3,192 m the *LBT* is located on *Mount Graham*[1] in the *Pinaleño Mountains* in south east Arizona. It is part of the *Mount Graham International Observatory (MGIO)* together with the 1.8-m *Vatican Advanced Technology Telescope (VATT)* and the 12-m *Heinrich Hertz Submillimeter Telescope (SMT)*. *Mount Graham* was chosen in the 1980s to host an observatory due to the low ambient light pollution, the small amount of atmospheric water vapour and finally the clear skies throughout the year. The excellent infrastructure in that area with a paved road nearly to the top, local technical support and enough space to build telescopes have been other arguments to choose this site. CROMWELL ET AL. (1990) carried out an extensive site testing over years. They determined a median seeing between $0\overset{\prime\prime}{.}55$ (7,165 Å) and $0\overset{\prime\prime}{.}59$ (5,000 Å) compared to $0\overset{\prime\prime}{.}43$ at the *Mauna Kea Observatory* on the island of Hawaii. Although showing a better result, the latter site has the disadvantage of being outside the continental *USA* and of very limited space for new telescopes.

In 1996 the construction of the foundation of the *LBT* was started. Almost 10 years later, in October 2005 the "First Light" image of *NGC*891 was taken with one of the prime focus cameras. In the near future more and more instruments will be attached to the telescope until the full planned performance is reached. This first-generation instrumentation of the *LBT* is described later.

The enclosure of the *LBT* was designed for temperature stability, compact size and uninterrupted airflow between the outside and the telescope chamber. The costs and the influences on the environment have been taken into account, too. Often the drawbacks of older observatories designs were seeing effects caused by thermal turbulences around the optics, heat generating equipment and uncontrolled airflow around the observation slit. Therefore the enclosure of the *LBT* opens to all sides to allow preferably laminar

[1] *Dził Nchaa Si An* (in the Western Apache language) is a holy place of the Apaches and was one of the refuges of GERONIMO.

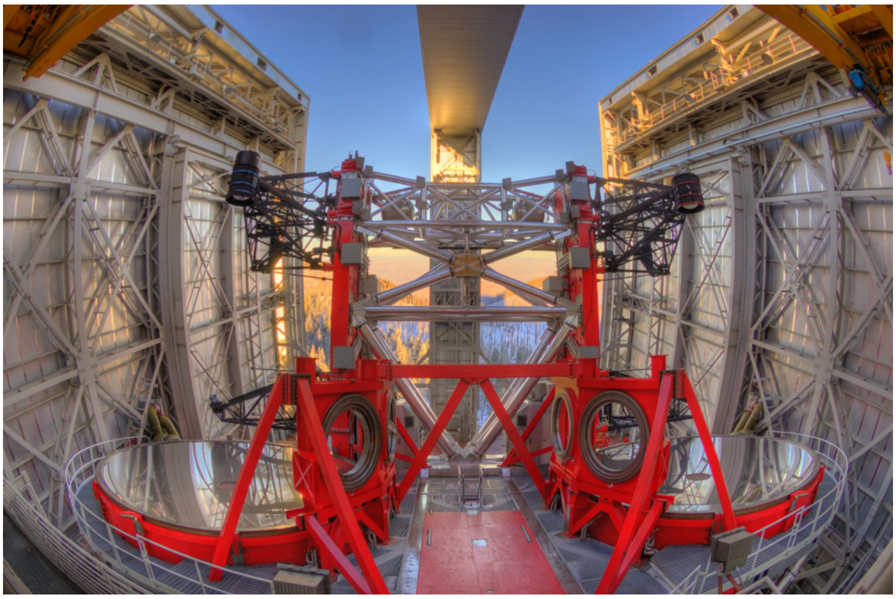


Figure 1.2: Fisheye image of the telescope by Marc-Andre Besel and Wiphu Rujopakarn (Feb. 2008). Above both main mirrors the prime focus cameras are mounted. The bent *Gregorian* focal stations between the main mirrors are still empty. Each of the open shutters in the background have a slit of 10.4 m. Additional lateral slits can be opened. (Image taken from http://medusa.as.arizona.edu/lbto/, 2009)

airflow around the telescope. To control the inside temperature and keep it stable during daytime, the *LBT* dome is equipped with 4 ventilation pipes that are as high as a man (see Figure 1.1). All temperature generating equipment is placed beneath the telescope chamber or resides in special air-conditioned and temperature controlled areas. This new kind of dome design was first developed and used for the *New Technology Telescope (NTT)* (see Wilson, 1983). The whole telescope building consists of an approximately 25 m high structure that contains all technical equipment, the control room, several workshops, offices, clean rooms, a high bay area and the dormitories, kitchen and living room of the staff and astronomers. On top of this structure the telescope enclosure is mounted. This rotatable telescope chamber is a 25 m×28 m×29 m wide cuboid with a total moving mass of about 1,600 tons. Hill and Salinari (2004), Hill et al. (2006) and Hill et al. (2008) give a comprehensive description of the *LBT* project.

The telescope structure and mounting was designed and built in Italy. In 2002 the completed structure was disassembled and shipped to Arizona. To allow an accurate smooth motion of the telescope the whole structure floats on hydrostatic oil bearings operating at 120 bars and is moved by geared electric motors. By using high resolution band encoders and dynamic balancing systems that pump fluids to special ballast tanks, the accurate motion of an estimated total weight of 580 tons is reached. The telescope structure and co-rotating enclosure is able to turn with 1.3°per second. According to Watson (1978) this leads to a zenithal blind spot of <0.5° size.

Two mirrors are mounted side by side onto the telescope structure. Each of these primary mirrors has a diameter of 8.4 m and a focal ratio of F/1.142. They have been designed, spin-casted, ground and polished with an accuracy of 20 nm by the *Steward Observatory Mirror Lab*[2] in Tucson. The finished mirrors are aluminised at the telescope without the

[2]The *Mirror Lab*, founded in 1980 by Roger Angel, is placed beneath the American Football stadium that hosts the local university team, the *Arizona Wildcats*. This laboratory is one of the world leading facilities for lightweight, huge and powerful mirrors.

primaries	figure:	parabolic concave
	size:	8.4 m ∅, 894 mm edge thickness
	central hole:	889 mm ∅
	focal ratio:	F/1.142
	material:	E6* borosilicate glass in honeycomb layout
	weight:	16 tons
	mounting:	160 actively compensating pneumatic actuators
	cooling:	by air flow
secondaries	figure:	parabolic concave
	size:	911 mm ∅, 1.6 mm thick
	focal ratio:	F/15
	weight:	≈10kg (shell), 0.6 tons (total adaptive secondary)
	mounting:	adaptively on 672 electromagnetic actuators
	attachment:	with hexapod on movable swingarm
tertiaries	figure:	flat
	size:	500 mm × 640 mm
	attachment:	on movable swingarm 2.25 m above primary vertex

Table 1.1: Parameters of the *LBT* optics.
*The *E6* glass of the *Ohara Corporation* in Japan is used. This material has good processing characteristics and a thermal expansion coefficient of 2.9 *ppm*/K. For comparison *ZERODUR*® produced by *SCHOTT Germany* has a near-zero expansion coefficient with a homogeneity between 0.1 to 0.01 *ppm*/K.

necessity to remove them. A honeycomb layout is used, that provides structural stiffness while significantly reducing the weight of each mirror from 100 tons of a solid one to merely 16 tons. Due to the relatively high thermal expansion coefficient of the used borosilicate glass and the deformation caused by the Earth's gravitational field, active regulation of the mirror within its cell is needed. Because of the low mirror mass the thermal expansion is compensated quite easily with air cooling. Additionally, the diurnal temperature variation is reduced by keeping the mirror at night temperature. To compensate for the gravitational field, active optics is needed. Therefore the primary mirrors are mounted each on 160 actively regulated pneumatic actuators. The active optics system and its optimisation is described in Martin et al. (2004).

Both mirrors combined provide a light collecting area that is equivalent to a single 11.8 m mirror. Mounted at a 14.4 m distance (centre to centre), interferometricly combined the mirrors have an effective resolution of a 22.6 m mirror. This baseline fills the gap between current 10-m class telescopes and long baseline interferometers like *VLTI*[3] (46 m – 200 m baseline) or *Keck*[4] (85 m baseline). Herbst and Hinz (2004) and Wagner (2007) give an overview on the interferometric instruments and the planned experiments with the *LBT*.

To achieve full optical performance with an interferometric resolution of 5 *mas* in visible light and 20 *mas* in the *near-infrared*, the distortion of the wavefront generated by atmospheric turbulences needs to be compensated. Therefore *Adaptive Optics* is required. Until the first adaptive secondary is attached and fully functional [5] a rigid secondary is

[3] *Very Large Telescope Interferometer (VLTI)* is part of the *Very Large Telescope (VLT)*, four 8.2 m telescopes placed on *Cerro Paranal* (2,635 m) in Chile. The *VLT* is maintained by the *European Southern Observatory (ESO)*. *VLTI* combines the light of the four main and several auxiliary telescopes to obtain the different baselines.
[4] The *Keck Observatory* consists of two segmented 10 m mirrors, located on *Mauna Kea* (4,145 m).
[5] The adaptive secondary is expected to be operational by the end of 2011.

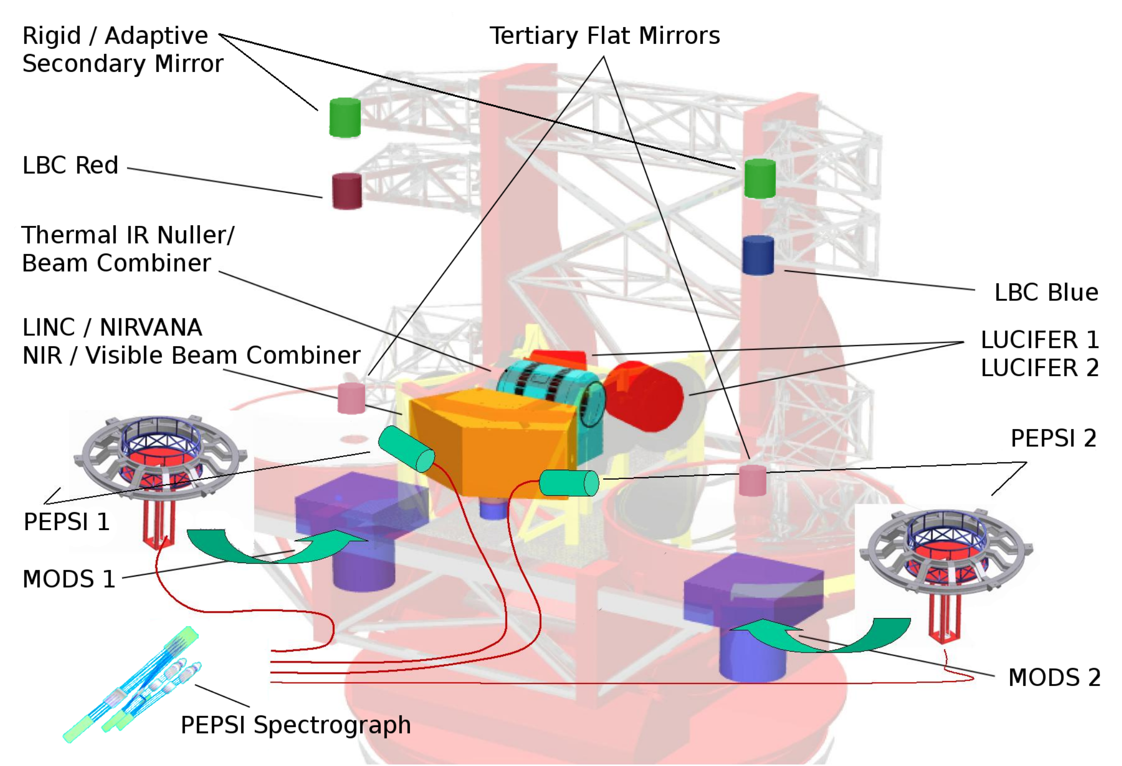


Figure 1.3: Instruments of the *LBT* at the different focal stations. (image adapted from `http://www.aip.de/groups/pepsi/`, 2009)

used. The concave secondaries with a focal ratio of F/15 are mounted on 672 electronical actuators. In order to cancel the atmospheric effects, these actuators can transform the 911 mm wide and 1.6 mm thick secondary at kilo Hertz rates. A real time computer and control electronics system, with a parallel computational power of 163 *Gflop/s*[6], analyses the distorted wavefront and calculates the correction at each individual actuator. The thin mirror shell was built by the *Mirror Lab* and shipped to *Arcetri* (Italy) where the *Adaptive Optics* system is manufactured. Riccardi et al. (2003) describe the design of the adaptive secondaries of the *LBT*. For the current status see Riccardi et al. (2008).

1.2 The Instruments of the LBT

Depending on the used optical setup, the entering light can be bypassed to one of the instrument foci. To make use of the short focal length of the primary mirrors, cameras have been built on each side. They are mounted on swingarms for a flexible use in the optical beam. When the *Gregorian* setup is used the prime focus cameras are replaced by rigid or adaptive concave secondaries. On each side a tertiary flat mirror reflects the beam to one of the three focal stages between the primaries. Without this flat mirror the beam is directly passed to the primary *Gregorian* focus. The *LBT* instruments and their focal stations (see Figure 1.3) are described in the following subsections. In Table 1.2 fundamental instrument parameters are compared. The *LUCIFER* instrument is described in detail in the Section 1.3.

[6]For comparison a *PC* processor (*Intel Core i7 965XE*, quad-core) performs with ≈ 70 *Gflop/s* while a *GPU* (*nVidia GeForce GTS 260M*) provides ≈ 400 *Gflop/s* in double precision.

Instrument	**Focal Station**	**Modes**	**Spectral Coverage**	**Spectral Resolution**	***FOV***
LBC-Blue *LBC-Red*	Prime	*CCD-Mosaicing*	0.32 – 0.5 μm 0.5 – 1.0 μm	4 – 50	$27' \times 27'$
MODS-Blue *MODS-Red*	Direct F/15	Imaging, *MOS*, *Long-Slit*-Spectroscopy	0.32 – 0.55 μm, 0.55 – 1.1 μm	2k opt.5k – 8k	$6.'5 \times 6.'5$
LUCIFER1 *LUCIFER2*	Front-Bent	Imaging, *MOS*, *AO*, *Long-Slit*-Spectroscopy	0.9 – 2.5 μm	5k – 10k *AO*≈40k	$4' \times 4'$ $30'' \times 30''$
LBTI	Centre-Bent	a)*Fizeau*-, b)*Nulling*-Interferometry	a) 5 – 28 μm b) 3 – 5 μm and 8 – 13 μm	2 – 30	$40'' \times 40''$
LINC-NIRVANA	Rear-Bent	*Fizeau*-Interferometry	0.6 – 2.4 μm	5 – 20	$10'' \times 10''$– $20'' \times 20''$
PEPSI	Rear-Bent Direct F/15	Spectroscopy Spectro-polarimetry	0.39 – 1.1 μm	40k – 300k	$0.''5$ – $1.''4$

Table 1.2: Comparison of the characteristics of the *LBT* instruments. (Table adapted from Wagner, 2008)

1.2.1 The Large Binocular Camera (LBC)

The *Large Binocular Cameras (LBCs)* are built in a joint project of different Italian *INAF* observatories, the *Osservatorio Astronomico di Roma*, the *Osservatorio Astrofisico di Arcetri* (Florence), the *Osservatorio Astronomico di Padova* and the *Osservatorio Astronomico di Trieste*. Both instruments have been the first scientific instruments attached to the telescope. Mounted on moveable swingarms they can be brought into the prime foci of the F/1.142 main mirrors to cover a wide field of $23' \times 23'$ in the sky. A group of 6 lenses corrects the optical beam and produces a flat image on the detectors. The biggest lens has a diameter of approximately 0,8 m. Each camera uses an array of four *CCD* chips with 2,048×4,608 pixels each. The positions of the *CCDs* are optimised to cover the corrected scientific field of view leading to an effective detector size of 6,150×6,650 pixels. Each camera hosts two filter wheels with a total of 8 usable filter positions. One of the cameras is optimised for the wavelength range between 0.5 μm and 1.0 μm. This *LBC-Red* uses *IR* coated detectors and lenses with good optical long wavelength characteristics. The other camera, *LBC-Blue*, is optimised for the UBV bands (0.32 μm – 0.5 μm) and uses *UV* coated detectors as well as lenses with good short wavelength characteristics. By observing the same object with both instruments the efficiency of the *LBT* can be doubled. Speziali et al. (2008) describe the instrument and evaluate the performance of the first binocular observation runs.

1.2.2 The Multi-Object Double Spectrographs (MODS)

MODS is the optical counterpart of the *LUCIFER* instrument (see Section 1.3). Both identical *MODS* instruments are being built by the *Ohio State University*. As described in Pogge et al. (2006), these low- to medium- dispersion spectrographs are placed in the direct F/15 focus behind the central hole of the primary mirrors. In each instrument the optical beam passes a dichroic splitter and is guided to a separate wavelength optimised red and blue channel. Each channel will have its own *CCD* detector with a size of

4,096×4,096 pixels in the commissioning phase. For scientific use these detectors will be replaced by 3,072×8,196-pixel detectors with the same pixel size of 15 μm×15 μm. By using a custom designed stabilisation system to compensate actively the image motion, long time observations are not affected by gravitational deformation of the large instrument structure. A moveable turret stores the gratings and a prism for different spectroscopic resolutions as well as a plain mirror for imaging. The focal plane of the *MODS* instrument can be equipped with a mask during observation. These masks are stored in a cabinet with 24 positions and can either be used to for common *long-slits* or user-defined multiple object masks for *Multi-Object Spectroscopy (MOS)*.

1.2.3 The Large Binocular Telescope Interferometer (LBTI)

NASA[7] and the *University of Arizona* are going to build the *LBT Interferometer (LBTI)*, that is located in the central bent *Gregorian* focus. *LBTI* is designed as a pathfinder project for further space missions. By destructively interfering the signals from both mirrors, the light of a central star can be reduced by factor 10^4. This allows imaging of young planets and determination of dust properties in circumstellar disks. To reduce the instrument related thermal noise, typical for the observed wavelengths, all optical components of *LBTI* need to be cryogenically cooled down to 77 K with liquid nitrogen. In order to reach the aimed spatial resolution of a 22.6 m wide telescope, both secondaries need to be working adaptively. With a spatial resolution of $0\farcs04$ and a $40'' \times 40''$ field of view, the *mid-infrared* *Fizeau*-mode camera provides unique scientific capabilities especially for extrasolar planet research and analysis of highly complex sources (see Mainzer et al., 2006). There are two other cameras for *nulling* and imaging. Their super high spacial resolution allows to examine the zone of habitable planets, that corresponds to 0.3 *AU* - 15 *AU* at a distance of 10 *pc* to 500 *pc*, respectively. Hinz et al. (2008) give more detailed information on the *nulling* and imaging camera of the *LBTI*.

1.2.4 The LBT Interferometric Camera (LINC) - Near-Infrared/Visible Adaptive Interferometer for Astronomy (NIRVANA)

LINC-NIRVANA is a *Fizeau* interferometric imager, built for the rear bent *Gregorian* focus by a consortium of the *Max-Planck-Institut für Astronomie (MPIA)* (Heidelberg), the *Istituto Nazionale di Astrofisica (INAF)* represented by *Osservatorio Astrofisico di Arcetri* (Florence), the *Erstes Physikalisches Institut der Universität zu Köln* and the *Max-Planck-Institut für Radioastronomie (MPIfR)* (Bonn). In *Fizeau* interferometers, the wavefronts interfere in the focal plane instead of the pupil plane. Therefore *LINC-NIRVANA* will be able to produce real images with a spatial resolution of e.g., 10 *mas* in the J band. The field of view is theoretically limited by the ability of the adaptive secondaries to produce a flat wavefront over the maximum possible scientific $2' \times 2'$ field. In this project the field of view is limited by the costs of the used 2,048×2,048-pixel *HAWAII-2* detector arrays, that cover merely a $10'' \times 10''$ field of view. For the full field of view 144 detectors of this type would be required. It would be a technical challenge to arrange, control and read out this multitude of detectors. The complexity of controlling a single detector array of this type is described in Subsection 1.3.5. The main task of the *LINC-NIRVANA* instrument

[7] *National Aeronautics and Space Administration (NASA)* is the agency of the United States that is responsible for the *US* aerospace research and space program. *NASA* was founded in 1958 under president Eisenhower as a response to the Soviet Union's *Sputnik* project, sending the first artificial satellites into Earth orbit.

is to control and stabilise the interfering focal planes. This is done by using complex opto-mechanical devices, that compensate e.g., the field rotation and the optical path length. Mounting two mirrors side by side reduces the problems of interfering the light of two telescopes. Due to the anticipated spatial resolution of 10 *mas* any vibrations caused by the own motors of the instrument, the telescope mounting or other instruments must be eliminated or at least be significantly reduced. Astronomical observations are performed under different rotation angles to compose a high resolution image. This observation technique is necessary because one axis of the optical setup synthesises a resolution of a 22.6 m telescope while the resolution of the other axis is equal to the size of a single mirror. *LINC-NIRVANA* is described in detail in Herbst et al. (2004) while the current status is reflected by Herbst et al. (2008).

1.2.5 The Potsdam Echelle Polarimetric and Spectroscopic Instrument (PEPSI)

PEPSI, as described in Strassmeier et al. (2003), is a *PI* instrument that uses the special *LBT* design of two mirrors mounted side by side. This instrument will allow simultaneous observations of circularly and linearly polarised light. *PEPSI* will have a very high spectral resolution between 40,000 and 300,000 for wavelengths of 0.45 μm – 1.1 μm, respectively. Furthermore *PEPSI* will also be able to resolve short-time effects. The air-conditioned *Echelle* spectrograph is located inside the telescope building and is attached to the telescope via fibres mounted to the polarimetric units at the direct F/15 and rear bent *Gregorian* foci. Both rear bent units are permanently mounted and can be used without the exchangeable direct F/15 units for non-polarimetric spectroscopy. To reach full resolution, both adaptive secondaries need to be working. Two *Acquisition, Guiding and Wavefront Sensing Units (AGWs)* will be mounted to the direct F/15 foci. Their task is to get the necessary data to control the active and adaptive optics of the telescope. Two similar *AGWs* are also mounted to the front bent *Gregorian* foci in order to support the *LUCIFER* instrument during observations.

1.3 LBT NIR Spectroscopic Utility with Camera and Integral Field Unit for Extragalactic Research (LUCIFER)

Both *LUCIFER* instruments are built by a collaboration of five German institutes. The *Landessternwarte (LSW)* (Heidelberg) is the head institute of this consortium and responsible for project management and coordination of the different partners. The mechanical and optical instrument design as well as the system integration and testing was done by the *LSW*. This design process of the *LUCIFER* hardware was supported by the *Fachhochschule für Technik und Gestaltung (FHTG)* in Mannheim. The robot mechanism that exchanges user-defined masks in the *LUCIFER* instrument was designed and built by the *Max-Planck-Institut für Extraterrestrische Physik (MPE)* in Garching. This most critical part allows *Multi-Object Spectroscopy (MOS)*. Another partner in Heidelberg, the *Max-Planck-Institut für Astronomie (MPIA)*, contributed the instrument control electronics, the detector and the cryo-design. The *Astronomisches Institut der Ruhr-Universität Bochum (AIRUB)* is responsible for developing the *LUCIFER* instrument control software, that is part of this thesis.

LUCIFER is a pair of *near-infrared* spectrographs and imagers with multiple observation modes. One of the first presentations of the *LUCIFER* project was given in Mandel et al. (1999). In (2008/2009), 10 years later, it was installed at the telescope and ready

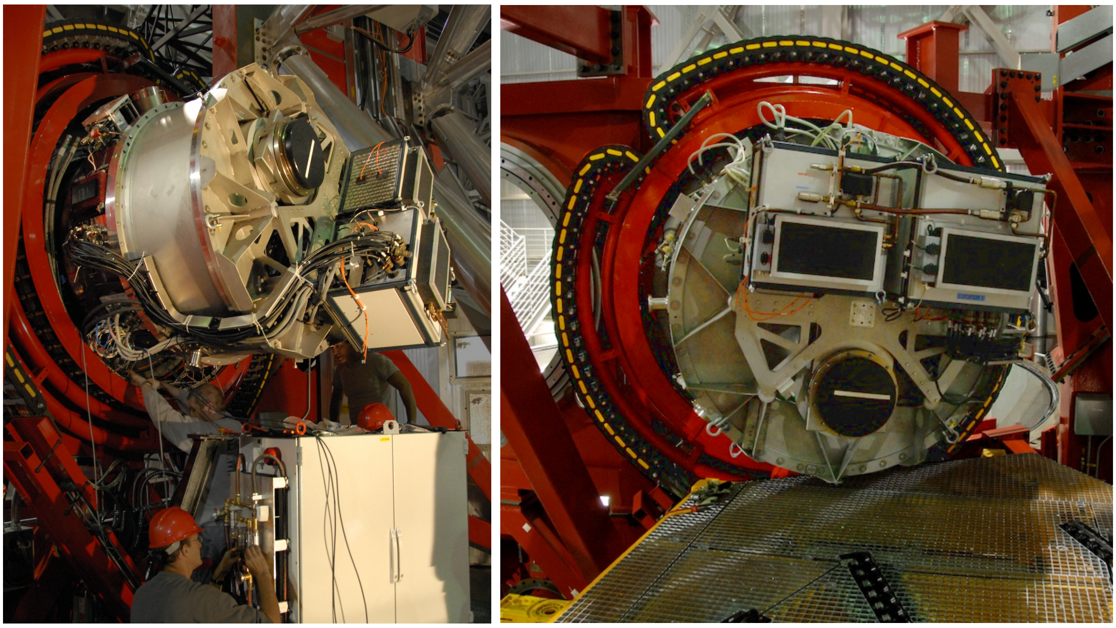


Figure 1.4: *LUCIFER* attached to the *LBT*. **Left:** The control and readout electronics mounted to the end of the dewar. Beneath *LUCIFER* the main electronics rack is placed. **Right:** Cable wrap and instrument behind the auxiliary cryostat gangway. (Images by Peter Buschkamp).

for scientific use. The progress of the *LUCIFER* project can be traced by looking at the different status reports (Seifert et al., 2003; Mandel et al., 2004, 2006, 2007, 2008). Both instruments comprise the *near-infrared* equipment of the *LBT*, covering the wavelengths between 0.85 μm and 2.5 μm. Depending on the instrument setup selected, *NIR* imaging can be done by using the 2,048×2,048-pixel *Rockwell HAWAII-2* detector (see Subsection 1.3.5). The field of view of *LUCIFER* is $4' \times 4'$ in seeing- and $30'' \times 30''$ in diffraction-limited observation mode. Combined with the three internal camera optics *LUCIFER* offers a spatial resolutions of $0''.25$, $0''.12$, and $0''.015$ per detector element. Other instrument setups allow *long-slit* and *Multi-Object Spectroscopy* with resolutions of 5,000 to 10,000 in seeing limited and 40,000 in diffraction limited mode, respectively. These multiple observation modes make *LUCIFER* the so called Swiss army knife of the *LBT*. In order to reduce the thermal interference of the instrument, the *LUCIFER* mechanics and optics are placed in a cryostat. As other infrared instruments this cryostat is evacuated and cooled down to temperatures of ≈60-70 K. Because of this cooling in the majority of cases *NIR* instruments are more costly and complicated than optical ones. The *LUCIFER* cryostat is 1.6 m high and has a diameter of 1.6 m and is one of the largest cryostats used in astronomical applications (see Image 1.4). In the following the different functional groups of the *LUCIFER* instrument and their implied requirements are described more precisely. These specified hardware requirements are significant for the control software.

1.3.1 The Optics

Even though *LUCIFER* has a comparably large cryostat, it requires a compact optical design. Therefore the optical path is several times folded in order to fit into the cryostat. This optical design is described in detail in Seifert and Xu (2002). The scientific *NIR* beam, coming from the telescope, enters the instrument through the tilted dichroic entrance window. On the other hand the optical part of the scientific beam is reflected by

		Seeing Limited		Diffraction Limited
		N1.8-Camera	N3.75-Camera	N30-Camera
Camera	Focal Ratio	$f/1.8$	$f/3.75$	$f/30$
	Focal Length	180 mm	375 mm	3,000 mm
Telescope	Focal Ratio	$f/15.0$		
	Focal Length	123,769 mm		
Collimator	Focal Length	1,500 mm		
Effective	Focal Length	14,850 mm	30,940 mm	247,540 mm
Pixel Scale		0″.25/pixel	0″.12/pixel	0″.015/pixel
FOV	Imaging	$4' \times 4'$		$30'' \times 30''$
	Spectroscopy	$4' \times 3'$		

Table 1.3: Basic optical data of the *LUCIFER* instrument. Compiled from Seifert and Xu (2002)

the plan-parallel 160 mm×225 mm entrance window. The *Acquisition, Guiding and Wavefront Sensing Unit (AGW)* gathers the necessary data out of this visible beam to analyse the wavefront for the *Adaptive Optics*. In the cryostat the beam passes the *Focal Plane Unit (FPU)* that is used to reproducibly position *long-slit* and *MOS* masks in the focal plane. Immediately behind the focal plane the first lens of the collimator is mounted. Then the beam passes 3 folding mirrors, before the last two collimator lenses are reached. The whole folded refractive collimator has an effective focal length of 1,500 mm and produces a collimated beam of 102 mm size. Two of the four folding mirrors can be moved during observation in order to compensate the instrument flexure. Behind the folded collimator optics the last folding mirror bends the beam towards the grating turret. The *Grating Unit* is placed at the pupil of the instrument. It contains a flat mirror for imaging and a high dispersion plus two low dispersion gratings for spectroscopy. Reflected by the *Grating Unit* the beam enters the *Camera Unit*. To change the image sampling resolution during observation, three different cameras are mounted on a wheel. The N3.75 camera has a spatial resolution of 0″.12 per pixel[8] and can be used for seeing limited imaging and spectroscopy, while the N1.8 camera with 0″.25 per pixel is optimised for spectroscopy. With 0″.015 per pixel the N30 camera is used for diffraction limited observation modes. See Table 1.3 for detailed information on the *LUCIFER* optics. Both N1.8 and N3.75 cameras have a space between the main optics and the necessary field lenses where the *Filter Wheel Unit* intersects the beam. One of the field lenses is shown in Figure 1.5, just right of the camera body (in red). The *Filter Wheel Unit* consists of 2 rotating wheels with 27 scientific filter positions. Behind the *Filter Wheel Unit* the detector is mounted on an adjustable stage to correct the focal length depending on the camera and filters used.

In the first installed version of the *LUCIFER* instrument the intended *Integral Field Unit*, the *Slit Viewer*, the *Atmospheric Dispersion Corrector (ADC)* and the *NIR Tip-Tilt (TT)* system are missing. Although these parts are not installed the required space is reserved and they can be installed during later instrument upgrades. The *ADC* is required for *AO* observations and will be installed in the near future.

1.3.2 The Mechanics

Due to the requirement of reducing thermal interference, *LUCIFER* needs to be cryogenically cooled down to 60 K. Therefore the optical parts of the instrument have to be placed in a dewar. All mechanical movable parts inside of *LUCIFER* are utilised to change

[8] 0″.24 per pixel with *Nyquist Sampling*.

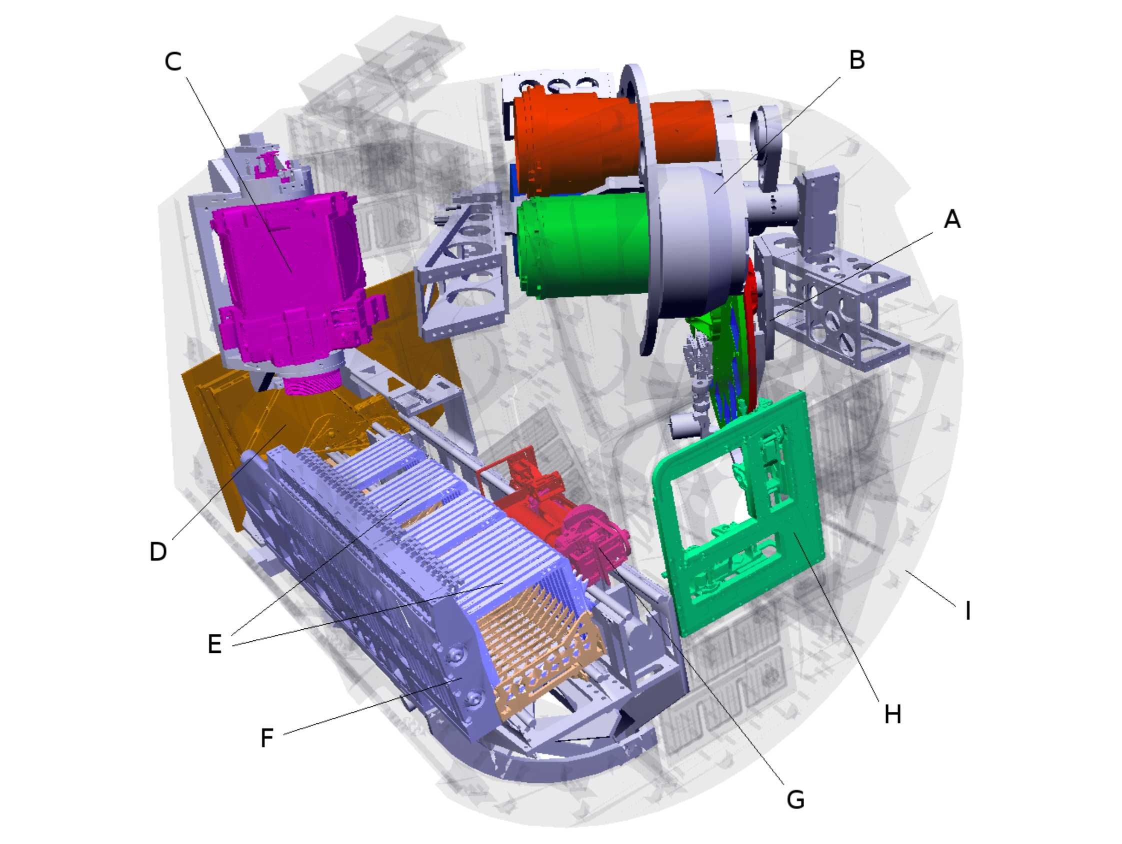


Figure 1.5: *LuciferVR* image of the different mechanical *LUCIFER* instrument units. **A:** filter wheel, **B:** camera wheel, **C:** grating selection, **D:** shield shutter, **E:** exchangeable and fixed mask cabinet, **F:** *MOS Unit* structure and mask retainer, **G:** mask handling robot, **H:** focal plane mask interface, **I:** inner structure.

the optical instrument setup. Building opto-mechanical parts that can be moved at liquid nitrogen temperatures (77 K) and below is a challenging engineering task. Lubrication of bearings or gears is very complex in a low pressure/temperature environment. Error diagnostics, in case of a stuck or not appropriately moving element, cannot be performed so easily as under non-cryogenic conditions. The visual analysis of the problematic parts can only be accomplished through small inspection windows inside the dewar vessel. Additionally the different thermal expansion coefficients of the used materials have to be taken into account. There is no guarantee that an assembled section that moves flawlessly at ambient temperatures, will behave the same at cryogenic temperatures.

Table 1.4 gives an overview of the mechanical units and the assigned functions. By a total of ≈150 switches and 30 stepper motors the motion of the opto-mechanical elements of *LUCIFER* is realised. Additionally electromagnetic tilt units, cold-clamps, electromagnetic locks, strain gauges, reed contact sensors and angular resolvers/incremental position encoders are utilised for particular tasks. The implemented hybrid stepper motors combine the advantages of reluctance and permanent-magnet stepper motors. This type of motor minimises the magnetic holding torque and the amount of steps lost. *LUCIFER* is delivered in a stripped-down configuration with 3 of the anticipated units missing. This fact leads to a reduced number of 22 motors. 12 of these motors are used by the *MOS Unit*, that is responsible for reproducibly transporting a mask to the focal plane and back to the storage again. The required positioning accuracy of a mask is 10 μm. The *MOS Unit* additionally has a cryogenic cabinet exchange mechanism to change masks without the

Instrument Unit	**Subunit**	**Task/Hardware/Comment**
Grating Unit		to exchange gratings/plain mirror
	Grating Turret	**1** motor, **3** switches, motor switches*,
	Grating Tilt	**3** tilt selection units
Camera Unit		to exchange camera optics
	Camera Wheel	**1** motor, **2** switches, motor switches*
Filter Wheel Unit		to exchange filters
	Wheel One	**2** motors, **8+2** switches, **2** cold-clamps
	Wheel Two	one position contains a pupil viewer lens
Pupil Viewer Unit		to move lens for pupil image analysis
	In-Out	**1** motor, motor switches*
Flexure Compensation Unit		to compensate for instrument flexure
	Mirror One-1	**4** motors, motor switches*
	Mirror One-2	
	Mirror Four-1	
	Mirror Four-2	
Detector Focus		to bring the detector into focus
	Focal Stage	**1** motor, motor switches*
ADC Unit		to compensate for atmospheric dispersion
	In-Out	**3** motors, motor switches*
	Prism A	
	Prism B	NOT CONSTRUCTED YET
Slit Viewer		to put additional detector in front of mask
	In-Out	**1** motor, motor switches*
		NOT CONSTRUCTED YET
NIR Tip-Tilt		to move *NIR TT* sensor into beam
	In-Out	**4** motors, motor switches*
	X Position	
	Y Position	
	Focus	NOT CONSTRUCTED YET
MOS Unit		to bring mask into focal plane
	FPU Clamp X	**12** motors, **3** electromagnetic locks,
	FPU Clamp Y	**6** encoders, **114** switches, **2** strain-gauges, **2** reed contact sensors
	MHU Translation	**1** gravitational 3-axis accelerometer
	MHU Rotation	
	Mask Grabber	
	Mask Selection	
	Mask Locking	
	Cabinet Locking	
	Cabinet Translation	
	Shield Shutter	
	AC Translation	
	AC Thermal Bridge	

Table 1.4: Overview of the units of the *LUCIFER* hardware.
*Limit switches and an optional reference switch

need of warming up *LUCIFER*. For that a pre-cooled *Auxiliary Cryostat (AC)* is attached vacuum-tightly to the instrument dewar and the rearmost cabinet part that contains 23 masks is exchanged. See Figure 1.5 for the exchangeable cabinet part. In Figure 1.4 the rails of the *AC* are visible and the vacuum lock is covered with a black protective cap. Compared to the other instrument units which can simply and independently translate or rotate without the risk of collision, the motors of the *MOS Unit* have to interact complexly to perform a mask exchange. This complex motion is made possible by the application of angular resolvers and encoders. Due to this complexity the control software of the *MOS Unit* is described separately in Chapter 6.2.

1.3.3 The Electronics

The main tasks of the *LUCIFER* electronics are to drive the instrument mechanics, monitor the instrument parameters and control the detector. The electronics responsible for the opto-mechanics and the detector control electronics are placed in two boxes at the end of the instrument dewar. The remaining main electronics is mounted in a separate rack beneath the dewar. This rack is divided in instrumentation and auxiliary electronics (see Figure 1.4). All connections between the electronics rack and the instrument have to pass the cable wrap at the de-rotator.

Several electronic systems work together to control the motion of the independent instrument units. There is the motion control electronics developed by the *MPIA* which is used to drive the stepper motors and to control the encoders and electromagnetic motion lock systems. Another *MPIA* development by Michael Lehmitz is the *High Resolution Analog Measurement and Output Board (HIRAMO)*. It is used to check the instrument cabling, to control the cold clamps, to provide 16 additional position switch connectors and to adjust the tilt of the gratings. The *MPE* developed the *Switch Box* electronics, which is used to access all switches of the *MOS Unit*[9], to evaluate the strain gauges and to measure the orientation.

Other electronic systems are responsible for monitoring and controlling important instrument parameters. The temperature controller and the temperature monitor are responsible for regulating the detector and fanout board temperature and monitoring the temperatures inside of *LUCIFER*, respectively. The pressure monitor supervises the vacuum of the dewar. Another task of the electronics is to control the vacuum pumps and the cold heads that are used to keep *LUCIFER* cold. The cold heads of the closed cycle cooler need to run asynchronously to minimise vibration transfer to the telescope. The thermal energy is transported from the cold heads through helium flex lines down to the compressors. Beside the electronics used to keep *LUCIFER* cold, special heater electronics exists to control the heat up process of the dewar. The spectral lamps of the *Calibration Unit* can be remotely controlled by additional electronics. All electronic boxes are equipped with their own power supply, are thermally insulated and use chilled fluids to regulate the temperature. This regulation is done by the rack cooling control electronics that measures the electronics temperature and controls the flow of the coolant.

The detector control electronics is described separately on page 16. The detector data is transferred directly via exclusively reserved fibres into the control room. To access the other electronic systems from the main instrument server in the control room, the serial data is transcribed by a port server into a *TCP/IP* communication. Additionally this port server has the ability of short time data buffering that reduces the risk of data loss.

[9]Currently 114 of the optional 256 switches are used.

Type	*SUN* Fire V880 Server
Processors	4× 1.2 GHz Ultra*SPARC* III (8 MB L2-Cache) 1× system service processor
Main Memory	32 GB *ECC RAM*
Hard Disk	6× 73 GB *HDD* over internal *FC-AL* controller
Network	1 Gb *Ethernet* and 10/100 *BASE-T Ethernet*
Input/Output	9.6 GB/sec. system bus with 9 full size, hot-swap *PCI* slots
Frame Buffer	*SUN PGX 64 8/24-bit*
Operation System	*SUN Solaris 10*
Miscellaneous	3× redundant power supplies (1,500 W each) external *FC-AL RAID* array

Table 1.5: Overview of the *LUCIFER* control computer.

1.3.4 The Control Computer

The *LUCIFER* instrument electronics will be controlled by a single server, that is placed in an air-conditioned computer room. The detector data is transferred from the custom-made readout electronics via fibre based transfer channels and fed into the server hardware with two *edt PCI CD-60* interface boards. Each interface board provides a high speed 16-bit parallel *DMA* channel with a data transfer rate of 60 MB/sec. In the final stage of extension with both *LUCIFER* instruments mounted, the server will be equipped with 4 interface boards. For historical reasons of the readout software development process and the demands of high data *IO* rates, a *SUN* computer hardware is needed. The main control server of the *LUCIFER* project is a *SUN Fire V880* (see Table 1.5). The *SUN Fire V880* uses the *Solaris 10 OS* of *SUN*. Therefore all software packages have to be developed for and tested on this target platform. During *LUCIFER* observation runs this server is accessed via terminal applications running on workstations that are located in the control room. This way of interaction allows remote observing, too.

1.3.5 The Detectors

Each *LUCIFER* instrument is equipped with a *HgCdTe Astronomical Wide Area Infrared Imager-2 (HAWAII-2)* detector manufactured by *Rockwell Scientific*. In contrast to common *CCD* detectors silicon with a bandgap of ≈ 1.1 eV cannot be used to detect *NIR* photons. The *HAWAII-2* hybrid detector consists of a thin *IR*-sensitive HgCdTe[10]-layer on top of a multiplexed readout array which gives direct access to each of the 2048×2048 pixels. The detector is subdivided into 32 data channels and 4 reference channels. These reference channels can be used to determine the readout characteristics of the quadrants which contain 8 data channels each. To operate the detector custom-made control electronics is used, developed by the *MPIA*. This electronics consists of a pattern generator which is needed to clock through the pixels accordingly. The pixels of the readout array selected at a time are directly connected to the individual data channels. The fanout board on which the detector is mounted pre-amplifies the output signals. Each of the 36 channels is connected to an individual *Analogue/Digital Converter (AD-C)*. By this separate signal conversion the data on the detector is preserved. Thus multiple readout modes can be applied to the detector in comparison to conventional *CCD* array with destructive and consecutive signal processing. Effects like the bleeding of saturated pixels along the readout column as

[10] Mercury cadmium telluride has a bandgap of 0.25 eV – 0.5 eV and a corresponding cut-off wavelength of 2.4 μm – 4.8 μm. The bandgap is adjusted by the doping process.

found by *CCD* detectors do not exists. Although other artifacts like residual ghost images of previously observed bright stars can be encountered. The detectors of the *LUCIFER* project and the available readout modes are described in detail in Muhlack (2006).

1.4 Infrared Astronomy

The electromagnetic radiation in the wavelength range from 1 mm (300 GHz) to 1 μm (300 THz) is called *infrared (IR)* radiation. With wavelengths between 1 μm and 10 μm the *near-infrared (NIR)* band is located at the lower end of the *IR* spectrum intersecting with the upper end of the optical band. The remaining parts of the *IR* spectrum are the *mid-infrared (MIR)* band (10 μm – 50 μm), the *far-infrared (FIR)* band (50 μm – 0.3 mm) and the sub-millimetre band (0.3 mm – 1.0 mm). The sub-millimetre band overlaps with the radio band of the electromagnetic spectrum.

As a benefit of the long *IR* wavelengths adaptive compensation of the atmospheric distortion is possible. This allows to achieve the maximum theoretical resolution of a ground based telescope that is comparable with optical resolutions only reached in space missions. With increasing computational power as well as faster and more accurate opto-mechanical actuators *Adaptive Optics (AO)* will also become available for optical observations.

1.4.1 The History of IR Astronomy

The *IR* radiation was first discovered by Sir John Herschel in 1800 when dispersing solar light through a prism onto several thermometers. The detection of radiation invisible to the human eyes led to many experiments observing the Moon, the planets and bright stars. In the 1920's the first systematic *NIR* observations have been carried out by Seth Barnes Nicholson and Edison Pettit. Their work was continued by Gerad Peter Kuiper and Harold Lester Johnson in the 1950's. With the upcoming techniques to fabricate artificial semiconductors the doping process was used to produce very sensitive bolometers. In 1961 Frank James Low presented a gallium-doped germanium detector that was cooled with liquid helium (4 K) and allowed to carry out low-noise *IR* observations at higher wavelengths. By observing at different altitudes on Tenerife 1856 Charles Piazzi Smyth already found out that the quality of *IR* observations is strongly connected to atmospheric effects. Therefore the first efficient *FIR* observations with gallium doped bolometers were done during balloon, airborne or satellite experiments.

1.4.2 IR Radiation Mechanisms

Emissions from rotational transitions are the most prominent line emission mechanisms in the sub-millimetre band. In the *MIR* and *NIR* the most common line emissions are generated by rotational-vibrational transitions in molecules and by recombination. These recombination lines are dominated by ionised hydrogen and helium atoms interacting with free electrons. E.g., the low-energy *Paschen*, *Bracket* and *Pfund* series of transitions of the hydrogen atom generate emission in the *NIR* wavelength band. Additionally forbidden line emissions like $FeII$ can be observed.

One of the most common *IR* continuum radiation mechanisms in astrophysical objects is the *black-body radiation*. Following *Planck's law* every object radiates in the *IR* with an intensity peak between 1 μm to 1 mm for temperatures of 2,900 K to 2.9 K, respectively. These sources of *black-body radiation* are relatively cold and can be dust, planetary objects and stars. *Bremsstrahlung* and synchrotron radiation that is produced in thermal and relativistic plasmas are other continuum radiation mechanisms in the *IR*. This kind of

Name	Centre Wavelength(μm)	Width (*FWHM*)(μm)	Sky Transparency	Sky Brightness
J	1.25	0.3	high	low at night
H	1.65	0.35	high	very low
K	2.2	0.4	high	very low
L	3.5	1.0	<3.5 μm fair >3.5 μm high	low
M	4.8	0.6	low	high
N	10.6	5.0	8-9 μm/10-12 μm fair others: low	very high
Q	21	11.0	very low	very high

Table 1.6: *IR*-windows in the Earth's atmosphere. Taken in parts from McLean (1997)

radiation is generated in the environment of star forming regions as well as in *Active Galactic Nucleuss (AGNs)* of *Quasars* or *Seyfert galaxies*.

Ground based observations in the *IR* are negatively affected by atmospheric effects. In comparison to the optical band with constantly high transmission rates the atmosphere is transparent only in few wavelength windows in the *IR* band (see Table 1.6). These windows of high transmission rates are separated by opaque bands with high spectral absorption by atmospheric gases like water vapour, oxygen and carbon dioxide. The strength of these *telluric lines* of atmospheric absorption is not necessarily linearly correlated with the *air-mass*. At wavelength above 2.3 μm the *telluric features* appear additionally as emission lines in the spectrum. High resolution spectra in the *NIR* wavelength range of the atmosphere above *Kitt Peak Observatory* are presented in Appendix F. The broad emission bands of rotational-vibrational transitions of hydroxyl (OH) molecules dominate the non-thermal atmospheric radiation. Excited by *UV* photons from the Sun a thin layer of OH molecules in the upper atmosphere[11] emits *IR* photons. The intensity of these emissions varies shortly within time[12]. At longer wavelengths this kind of emission is excelled by the thermal *black-body radiation* of the atmosphere itself. See McLean (1997). Especially the water vapour is strongly connected to climatic effects and varies highly with time. Therefore *IR* observations are done at dry places and at high altitudes preferably outside of the Earth's atmosphere. For ground based observations one needs to reduce the negative impact of atmospheric influence with adequate and close in time calibration data of the sky.

1.4.3 Science in the IR-Regime

Scientific observations in the *IR* regime are very effective for sources embedded in optical dense medium. In cases of dust covered objects extinction prevents direct visual observations. The emission of dense molecular clouds that are heated by stars and *AGNs* are other bright *IR* radiation sources. The following science cases are representative for the widespread use of *IR* observations.

Search for Exoplanets

The direct imaging of *exoplanets* in the optical is complicated by the necessary high contrast ratio between the stellar component and the planet. For Jupiter-like *exoplanets*

[11] At an altitude of approximately 90 km

[12] Intensity changes by factor 2 or higher in 30 minutes

this ratio is 10^9 and is increasing for Earth-like objects to 10^{10}. Future space missions that use e.g., *Non Redundant Masking (NRM)* or an external occulter to screen the light of the central stellar component could theoretically reach the required limits (see Levine et al., 2009; Sivaramakrishnan et al., 2009). A first optical image of a possible *exoplanet* is presented in Kalas et al. (2008) showing an object orbiting at 119 *AU* around *Formalhaut*. In comparison to optical imaging of *exoplanets* ground based *NIR* observations could make use of *AO* systems to increase the spatial resolution. In particular young *exoplanets* radiate thermally by converting their potential energy. This effect in addition to the thermal radiation characteristics of the central star reduces the contrast ratio for Jupiter-like *exoplanets* to 10^4. First direct *NIR* images of a possible *exoplanet* have been presented in Neuhäuser et al. (2007).

The Galactic Centre

In the dense and luminous star cluster at the centre of the Milky Way a very compact radio source called *Sagitarius A** is embedded. *Very Long Baseline Interferometry (VLBI)* observations of this source have confined the size of this component to less than 10 light minutes. To determine the nature of the centre of our Galaxy the contained mass is important. Therefore diffraction limited observations in the *NIR* wavelength band are used. These observations allow to see through the concealing dust and provide high spatial resolution. By tracing the *Keplerian orbits* of the stars around *Sagitarius A** the mass was determined to $3.6 \times 10^6\,\mathrm{M}_\odot$. Due to the orbit of the innermost stars this mass needs to be concentrated within $\approx 10 - 20$ light hours. Even with a radius 1,000 times higher than the event horizon of a black hole with a comparable mass[13] these observations give the best evidence for the existence of a super massive black hole in the centre of our Galaxy (Genzel and Karas, 2007).

ULIRGs / Dusty Starbursts

Another interesting group of objects are the *ultra-luminous infrared galaxies (ULIRGs)*. Dasyra et al. (2006) observed the stellar kinematics of local *ULIRGs* in the *NIR*. The galaxies of their sample have been selected to show a remaining single merged nucleus. The results of their observations demonstrate that most of the galaxies in their sample arise from major mergers between gas-rich spiral galaxies. These merging processes form elliptical galaxies with random stellar motions. Triggered by the merging process a starburst event takes place in the host galaxies and generates *IR* radiation. Combined with the emission from an *AGN IR* luminosities greater than $10^{12}\,\mathrm{L}_\odot$ are reached.

The Early Universe

The bright *UV* and optical transition lines of highly red-shifted objects can only be observed in the *IR* wavelength band. Therefore the investigation of the early universe e.g., highly red-shifted extragalactic objects is done with *NIR* observations.

[13]The *Schwarzschild radius* of a $3.6 \times 10^6\,\mathrm{M}_\odot$ object is $\approx 1 \times 10^7$km ≈ 33 light seconds.

CHAPTER 2

Software Development

This chapter describes the software development process of the LUCIFER project. The specified general aspects can be useful as a base for managing other instrumentational software projects by adopting the described process and model. The chapter starts with a short overview of modern software development techniques and includes a description of the applied process. In addition an integrated environment is presented which assists the developers in performing the different tasks. Finally the benefits of the Object Oriented (OO) programming approach are depicted which lead to the successful realisation of the LUCIFER Control Software Package (LCSP).

One of the big issues when developing a control software for a complex instrument is to stay in control of the software itself. On one hand, the more complex software gets the harder it is to keep an overview of the different parts of the software and their intended functionalities. On the other hand a very specific knowledge of the source code is essential to manage software change requests, to find errors and to remove them. Theoretical knowledge of software development to choose an appropriate process (see Section 2.2), the used development environment (see Section 2.3) and the software design (see Section 3.2) are equal parts in handling the software development requirements.

The *Divide and Conquer* approach, used by the Romans to control a big empire, is one of the most used solutions to reduce complexity in computer science. It can be found in small algorithms as well as in big software packages. A design that divides the software in small manageable chunks allows isolated handling of these parts. Identification of failures can be simplified by strong associations between the tasks and the corresponding software parts. An abstract high level design provides a simple overview that is easy to understand, even without knowing in detail how a task is solved. This allows to concentrate on only one complex part at a time while masking out the issues of the other parts of the software. By defining interfaces the complexity of subsystems is hidden and changes or even their complete replacement does not affect the software that uses these subsystems. The detailed design of the *LUCIFER Control Software Package* is presented in Section 3.2.

2.1 Software Development Models

Going back in time to the beginning of the computerised era, a software developer was a craftsman. Based on the diversity of computer hardware, software was a product just built to do one job on one hardware. Most people that had been responsible for software, came from the electronics engineering companies that built the hardware. Some people just found their way into business by using computers. Due to the fast evolving computer sector, software was not able to deal with the increasing complexity of hardware. This finally led to what is known as the software crisis. One of the major problems was the lack of documentation. In addition there was no uniform way in solving problems. Pieces of code had just been copied and pasted instead of writing reusable code. Only the one that wrote the code was able to debug, manipulate and change it. Software quality was so bad in the late 1960's, starting 1970's that companies started adopting development models which had proven to work. Attempts to describe the development process in a formal way and the starting research in the field of software development models have been the first steps towards professional software engineering. Shaw and Garlan (1996) describe this development process starting with a talented amateur that uses a lot of time and resources. The next evolution step is the skilled and trained craftsman who has knowledge of established procedures. The last evolution step leads towards the educated professional that makes use of analysis and theory to guarantee a certain quality standard. The comparison of software development with other engineering professions indicates that the last transition from a handcraft to an engineering approach has not been finished yet. The wide range of proven solutions other engineering domains can rely on is still missing.

With modern software development techniques a wide range of challenges is handled. The keywords often mentioned in the context of software are quality, reliability, usability, functionality and cost effectiveness. These keywords are unclear, ambiguous and have no fixed meaning. E.g., a manager defines the quality of software by its revenue and neglects the usability of documentation. On the other hand the user appreciates a stable, usable and well documented software and uses these criteria as quality benchmarking items. The developers tend to neglect cost effectiveness and concentrate on reusability or code-style. This example demonstrates how different the valuation of these words and how inexact their meaning is. To guarantee a certain comprehensible level of quality in the end, the whole process must be standardised and methods to measure quality need to be formulated. These standardisation efforts lead to an approach where detailed planning come to the fore instead of cobbled solutions. E.g., ISO 9000/ISO 9001 can be used to apply a certified quality management, or ISO 15504 defines a standard to improve the software development process itself. Nowadays people still tend to rate software by its result and the represented functionality instead of caring about the internals. These people neglect that tidy internal structures lead to better changeability until they find out that a small structural software change requires an entire re-engineering. "It isn't enough for a computer to produce the correct outcome. Other software qualities are also important and can be achieved by careful structuring." (see Shaw and Clements, 2006)

There are several development models that can be used to structure software creation. The *Build and Fix Model* is just the unstructured approach that has been used by every programmer. It starts with an idea and simply implements the developers own requirements. Winston Royce introduced in 1970 the first formal description of the *Waterfall Model* (see Royce, 1970). In this model the progress flow follows the phases from top to bottom, comparable to water flowing down a waterfall. Even though the *Waterfall Model* was published by the author as a process that is not feasible and cost efficient for larger projects, this model has widely become accepted. The *Lifecycle Model* (see Pomberger and Blaschek, 1993) and the *V-Model* (see Bröhl and Dröschel, 1993; `http://www.v-modell.iabg.de`, 2009) are other common approaches to structure and improve the development process. The *V-Model* is used by the German federal administration to manage software development. It opposes the defining/planning phases to the phases used to test and validate the product. Several specialised development models exist that try to address the different aspects that arise during the software development process. For example the *Spiral Model* presented in Boehm (1988) takes special care of the risks and costs of a software project while the *Incremental Build Model* by Balzert (1998) covers the aspect of the late start of coding. Further information on development models can be found in the literature e.g., Zuser et al. (2004). The previously mentioned development models are divided into phases. By breaking down the software development into manageable sections the complexity in understanding the whole process is reduced. The following phases can be treated together as composed stages of project planning, creation and verification.

- *Analysis* and *Design Phase*
- *Coding* and *Implementation Phase*
- *Testing* and *Verification Phase*

As these phases are part of the software development model of the *LCSP* the following subsections contain a detailed description.

2.1.1 The Analysis and Design Phase

After gathering the project related requirements and composing the specifications of the project, the individual tasks are analysed. This is done by generating and examining concretised use cases. During the *analysis phase* the use cases are refined by breaking down their activities. Another important part of the *analysis phase* is to carry out a feasibility study and to evaluate whether other, already existing products could be included. The result of the *analysis phase* is a hardware independent description of the functionalities. Any kind of information on the technical realisation is omitted.

As a result of the inspection of the requirements, a high level design is created. In successive steps of the *design phase*, this coarse design is more and more refined until a design exists on module level. This can be used as a blue print for the *coding phase*. The final design should contain all interface specifications that are necessary to formulate the module tests. Part of the interface specification is the definition of user interactions, which may lead to a preliminary *Graphical User Interface (GUI)* without functionality. This specification should also state clear boundaries between the modules and a stringent module interaction policy. This easily allows to distribute the development of the different modules between independent working groups.

One possibility to formalise the results of the *analysis* and *design phase* especially of an *Object Oriented (OO)* software solution is given by the *Unified Modelling Language (UML)*. See Section 2.4 for more details on the *OO* programming paradigm. The best industrial engineering practices to model and describe a complex system are combined in *UML*. This unification is closely connected to the work of Jim Rumbaugh, Grady Booch and Ivar Jacobson called the *Three Amigos*. They started with individual methods using their own syntax and semantics. As their methodology was evolving, they recognised the similarities of their approaches and started to unify syntax, semantics and procedures. Initially *UML* was intended to be used for *OO* software analysis and design. Nowadays *UML* is very flexible, scalable and capable of describing any kind of system, not necessarily limited to software systems. In September 1997, *UML* was standardised in its first version by the *Object Management Group (OMG)* *Object Analysis and Design Task Force*. The *OMG* is also responsible for the *CORBA*[1] standard.

Since then the modelling approach of *UML* has been evolving and has become an accepted industrial standard. *UML* comprises a wide range of diagrams to specify, visualise and document any kind of problem and its solution. This unified language consists of clearly

[1] *Common Object Request Broker Architecture (CORBA)* specifies a middleware architecture that allows distributed server-client interaction in a heterogeneous environment.

formulated semantical elements that structure the problem solving process. One of the big advantages of *UML* is to provide specialised views with individual capabilities to assist the problem analysing, solving and documenting process. The view of the user is covered by the *use case diagram* type, while the environment is examined with the *deployment diagram* type. To handle the implementation issues the *component diagram* exists. For a more detailed structural view of a problem *class diagrams* can be utilised. Behaviour can be modelled with *sequence diagrams*, *collaboration diagrams* and *activity diagrams*. More information on the different diagram types can be found in Alhir (1998) and Pilone and Pitman (2005).

2.1.2 The Coding and Implementation Phase

The *coding phase* is used to transfer the design of the previous phase into an executable software solution. During the *coding phase* weak parts of the software design are revealed and may induce a re-analysis of the design. Another important task during the *coding phase* is testing. The strong interaction between the tasks of the *coding phase* and the *testing phase* makes a clear separation difficult. Coding, module design and module testing rely tightly on each other.

As a result of the *coding phase* the source code is created. Although the source code is a very important part of the software product, other results of the *coding phase* aren't less relevant like the documentation and the layout of the source code. It is also important to pay attention to stability, flexibility and reusability of the produced code. To assure a certain level of documentation and code style automated tools can be applied. The tools used for the *LUCIFER* project are given in Table 2.3. Another important aspect of the *coding phase* is the *change management*. It allows to manage and trace back software changes and provides unique version information for each stage of development of the sources, configurations and setups.

Several strategies exist on how to implement the design. In the *top-down approach*, the developer starts at the highest level of the software. This allows to present a *GUI* in a very early project phase. For a first test version of the software the non-existing modules need to be replaced by inoperable dummies, which satisfy just the module interface specification. In contrast the *bottom-up approach* to create the source code, starts at the bottom of the design, goes up level by level and connects functionalities until the software is ready for release. This reverse approach needs a long time prior presenting the first *GUIs*. Problems that arise during the assembly of the individual modules show up in a very late phase of the project and may delay the completion. On the other hand, errors of the *analysis* and *design phase* that manifest during coding, do not lead to a re-design and re-implementation of the higher levels as they would in a *top-down approach*. The *bottom-up* implementation strategy was used for the *LUCIFER Control Software Package (LCSP)*. Towards the end of the *LCSP* implementation, multiple engineering interfaces have been developed in order to support the engineers in assembling the *LUCIFER* hardware. This strategy enabled the engineers to test the low level software modules together with the newly produced hardware and gave additional time for the development of higher level software parts.

2.1.3 The Testing and Verification Phase

The verification and testing of software is the most important task to guarantee a certain level of quality. It is important to define the targets and procedures before testing starts. Continuous testing significantly improves the software and reduces the amount of undiscovered errors. Therefore in each development model of the previous section there is a dedicated software *testing* and *verification phase*. Software tests are done to reveal errors of the *coding phase* and to demonstrate that the software complies with the requirements. The use cases that have been driving the development process, define the functionalities of the software that have to be verified.

Test strategies can be divided into *white-box* and *black-box testing*. Which strategy is used depends on the available module descriptions. *Black-box testing* concentrates on testing the interface specification and ignores any internal technical details. *White-box tests* use the knowledge of the internal implementation structure to define the test cases. The module specification allows to test every logical path instead of just testing the interfaces. Therefore errors that occur during a *black-box test* are more difficult to localise in comparison to errors found with *white-box tests*. Additionally one also distinguishes between *structural* and *functional tests*. *Structural tests* are valuable to examine whether the software was assembled correctly and modules interact as intended. The wide range of *structural testing* techniques contains tests for system performance (e.g., *stress tests*), tests for system execution, system recovery, operation, compliance and system security. In *functional tests* the focus is on the system specification. *Functional tests* are tests of the compliance with the requirements, correct error handling, computer-user interaction and data exchange between subsystems. *regression testing* is an important *functional test* during the *coding phase*. These tests ensure that changes made to one subsystem do not unintentionally interfere with other subsystems. Automated tools can be used to implement *Regression testing* into the software development process (see Section 2.3). See Perry (2006) for practical applications of the different test methods.

Simulators are often used to test complete software systems. A simulator was built to test the software of the *LUCIFER* project. This virtual *LUCIFER* instrument is described in Chapter 8. The complexity of a simulator depends on the completeness of the software system required to execute the *system tests*. This kind of simulators basically creates and evaluates data input and output. Most often simulators are limited to only one kind of input, e.g., only keyboard or mouse interaction is considered. In cases with a very high reliability level of the system, e.g., nuclear power plant or aircraft industry, the complexity of the simulator can easily reach the level of the tested system itself.

For any kind of testing it is important to keep in mind: “Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.” (Dijkstra, 1972)

2.1.4 Other Important Development Tasks

Beside an adequate development model several other tasks need to be taken into account in order to create a successful software solution with a specific guaranteed level of quality. Every software project needs a management to administrate and supervise these tasks. Beside the scheduling of tasks the management has to ensure that the developers follow the predefined proceedings. In order to be able to define the development process a manager needs to have theoretical knowledge of quality management standards, software design and fundamental programming techniques (see Zuser et al., 2004). Missing of abstraction and breaches in the approach of solving tasks are the most common issues to deal with. The chosen development process has to be open for regular changes of the software. An appropriate documentation is fundamental to allow for later changes by different programmers. By enforcing a reusable design and encouraging the developers to generate frameworks instead of single software solutions, the quality and value of the developed software can be increased. The process of supervising the developers can be improved by using automated tools to control the quality and to measure e.g., the complexity of the source code (see Section 3.5). A wide range of automated tools can also be used to test software and to find out whether the software complies with its requirements. The tools used to create the *LUCIFER Control Software Package* are described in Section 2.3.

2.2 The *LUCIFER* Control Software Development Model

It's most important for a development process to match the size and requirements of the software project. A process needs to be flexible for adjustment to the project. An oversized process most often includes a wide range of bureaucratic workflows that minimise efficiency.

The development process of the *LUCIFER* software can not be described by a monolithical model due to numerous changes within the project realisation time. In the beginning there was no model defined and the software was developed by applying the *Build and Fix Model*. First approaches to implement the software requirements were programming language dependent and started without an *analysis* and *design phase* (see Jütte et al., 2002).

There were several major issues the *LUCIFER* software project had to deal with. First of all only a few requirements were defined in the beginning of the project. Partially this is a consequence of the uniqueness and technological challenges of the *LUCIFER* instrument. Building a functional scientific instrument without the experience of a prototype is a usual case in astronomical technological projects. Of cause there are comparable instruments, but each instrument has its own specifications. No standard software solution exists that would just need some modifications to comply with the requirements of any instrument. The requirements of the *LUCIFER Control Software Package* are discussed in detail in Section 3.1. The next problem to deal with are the very limited human resources. This demanded from each team member to work in the different roles of an analyst, designer and programmer. Zuser et al. (2004) defines such a unique project like *LCSP* as a large project and suggests to have at least 40 team members. This amount of staff is definitely adequate for space missions, for a ground based project like *LUCIFER* a few team members should be enough. For other pure software or data mining projects the number of required personnel could easily be larger. The *LUCIFER* software project started with 1 person and had a maximum of 4 members actively developing the software. Missing requirements and a small development team can also be found in other projects. These surrounding conditions needed to be considered to find an appropriate development process.

The final *LUCIFER* development process presented in Jütte et al. (2004a) and Jütte et al. (2004b) is an adoption of the *Unified Process*. Ivar Jacobson started 1987 the development of such an *OO* process. This development was integrated in the *Rational Process*. *Rational, Inc.* published several papers concentrating on architecture oriented software development as well as on articles describing an incremental and iterative process (see Figure 2.1). The *Unified Process* was published in Jacobson et al. (1999). A detailed analysis of the requirements is the key element of the *Unified Process*. The design of the software is based on the use cases that are a result of this requirement analysis. By incorporating the same use cases the test plan for the *verification phase* is build. This demonstrates how the requirements affect the whole development process and how the use cases connect each phase of the *Unified Process*.

In the beginning of the *LUCIFER* project the requirements have never been defined in form and content. Only a few instrument hardware specific requirements existed (see Section 3.1). Therefore these specifications needed to be gathered continuously by the software engineers. With the progress of the project new use cases came up and had to be taken into account. As a result of the incremental and iterative proceeding of the *Unified*

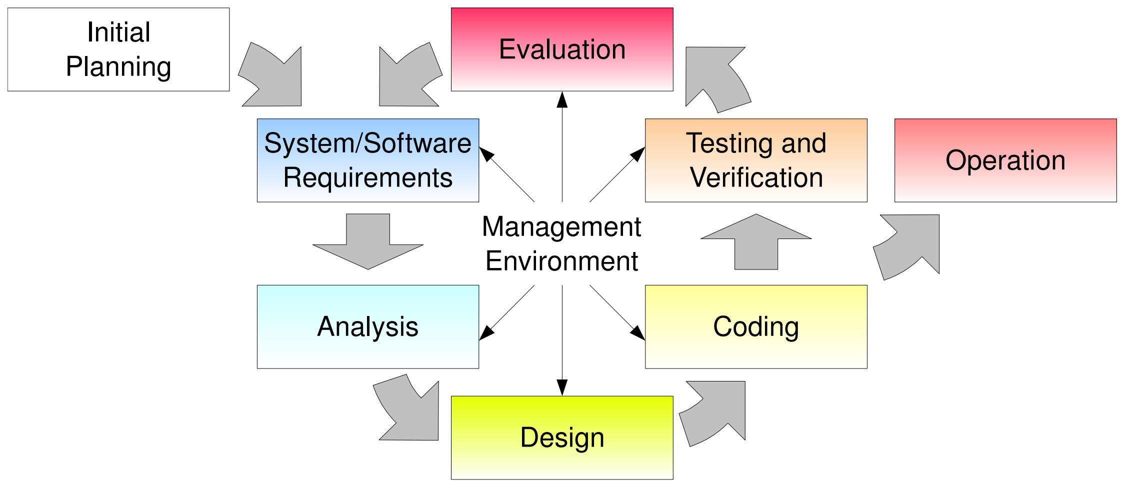


Figure 2.1: Combined incremental and iterative software development process of the *LUCIFER* project. (based on KRUCHTEN, 1999)

Process these new requirements are added in the next process cycle. With an overall design that is open for module changes the *Unified Process* does not require to step back to a previous phase as demanded in the *V-Model* or the *Waterfall Model*. With each cycle the quality of the design improves and the code of the project advances. This allows an early release of the software. Due to the repeated cycling through the development phases the *Unified Process* could be easily parallelised by concentrating on only one use case of a module. This offers scaling of the human resources by assigning one developer to one task. With an increasing number of software engineers the overall speed of the development process could be raised. This scalability of the *Unified Process* makes it ideal for instrumentational software projects where the number of the developers fluctuates.

To profit from the benefits of the *Unified Process* the global design of the software architecture needs to be very flexible for changes. Without a flexible design new requirements could result in re-design and rewriting of already existing modules. Then the advantage of the *Unified Process* in comparison to the *V-Model* or the *Waterfall Model* would be lost. The prominent role of the architecture is at the same time weakness and strength of the *Unified Process*. An insufficient design would lead to many avoidable changes, while a strong and flexible design would allow fast and parallel development.

2.3 The Integrated Development Environment

A customised *Integrated Development Environment (IDE)* for the *LUCIFER* project was created by combining several tools. Table 2.1 provides an overview of the used tools. The requirements of the *LUCIFER* readout hardware and the heterogeneous computer infrastructure demanded that the development environment runs under multiple *Operating Systems (OSs)*. *Windows*[2] and *Linux*[3] have been used for software development while *Solaris 10*[4] was the target platform (see Section 1.3.4).

To provide version management for the *analysis* and *design phase* as well as for the source code, a *Concurrent Versions System (CVS)* repository was created. *CVS* is based

[2] *OS* for personal computers, developed by *Microsoft*.

[3] Free multi-plattform and multi-user *OS* based on *UNIX* philosophy. *UNIX* was built in the early 1970s at *Bell Laboratories* to support software developers.

[4] *UNIX* *OS* of *Stanford University Network (SUN)* Microsystems, a leading manufacturer of software and computer hardware.

Tool	***Plug-In***	**Scope**
Together	(*built-in*)	– analysis and design – *UML* diagram creation – coding/refactoring – audits/metrics
Enterprise Architect	(*built-in*)	– analysis and design – *UML* conform modelling
Eclipse	(*built-in*)	– coding/refactoring – *CVS* repository access – *JavaDoc* documentation – integration of *Java* compiler/debugger – integration of *Ant* build tool
	JUnitRunner	– automated *JUnit* tests
	DBEdit	– database access – *SQL* statement evaluation
	XML viewer	– *XML* editing/verification
	GENADY's *RMI*	– registry inspection – automates skeleton and stub creation
	Checkstyle	– *SUN* conform code style inspection
	eSpell	– spellchecker
	Metrics	– generating metrics

Table 2.1: Overview of the different tools of the *LUCIFER* development environment.

on the *Revision Control System (RCS)* and provides command line tools for versioning on almost any *OS*. Another advantage of *CVS* is its easy way of integration into modern development software. See PURDY (2001) for more information on using *CVS*.

At the beginning of the project, *Java*[5] was defined as the main programming language. Therefore the tools of *SUN* for compiling, distributing, debugging and system execution have been used. The conventions of *SUN* for naming and documenting were adopted and own tools to provide special documentation tags have been created. By extending the existing annotations standards the *JavaDoc* tool was enhanced to protocol the history of a file, to provide examples on module usage, to protocol the change history and to document special requirements of a module. See Appendix D for a complete list of project relevant *JavaDoc* annotations. To automate the build process the *Ant* tool was added to the *IDE*. *Ant* is part of the *Apache*[6] project. By defining the build process in an independent configuration file a repeatable build and software distribution procedure was achieved. In Appendix C the *Ant* build file of the *LCSP* is presented. Due to the broad acceptance and industrial usage of the *Ant* tool, many development solutions provide an interface. *Ant* is comprehensively described in TILLY AND BURKE (2002).

To assist the developers in executing automated unit tests, *JUnit* provides a *Java* based framework. *JUnit* allows to bundle tests together and to protocol the test results in a compact way. By automatically setting up a test environment and cleaning up at the end, no user interaction is required. To develop tests with *JUnit*, the expected results of

[5] *OO*, hardware independent programming language developed by *SUN* Microsystems.

[6] *Apache* is an *Open Source* web server software package that is maintained by one of the biggest *Open Source* communities and includes a wide range of software frameworks.

method calls need to be specified. This enables the testing environment to compare results and determine variations. *Ant* provides special tasks for unit testing with *JUnit*, which allows to include the *regression testing* into the build process of a software package. Beck (2005) and Link (2003) provide a short introduction on how to use *JUnit*.

Especially when developing distributed applications, it is important to have tools to observe the running distributed services and execute simple commands. The services of the *LUCIFER* project are based on *Remote Method Invocation (RMI)* of *Java*. All services are managed by a central registry. Therefore the registry inspector of Genady Beryozkin (`www.genady.net`) was included into the development environment. More information on the distributed service architecture of the *LUCIFER* project can be found in Section 4.1

Nowadays a development environment is more that a simple editor. Programmers in the 1970s have got used to simple line oriented editors. In the 1980s page oriented editors came to the fore. Editors like *Vi*[7] or *Emacs*[8] have been the standard software to develop applications for a long time. Over the years these page oriented editors have been improved and provided syntax highlighting, code completion and compiler integration. The *IDEs* of today have the same capabilities of syntax highlighting and code completion, too. Compiler, linker and debugger functionalities are also integrated.

Modern *IDEs* are characterised by their possibilities to attend the whole development process. They provide a user friendly way to trace and debug software, to manage resources like software documentation and to cooperate with other team members. An ideal environment can be fully adjusted to the specific needs of each phase of the development process. This includes changing the graphical layout to assist analysts, designers or developers best. It is important to have efficient tools to speed-up the development process, e.g., to search for pieces of code, references and occurrences, even in different files or to re-factor software structures. Current *IDEs* are able to transform a *UML* design into source code and vice versa. Generated code contains only empty structures that need to be filled. These skeletons save time that could be better spent on more challenging tasks. In addition to this build-in multitude of functionalities flexible interfaces allow to integrate other software solutions, like version management or build tools.

The *LUCIFER* project started by using *Together* of *TogetherSoft, Inc.* which was later sold to *Borland Software Cooperation*[9] an American company founded by Philippe Kahn, a former French math teacher. *Borland Software Cooperation* is known for the *Turbo Pascal* development environment. *Together* is a graphical analysis and design tool, based on the *UML* notation. Although *Together* is intended to support the first two phases of the *Unified Development Process*, it provides an extensive coding environment with tools for refactoring, source code auditing and metric analysis. The metrics of the *LUCIFER* project are discussed in Section 3.5. An outstanding feature of *Together* compared to its competitors was the early availability of generating *class diagrams* and *activity diagrams* from existing source code by maintaining the *UML* notation standard. Another benefit of *Together* is its platform independent realisation with the *Java* programming language and therefore the possibility to run on the project specific *OSs*.

As the *LUCIFER* project evolved, the project migrated to the *Eclipse* development environment. *Eclipse* is a software development platform that consists of a conventional *IDE* combined with a very flexible *plug-in* system. Table 2.1 lists all used *Eclipse plug-ins*. Even though it is written in *Java Eclipse* supports the realisation of software projects in

[7] *Vi* (visual), a page oriented text editor developed by Bill Joy in 1976. *Vi* is one of the most common command line editors in the *UNIX* world.

[8] *Emacs* (editor macros) started 1976 as a collection of macros at the *Massachusetts Institute of Technology (MIT)* and was reimplemented and improved by Richard Stallman in 1984.

[9] The name *Borland* was inspired by the American astronaut Frank Borman.

C, *C++*, *Python*, *Perl*, *Cobol* and many more. It can be even used to manage database servers, to create web pages or to write LaTeX files. *Eclipse* started as a project of the Canadian *IBM*[10]. Today *Eclipse* is an *Open Source* project managed by the *Eclipse Foundation*. Beside the extensions, already mentioned in this section, a *plug-in* to control the source code quality and to follow the *SUN Java* coding standards was added. This *Checkstyle plug-in* provides a wide range of configurable rules to monitor during coding, starting with simple code indention and formatting and ending at possible coding problems like missing variable calls. Breaches of the defined coding standard are highlighted automatically. The *LUCIFER* project is an international project. Thus code and documentation have to be written in English. To fulfil this requirement adequately the *eSpell plug-in* integrates a simple spell checker. To improve the access to the databases used by the control software, the *DBEdit plug-in* was added to the *Eclipse IDE*. This *plug-in* contains a viewer to browse the data and tools to evaluate *SQL*[11] statements. Because the *XML*[12] data format is extensively used for all persistent configuration purposes within the project, the *XML Viewer plug-in* is applied. Finally, to observe the metrics of the *LUCIFER Control Software Package* the *Metrics* extension was integrated in the *IDE*. This *plug-in* allows the evaluation of numerical observable parameters of the source code on the fly. By monitoring parameters like the comment ratio or line numbers per method, the quality of the produced software can be increased. In Section 3.5 some basic metrics of the *LUCIFER* project are discussed.

2.4 Object Oriented Software

The *OO* programming paradigm is an approach to deal with the increasing complexity of software projects. In a first approach to structure software functionalities where collected in libraries allowing to spread the source code across different files. The purpose of the *OO* paradigm is to map the real world in the software. This is done by bundling object representing data (attributes) with object manipulating functions (methods). Due to the fact that humans are accustomed to categorise objects in their daily life, this bundling is a user-friendly way of modelling and representing problems. An object should not be confused with a simple data structure. It is characterised by the possibility to manipulate its data. Furthermore objects can encapsulate their data by using methods and by hiding non-public functionalities. Therefore the visibility of methods and attributes can be defined in order to control their accessibility in the project. A class defines the structure and the interfaces of an object on the other hand an object is a real instance, containing own data. E.g., person is a class whereas the reader of this thesis is an instance of person and for this reason an object. By providing the ability of abstraction and inheritance the complexity of the software design can be significantly reduced. A functionality exists only once and therefore typical copy and paste errors can be prevented. *OO* languages also use *polymorphism* to access inherited objects as objects of their *super-class*. Inheritance allows to overwrite the methods of a *super-class*. *OO* software can be reused easily, because all necessary data and methods are bundled together. If a person class is needed in another software it can just be imported due to the small number of dependencies with other

[10] *International Business Machines (IBM)* Corporation is one of the world leading hardware, software and service providers in the sector of information technologies.

[11] *Structured Query Language (SQL)* is a standardised language to define, query and manipulate data of relational database systems.

[12] *Extensible Markup Language (XML)* is a language to structure data by utilising user-defined markup elements.

classes. For that reason the person class should be designed and built to exist alone and contains all necessary information and methods.

Most of the mentioned benefits of the *OO* approach already existed before *OO* languages appeared. In contrast to other approaches the *OO* programming paradigm induces a deeper structural analysis of the problem domain and therefore produces a better design and source quality. Nevertheless it is still important to obey fundamental programming guidelines like information hiding, separation of concerns and reusability by modularisation. This means that the use of an *OO* language does not automatically guarantee a successful software project.

2.4.1 History of the Object Oriented Paradigm

Today, the *OO* paradigm is a common programming technique. Back in the early 1960s, when *Simula* was introduced by Ole-Johan Dahl, Bjørn Myhrhaug and Kristen Nygaard, it was a revolutionary approach on writing software. *Simula* was developed in order to generate simulation programs. In Dahl et al. (1968) all central characteristics and constructs of a modern *OO* language have already been mentioned. The concept of classes to bundle the attributes with methods as well as to inherit properties and functionalities was introduced. Information hiding and dynamic object creation represent other fundamental concepts that can be found in other *OO* languages. Because the instances of *Simula* classes act like coroutines, they could be used to simulate concurrency. Until the 1970s *Simula* was just used only by a small community. This changed rapidly when the *OO* paradigm was used as base of *Smalltalk*. The innovative idea of *OO* programming was transferred into other languages like *C*[13]. Stroustrup (1986) started his work on developing *C++* in 1979 by upgrading *C* with *OO* structures and functionalities. From that point on, the *OO* approach became a standard and is nowadays part of languages like *Python*[14], *C++*, *C-Sharp*[15], *BETA*[16] and *Java*.

2.4.2 The Java Programming Language

James Gosling, working at *SUN* Microsystems, started the development of the *Java* programming language in 1991. In 1995 the first version of *Java* was published. Since 2007 *Java* is *Open Source*. One of the main intentions of *Java* is to provide a programming language independent of the hardware platform and the *OS*, that allows developers to write software once and run it anywhere. Therefore the source code is not translated into a native hardware dependent machine code. Instead the compiler generates a hardware independent byte code for a virtual machine. These virtual machines are provided free of charge by *SUN* for many different hardware systems. Certainly interpreted code cannot be as fast as highly optimised hardware dependent code. But a highly optimised virtual machine is as fast as any conventional compiler that generates hardware code. The optimisation of the virtual machines is achieved by implementing different techniques, like just-in-time compilation of byte code to hardware code and dynamic recompilation based

[13] *C* is a programming language, developed in 1972 by Dennis Ritchie at the *Bell Telephone Laboratories*. It was initially written to develop the *UNIX* *OS*.

[14] *Python* is a high level programming language developed in the 1980s by Guido van Rossum at the *National Research Institute for Mathematics and Computer Science* in the Netherlands.

[15] *C-Sharp* is developed by *Microsoft* and supports multiple programming paradigms like functional, imperative and *OO* programming.

[16] *BETA* is a pure *OO* programming language originating in the Scandinavien roots of the *Simula* language (see Madsen et al., 1993).

on runtime analysis data. In comparison to other programming languages, *Java* was designed to be a true *OO* language instead of just having additional *OO* features. Although the notation style of *Java* looks similar to *C* and *C++*, there are major differences between these languages. *Java* implements natively an exception handling that provides an extreme robustness. Exception handling together with memory management prevents runtime errors from interfering with the *OS*. Errors can be traced back and handled. These functionalities combined with the ability to run an application in a so called sand-box that manages the hardware resources, makes *Java* a save language from scratch. It is designed to work in a distributed environment. Since many years *SUN* has a leading position in network technologies and therefore implemented a networking interface and distribution mechanism which is simple to use and efficient. Due to the fact that *Java* runs in a virtual machine the management of memory or multiple processors needs no special treatment. Multithreading is another feature that is natively included via simple mechanisms to create, synchronise and terminate threads. In comparison to other programming languages, where the developer has to take care of memory allocation and deallocation, *Java* provides an automated garbage collector. This garbage collector observes the state of objects and decides which memory blocks are no longer used and can be released. *Java* also includes a comprehensive collection of function libraries and interfaces. Therefore nobody needs to implement basic data structures like *Vectors* or *Hashtables* or has to write code for a well known algorithms like *Quick Sort* or string formatting methods.

The requirements of a distributed and heterogeneous system as well as the fact that the control software has no real-time requirements lead to the choice of the *Java* programming language as basis of the *LUCIFER Control Software Package (LCSP)* (see Jütte et al., 2006). It is not negligible that the time to learn this language and the development time are shorter in comparison to other programming languages. Finally the stringent way of *OO* programming helps to avoid structural errors and to improve the code quality.

Part II

Control Software

Chapter 3

The Control Software Basics

Due to the cutting edge design of the LUCIFER instrument the control software is unique and is developed as a prototype. This chapter presents the LUCIFER Control Software Package (LCSP) and starts with the description of the very specific software requirements. Next the chosen multi-tier architecture of the LCSP is outlined. Directly connected to a successfully operating distributed system is the service start mechanism. This software start framework together with the used external software packages is detailed. Finally the LCSP is analysed by discussing basic metrical numbers.

Before starting to describe the design and architecture of a software solution it is helpful to discuss the requirements. *IEEE* 830 gives a standard to record these software requirement. As a prototype, the *LUCIFER* project has to deal with changing and late defined requirements. Therefore no uniform document that contains all the requirements exists.

3.1 The Requirements

Software requirements can be coarsely classified as user or developer requirements whereas both classes may intersect. Another classification distinguishes between functional and non-functional requirements. In the following the requirements are classified even though some may also fit in other classes. The first of these classes to discuss contains the user requirements.

The users of the control software can be either observing astronomers or technical staff. For these users the *LCSP* needs to ...

- ... [U1] allow to conduct efficient observation runs. Therefore overhead times created by the software as well as software re-starts or system crashes need to be minimised.
- ... [U2] implement a quick and simple re-start procedure of the complete control software or parts of it. This is essential in case of software system failures and/or power outages.
- ... [U3] provide a simple configuration mechanism that allows to change the software execution characteristics without the need of re-compilation.
- ... [U4] control the mechanics of the instrument. The complexity of the software - hardware interaction must be hidden from the user while providing all necessary functionalities to the engineers. On one side the astronomer issues complete instrument setups while on the other side the engineer has to control individual motors to determine configuration parameters. An early version of the control software is mandatory to enable the engineers to build up the opto-mechanical units of the instrument and carry out test-runs in the laboratory.

... [U5] interact with the electronic devices that control/monitor the environment of the instrument. A part of this electronics is off-the-shelf while the rest is custom-made by the participating institutes. This custom-made electronics requires special treatment because its interfaces may change with further developments.

... [U6] protocol and preserve all instrument states and parameters. Additionally the software has to supervise the instrument and its environment and create alerts if parameters run out of their specifications.

... [U7] have a telescope interface that allows to observe the designated astronomical targets and to perform the necessary telescope motions.

Form the developers point of view the *LCSP* needs to ...

... [D1] log system status and error messages. This logging is important for tracking of system status as well as for debugging.

... [D2] allow a simple recording of communication between the software and the control electronics in order to analyse the data while debugging.

... [D3] run without instrument hardware to carry out integrated software system tests. Therefore an emulation of the hardware-software interaction is required.

... [D4] be built within a unified software build cycle. This allows several developers to work independently on the software. A quick and simple software compilation and distribution mechanism is necessary for a fast bug-fixing cycle.

... [D5] be written in an easy to use and multifunctional development environment.

The previously presented requirements describe functions of the control software. From the non-functional point of view the control software needs to ...

... [F1] run on a *SUN* computer hardware. The detector readout electronics and software are a development of the *MPIA*. For historical reasons this software runs only on *SUN* computers.

... [F2] communicate with the electronics via a port server. This port server translates the serial *RS232* data in a *TCP*/*IP* communication.

... [F3] use a database to store all messages, system states and communications. This database storage allows for later analysis with special data query tools.

... [F4] run as a distributed system. This allows to distribute the tasks and system load on services that may run on individual computer hardware. Distributed computing enables the user to run multiple *GUIs* independently from the control software.

... [F5] provide a central configuration mechanism. Especially in a distributed environment the configuration management can become confusing.

... [F6] use a unified persistence storage mechanism. All data should be stored in *XML* files.

... [F7] have an intuitive and clearly arranged *GUI* that minimises the risk of faulty operation by the user.

In the beginning of the *LUCIFER* project only the requirements [U4], [U5], [U6], [F1] and [F2] had been specified. The other requirements are the result of long discussions during a requirement gathering phase or arose during one of the software development cycles (compare Chapter 2).

Finally there are several adjectives left that are likely used to describe software without explicitly specifying their semantic. The following requirements with their project specific interpretation are some of these more general software characteristics.

efficient Efficiency for the *LCSP* means to use as much as possible of the night time for observations. Therefore tasks should be done in parallel and controlling overheads need to be minimised. The limiting parameter in controlling the instrument should be the communication speed between the electronics of the instrument and the control computer as well as the motion time of the opto-mechanical parts. Additionally efficient usage of external packages or libraries and concentration on implementing the necessary tasks reduces the manpower.

reliable To create a reliable software solution *GUI*, data structures and data manipulation methods need to be separated. A distributed system improves reliability by separating tasks on individual services. Like in modern *OSs* a faulty service can be restarted without the necessity of restarting the whole system. An appropriate exception handling mechanism should be implemented in the control software that e.g., automatically reacts on software-electronics communication errors, generates timeouts or restarts broken services. Another important part is to track the instrument state reliably for later data analysis.

documented All classes, attributes and methods need to be well documented to allow for later changes and debugging. Thereby it is important to document the complex behaviour of methods or the data structures used. Although this may seem obvious documentation is frequently neglected. A good documentation is important to have a maintainable and extendable software.

maintainable To increase maintainability simple approaches are preferred. An elaborate but compact architecture enables the developer to understand the concept of the control software. Each service should be responsible for only one problem domain. This simplifies the complexity of a single service enormously and allows fast allocation of problems.

extendable The chosen design needs to be flexible to add new functions/services. A clear structure of the source code is important to be able to add new functions. When designing software packages abstract classes and interfaces should be preferred to generate external access points.

secure In the domain of the *LCSP* security does not mean to protect the instrument from hackers and to encrypt the inter-service communication. Security means to ensure that the observing astronomer could do no harm to the instruments hardware. At all events it has to be ensured that observations are not interrupted by faulty performing services. In case of several connected *GUI* clients it has to be guaranteed that each client displays self-consistent information.

Based on these requirements the architecture of the *LCSP* was developed.

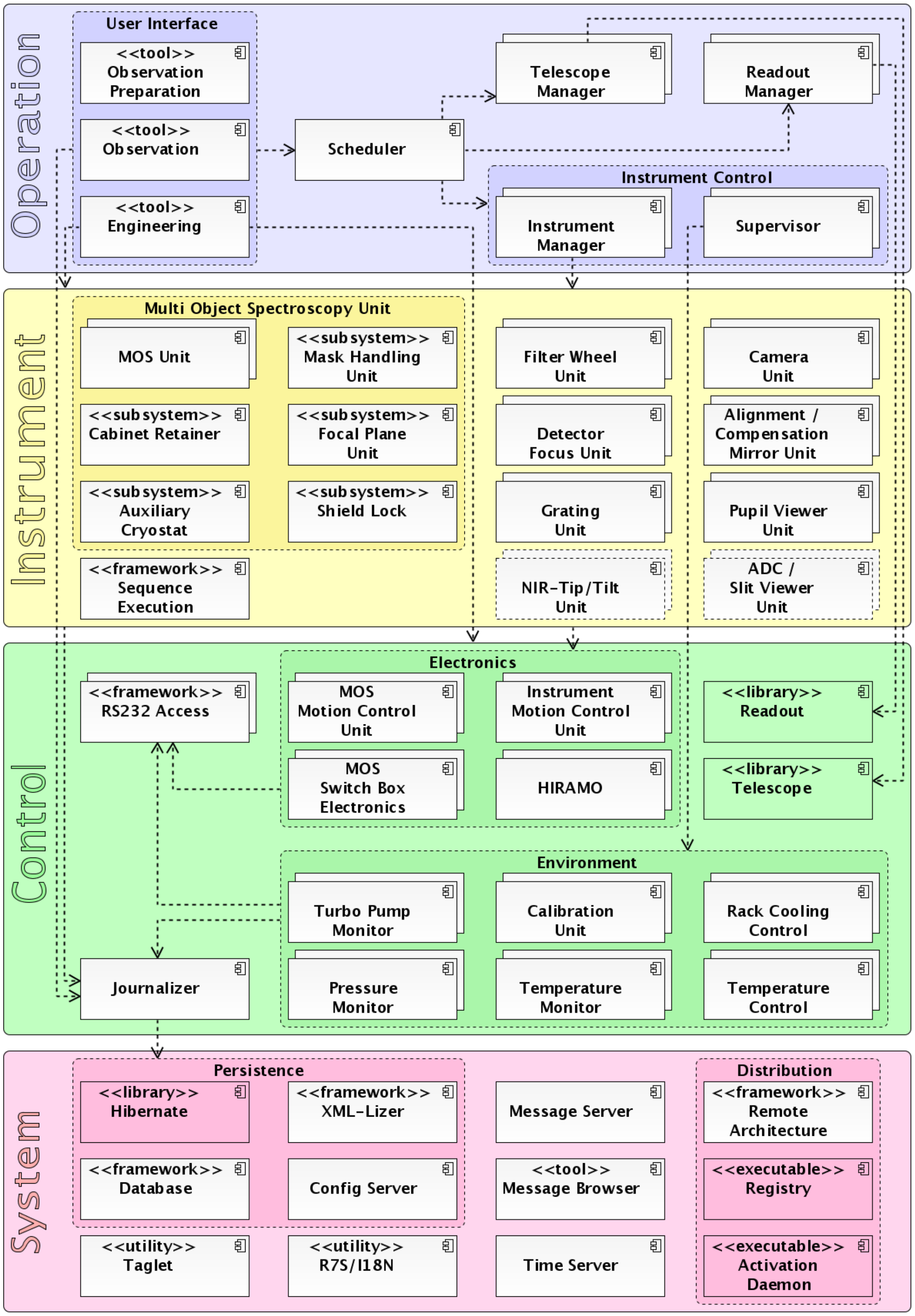


Figure 3.1: The Architecture of the *LCSP*. Boxes represent the services of the system if not specified else. Doubled boxes are used if separate services exist for each of the *LUCIFER* instruments. Translucent objects symbolise the use of external software packages. Dashed boxes are used for currently not existing services.

3.2 The Architecture of the *LCSP*

DIJKSTRA mentioned in his 1972 *Turing Award* lecture the virtues of a software developer. "We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers." In his eyes a developer should aim for a simple and comprehensible solution with clearly readable source code. Software developers tend to use large and oversized development processes and powerful frameworks even without the necessity for their use. Reasons for this miss-use are in many cases fast changing hypes of new techniques and the developers compulsion to integrate as many buzzwords as possible into the software project. In the *LUCIFER* project a simple and flexible development process was chosen (compare Section 2.2). The software architecture of the *LCSP* was designed to be as simple as possible and at the same time very flexible. In TATE AND GEHTLAND (2004) arguments for lightweight software architectures and against heavyweight enterprise solutions can be found.

Multi-tier client-server and *Service-Oriented Architectures (SOAs)* are examples of extremely effective design patterns (see SHAW AND CLEMENTS, 2006). Even though *SOAs* get out of style, individual services provide an easy way to separate tasks, to increase abstraction and reusability, to allow autonomous execution and to encapsule complex algorithms as well as data structures. A typical example of *SOAs* are modern *UNIX*-like *OSs* that gain stability with independently start-/stoppable daemons.

A multi-tier architecture allows to hide complexity within a tier as well as to provide simple and powerful functions to services of a higher tier. This proven concept was applied in the development of the high level service architecture (see Figure 3.1). A distributed system was chosen to increase scalability and interoperability of the software (compare [F4]). The individual services are grouped into four tiers, the *System Tier*, the *Control Tier*, the *Instrument Tier* and the *Operation Tier*.

The *System Tier* (see Chapter 4) contains all frameworks to run and control a distributed system, to allow central configuration management, persistent storage of data in a database and message generation in order to track the state of the system. Beside the basic services for messages and configuration management, frameworks to implement internationalisation and resource bundling as well as tools to improve source documentation and message database-mining are included in the *System Tier*. This tier is responsible to comply with the requirements [U2], [U3], [D1], [F3], [F5] and [F6].

In the *Control Tier* (see Chapter 5) the hardware-software interaction is reflected. Therefore an *RS232* communication framework is used in each of the hardware controlling services (see [D2] and [F2]). These services are grouped in environment supervising services and into services that communicate with the motion control electronics (see [U4] and [U5]). As a central logging mechanism this tier contains a service that tracks the instrument state (requirement [U6]). All hardware interacting services notify this logging service. The detector readout software *GEIRS* by CLEMENS STORZ and an *ICE* interface to the telescope (see [U7]) by JOSÉ BORELLI are also part of this tier.

The next tier is the *Instrument Tier* (see Chapter 4). This tier is responsible for hiding the complexity of the motion logics needed to set up the opto-mechanical components of the instrument. Therefore this tier is based on the usage of the hardware communication provided by the *Control Tier*. As a central part of this tier the sequence execution framework provides easy composing and execution of complex motion sequences. This tier is fundamental to an efficient usage of the *LUCIFER* instrument (see [U1]).

As the topmost tier the *Operation Tier* (see Chapter 7) contains all services needed to operate the instrument. This includes the coordination of the services of the *Instrument Tier* and the supervision of the instrument environment as well as the interaction with the external software packages of the readout and telescope control software. The *GUIs* used to grant engineering access as well as an observations scripting mechanism are another part of this tier. This tier complies with the requirements [U1] and [F7].

Due to the massive use of frameworks the development of a service could be minimised on solving its core tasks. Remote service distribution, database access, configuration management, message creation and handling, *XML* file access, *RS232* hardware communication, logical sequence execution and even parts of the *GUI* programming is done in frameworks. Thus every service is based on at least one of these frameworks. Typically a service is implemented as a multi-layer application. The lowest layer is responsible for remote service distribution and inter-service communication. The next layer uses the configuration service or *XML* files to configure the behaviour of service. This layer also integrates message generation and if needed database access. E.g., in the *Instrument Tier* the highest layer is responsible for implementing the motion logics that is achieved by using the corresponding framework. In the *Control Tier* the highest layer may use the communication framework to access the hardware. When using these multi-layer services the remote command is received by the lowest layer, processed in the middle layer (e.g., a message is generated or data is written to the database or file) and finally executed in the highest layer (e.g., a motion sequence is started or a command is send to the electronics). After the command was processed by the highest layer the results are sent back through the middle layer[1] to the lowest layer where the remote command is finished. The lowest layer does all error handling regarding remote operations while the middle layer ensures proper data exchange/logging. Therefore the highest layer can be kept simple and clear while the basic features of the independent layers can be sourced out into individual frameworks to enhance reuse and speed up the service development process.

3.3 Service Deployment and Software Start

When developing a multi-service application one of the most important tasks is the service deployment and monitoring. The *LCSP* consists of services and applications that can be started directly in the command shell. In an early phase of the project the individual services have been started manually in the *IDE*. With an increasing number of services a uniform and fast way of starting and stopping the services became indispensable. In the domain of this thesis a *Start Manager* application was developed to support service management on just one machine because a single hardware platform was anticipated. When running the distributed control software on separate computers a *Start Manager* application has to run on each of these machines. The architecture of the service start application was designed open for extensions. This allows to create a distributed version of this application by adding a slave version that runs on each of the computers and connects to a central master. This master can then be used to issue service start and stop commands on the connected slaves. Although a centralised start process could be achieved with a command shell script the functionality to individually manage and supervise service activities demands a user friendly service access and status visualisation.

The *LUCIFER Management Console (LMC)* combines both, an easy to use and fast application start/stop option as well as service management functions (see Figure 3.2).

[1]This may cause new message or database interaction.

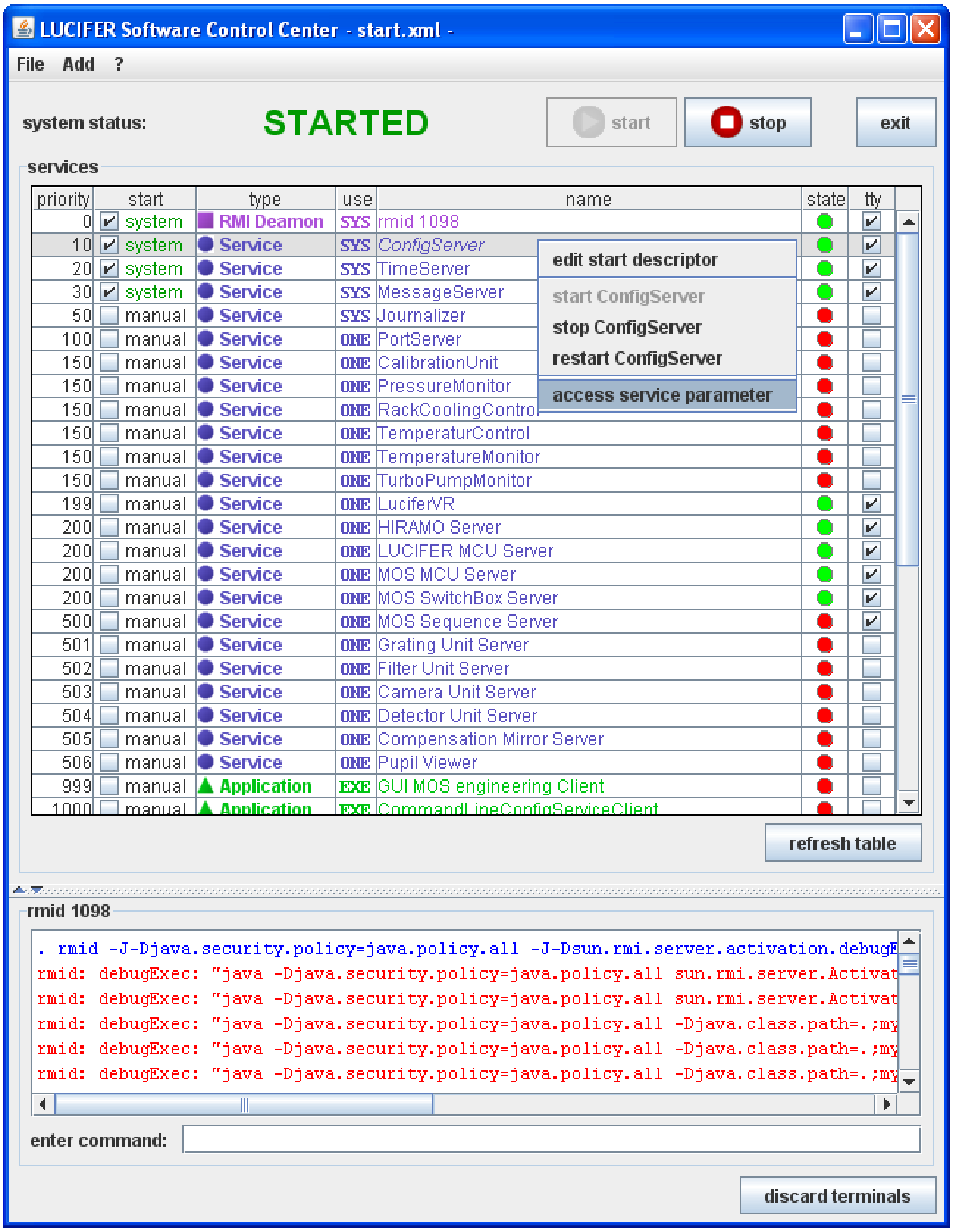


Figure 3.2: The *LMC* main window that allows to manage, configure and supervise the services. The context menu of the configuration server is opened by right clicking on the service.

The start process of the applications and services can be freely configured. Therefore a start descriptor needs to be specified for each service.

Four different types of start descriptors exist. A general descriptor to specify the start configuration of any kind of application/command. This kind of descriptor can additionally be used to call command shell scripts that e.g., clean up a temporary data directory or dump entries from the database to a file. The *RMID* start descriptor is needed to start an activation daemon and its internal registry (compare Section 4.1). To start plain *Java*

Figure 3.3: The *LMC* start screen displaying start progress of applications and services.

applications the *Java* start descriptor can be used while the start-up procedure of more specialised services is implemented by the service start descriptor. The basic capabilities of a start descriptor are passed from the **StartDescriptor** class to the specialised start descriptors that are presented in Table 3.1. In this inheritance hierarchy the **ServiceStartDescriptor** class is based on the **JavaStartDescriptor** class.

After the configurations have been defined the resulting start descriptors can be saved to an *XML* file. This configuration file can be edited manually (see Appendix A). The status of the complete software system is displayed by the *LMC*. All applications and services that are marked to automatically start with the system are executed by a single mouse click. The sequence commands are scheduled according to their priorities. Thereby the *LMC* allows parallel execution of commands to speed-up the start-up process. The progress of the system start is visualised by an animated start screen (see Figure 3.3). After the system has been started the state of the services is displayed in the *LMC*. Green buttons represent running applications and services while red buttons symbolise non-available ones (compare Figure 3.2). By starting a *RMI Activation System Daemon (RMID)* as a background daemon the control software services will stay alive even when the *LMC* is closed. When the management console is started it automatically scans for a running registry and analyses the status of the registered services. This is done to ensure a consistent software state. All running applications and services are executed in an individual console. These consoles can be used to directly manipulate the commands by entering command line text that is processed by the running applications and services. The main difference for services is that these services just register to the activation daemon and terminate. Then the activation system takes care of initialising a new virtual machine which hosts the according service (compare Section 4.1).

Beside a centralised start and stop of applications and services the *LMC* provides context sensitive access. Services can be restarted, too. The main benefit of the *LMC* is the ability to access the parameters of a service (see Figure 3.4). These parameters contain plain information, e.g., the service name, the host the service is running on, the version number or the up-time. If the *Time Service* is used the time drift is displayed. In the screen shot of Figure 3.4 no time drift is displayed because *Time Service* and client are

Class	Attribute	Scope
StartDescriptor	name	– name to be displayed in *LMC*
	command	– command string that should be executed
	commandArgument	– additional command arguments
	workingDir	– working directory the command is executed in
	autostart	– whether or not to include the command in the start procedure
	priority	– priority when starting the complete system
	usage	– usage identification
	executionTime	– required application start time
ProgramStartDescriptor		
RMIDStartDescriptor	serviceName	– name of the activation system daemon used to determine whether the system is running
JavaStartDescriptor	mainClass	– main *Java* class to execute
	programArguments	– additional program arguments passed to the executed main class
ServiceStartDescriptor	serviceName	– name of the service that is used for the service status display

Table 3.1: Overview of the available *LMC* start descriptors and their attributes.

running on the same host and therefore no drift can be measured. Next to this basic information the command line arguments that have been passed by the activation daemon to the virtual machine can be inspected. These arguments can only be changed in the service start descriptor and demand a complete stopping and starting of the service. A grace time can be specified and a terminate/restart signal can be send to a service by pressing the corresponding buttons. Then the service will wait for the specified amount of time and exit thereafter. The main functionality of the service parameter dialogue is to manage the configurations used by the service. All used configurations are automatically collected by the configuration management framework (compare Section 4.4). The functionality of this dialogue is available for every service of the *LCSP*. This is made possible by integrating these functions into the basic remote communication framework of the project (see Section 4.1). In case of modifying the configuration of a service the service itself can determine how to process and save the changed configuration values. This is especially important when distributed and centralised configuration files co-exist.

3.4 External and Utility Packages

The individual services and frameworks of the *LCSP* are described in the following chapters. To run these services external software packages are needed. This collection of external packages and programs is fundamental to a successful operation of the *LUCIFER* instrument. First of all this collection includes an *OS* that provides basic functionalities

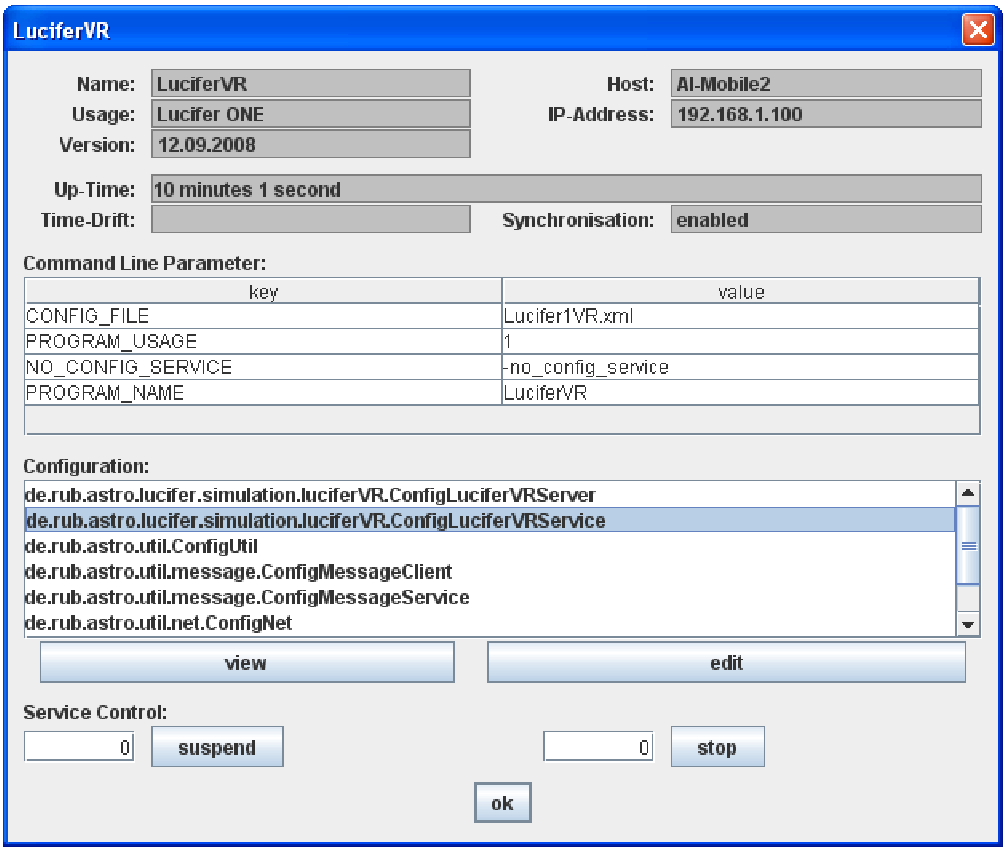


Figure 3.4: The *LMC* service parameter window that allows to view and edit service parameters.

like multi-tasking, file/network access and graphical data output. To run the *Java* applications and services a *Java Runtime Environment (JRE)* is needed that includes the virtual machines, compiler/software development tools, remote service deployment and debugging facilities. For persistent database storage a database service has to be set up. In the *LUCIFER* project a common *MySQL* database was chosen. This relational database management system allows data access and modification with *SQL* statements. To easily access the data stored in the database a web server is applied. *PHP* generated web pages are made available by an *Apache 2* web server. The *CoolStack* software package from *SUN* includes all needed programs[2] to include web service functionalities in the *Solaris 10 OS*.

3.4.1 The Hibernate Framework

Hibernate is a persistent storage framework for the *Java* programming language. It is part of the *JBoss*[3] enterprise middleware system suite and developed/maintained as a professional *Open Source* project with the availability of professional support. In the *LUCIFER* project it was mainly used to store the information that represent the instrument status. *Hibernate* provides mechanisms to transparently persist *OO* data structures, including dependencies like inheritance, association as well as aggregation to a relational database. Any kind of multiplicities in the dependencies is covered whereas one can specify how to propagate data changes. Eager and lazy loading as well as cascading or flat saving of objects is available. To use these multiple storage options this framework needs only an

[2] *MySQL*, *Apache 2* and *PHP* 5 are the most prominent services that are included in *CoolStack*.
[3] *JBoss, Inc.* is a department of *Red Hat*.

appropriate mapping information of the classes. This mapping information can either be specified in separate mapping files or as in-line source code annotations. Appendix E contains an example of such a mapping file. Once a mapping between an *OO* data structure and its relational database representation has been defined, *Hibernate* takes care of the data transformation, object caching and data integrity.

Hibernate has become an industrial standard for medium to large scale software projects with database access. By using this service a lot of time developing specialised database access routines could be saved. No explicit *SQL* commands need to be composed by the developer. He just needs to concentrate on one simple mapping that may be changed without any changes to the source code. This allows fast adoption to changing database requirements. Even thought this framework seems to be oversized for the *LUCIFER* project the usage of the *Hibernate* framework paid off. The time used to integrate this framework and to defined the individual class mappings was much less than the estimated time to write individual *SQL* data export routines. Moreover one should not forget that a mapping automatically provides object import functionality. This option may be used in further software releases to implement instrument status replay capabilities to the virtual instrument (see Chapter 8).

Further information on *Hibernate* can be found in Bauer and King (2006) and on the project's web page `http://www.hibernate.org` (2009).

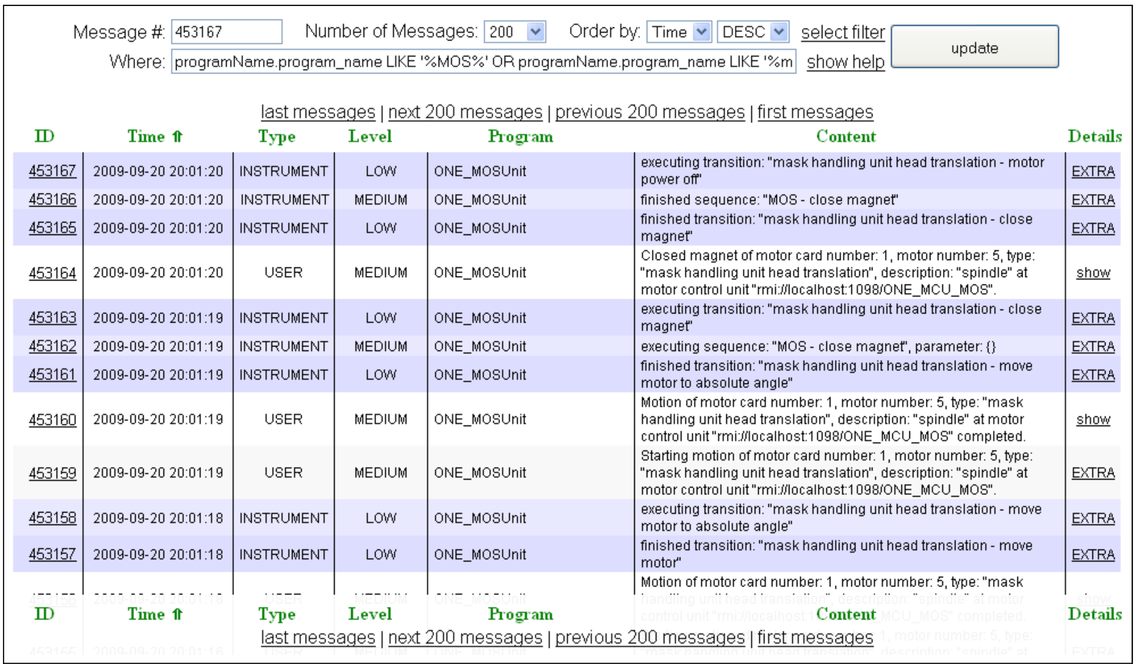
3.4.2 The Message Browser

The storage of program messages is necessary for system status tracking, user notification as well as error debugging. Therefore the messages are processed by a central service and stored in a relational *MySQL* database (compare Section 4.5). Even though a *GUI* application exists to read generated messages on the fly this tool is not able to roll back older messages. For efficient database mining operations a tool which is simple to use is needed. Therefore the message browser that is presented in Figure 3.5 was developed. This browser is realised as a *PHP* web page and allows to specify a message identification number. This unique number can be included into error reports and reduce the amount of time needed for the hardware/software engineers to find and solve a problem.

Besides sorting options, fast browsing capabilities are provided. The selection of messages that are displayed can additionally be influenced by applying filter options. The easiest way to specify filters is to select one out of five message levels for each of the five different message types. These message types are: instrument, system, user, error and debug messages. See Section 4.5 for more details on the individual message levels and types. For a more experienced user *SQL Where-statements* can be specified. By pressing on the identifier the selected message becomes the top message on the display. Each message can be inspected in detail with a separate message overlay. This overlay contains information that can be used to localise the origin of the message in the source code or the exact time the message was created. If auxiliary information has been attached to a message this extra data is displayed in the overlay, too. The presence of such appended data is indicated by naming the details link as <`EXTRA`> instead of <`show`>.

3.4.3 The GEIRS Detector Readout Software

The *Generic Infrared Software* package is an in-house development of the *MPIA* in Heidelberg and is maintained by Clemens Storz. It has been written as a mix of *C/C++* sources without a previous analysis and design phase. Together with the first readout electronics that has been developed to control *IR* detectors this software emerged. Today



ID	Time	Type	Level	Program	Content	Details
453167	2009-09-20 20:01:20	INSTRUMENT	LOW	ONE_MOSUnit	executing transition: "mask handling unit head translation - motor power off"	EXTRA
453166	2009-09-20 20:01:20	INSTRUMENT	MEDIUM	ONE_MOSUnit	finished sequence: "MOS - close magnet"	EXTRA
453165	2009-09-20 20:01:20	INSTRUMENT	LOW	ONE_MOSUnit	finished transition: "mask handling unit head translation - close magnet"	EXTRA
453164	2009-09-20 20:01:20	USER	MEDIUM	ONE_MOSUnit	Closed magnet of motor card number: 1, motor number: 5, type: "mask handling unit head translation", description: "spindle" at motor control unit "rmi://localhost:1098/ONE_MCU_MOS".	show
453163	2009-09-20 20:01:19	INSTRUMENT	LOW	ONE_MOSUnit	executing transition: "mask handling unit head translation - close magnet"	EXTRA
453162	2009-09-20 20:01:19	INSTRUMENT	MEDIUM	ONE_MOSUnit	executing sequence: "MOS - close magnet", parameter: {}	EXTRA
453161	2009-09-20 20:01:19	INSTRUMENT	LOW	ONE_MOSUnit	finished transition: "mask handling unit head translation - move motor to absolute angle"	EXTRA
453160	2009-09-20 20:01:19	USER	MEDIUM	ONE_MOSUnit	Motion of motor card number: 1, motor number: 5, type: "mask handling unit head translation", description: "spindle" at motor control unit "rmi://localhost:1098/ONE_MCU_MOS" completed.	show
453159	2009-09-20 20:01:19	USER	MEDIUM	ONE_MOSUnit	Starting motion of motor card number: 1, motor number: 5, type: "mask handling unit head translation", description: "spindle" at motor control unit "rmi://localhost:1098/ONE_MCU_MOS".	EXTRA
453158	2009-09-20 20:01:18	INSTRUMENT	LOW	ONE_MOSUnit	executing transition: "mask handling unit head translation - move motor to absolute angle"	EXTRA
453157	2009-09-20 20:01:18	INSTRUMENT	LOW	ONE_MOSUnit	finished transition: "mask handling unit head translation - move motor"	EXTRA

Figure 3.5: The Message Database Browser. A *PHP* based web browser tool to display, search and sort the contents of the message database.

GEIRS is used to control almost all infrared instruments built at the *MPIA*. This includes the *IR* instrumentation at *Calar Alto Observatory* (*CAHA*). To access the *IR* arrays the software sends commands to the control electronics via a serial *RS232* connection. These commands are used to define basic parameters like the integration time or the number of integrations as well as more complex parameters that specify the readout mode, sub-frames or electronical pattern generation that is needed to clock the array. Because the electronics is modularised and free configurable it can be adopted to different detector types and match the individual requirements. After an integration the detector content is amplified, digitised and sent back to the control computer via a high-speed fibre connection. The readout software converts the received data together with the instrument configuration into a *FITS*-file and optionally saves it on disk. More detailed information on *GEIRS* can be found e.g., in the technical report of the *MIDI* instrument (see Storz, 2001).

For historical reasons the detector readout is tightly coupled by *GEIRS* with the instrument control. This complicates a stand-alone operation. In the *LUCIFER* project *GEIRS* is just used for the detector access. The lack of separation of concerns complicates the creation of the *FITS* header information. *GEIRS* was intended to write parameters like instrument temperatures and filter wheel positions directly. This is the reason for similar keyword entries in the *FITS* header. More details on the *FITS*-header creation can be found in Knierim (2009).

Besides the basic functionality to control the detector the *GEIRS* software provides a *GUI* to inspect the results of an integration. Either this *GUI* or a command line language can be used for user interaction. The *GEIRS* package was embedded into the *LCSP* by a socket connection. Therefore the command server accepts incoming commands from other remote services. These remote commands correspond to those commands that can locally be entered in the command line.

A first functional version to communicate with the *GEIRS* command server and integrate the detector interaction in the *LCSP* was developed in the diploma thesis Muhlack (2006). More details on the available and used commands of the command server can

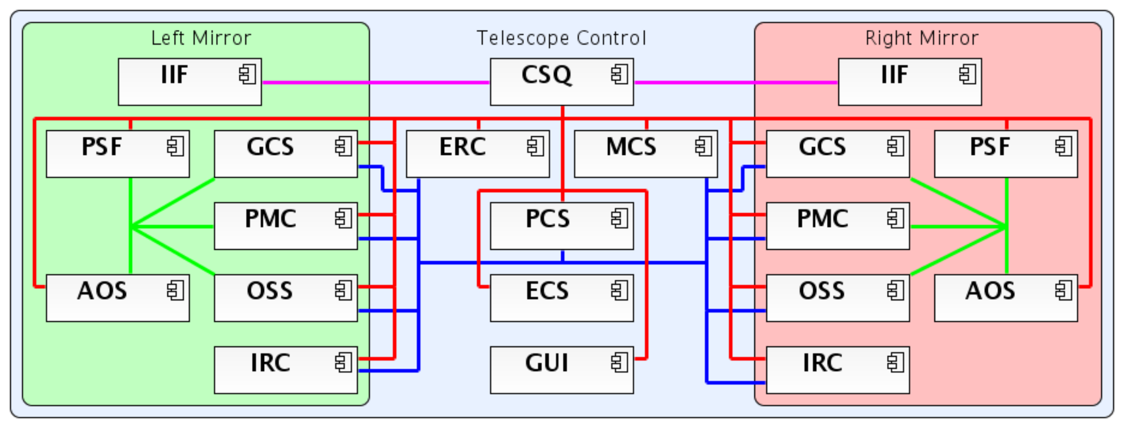


Figure 3.6: Subsystems of the Telescope Control Software (distilled from Axelrod, 2005). RED: Command Exchange Channel, GREEN: PSF Control Channel, BLUE: Pointing Control Channel, MAGENTA: Instrument Control Channel.

be found there. The *Readout Service* that embeds the *GEIRS* package is based on previous work that had been done in an early phase of this thesis. A test readout electronics equipped with simulators instead of *AD-Cs* was brought into service. This setup needs no cryogenic cooled detector array. This was necessary to run first tests with the *GEIRS* to define the requirements[4] in more detail and develop a first software integration plan (compare Section 3.1).

3.4.4 The Telescope Control Software

The *LBT* is controlled by a framework of subsystems called *Telescope Control Software (TCS)*. These subsystems are presented in Figure 3.6. To separate the communication channels and improve performance four independent data paths are used. For instrument interaction the instrument control channel is used. Via this channel the *Instrument Interfaces (IIFs)* communicate with the *Command Sequencer (CSQ)*. Depending on the actions of the instruments the *CSQ* generates the appropriate commands and notifies the subsystems that are responsible for pointing and optical alignment. The command exchange channel ensures this fundamental communication between all subsystems. It connects the independent and separated subsystems of both mirrors with the overall telescope control services. The pointing of each mirror is supervised by the *Point Spread Function Optimisers (PSFOs)* that has access to the *PSFO* control channel. *Adaptive Optics System (AOS)*, *Optics Support Structure Control System (OSS)*, *Primary Mirror Control System (PMC)* and *Guiding Control System (GCS)* need to work together in a coordinated way to ensure stable optical setups. To control the pointing of the telescope the *Pointing Control System (PCS)* communicates via the pointing control channel. In addition to the *GCSs*, *PMCs*, *OSSs* and *IRCs* of the individual mirrors the *Enclosure Rotation Control (ERC)* to move the enclosure and the *Mounting Control System (MCS)* to move the telescope are needed to point, track and guide to selected coordinates. In Terrett (2006) the complexity of pointing a binocular telescope is described. The *Enclosure Control System (ECS)* is an independent subsystem that controls the enclosure and allows to move the shutters and regulate the lights as well as to control environmental elements like air-conditioning and roof heating. More details on the *TCS* and its subsystems are presented in Axelrod (2005). For the realisation of the *TCS* the *C/C++* programming language and *Remote Procedure Calls (RPCs)* based inter-subsystem communications have been

[4][F1] was one of the few predefined requirements.

used. Even though this approach is old-fashioned it has been heavily recommended to the external instrument software teams.

As a fundamental weak spot of the *TCS* design the instruments have to take over control of the telescope. This kind of command structure complicates the allocation of responsibilities. Currently instruments can not issue commands to adjust optical components directly and don't get direct feedback on running operations while their topmost position in the command hierarchy would demand this. Two independent instruments are anticipated for the two mirrors but the design does not allow binocular instrumentation to use both mirrors concurrently.

The first instruments that used the *TCS* were the *LBCs*. The requirements of wide field cameras differ from *NIR* cameras. To run a prime focus instrument only the main mirror needs to be adjusted and the wide *FOV* reduces the demands for pointing and tracking precision. With the installation of the *IR* test camera in one of the bent *Gregorian* focal stations the primary, secondary and tertiary mirrors must work together to produce a proper optical beam. The optical setup as well as the new demands of increased telescope interactions lead to *IIF* functionality enhancements. Even though the *IIF* should be used as an interface to the *TCS* no real system independent access is granted. A complete instance of the control software source is mandatory to access the *CSQ*. This means that any changes to the software force a recompilation of the accessing instrument software.

The control software of the *IR* test camera was written in *C++* by José Borelli. By implementing an *ICE* interface a middleware solution to independently access the *TCS* was created. See Borelli (2008) for detailed information on the commands supported by the *IR* test camera interface. This simple to use interface is included in the *LCSP* to allow telescope interactions instead of accessing the *CSQ* directly. For this reason the *Telescope Service* of the *LCSP* communicates via an *ICE* interface with the *IIF* that sends requests to the *CSQ*. This cumbersome way of instrument-telescope interaction needs to be simplified in the future by implementing a direct interface in the *TCS*.

3.4.5 The JavaDoc Extension Package

Even though this package is not fundamental for the execution of the control software it is important to improve the *API* documentation. Besides a good architecture and design, a good documentation reduces the time for software improvements as well as failure localisation and therefore increases the efficiency and reliability of the control software.

After defining the basic coding style specifications the software documentation conventions have been fixed (compare Chapter 2 and Appendix D). Therefore additional *Taglets* have been implemented. A *Taglet* is an extension to the *JavaDoc* tool that allows annotation tag evaluation. All project dependent *JavaDoc* extensions inherit the basic annotation processing capabilities from the **BaseTaglet** class. This class is based upon the **com.sun.tools.doclets.Taglet** class of *SUN*. The newly created *Taglets* are presented in Table 3.2.

With the exception of the **ExampleTaglet** class all *Taglets* are realised in one class. To permit syntax highlighting of the examples the source code needs to be parsed and split into tokens that can be represented in *HTML*. A first implementation of syntax highlighting was taken from the web. But it was found insufficient for the example presentation because of the missing separation between the individual *Java* keywords. Therefore the classes used to format *Java* source code have been re-design and created together with the other *Taglets* in this thesis.

Annotation	Class	Scope
@changes	**ChangesTaglet**	– logging of change history
@testcase	**JUnitTaglet**	– dynamic linking of *JUnit* test cases
@todo	**ToDoTaglet**	– recording of unsolved tasks
@example	**ExampleTaglet**	– including of source code examples
	JavaToHtmlWriter	– syntax highlighting
	JavaFormatter	– tokeniser and parser
	JavaFormatterResult	– result of parsing

Table 3.2: Overview of the *JavaDoc Taglet* extensions.

3.5 The Metrics of the LCSP

In computer science metrics is used to quantitatively evaluate source code in order to discuss software packages and to monitor code quality. One benefit of metrics is that it can be collected automatically. The tools that have been used to generate the metrics of the *LCSP* are described in Section 2.3. A quantitative inspection of software allows a comparison of software projects, identification of possible weaknesses in design or implementation and an estimation of complexity and costs. The size of a project is somehow correlated with the efforts to create it. E.g., *Red Hat Linux 7.1* consists of 30 million lines of source code and was estimated to cost roughly 1 billion *US*$ if it would have been built in a conventional development process (compare `http://www.dwheeler.com/sloc/`, 2001).

3.5.1 Definition of Metrics

Besides the size of the source code more important metrics exists. In the following subsections the metrics that has been collected for the *LCSP* is specified.

Lines of Code

Even though the *lines of code (LOC)* are one of the less significant metrics they are most often used to quantitatively describe a software project. These *LOC* represent the amount of source code lines that are not empty. Since this measure is directly connected to the skills and formatting preferences of the software developer it can only be used to determine the order of magnitude of a project. The Listings 3.1, 3.2 and 3.3 are a simple example how the same result can be produced by 1, 3 or 5 *lines of code*. The first example shows how inappropriate structuring can lead to many source lines while the size differences of the latter examples result in different formatting. On one hand the occurrence of many copy-and-paste sources result in a higher *LOC* measure while the maintainability of the code is reduced. One the other hand highly compact and efficient code that needed a lot of development time can be covered with only a few *lines of code*.

To overcome these disadvantages several other *LOC* definitions have been introduced. In addition to simply count all non-empty lines the *physical lines of code (PLOC)* and *logical lines of code (LLOC)* measures have been defined. The *PLOC* reflects all lines that physically contain program source. Any kind of documentation and comments are ignored in the calculation of *PLOC*. The *LLOC* is the amount of logical instructions in the sources. To calculate the *LLOC* is more difficult because a logical analysis of the source is needed. A simple way to create this measure in *Java* is to count the number of “;” that are used to terminate an instruction. For *OO* programming languages the *method lines of*

Listing 3.1: LinesOfCodeExample1.java (Java Source File)

```java
System.out.println("Row 0");
System.out.println("Row 1");
System.out.println("Row 2");
System.out.println("Row 3");
System.out.println("Row 4");
```

Listing 3.2: LinesOfCodeExample2.java (Java Source File)

```java
for (int i=0; i< 5; i++) {
  System.out.println("Row " + i);
}
```

Listing 3.3: LinesOfCodeExample3.java (Java Source File)

```java
for (int i=0; i< 5; i++) System.out.println("Row " + i);
```

code (MLOC) is important. This measure counts the source lines that are embedded in methods and therefore represents the executing part of the software and ignores declarative expressions. A high number of source code lines in a specific method directly indicates a structural problem and the need for refactoring.

In January 2010 the *LCSP* consisted of 1,661 files containing ≈ 293 k lines in total. ≈ 269 k of these lines are not empty and contain java sources, *Hibernate* mapping information or internationalisation properties. Configuration files have been excluded from these measurements. The *PLOC* of the software package is ≈ 137 k lines while the *LLOC* is ≈ 98 k lines. The average cost for one *logical lines of code* is ≈ 10 € based on the 22 man-years[5] of work needed to create the software. Compared to the costs of *Red Hat Linux 7.1* this is one third.

Comment Ratio

The *comment ratio (CR)* measures the ratio between *PLOC* and the lines containing comments. These lines can either be the *JavaDoc* annotations or comments embedded in the source code. Empty comment line are omitted from the calculation. The *CR* is used to test an appropriate source documentation. A value of 0.0 is found in undocumented source files while 1.0 represents an equal amount of comment and source lines. Higher values denote more comment than source lines. As for other quantitative ratings the quality of the comments is not considered and hence the *CR* only represents the order of magnitude of source documentation. Another disadvantage of this metric is that software developers tend to comment out source line of former versions or lines that contain debug code. Even though this code is no longer used it is not deleted and therefore distorts the calculation of the *CR*.

The *CR* for the individual files of the *LCSP* was accumulated together with the *LOC* and *PLOC* by a tool that was developed as part of this thesis. The *CR* that was previously calculated with the *Together IDE* was based on the *LOC* and the summation of comment lines did not exclude empty comments. Thereby the ratio for files with numerous comments was undervalued. A comparison of the *PLOC* measurements done with the *Metrics plug-in* and of those done with the especially developed tool showed no difference.

[5]Assuming a man-year of work costs ≈ 50,000 €

Number Metrics

Other metrics counts the amount of structuring elements of the same type. In *OO* programming languages these are in the first instance the *number of classes (NOC)* and the *number of interfaces (NOI)*. These measurements are used to estimate the size and assess the design of a software package. When the number of abstract classes and interfaces is compared with the cumulative number of classes the abstractness of a package can be calculated. This measure reflects whether a package was designed to be abstract or precised. Abstract packages are intended to provide generic solutions that can be specified, concretised and extended by other packages while precised packages solve a concrete task. The *number of methods (NOM)* and *number of attributes (NOA)* metrics reflect the inner structure of programming elements. Too large numbers are often a sign of inadequate design. For the number of parameters of a method this applies accordingly. In case of too many parameters that are passed to a method these parameters should be grouped and encapsulated inside classes. The *NOC* and *NOM* metrics that are presented in Table 3.3, 3.4, 3.5 and 3.6 are added up for programs/frameworks that consist of several software packages.

Dependency Factors

Coupling factors are used in computer science to measure the concatenation of software packages. Thereby it is important to distinguish between *afferent coupling (CA)* and *efferent coupling (CE)* metrics. The *CA* measures the amount of external assignments to classes within a package. This measure reflects how often a package is used from external and therefore points out its relevance for the software product. Contrary to the *CA* the *CE* represents the dependency on other software packages. The total coupling of a package is calculated as the sum of *CA* and *CE*. In Table 3.3, 3.4, 3.5 and 3.6 the maximum *CA* and *CA* values are stated for composed programs/frameworks.

The capability of packages to resist external changes is expressed by the ratio between the *CE* and the total coupling. This metric is called instability. A value of 0 indicates a package with no external connections that is completely stable to external changes. A value of 1 represents an instable solution. In one of the diagnostic diagrams of computer science the instability is plotted versus the abstractness. Due to the anticipated application the instability should be near 0 for an abstract package and near 1 for a concrete solution. The correlation between instability and abstractness forms an ideal line between these both extreme examples and is called *Main Sequence* (see Martin, 1994). The Euclidean distance of a package from this *Main Sequence* reflects the variance in balance between abstractness and instability.

Besides the coupling factors that quantitatively describe the inter-package dependencies cyclic dependencies between packages have to be minimised. Otherwise changes to a package could lead to unpredictable influences on the other cyclically connected elements. This would reduce the maintainability of the source code enormously.

3.5.2 Tier Metrics

Basic metrics of the different software tiers of the *LCSP* (compare Section 3.2) is presented in Table 3.3, 3.4, 3.5 and 3.6. The total sum of analysed *PLOCs* is $\approx 21\,\text{k}$ lines lower than the number of all physical source lines. This difference results in automatically generated source code and used third-party software packages that have been excluded from the analysis.

Program/Framework	*PLOC*	*CR*	*MLOC* avg	*MLOC* max	*NOC*	*NOM*	*CA*	*CE*
Configuration/XML-Lizer*	1,672	1.44	11.4	64	12	97	194	14
Database Access*	590	1.27	10.6	71	7	36	10	6
Message Creation*	1,505	1.57	9.8	75	21	110	210	21
Remote Architecture*	1,489	1.58	7.9	56	24	129	205	19
Taglets*	507	1.08	8.3	54	9	35	0	6
*Time Service**	440	1.38	8.1	65	9	29	46	11
Utilities[1]/R7S*/I18N*	1,108	2.21	7.3	30	14	62	373	8

Table 3.3: Metrics of the *System Tier*.
* 100% developed in the context of this thesis, [1] more than 50%.

Although the *System Tier* consists of merely 7 k lines it is the base of the whole *LCSP* (see Chapter 4). The metrics of the *System Tier* visualises the importance of the *System Tier* (compare Table 3.3). The high *CA* values indicate a strong usage of these packages. The 1.5 k lines of the remote architecture package provide all functionalities to create automatically activatable remote services together with their clients. Every service was created by using a template service and applying simple modifications to its source code. Therefore ≈ 300 *PLOCs* need to be subtracted from every service. E.g., the *Time Service* that was used as an prototype of a first service was realised in only ≈ 150 *PLOCs* by adding methods to exchange time information.

The *Control Tier* is responsible for the interaction with the electronics and the logging of the instrument state. The diversity of the accessed electronics and their different command interfaces lead to a large *PLOC* value of ≈ 25 k lines. Only the temperature controller and monitor use the same command interface language. Therefore the temperature controller was built as a copy of the *Temperature Monitor Service*. Their similarities are reflected by the metrics (compare Table 3.4).

Program/Framework	*PLOC*	*CR*	*MLOC* avg	*MLOC* max	*NOC*	*NOM*	*CA*	*CE*
Journalizer[1]	1,069	1.20	10.3	66	12	65	41	13
*Calibration Unit**	786	1.39	11.0	58	8	45	12	10
RS232 Communication*	852	1.69	7.5	37	14	66	24	16
Electronics/*MCU*[2]	5,887	1.51	10.1	81	69	409	238	15
HIRAMO[3]	1,568	1.53	10.9	64	11	81	61	11
*LuciferVR**	3,679	1.43	9.2	117	39	315	19	19
Pressure Monitor[3]	1,312	1.51	10.0	71	11	90	11	11
Rack Cooling Control Unit[3]	1,586	1.38	10.0	80	12	99	8	11
Readout Service[4]	1,936	1.06	9.1	85	11	146	8	10
*Switch Box Service**	1,252	1.41	10.5	126	11	78	145	10
Telescope Service[5]	1,997	1.01	6.2	98	18	216	20	13
Temperature Control[3]	1,155	1.49	10.5	71	8	72	9	10
*Temperature Monitor**	1,205	1.36	10.5	71	10	76	12	11
Turbo Pump Monitor[3]	885	1.33	10.3	71	10	55	3	11

Table 3.4: Metrics of the *Control Tier*.
* 100% developed in the context of this thesis, [1] more than 75%. [2]: developed in cooperation with Andreas Zeh and 100% reviewed and redesigned. [3] by Volker Knierim. [4] by Tobias Muhlack and Volker Knierim. [5] by Marcus Jütte.

Program/Framework	PLOC	CR	MLOC avg	MLOC max	NOC	NOM	CA	CE
Camera Unit[1]	871	1.25	8.7	43	14	61	14	15
Compensation Mirror[1]	2,242	1.14	9.6	141	16	176	13	16
Detector Unit[1]	1,391	1.17	8.9	97	13	111	12	14
Filter Unit[1]	1,544	1.11	9.2	79	15	116	21	16
Grating Unit[1]	2,298	1.16	8.9	112	28	173	24	27
Instrument Framework*	584	1.23	7.1	28	7	47	92	4
*MOS Unit**	7,964	1.19	8.2	144	138	337	154	12
Pupil Viewer Unit[1]	701	1.13	9.1	42	12	49	11	13
Sequencing Framework*	2,922	1.49	5.7	89	70	295	195	58

Table 3.5: Metrics of the *Instrument Tier*.
* developed in the context of this thesis. [1] by Volker Knierim.

Program/Framework	PLOC	CR	MLOC avg	MLOC max	NOC	NOM	CA	CE
Start Manager[1]	4,120	1.15	8.2	108	47	321	13	17
Pluggable Engineer *GUI**	1,534	0.84	11.8	90	12	76	0	7
*LuciferVR GUI**	1,776	0.55	15.9	146	9	78	0	8
Messaging *GUI* Client*	1,242	0.92	13.2	90	12	52	3	13
Sequencing *GUI**	1,522	1.03	10.0	90	13	69	7	13
MOS GUI Client*	1,283	0.82	12.5	128	4	36	1	4
MCU GUI Client*	2,333	1.12	9.8	73	22	97	5	21
Switch Box *GUI* Client*	1,602	0.84	11.9	82	10	53	2	10
10 Other Engineer *GUIs*[2]	10,536	0.72	10.2	167	59	691	3	13
Readout Service[2]	835	1.22	7.3	75	7	71	8	6
... *GUIs*[2]	958	0.15	8.3	85	7	78	0	6
Supervisor[2]	1,202	1.06	10.2	94	7	75	4	7
... *GUIs*[2]	402	0.71	7.8	30	3	31	0	3
Scripting[2]	2,147	0.41	8.9	160	24	171	5	10
Instrument Manager[3]	2,397	0.69	8.1	74	29	204	30	14
... *GUIs*[3]	3,004	0.41	8.8	790	26	244	8	19
Telescope Service[3]	199	1.18	3.8	18	4	23	4	5
... *GUIs*[3]	3,044	0.14	16.6	1277	12	139	1	8
Acquisition[3]	465	0.69	10.6	36	5	32	4	3
Instrument Data Structure[4]	2,854	1.07	3.3	52	43	451	149	6
Astronomical Functions[4]	611	0.03	6.2	30	4	67	7	4
Observation Templates[4]	580	0.10	24.2	84	6	19	1	5
OPT[4]	17,982	0.24	14.2	158	135	920	14	82

Table 3.6: Metrics of the *Operation Tier*.
* 100% developed in the context of this thesis, [1] more than 50%. [2] by Volker Knierim.
[3] by Marcus Jütte. [4] by Jan Schimmelmann.

The most important services that physically control the mechanical parts of the instrument are the *Motion Control Unit (MCU)*, the switch box and the *HIRAMO Service*. These services have the highest *CAs* because they are widely used in the *Instrument Tier* and have sophisticated *GUIs* in the *Operation Tier*. Both, the *Journalizer* that loggs the instrument state and the framework that communicates with the electronics via a serial connection present high *CA* values.

In the *Instrument Tier* all elemental logics to move the opto-mechanical parts are covered by $\approx 20\,\mathrm{k}$ *PLOCs*. All services of this tier depend upon the sequencing framework and the generic *Instrument Service*. Therefore both frameworks have high *CA* values. The sequencing framework additionally shows a high *CE* value because of its requirement to provide an interface to the control electronics. The most complex service of the *Instrument Tier* is the service used to control the *MOS Unit*. Its size and complexity originates in the construction of the *MOS Unit* and its task to transport freely movable masks from a storage to the focal plane of the instrument. This exchange is realised for any orientation of the instrument by compensating and correcting unavoidable mechanical bending and stepper motion errors.

The *Operation Tier* is responsible for operating the instrument. Central services that combine the functionalities of lower tiers provide all functionalities to perform observations with the *LUCIFER* instrument. Beside these services many *GUIs* that grant engineering access to the services of the lower tiers and *GUIs* to prepare and perform observations have been developed. *GUI* applications easily reach large *PLOCs* with typically low *CRs* especially when created with automated *GUI* builders. The creative effort to build a *GUI* is negligible compared with the work to develop the functionalities of a service.

In the *Operation Tier* of the *LCSP* $\approx 49\,\mathrm{k}$ lines of the $\approx 63\,\mathrm{k}$ *PLOCs* are used for *GUI* applications. Most of these *GUI* applications are only utilised for engineering tasks. The remaining lines provide the capability to perform observations and coordinate the instrument, the detector and the telescope. As expected for the highest tier all applications should have lower *CA* than *CE* values because functionalities are only centralised without the need of propagation (see Table 3.6).

Theoretically the instrument could be used with the engineering applications only, but for a more convenient operation this is not anticipated. The elements that are fundamental for the execution of the *LCSP* are reflected by their high *CA* values (compare Table 3.3, 3.4, 3.5 and 3.6). Most of these significant programs/frameworks have been developed in the context of this thesis.

CHAPTER 4

The System Tier

The System Tier contains all basic frameworks and services to run the LCSP. As a fundamental factor of a robust and high-performance distributed system the inter-service communication framework is presented. Resource management, internationalisation and time synchronisation are part of this tier. The persistent storage of data used either for system configuration or logging is introduced. Finally the messaging framework and its service is described.

In Section 3.5 the metrics of the *System Tier* exhibit very high *CA* values. This can be used as an indicator of the importance of the embedded services and frameworks. The *System Tier* ensures a stable and fast running distributed system and provides basic functions to the services and applications. In a distributed environment it is important to unify the distribution mechanism of the functionalities and the service creation process. This includes a centralised storage of data and service configurations to increase the maintainability. Time synchronisation is needed when a distributed system spreads across multiple hardware instances. To track the state of the software system and to create human readable information conveniently a distributed messaging system is mandatory.

4.1 The Remote Service Framework

All distributed services and applications of the *LCSP* are based on the remote service framework. This framework automatically integrates the time synchronisation, configuration management and message exchange into every service of the *LCSP*. Besides integrating project specific functionalities the framework implies all necessary skeletons to create services and their corresponding clients. The implementation of the remote service framework is based on the *Remote Method Invocation (RMI)* capabilities of *Java*.

4.1.1 Remote Method Invocation

RMI is comparable to *RPC* as remote methods/procedures are transparently called via a networking connection. As other programming languages *Java* allows for sequential processing of data. Such a sequence is called a stream. It can either be used to send/receive data to/from a file, a terminal or another program. To be able to do this with *OO* data structures all the objects must specify their mode of data serialisation and de-serialisation. In the simplest way this is done by implementing the empty **java.io.Serializable** interface. This denotes an object to be automatically serialised in which all transient attributes are omitted. For complex data structures the object serialisation and de-serialisation can be specified by implementing the *.writeObject(out)* and *.readObject(in)* methods. To prevent different versions of serialisation from compromising the data integrity the use of the `serialVersionUID` attribute is recommended for versioning. This attribute allows to label an implementation. In distributed *OO* applications the transport of objects instead of primitive types is essential. The mechanisms of streaming the data structures is the basis of the inter-service communication.

Sockets provide an interface for network communications. A *socket* communication connects to processes across an *IP* network and allows data exchange. These communications are not necessarily limited to inter-computer data exchange. They are commonly used to communicated between applications/processes that run on one and the same computer. On top of the network layer that ensures the data transportation simple data streams are used to realise the bidirectional communication. The *IP* address of a host together with a port number is used to specify a *socket*. For many applications string messages are used for the client-server communication. This demands a specification of commands and data exchange formats. Such a string based *socket* connection is used to interact with the *GEIRS* server (compare Section 3.4.3). A benefit of this communication paradigm is that the applied programming language and its data type conventions are of no importance to both the server and its clients. This leads directly to the disadvantage of this approach. The data parsing and conversion must be done by the developer. Middleware solutions like *CORBA* or *ICE* have been built to minimise this work. These solutions introduce a programming platform independent interface definition language. Once the interfaces are specified all data and type conversions are done and the required source code is generated automatically by the middleware.

As *RMI* is used to connect two *Java* applications no data unification/transformation is needed. The hardware platform independence of *Java* already requires an internal data transformation. E.g., the endianness or word size of a computer architecture must not be considered when writing an application. Therefore the *JRE* intrinsically ensures a uniform data representation and no data mapping between the applications needs to be specified. The data is exchanged on the basis of the underlying object serialisation and de-serialisation capabilities of *Java*. Together with the data the method calls must be transported over the *socket* connection. This means that both the method calls and their responses must be wrapped into a byte stream. The marshalling and de-marshalling of data, the handling of the *socket* connection by threads and the remote execution of method calls is transparently done by *RMI*. The only difference between a remote and a local method call is the required handling of possible exceptions caused by the network transportation. To make a method callable from remote an object needs to implement an interface that extends **java.rmi.Remote**. This interface is used to specify the available remote callable methods. To allow handling of errors that arise from the *socket* communication all remote methods must throw a **java.rmi.RemoteException**. This increases accordingly the fault tolerance of an application.

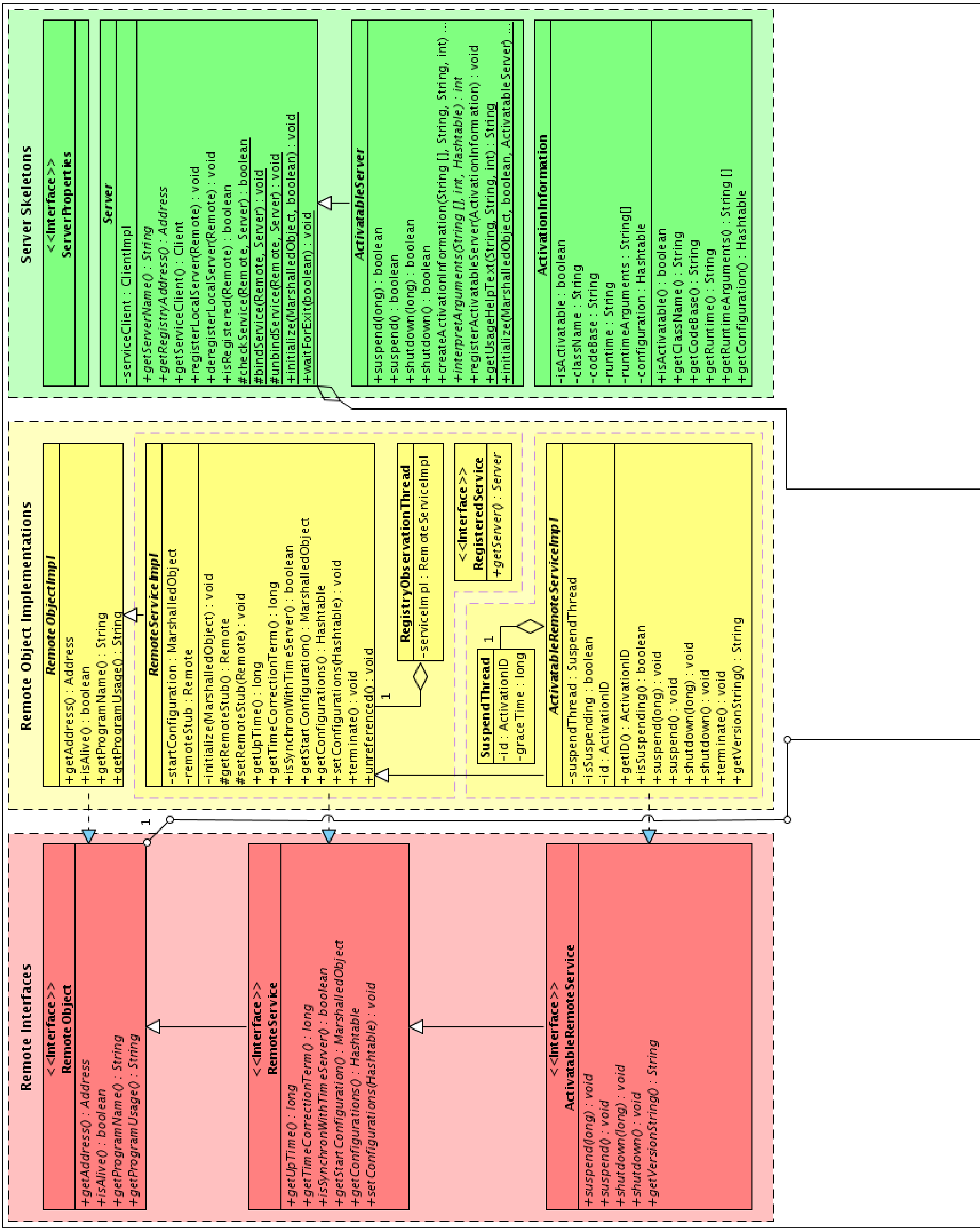
On each side of the *socket* communication a proxy is used to connect the data stream with the virtual machine of *Java*. On the client side this proxy is called *stub*. This *stub* acts like an object that is part of the application nonetheless the actual instance is on the remote side. Therefore the parameters are marshalled before the method call and the return value is de-marshalled after it. On the server side a *skeleton* grants access to the remote instance. Incoming method calls are passed together with the de-marshalled parameters to the instance of the remote object and the return value is sent back to the client application. Since *RMI* does not guarantee that each method call is handled by one and the same thread it is important that the remote method implementation is thread safe. In the current version of *Java* the creation of *skeleton* files is not mandatory however each *stub* demands a corresponding class file. These classes are created by the *RMI* compiler. In the *LUCIFER* project the *stub* creation was integrated in the build process (see Appendix C).

As a final step of creating a *RMI* connection between two applications a remote object must be exported. This includes the creation of the remote object as well as the *skeleton* on the server side proxy. For remote access the created instance is connected to a network *socket*. To establish a connection the client implementation uses a remote reference realised by a *stub*. However this reference requires the address information of the *socket*. To manage the remote references centrally and distribute the address information *Java* provides a registry service. Remote objects are bound to the registry with a unique identifier. This allows clients to query for a specific object. The security policy of *Java* prevents remote objects from being registered at remote computers. For that reason at least one registry is required on every computer that should host remote objects. On the client side the remote object look-up mechanism demands the specification of the remote host. Once a remote reference is acquired a direct communication between the remote object and the client application is established. Besides the simple export of a remote object *Java* supports a dynamic creation and registration. To create remote objects on demand the activation system daemon can be used. This daemon is capable of creating new virtual machines with remote objects. To support this activation the implementation of a remote interface has to extend the abstract **java.rmi.activation.Activatable** class. Once an activatable object has been bound to the registry service incoming look-up calls are passed to the activation system. This system returns the remote reference to already existing instances while the non-existing are spawned. This automatic and on demand creation of remote objects increases the stability of the system. E.g., an unintentionally ended instance of a virtual machine is recreated and does not affect the execution of the depending clients.

More information concerning *RMI* especially on distributed threading, distributed garbage collection and the activation system can be found in GROSSO (2002).

4.1.2 The Remote Interfaces

As the design of the *LCSP* demands a distributed service architecture (compare Section 3.2) a framework to create remote services was developed. The class diagram in Figure 4.1 presents the core classes of this framework. Besides these classes other classes exist that are used e.g., to store the address or to configure the networking behaviour of a remote service. The upper part of the class diagram contains all classes and interfaces that are required to realise the server side of a service while the lower part represents the client side. The server side is divided into the remote interfaces, their implementations and a server skeleton. The remote interfaces specify the remote capabilities of a service. To hierarchically group the remote capabilities in dependence of the kind of service an inheritance structure was chosen. The root of this structure is the **RemoteObject** interface. This interface defines rudimentary methods to test whether a service is alive and to retrieve its address, name and its designated use. The **RemoteObject** interface is extended by the **RemoteService** interface to add functionalities for querying the up-time and *Time Service* connection information. Methods to exchange the configuration set of a service are added, too. The **ActivatableRemoteService** interface is the last element of the inheritance hierarchy. This interface adds methods to stop or suspend a service. This is necessary to ensure that automatically activated services can be stopped at all. Otherwise the activation system would restart them. Even though these remote interfaces define remote methods that throw a **java.rmi.RemoteException** none of these interfaces directly extends the **java.rmi.Remote** interface. The **java.rmi.Remote** interface is implemented directly by the remote implementations. This allows to use these interfaces more generally.


Server Skeletons
<<Interface>>
ServerProperties
Server
-serviceClient : ClientImpl
+getServerName() : String
+getRegistryAddress() : Address
+getServiceClient() : Client
+registerLocalServer(Remote) : void
+deregisterLocalServer(Remote) : void
+isRegistered(Remote) : boolean
#checkService(Remote, Server) : boolean
#bindService(Remote, Server) : void
#unbindService(Remote, Server) : void
+initialize(MarshalledObject, boolean) : void
+waitForExit(boolean) : void
ActivatableServer
+suspend(long) : boolean
+suspend() : boolean
+shutdown(long) : boolean
+shutdown() : boolean
+createActivationInformation(String [], String, String, int) ...
+interpretArguments(String [], int, Hashtable) : int
+registerActivatableServer(ActivationInformation) : void
+getUsageHelpText(String, String, int) : String
+initialize(MarshalledObject, boolean, ActivatableServer) ...
ActivationInformation
-isActivatable : boolean
-className : String
-codeBase : String
-runtime : String
-runtimeArguments : String[]
-configuration : Hashtable
+isActivatable() : boolean
+getClassName() : String
+getCodeBase() : String
+getRuntime() : String
+getRuntimeArguments() : String []
+getConfiguration() : Hashtable
Remote Object Implementations
RemoteObjectImpl
+getAddress() : Address
+isAlive() : boolean
+getProgramName() : String
+getProgramUsage() : String
RemoteServiceImpl
-startConfiguration : MarshalledObject
-remoteStub : Remote
-initialize(MarshalledObject) : void
#getRemoteStub() : Remote
#setRemoteStub(Remote) : void
+getUpTime() : long
+getTimeCorrectionTerm() : long
+isSynchronWithTimeServer() : boolean
+getStartConfiguration() : MarshalledObject
+getConfigurations() : Hashtable
+setConfigurations(Hashtable) : void
+terminate() : void
+unreferenced() : void
1
RegistryObservationThread
-serviceImpl : RemoteServiceImpl
<<Interface>>
RegisteredService
+getServer() : Server
SuspendThread
-id : ActivationID
-graceTime : long
1
ActivatableRemoteServiceImpl
-suspendThread : SuspendThread
-isSuspending : boolean
-id : ActivationID
+getID() : ActivationID
+isSuspending() : boolean
+suspend(long) : void
+suspend() : void
+shutdown(long) : void
+terminate() : void
+getVersionString() : String
1
Remote Interfaces
<<Interface>>
RemoteObject
+getAddress() : Address
+isAlive() : boolean
+getProgramName() : String
+getProgramUsage() : String
<<Interface>>
RemoteService
+getUpTime() : long
+getTimeCorrectionTerm() : long
+isSynchronWithTimeServer() : boolean
+getStartConfiguration() : MarshalledObject
+getConfigurations() : Hashtable
+setConfigurations(Hashtable) : void
<<Interface>>
ActivatableRemoteService
+suspend(long) : void
+suspend() : void
+shutdown(long) : void
+shutdown() : void
+getVersionString() : String

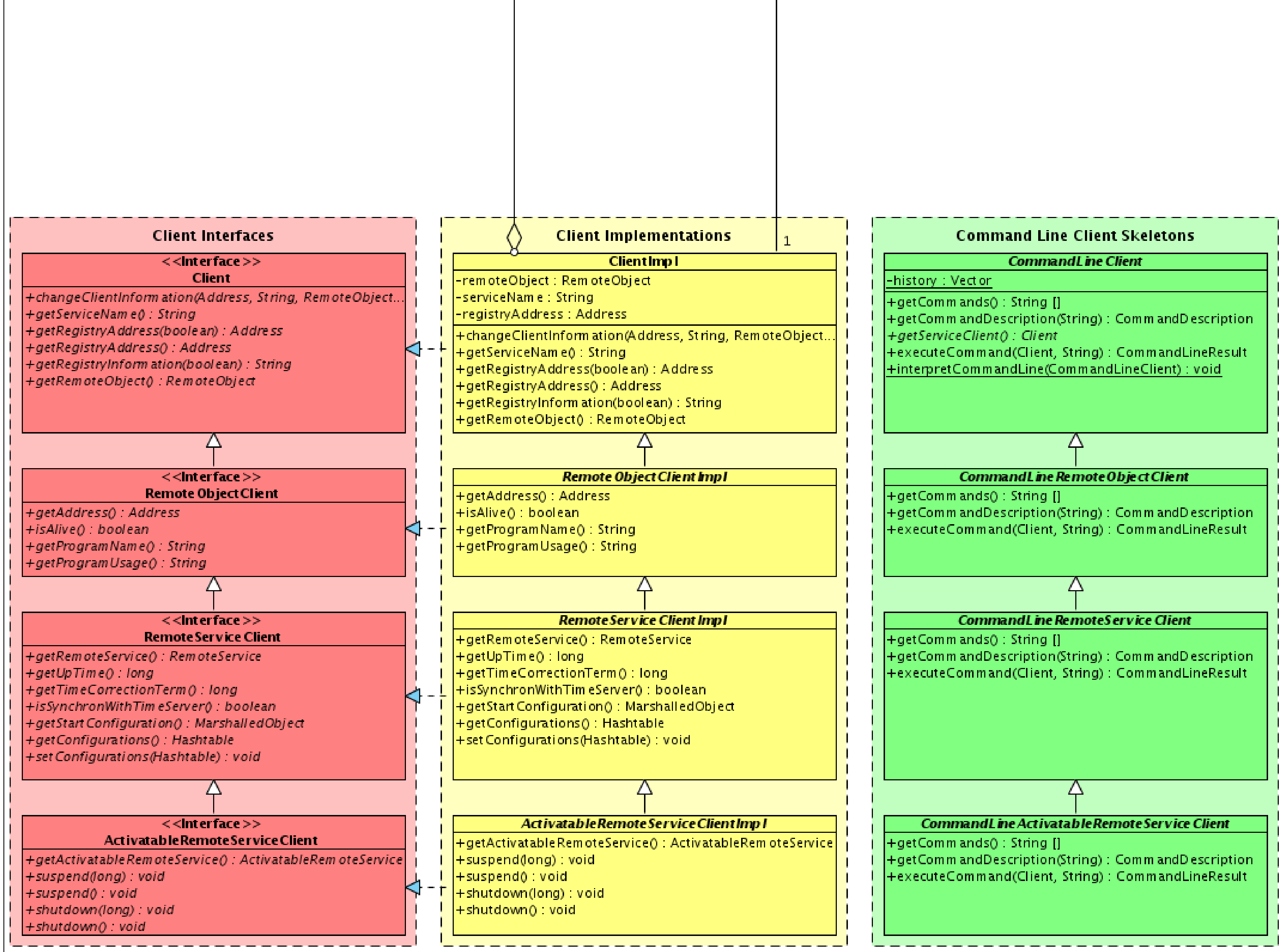


Figure 4.1: Class diagram of the remote service framework. The upper part covers the server side while the lower part represents the client side. RED : interface specifications, YELLOW : corresponding interface implementations, GREEN : software skeletons.

4.1.3 The Remote Object Implementations

The counterpart of the interfaces are the implementations of the remote functionalities. These implementations use the same inheritance hierarchy as the interfaces do. The abstract **RemoteObjectImpl** class realises the basic remote methods while the abstract **RemoteServiceImpl** class implements native access to the configuration sets and to the *Time Service* connection status. More important for the **RemoteServiceImpl** are the implementations of the **Unreferenced** and **Terminatable** interfaces. The **Unreferenced** interface is used to signalise to the registry service that its remote garbage collector can notify an unreferenced remote object. Concretised services that extend the abstract **RemoteServiceImpl** can override the corresponding method. The **Terminatable** interface is introduced to the remote service framework to enable a controlled service shutdown. The **RemoteServiceImpl** class is implemented in that way that an ending *JRE* will call the terminating method and allow overriding classes to handle the shutdown. For each **RemoteServiceImpl** object that implements the **RegisteredService** interface a **RegistryObservationThread** is created. This thread ensures that a service is ended as soon as the registered remote reference differs from the observed remote object. This is necessary to terminate remote services that are no longer accessible through the registry service and therefore can no longer be discovered by clients. Otherwise a service repeatedly registered with the same identifier would lead to only one reachable instance and thus waste resources. Finally the abstract **ActivatableRemoteServiceImpl** class provides the functionalities defined for an activatable remote service. This includes the usage of a **SuspendThread** to suspend an activated remote service after a specified grace time. The abstract implementations of the remote interfaces embed several project specific functionalities. These functionalities can be accessed e.g., by the user through the *LMC*.

4.1.4 The Server Skeletons

The missing element to create a remote service is a server skeleton that creates a remote object and binds it to the registry service or to the activation system daemon, respectively. The simplest way to realise a remote service is to create a remote object within a *JRE* and bind it to the registry. When the *JRE* ends the remote instance is no longer usable while the remote reference is still bound. Therefore the **RemoteServiceImpl** class ensures an automatic de-registering in case of a terminated service. The **Server** class provides basic functionalities to start such a local service. To access the implementation of a remote object the **Server** class uses a client implementation reference. This client implementation can also be used to get the address of the remote service. The skeleton that is realised by the **Server** class implements methods to bind, unbind and check the status of a remote instance. However the most import part of the **Server** class is the static *.initialize()* method. This method ensures that all project specific program arguments are processed and the corresponding clients are set up correctly. This configuration passing is mandatory to specify basic service parameters at service start. To realise a uniform configuration of all services a *Hashtable* is used that maps predefined keys to the configuration values. The fundamental keys are defined in the **ServerProperties** interface and are extended in inherited definitions of the concrete service realisations. Other static methods allow to bind and unbind a service, to check whether a service is still bound and running and to wait until a server is stopped. Those static methods that use a **Server** object rely on the abstract methods that grant access to the identifiers of a server. These methods must be implemented by the concrete services to exchange the service identification parameters with the skeleton.

As an extension of the **Server** class the **ActivatableServer** provides a skeleton to create an on-demand activated service. To allow the activation system to create a *JRE* that hosts the on-demand activated service several parameters need to be specified. These parameters are represented by the **ActivationInformation** class. In addition to some basic parameters like the server class name to execute, the code base, the *JRE* or the *JRE* arguments, the program arguments *Hashtable* is stored. Each descendant of an **ActivatableRemoteServiceImpl** class gets an instance of these program arguments that can be passed to the *.initialize()* method of the **Server** class. Therefore each activated service is started with the same program arguments as the server once was started with. The *Hashtable* itself is created from the command line arguments that have been parsed during the start process. The parsing of the fundamental keys is done in a method that is overridden by the concretised server implementations. These implementations define new keys and parse them. A short list of the available program arguments can be found in Appendix B. In comparison to services that have been started locally an activatable service is registered at the activation system daemon. Since this daemon has its own registry this registry is used to look-up the remote references. As activated services are started automatically by the activation daemon they can no longer be stopped by just ending the server application. Therefore the server application must be started to utilise the service client reference to send a suspend or stop command to the corresponding remote instance. As the **ActivatableServer** is only required to create a remote service that is managed by the activation system daemon it can be stopped after the binding processes have been finished.

Another challenge of a distributed system whose services are created automatically and on-demand is to obtain debugging access. Since the **ActivatableServer** extends the **Server** class all activatable services can be started as a local version. Then the debugger of the applied *IDE* can directly access the running service and trace external remote method calls of client applications. This simplifies enormously the inherently complex debugging of a distributed application.

4.1.5 The Remote Service Client Architecture

To use the remote methods of the services corresponding clients must be implemented. The lower part of Figure 4.1 represents the client side and is symmetrically designed to the server side. As well as on the server side a hierarchical interface structure is used to group client functionalities. The **RemoteObjectClient**, **RemoteServiceClient** and **ActivatableRemoteServiceClient** interfaces define the access to the functionalities that are specified in the **RemoteObject**, **RemoteServer** and **ActivatableRemoteService** interfaces, respectively. The most important difference between these structures is the root of the client interface hierarchy. The **Client** interface defines rudimentary methods to handle remote references. This includes accessing the service identification parameters as well as the registry address.

As on the server side the implementation of the client interfaces is a transfer of the interface structure. Just as the **Client** interface defines basic client functionalities the **ClientImpl** class implements all necessary methods to create a client. Therefore the reference to the accessed remote object is stored together with the demanded service identification parameters that are used to look-up the service in the registry. The stored remote reference is casted transparently by the extending classes into the required type. The internal storage and buffering of the remote reference in the client minimises calls to the registry and to the activation system of the applied underlying *RMI* system. Each extending client implementation ensures a direct connection to the remote object.

The command line client skeleton (see Figure 4.1) can be used as a basis for developing a simple client that provides service access through the command line. For all basic services of the *System Tier* such a client was developed. The **CommandLineClient** class and its descendants implement the parsing of command strings for the corresponding client functionalities in the hierarchy. First it was planned to include the command line client functionalities in every service client to grant fundamental access. This approach was given up in favour of a *GUI*. To implement this feature in future software releases the inheritance hierarchy must be modified slightly in such a way that the **CommandLineClient** class extends the client implementation of an activatable service. Additionally all concrete service client implementations must be changed to extend the command line client instead of the basic client and override the *.executeCommand(Client, String)* method.

The implementations on the client side of the remote service framework ensure that all extending clients offer the project specific functionalities. Therefore the client of a specific service just needs to implement access to its particular methods. All clients to remote services of the *LCSP* have been developed with the specification to have a static method to retrieve a client instance. This centralised client creation and management enables the software to load the configuration data demanded by each client. In each client implementation the pre-conditions to use the reference accordingly are guaranteed transparently. Thus an application that utilises a client can simply call its methods without the necessity to run an initialisation prior to the remote service access.

Another demand of the client implementation is to include messages into the client method calls. Besides those messages that are created on the service side these messages are needed to find errors in a client application (see Section 4.5). In the end all client implementations just bundle the remote reference management with the wrapped method calls to a uniform and compact service gateway.

4.2 The Time Synchronisation Service

Since the *LCSP* was designed as a distributed system the applications can be spread across different computers. This directly leads to the problem that the computers may have different system times. A synchronous time is important for the messaging system. To synchronise the time used by the services the *Time Service* was developed. The *Time Service* itself adds only one method to the service implementation of the remote service framework. This method returns the time of the system the service is running on.

For the *Time Service* two client implementations exist. One to directly access the service in order to suspend or stop it and another to provide synchronous time creation capabilities to an application. As the service only returns its plain system time the latter one implements all logics to calculate a time correction term. This correction term is then used to create synchronised system times on the client side.

Several solutions to synchronise computer clocks exist. E.g., the *Network Time Protocol (NTP)* includes time synchronisation via network in the *OS*. In that sense the *Time Service* is a reinvention of the wheel that synchronises client applications with a central server taking into account the network transportation delay. As the *Time Service* is the simplest service of the *LCSP* it was used as a testbed for the remote service framework. This service was developed as a prototype for all other remote services. An advantage of the *Time Service* is that it is natively integrated into the control software and its synchronisation calls can be used to trace the healthiness of the services. By default all applications and services synchronise every 60 minutes with the *Time Service*. The corresponding messages can be used to determine whether a service was running at a specified time by querying the database.

4.3 Resource Management/Internationalisation

The *LUCIFER* project is an international project and thus has special requirements on localising the software. The whole control software source was written and documented in English. On the user-side of the software it was intended to externalise all strings and formatting information to realise *Internationalisation (I18N)*. Therefore the *Resources (R7S)* handling **R7S** class and the formatting **I18N** class have been developed. These classes allow to load strings from external files and apply text formatting functions. All packages developed in the scope of this thesis contain classes that start with <R7S>. These classes provide access to the *I18N* data. This access is realised through static methods that use a default value in case of missing property files that contain the *I18N* mappings for different locations.

Besides externalising strings both classes provide formatting capabilities. By passing an array of objects to the *R7S* methods these parameter are automatically included into the resource string. For that reason special formatting tags need to be included in the resource string. E.g., <{0} likes {1}> realises a simple insertion of two values. As the text formatting capabilities of *Java* allow to specify formats independently of the used locale any methods to ensure a specific data format for each locale are dispensable. E.g., <{0,number,#0.000}> defines that the passed number object is always displayed with 3 decimal digits and may be padded with zeros. If no particular number format is specified the default format of the used locale is applied. Thereby the correct usage of decimal separators and date formats is guaranteed in any locale. Finally the choice format can be used to improve the readability of typified number values. E.g., <{0,choice,1#ONE|2#TWO}> transforms a number value of 1 or 2 into the corresponding word.

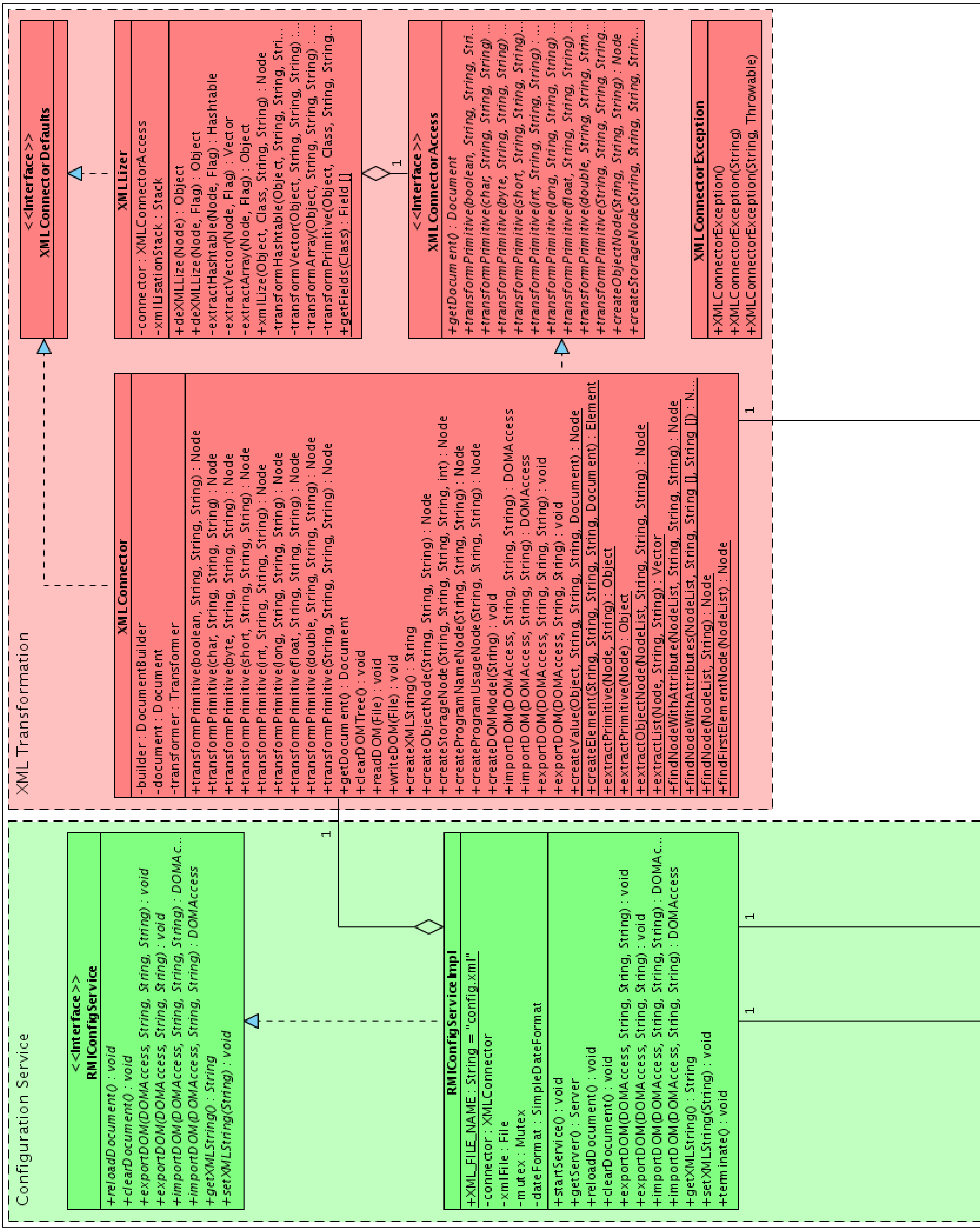
4.4 The Persistent Data Storage

As in any larger software project the *LCSP* needs to store volatile application data permanently. This persistently stored data can either be input or output of an application. The input can e.g., be configuration data that specifies the behaviour of the application or data that needs to be processed. Besides the science data of the detector, output generated by the *LCSP* are e.g., the messages that notify on the system status, the journalised instrument status or the logged transmissions of the electronics. Persistent data storage can be realised as plain file or database storage depending on the kind of data and the anticipated post processing. In distributed systems the persistent storage is often centralised to reduce the data management complexity that is required to combine the scattered data.

In principle *Java* is equipped with its own object storage capabilities that store and load serialised object data streams to/from disk (compare Section 4.1.1). The drawback of this storage solution is that the format of the serialised data is not human readable meaning that special tools to view and edit it are demanded. Therefore wherever applicable the *XML* data format was applied for plain file storage while content that requires to be searchable was stored in an *SQL* database.

4.4.1 The XML Transformation Framework

Many of the original tapes from the *Apollo 11* Moon landing have got lost in the *NASA* archives. Those tapes that still exist contain data sets that nobody can decipher and interpret because the knowledge of the used data format is lost (see Harold and Means, 2004). To solve such problems the *Extensible Markup Language (XML)* allows to structure data by user defined tags and ensures that the format is tightly coupled with the content.


XML Transformation
<<interface>>
XMLConnectorDefaults
XMLLizer
-connector : XMLConnectorAccess
-xmlLisationStack : Stack
+deXMLLize(Node) : Object
+deXMLLize(Node, Flag) : Object
-extractHashtable(Node, Flag) : Hashtable
-extractVector(Node, Flag) : Vector
-extractArray(Node, Flag) : Object
+xmlLize(Object, Class, String, String, String) : Node
-transformHashtable(Object, String, String, Stri...
-transformVector(Object, String, String, String) :...
-transformArray(Object, String, String, String) : ...
-transformPrimitive(Object, Class, String, String...
+getFields(Class) : Field []
<<interface>>
XMLConnectorAccess
+getDocument() : Document
+transformPrimitive(boolean, String, String, Stri...
+transformPrimitive(char, String, String, String) ...
+transformPrimitive(byte, String, String, String) ...
+transformPrimitive(short, String, String, String)...
+transformPrimitive(int, String, String, String) : ...
+transformPrimitive(long, String, String, String) ...
+transformPrimitive(float, String, String, String) ...
+transformPrimitive(double, String, String, Strin...
+transformPrimitive(String, String, String, String...
+createObjectNode(String, String, String) : Node
+createStorageNode(String, String, String, Strin...
XMLConnectorException
+XMLConnectorException()
+XMLConnectorException(String)
+XMLConnectorException(String, Throwable)
XMLConnector
-builder : DocumentBuilder
-document : Document
-transformer : Transformer
+transformPrimitive(boolean, String, String, String) : Node
+transformPrimitive(char, String, String, String) : Node
+transformPrimitive(byte, String, String, String) : Node
+transformPrimitive(short, String, String, String) : Node
+transformPrimitive(int, String, String, String) : Node
+transformPrimitive(long, String, String, String) : Node
+transformPrimitive(float, String, String, String) : Node
+transformPrimitive(double, String, String, String) : Node
+transformPrimitive(String, String, String, String) : Node
+getDocument() : Document
+clearDOMTree() : void
+readDOM(File) : void
+writeDOM(File) : void
+createXMLString() : String
+createObjectNode(String, String, String) : Node
+createStorageNode(String, String, String, int) : Node
+createProgramNameNode(String, String, String) : Node
+createProgramUsageNode(String, String, String) : Node
+createDOMModel(String) : void
+importDOM(DOMAccess, String, String, String) : DOMAccess
+importDOM(DOMAccess, String, String) : DOMAccess
+exportDOM(DOMAccess, String, String) : void
+exportDOM(DOMAccess, String, String) : void
+createValue(Object, String, String, String, Document) : Node
+createElement(String, String, String, String, Document) : Element
+extractPrimitive(Node, String) : Object
+extractPrimitive(Node) : Object
+extractObjectNode(NodeList, String, String, String) : Node
+extractList(Node, String, String) : Vector
+findNodeWithAttribute(NodeList, String, String, String) : Node
+findNodeWithAttributes(NodeList, String, String [], String []) : N...
+findNode(NodeList, String) : Node
+findFirstElementNode(NodeList) : Node
Configuration Service
<<interface>>
RMIConfigService
+reloadDocument() : void
+clearDocument() : void
+exportDOM(DOMAccess, String, String, String) : void
+exportDOM(DOMAccess, String, String) : void
+importDOM(DOMAccess, String, String, String) : DOMAc...
+importDOM(DOMAccess, String, String) : DOMAccess
+getXMLString() : String
+setXMLString(String) : void
RMIConfigServiceImpl
+XML_FILE_NAME : String = "config.xml"
-connector : XMLConnector
-xmlFile : File
-mutex : Mutex
-dateFormat : SimpleDateFormat
+startService() : void
+getServer() : Server
+reloadDocument() : void
+clearDocument() : void
+exportDOM(DOMAccess, String, String, String) : void
+exportDOM(DOMAccess, String, String) : void
+importDOM(DOMAccess, String, String, String) : DOMAc...
+importDOM(DOMAccess, String, String) : DOMAccess
+getXMLString() : String
+setXMLString(String) : void
+terminate() : void

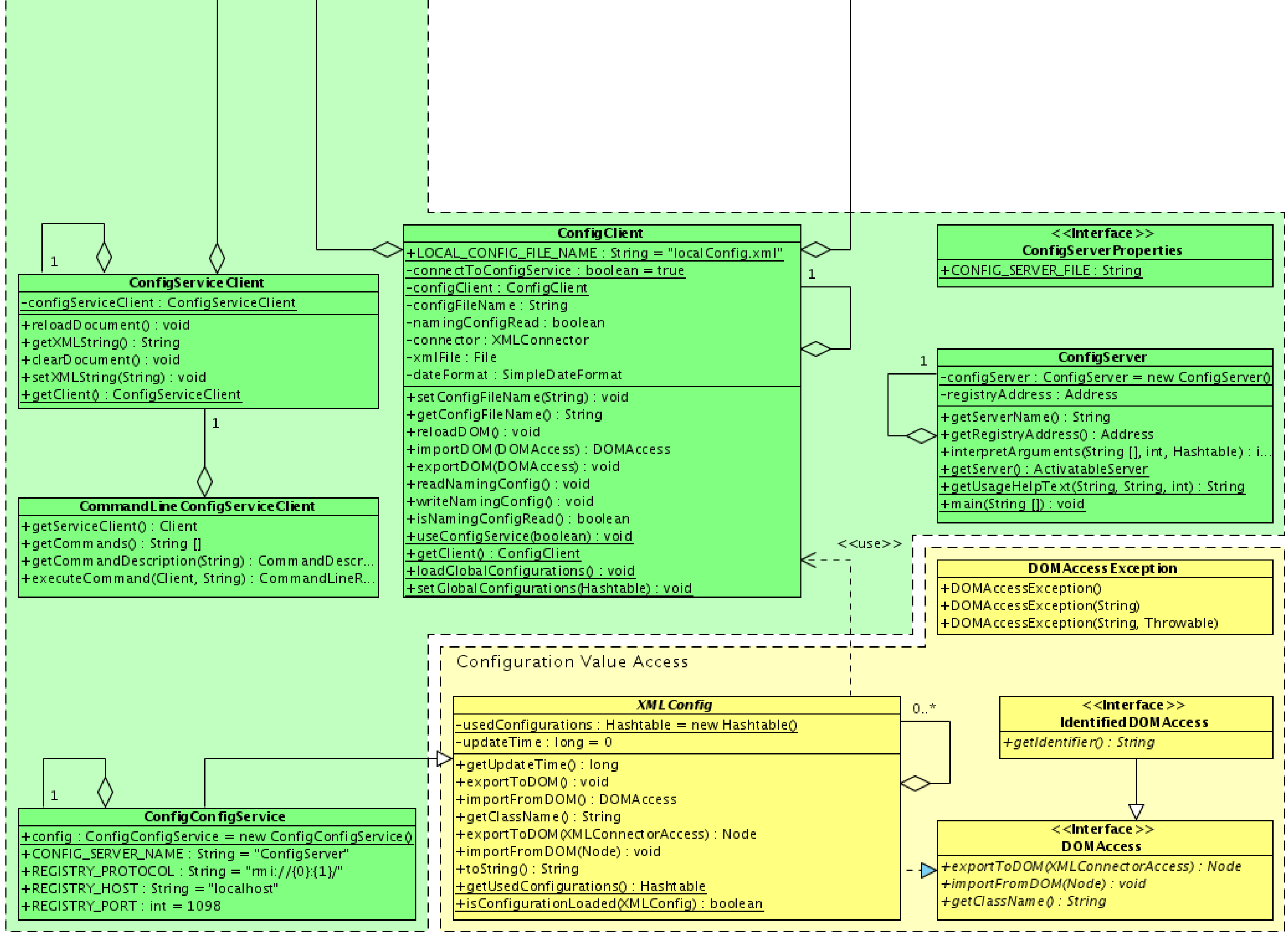


Figure 4.2: Class diagram of the *Configuration Service*.
RED : Framework to transform objects in an *XML* representation and vice versa. YELLOW : The classes and interfaces to transparently integrate configuration values into the *LCSP*. GREEN : The classes that realise the remote *Configuration Service* and its clients.

A framework was developed to realise the conversion between objects and *XML* structures. This framework is shown in Figure 4.2. It is part of the *Configuration Service* because its main task is to persist configuration values. Even though the framework is based on only two classes it offers very powerful data transformation capabilities and therefore is extensively used for *XML* data access. The *XML* transformation system uses the *Document Object Model (DOM) API* of *Java*. The **org.w3c.dom** package contains all functionalities to parse a syntactical correct *XML* representation into a structured tree. This allows to access the nodes that contain the *XML* information directly without manually parsing files or creating *XML* tags. The transformation between the structured tree and the *XML* representation is handled by the *DOM API*. The semantical interpretation of the nodes is done in the **XMLConnector** class. This class uses the predefined markup strings of the **XMLConnectorDefaults** interface to encapsule/extract all primitive data types in/from corresponding *DOM* nodes. Exceptions that are thrown during data transformations are typified as **XMLConnectorException** or **DOMAccessException** classes depending on the allocation of the exception.

The **XMLConnector** class uses the **org.w3c.dom** package to realise direct access to the file and the *DOM* representation. Static auxiliary functions allow to find nodes in the tree or extract data. As the *XML* transformation of the *LCSP* was designed for the configuration value storage the **XMLConnector** class is able to encapsule data sets within a name and usage tag. This allows a data separation based on the application/service that is associated with the data. The tight data-application coupling is implemented by the remote service framework that is responsible for embedding configuration capabilities in every service. Instead of directly using the **XMLConnector** class the **XMLConnectorAccess** interface grants access to the functionalities. This allows to exchange or modify the implementation of the connector without changing the utilising classes.

The **XMLLizer** class implements methods to execute the transformations between an object and a *DOM* representation. This transformation is realised by recursively analysing an object. All attributes of the analysed object that are primitive data types are transformed by an **XMLConnector** instance. Those attributes that are objects themselves are analysed and transformed accordingly. The recursive analysis of objects is based on the *Reflection API* of *Java*. This *API* allows to examine the attributes of an object during runtime and to exclude those attributes that are marked as transient from the transformation. This is comparable to the serialisation mechanism of *Java*. As an *XML* transformation is supported for both directions the *Reflection API* is also used to create objects from their *DOM* representation. Therefore an empty constructor is demanded for all objects that should be transformed. The attributes of the created objects can then be modified with direct *Reflection* access.

Essential for a correct recursive analysis is the detection of cycles in the data structure and an appropriate *XML* mapping. An infinite recursion due to a cyclic data structure leads to an overflow of the call stack. The detection of cycles is implemented by a stack that holds all transformed objects and therefore can be used to exclude already transformed objects from a repeated analysis.

4.4.2 The Configuration Service

Instead of implementing a persistence mechanism for the configuration values in every application of the *LCSP* it is included as a framework in the *System Tier*. The *Configuration Service* enables a central management of configuration values. This is realised by remote methods that persist the *DOM* representation of objects at the location of the *Configuration Service*. The *XML* transformation of objects is done by the framework presented in Subsection 4.4.1. The available remote methods are defined in the **RMIConfigService** interface and implemented in the **RMIConfigServiceImpl** class. Besides the methods that import or export a *DOM* tree the *XML* representation can be accessed directly. Other methods allow to clear or reload the *DOM* representation from file. Important for the *Configuration Service* is the implementation of the *.terminate()* method which ensures that the configuration values are written to disk before the service is ended (compare Subsection 4.1.3).

The remote export and import methods of the *Configuration Service* require a **DOMAccess** conform object. This interface specifies that an implementing class must have methods to support the two-way transformation. This is comparable to the *.writeObject(out)* and *.readObject(in)* methods that specify the serialisation process of an object. The extending **IdentifiedDOMAccess** interface can be used to add a special identifier tag to an object as it is needed in data structures like *Hashtables*. The **XMLConfig** class provides a master implementation of the **DOMAccess** interface. All classes of the *LCSP* that contain configuration values inherit their functionalities from this class. To reflect this dependency the names of those classes start with <`Config`>. The *JavaDoc* *HTML* pages of the configuration classes describe the individual parameters and their usage. The required export and import methods ensure the transformation of an object. Therefore all extending classes are analysed automatically. The **XMLConfig** class administrates the date the configuration was loaded from file. This is necessary to be able to distinguish between a default configuration and a dynamically loaded one. Figure 4.2 demonstrates the relation between the **ConfigConfigService** class and the **XMLConfig** class as well as its default values and static reference. The static reference is used to access the configuration values directly. By using the **ConfigClient** implementation the **XMLConfig** class provides two methods to initiate the export and import of the configuration values. Methods without a parameter use the remote access to the *Configuration Service* to pass the configuration objects to the remote service. Then the remote service uses their **DOMAccess** capabilities to transform these objects and insert or change their *DOM* representation.

The remote service by itself is only responsible for persisting the *DOM* representation centrally. Direct access to the service is granted by the **ConfigServiceClient** class. This class allows to perform all service related tasks that are not included in the **ConfigClient** implementation. The data transformation is done by the *XML* transformation framework that is included by the **XMLConfig** class in every extending object. The core implementation of the centralised configuration management is found in the **ConfigClient** class. As the *Configuration Service* is a remote service its location and access parameters must be specified. Each service must have a local configuration file to store just these access parameters. All other configurations are stored centrally at the *Configuration Service*. Therefore the client needs to manage its local configuration file. If the configuration service is not used all configuration values are stored locally by the client. The name of the local configuration file as well as the utilisation of the *Configuration Service* can be changed by the corresponding methods. These methods are included in the parsing process of the command line (compare Appendix A.1). This parsing is made available to all services/applications by the remote service framework.

The capability of the client to manage a local configuration file is used by the *Configuration Service* to realise the file storage. Apart from the remote methods the client implementation is in fact the service core. The advantage of the client is that every **DOMAccess** conform object which is exported or imported passes the corresponding methods. It makes no difference whether a configuration is kept centrally or locally. This enables the client to have a listing of all used configurations of an application. This capability is included in the remote service framework and allows to modify configuration values without knowing the currently selected storage process of the configuration values of a service.

4.4.3 The Database Storage Framework

The database storage framework was implemented to unify the database access. In the *LCSP* this framework is used to log the hardware communication of the *Control Tier*, to persist the current instrument status and to store the information of the message exchange framework. It is based on the *Java Database Connectivity (JDBC) API* of *Java*. This framework can be divided into three section (see Figure 4.3). These sections provide an interface to the objects that should be stored, prevent data loss and manage the accessed databases, respectively. All objects that should be persisted must implement the **SQLizeable** interface that enables each class to handle its database storage. The *.sqlize(Database)* method is called by the database client that passes the **Database** object that should be used for storage. If an object should be stored that does not implement this interface its string representation is generated and stored instead. Both, the data loss prevention and the database management is implemented in the **DatabaseClient** class.

Objects are written to local files if the database server could not be reached or produces an error during a write access or if the database storage in general is disabled. One part of these files contains the acronymXML transformation of the objects. The transformation is automatically split into parts to guarantee a certain file size. This *XML* storage is demanded e.g., to dump and analyse the hardware communication of the *Control Tier*. The other files contain a queue of **DatabaseObject** objects. These queues are stored with the internal serialisation mechanism of *Java*. If the database client detects that a previously not accessible database server is operational again a new **RebuildThread** object is activated. As every **DatabaseObject** contains the name of the database where the object should be persisted the **RebuildThread** can reload the queue and send each object to the appropriate database. This ensures a fully transparent and reliable data storage.

All available **Database** objects are managed by the **DatabaseClient** object. This is done in a *Hashtable* that maps an identifier to each **Database** object. These identifiers must be specified when calling one of the storage methods of the client. The client supports direct and buffered persisting of objects. If the buffered storage method is used the client stores the object in a queue that is processed later on. This allows the calling thread to continue its work without waiting for the database operation to be finished. Each **Database** object grants direct access to a *JDBC* database connection. Therefore each database has a **DatabaseAccessData** object that specifies the connection modalities. The specification of the correct driver is very important to create a database connection. In the *LUCIFER* project a *MySQL* server is used. Thus the corresponding *MySQL* connector needs to be specified as the driver.

4.5 The Message Exchange Framework and its Service

In a distributed system a centralised message processing is important to track the status of the sub-components. Without such a system the user would be responsible for gathering the information from every service. This would demand the scanning of the log files of every application to obtain the required information. In the *LCSP* the message exchange framework is used by every application to create and transport messages to a central service. This *Message Service* is responsible for persisting the messages and distributing them to the connected client applications.

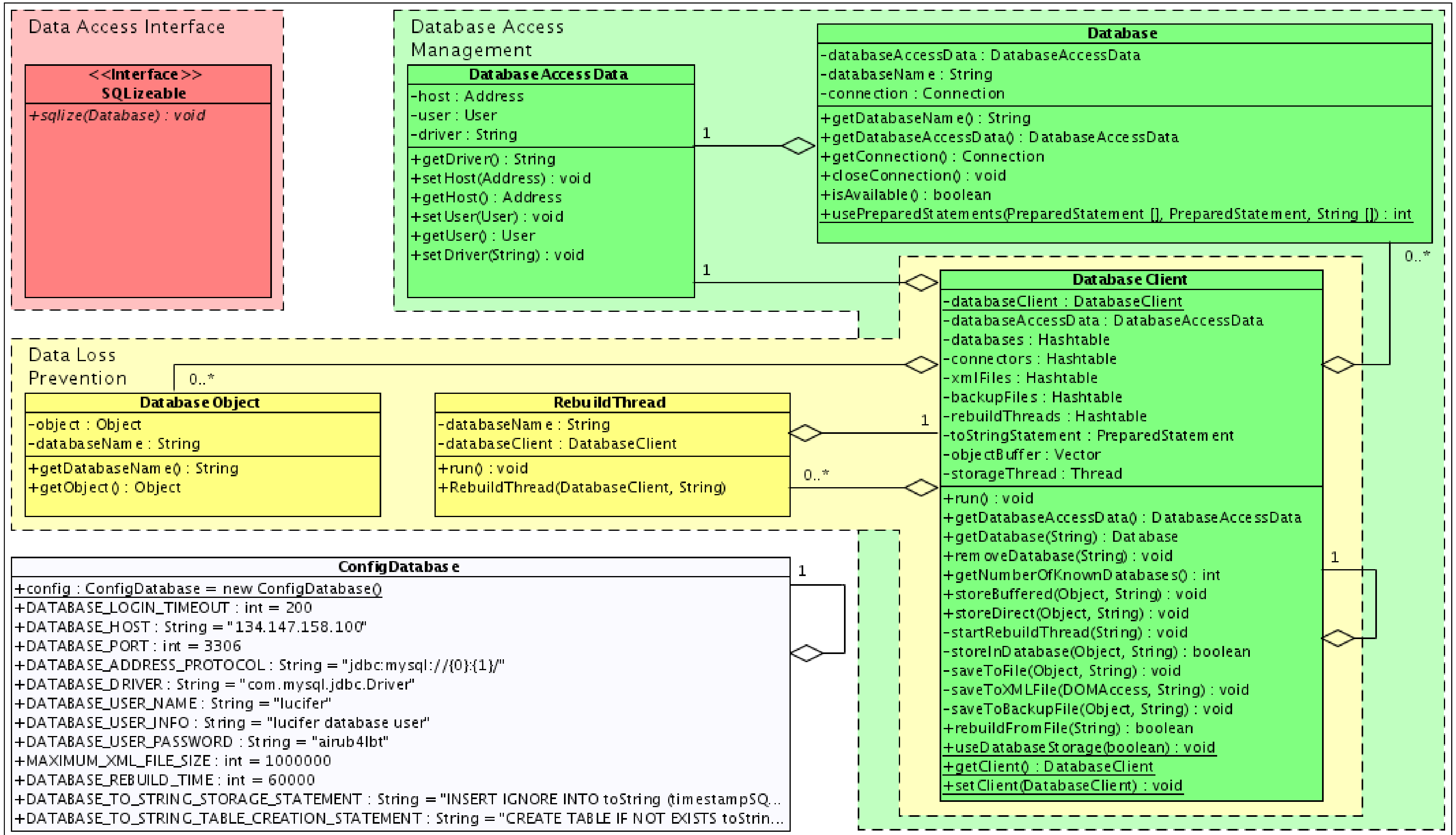


Figure 4.3: Class diagram of the database access framework. RED : The interface that enables *SQL* transformation and storage of objects in a database. YELLOW : The classes that prevent a data loss. GREEN : The classes that manage the databases and their access.

Message Service / Clients
ConfigMessageService
+config : ConfigMessageService = new ConfigMessageSe...
+MESSAGE_SERVER_NAME : String = "MessageServer"
+REGISTRY_PROTOCOL : String = "rmi://{0}:{1}/"
+REGISTRY_HOST : String = "localhost"
+REGISTRY_PORT : int = 1098
<<Interface>>
MessageServerProperties
+ENABLE_FILE_STORAGE : String
MessageServer
-messageServer : MessageServer = new MessageServer()
-registryAddress : Address
+getServerName() : String
+getRegistryAddress() : Address
+interpretArguments(String [], int, Hashtable) : int
+getServer() : ActivatableServer
+getUsageHelpText(String, String, int) : String
+main(String []) : void
CommandLineMessageServiceClient
+getServiceClient() : Client
+main(String []) : void
MessageServiceClient
-messageServiceClient : MessageServiceClient
+addListener(RMIMessageListener, MessageSelector) : void
+removeListener(RMIMessageListener) : void
+containsListener(RMIMessageListener) : boolean
+processMessage(MessageAccess) : void
+isFileStorageEnabled() : boolean
+enableFileStorage(boolean) : void
+getListeners() : Hashtable
+getClient() : MessageServiceClient
ConfigMessageClient
+config : ConfigMessageClient = new ConfigMessageClie...
+MESSAGE_CLIENT_RETRY_INTERVAL : long = 10000
+MESSAGE_CLIENT_SLEEP_TIME : long = 60000
+MESSAGE_CLIENT_GRACE_TIME : int = 1000
+MAXIMUM_MESSAGES_TO_QUEUE : int = 1000
<<Interface>>
RMIMessageService
+processMessage(MessageAccess) : void
+addListener(RMIMessageListener, MessageSelector) : void
+removeListener(RMIMessageListener) : void
+containsListener(RMIMessageListener) : boolean
+isFileStorageEnabled() : boolean
+enableFileStorage(boolean) : void
+getListeners() : Hashtable
RMIMessageServiceImpl
-listenerTable : Hashtable
-listeners : Hashtable
-messages : Vector
-notificationThread : Thread
-isActive : boolean
-isFileStorageEnabled : boolean
-logFile : File
-dateFormat : SimpleDateFormat
-mutex : Mutex
-initialize(MarshalledObject) : void
+startService() : void
+processMessage(MessageAccess) : void
-notifyListener(MessageAccess) : void
+addListener(RMIMessageListener, MessageSelector) : void
+removeListener(RMIMessageListener) : void
+containsListener(RMIMessageListener) : boolean
+getListeners() : Hashtable
+isFileStorageEnabled() : boolean
+enableFileStorage(boolean) : void
-saveMessageInFile(MessageAccess) : void
+getServer() : Server
+terminate() : void
CommandLineListener
-serverReference : RMIMessageService
+processMessage(MessageAccess) : void
+processDisconnect() : void
+main(String []) : void
RMIMessageListenerImpl
+processMessage(MessageAccess) : void
+processDisconnect() : void
<<Interface>>
RMIMessageListener
+processMessage(MessageAccess) : void
+processDisconnect() : void
MessageClient
-messageQueue : Vector
-isSendingMessagesFromQueue : boolean
-isActive : boolean
-messageListener : RMIMessageListener
-messageClient : MessageClient
+activate(boolean) : void
+isActive() : boolean
+sendMessage(MessageAccess) : void
+printMessage(MessageAccess) : void
+hasMessagesInQueue() : boolean
+printMessageQueue() : void
+useMessageService(boolean) : void
+ignoreMessages(boolean) : void
+getClient() : MessageClient
1
0..*

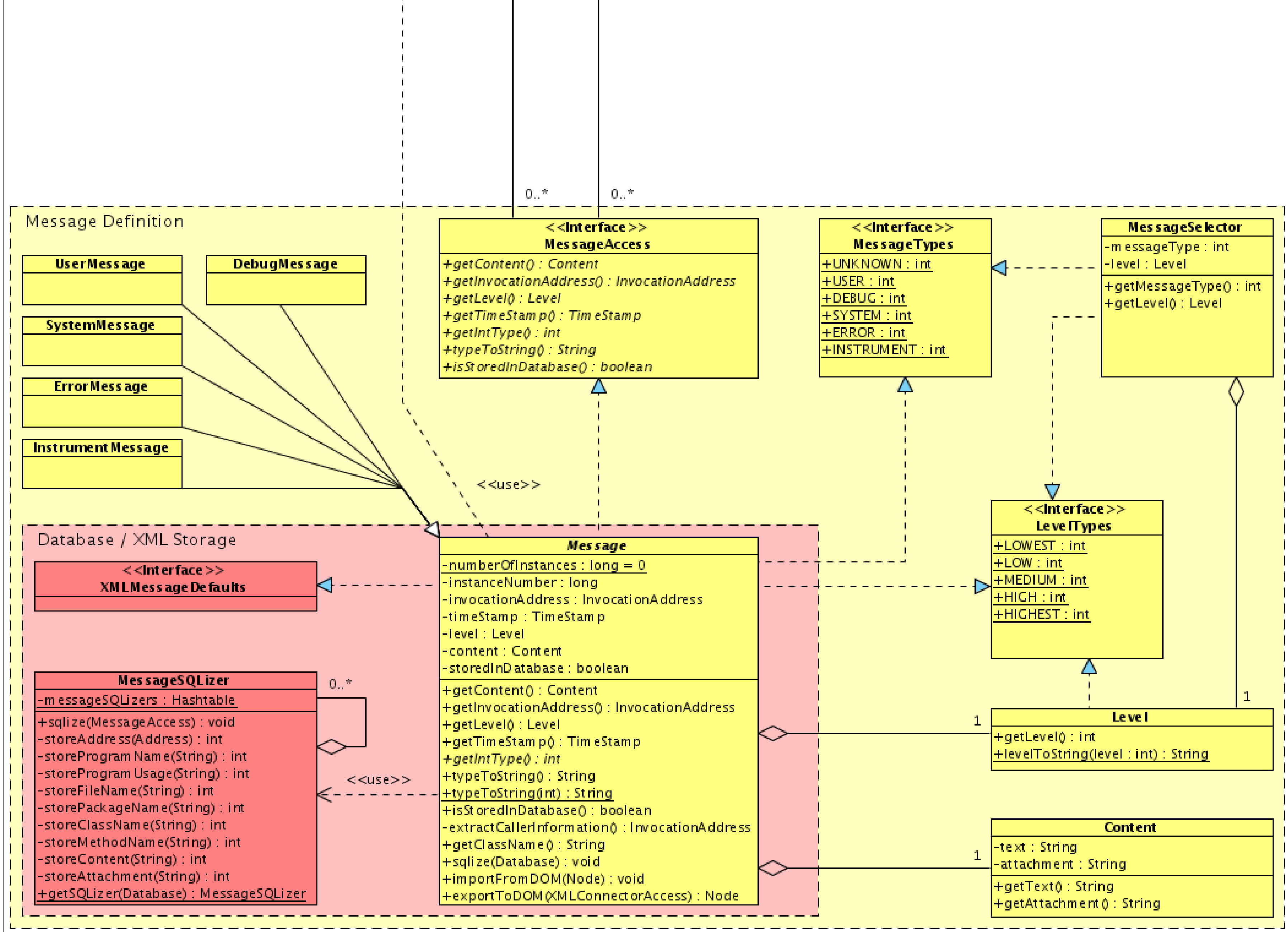


Figure 4.4: Class diagram of the *Message Service*.
RED : The *XML* and database storage of the messages. YELLOW : The classes and interfaces that represent the different messages.
GREEN : The classes and remote interfaces of the *Message Service* and its clients.

Type	Scope	Level
User	general user notifications, e.g., on service status or executed operations	highest to lowest
System	status of the *LUCIFER* system, e.g., of the opto-mechanical units or the calibration unit	medium
Error	description of errors that occur during software execution	highest to lowest
Instrument	information on executed motion sequences of the instrument hardware controlling services	highest to lowest
Debug	additional execution information that is used to maintain the software	highest to lowest

Table 4.1: Overview of the message types and available significance levels.

The message exchange framework is designed to handle several kinds of messages. Therefore the messages are subdivided into 5 classes which use a level to classify their significance (see Table 4.1). In Figure 4.4 the classes that represent the individual messages can be found. All specialised message classes inherit their capabilities from the abstract **Message** class. These classes implement the *.getIntType()* method and return the corresponding type defined by the **MessageTypes** interface. The specialisation of the classes is reduced to the type and the individual constructor. E.g., the **ErrorMessage** class supports only the creation of messages that specify a **Content** object instead of a simple string while the **UserMessage** class provides both options. The **SystemMessage** class does not allow to specify the level because all messages of this type use the medium level. The supported level values are specified in the **LevelTypes** interface while the concrete level of a message is represented by a **Level** object. Each specialised **Message** object has a **Level**, a **TimeStamp**, an **InvocationAddress** and a **Content** object. The **TimeStamp** and the **InvocationAddress** objects are mandatory to store time and place the message was created while the **Content** object stores the information. The constructor of the **TimeStamp** class uses a **TimeClient** instance to generate a time stamp that is synchronised with the *Time Service*. The **InvocationAddress** object is created directly in the **Message** object constructor. This is done by calling a private method that extracts the current process stack information and uses the *Reflection API* of *Java* to create a new **InvocationAddress** object.

The **Message** class itself implements the persistent storage of the messages. Therefore a **Message** object can be transformed automatically into its *XML* representation or can be stored in an *SQL* database. The definition of the *XML* tags is stored in the **XMLMessageDefaults** interface. These definitions are required by the **Message** implementation to create the *DOM* tree that is demanded for the *XML* transformation. The **MessageSQLizer** class that is used by the **Message** implementation contains all methods to store a passed **Message** object in a database. The storage of the messages is based on the generic frameworks presented in Section 4.4. Therefore changes to the data structure require only local adaptions without the need of modifications to the frameworks.

The design of the *Message Service* and its clients is presented in the upper part of Figure 4.4. The central component of this diagram is the **RMIMessageServiceImpl** class that realises the centralised message processing. This is done by receiving the messages from the connected **MessageClient** instances in the distributed applications and by forwarding them to the registered listeners. The **RMIMessageListener** interface defines the functionalities of the listeners that are used for remote call-back methods. These methods provide the ability to process incoming messages and the event of being disconnected from the server. When a listener is connected to the service a **MessageSelector** object is required. This object specifies the type and level of the messages that should be forwarded to the listener. All connected listeners are managed by the service in two *Hashtables*. One of these tables uses the **RMIMessageListener** objects as a key. This is required to support management functionalities. The other table uses a nested mapping to retrieve a vector of all **RMIMessageListener** objects that a registered for a specific message type and level. This mapping increases the speed of finding the appropriate listeners. The connected listeners are notified by an independent thread that consecutively processes the message queue. This is done to ensure an undisturbed processing of incoming messages. The used subscriber paradigm allows to implement an asynchronous communication between the server and the clients as well as to minimise the network traffic.

Besides forwarding the incoming messages to the connected listeners the *Message Service* is responsible for persisting the data. This functionality is realised on basis of the corresponding database storage framework. Every **Message** object provides the functionality to be transformed into *XML* or to be stored in a database. The central *Message Service* uses these functionalities. If the service is ended the *.terminate()* method is invoked. This allows the service to finish processing of messages and to store all queued messages or at least print them to the console.

The **MessageClient** class provides a direct connection to the service. When a new **Message** object is created the constructor sends this object directly to the service by using a **MessageClient** instance. This minimises the programming effort to create a message and process it. As the message creation process should not be dependent on the remote method invocation and corresponding network latencies the messages are stored in a local queue. This message queue of the **MessageClient** object is processed by a separate thread that handles the sending of the messages to the **RMIMessageServiceImpl** remote instance. Additionally the **MessageClient** object holds a reference to an **RMIMessageListener**. This listener is used to process the messages not transmitted if either the queue is full or the application is ending.

The user of the *Message Service* can directly view the messages by using one of the listener applications. This can be either a simple command line message display or a *GUI* application that enables the user to specify filter criteria. The benefit of this approach is that messages can be displayed at the moment they have been created. The disadvantage of an on-line display is that older messages must be dropped and a comprehensive search can not be supported. This is related to the pre-selection of messages by type or level directly at the central *Message Service*. The most powerful access to the messages is granted by the message browser. This browser allows to search all persisted messages in the database (compare Section 3.4.2).

Chapter 5

The Control Tier

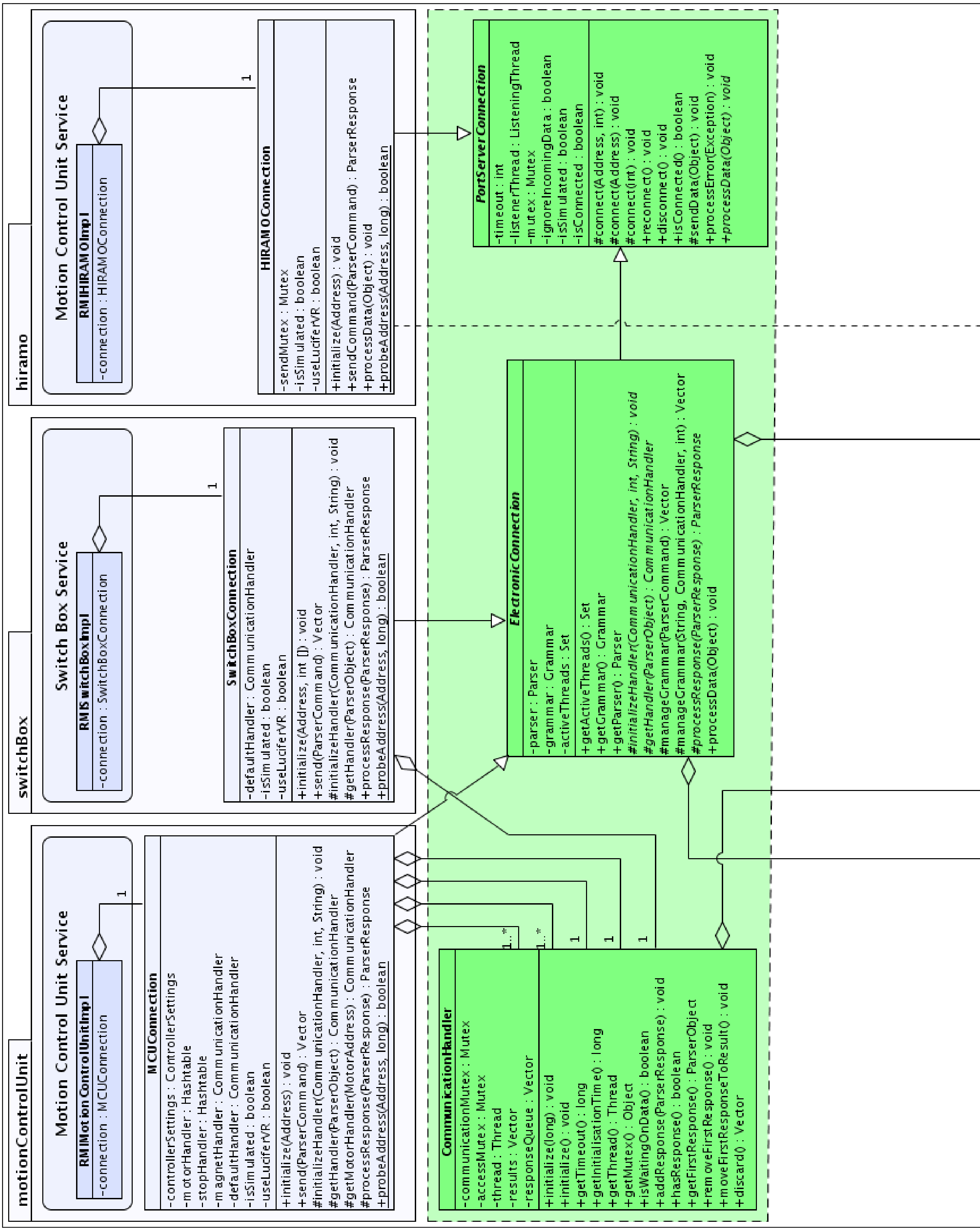
All communication with the hardware of the LUCIFER instrument is handled in the Control Tier. An abstract communication framework provides basic functionalities to exchange serial data with an electronic device. This tier operates the electronics that controls the opto-mechanical parts as well as the hardware that monitors environmental parameters of the instrument. As a central instrument status logging facility the Journalizer Service is presented.

The *Control Tier* is the interface between the instrument hardware and the *LCSP*. It contains the services that operate the electronics. This electronics is used to log and control environmental parameters of the instrument like temperature and pressure. Furthermore the opto-mechanical units of the *LUCIFER* instrument are accessed by electronics. The complex serial command exchange between the controller firmware and the software is covered by the services of the *Control Tier*. These services exhibit simple but powerful interfaces to the services of higher tiers (compare Section 3.2).

5.1 The Serial Communication Framework

The electronics of the *LUCIFER* instrument is equipped with serial *RS232* ports. These ports are connected to a port server that provides direct access to the serial ports via the *TCP/IP* network protocol. A documentation of this device is presented in Lehmitz (2008c). The **PortServerConnection** class is the core element of the serial communication framework. This class realises all functionalities that are required to connect to a serial port of the port server hardware. It includes the methods to connect/disconnect and to send data. As the hardware communication is asynchronous a **ListeningThread** object handles the incoming data and calls an abstract data processing method of the connection. This method must be implemented by every concretised connection class to perform the data processing. Primarily the **PortServerConnection** class implements the **Listener** interface to define the interaction with the **ListeningThread** object. If a concretised connection class implements the **IntegerListener** interface instead, the transmitted data is treated bytewise instead of doing an *ASCII* interpretation. This is required by some of the controlled electronics. Every data that is exchanged between the hardware and the software passes one of these classes. Therefore the data can be encapsulated in **Transmission** objects and stored in an *XML* file or *SQL* database. These **Transmission** objects provide additional information on the source/target and time of a data package. By implementing the logging capability in the core classes of the serial communication framework all data exchange over a socket connection can be analysed.

The central port server hardware that connects to the electronics is covered by the *Port Server Service*. This allows to centrally manage the mapping between the ports and


motionControlUnit
Motion Control Unit Service
RMIMotionControlUnitImpl
-connection : MCUConnection
MCUConnection
-controllerSettings : ControllerSettings
-motorHandler : Hashtable
-stopHandler : Hashtable
-magnetHandler : CommunicationHandler
-defaultHandler : CommunicationHandler
-isSimulated : boolean
-useLuciferVR : boolean
+initialize(Address) : void
+send(ParserCommand) : Vector
#initializeHandler(CommunicationHandler, int, String) : void
#getHandler(ParserObject) : CommunicationHandler
#getMotorHandler(MotorAddress) : CommunicationHandler
#processResponse(ParserResponse) : ParserResponse
+probeAddress(Address, long) : boolean
switchBox
Switch Box Service
RMISwitchBoxImpl
-connection : SwitchBoxConnection
SwitchBoxConnection
-defaultHandler : CommunicationHandler
-isSimulated : boolean
-useLuciferVR : boolean
+initialize(Address, int []) : void
+send(ParserCommand) : Vector
#initializeHandler(CommunicationHandler, int, String) : void
#getHandler(ParserObject) : CommunicationHandler
+processResponse(ParserResponse) : ParserResponse
+probeAddress(Address, long) : boolean
hiramo
Motion Control Unit Service
RMIHIRAMOImpl
-connection : HIRAMOConnection
HIRAMOConnection
-sendMutex : Mutex
-isSimulated : boolean
-useLuciferVR : boolean
+initialize(Address) : void
+sendCommand(ParserCommand) : ParserResponse
+processData(Object) : void
+probeAddress(Address, long) : boolean
PortServerConnection
-timeout : int
-listenerThread : ListeningThread
-mutex : Mutex
-ignoreIncomingData : boolean
-isSimulated : boolean
-isConnected : boolean
#connect(Address, int) : void
#connect(Address) : void
#connect(int) : void
+reconnect() : void
+disconnect() : void
+isConnected() : boolean
#sendData(Object) : void
+processError(Exception) : void
+processData(Object) : void
ElectronicConnection
-parser : Parser
-grammar : Grammar
-activeThreads : Set
+getActiveThreads() : Set
+getGrammar() : Grammar
+getParser() : Parser
#initializeHandler(CommunicationHandler, int, String) : void
#getHandler(ParserObject) : CommunicationHandler
#manageGrammar(ParserCommand) : Vector
#manageGrammar(String, CommunicationHandler, int) : Vector
#processResponse(ParserResponse) : ParserResponse
+processData(Object) : void
CommunicationHandler
-communicationMutex : Mutex
-accessMutex : Mutex
-thread : Thread
-results : Vector
-responseQueue : Vector
+initialize(long) : void
+initialize() : void
+getTimeout() : long
+getInitialisationTime() : long
+getThread() : Thread
+getMutex() : Object
+isWaitingOnData() : boolean
+addResponse(ParserResponse) : void
+hasResponse() : boolean
+getFirstResponse() : ParserObject
+removeFirstResponse() : void
+moveFirstResponseToResult() : void
+discard() : Vector
1
1
1
1..*
1..*
1
1
1

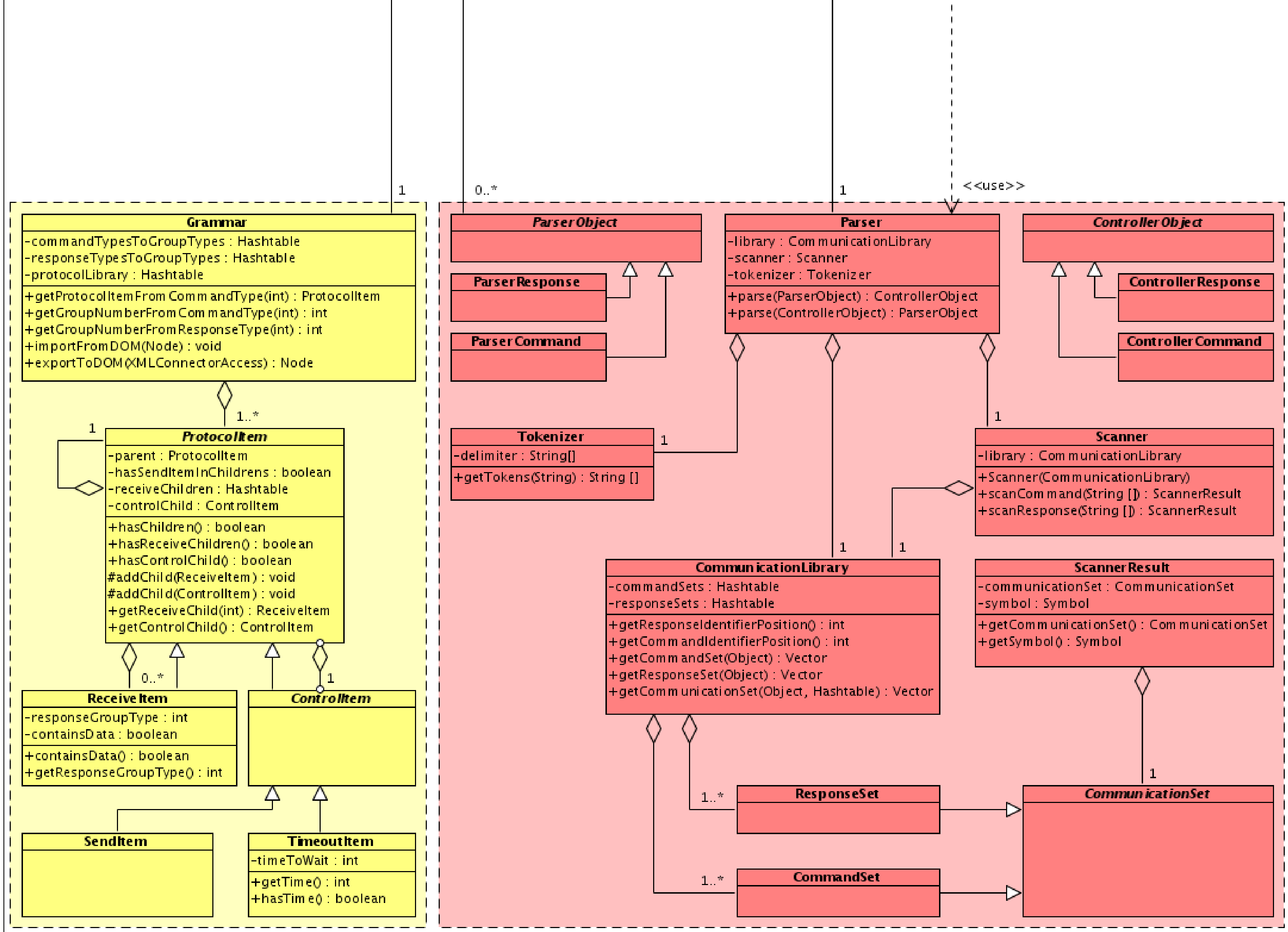


Figure 5.1: Class diagram of the control electronics services. The upper part covers the service implementations and the communication classes while the lower part represents the command analysing framework. GREEN : hardware connection and communication handling/synchronisation. YELLOW : processing of a grammar to realise a proper execution of commands, RED : command and response string parsing/generation.

the attached electronics. In case of a changed cabling only the mapping of this look-up service needs to be updated while the individual services can stay unchanged. Nonetheless the **PortServerConnection** class can still be used to create direct connections by specifying a socket address. For use with the virtual instrument it is even possible to create connections that are just simulated and do not establish a connection at all.

5.2 The Control Electronics Services

The opto-mechanical elements of the *LUCIFER* instrument are controlled by three electronic boxes. These are the *MCU*, the *Switch Box* and the *HIRAMO* electronics (compare Subsection 1.3.3). All three electronic boxes are custom-made by the institutes that contribute to the project. The firmwares of the applied micro-controllers use a similar communication scheme. Therefore all services make use of a common framework that provides all the functionalities for analysing and composing command or response strings. A first version of this framework was developed in cooperation with Andreas Zeh (see Zeh, 2005). The relation between the three control services and the command analysing framework is shown in Figure 5.1.

5.2.1 The Command Analysing Framework

The command analysing framework handles the communication with the control electronics. It consists of three parts: One to realise the connection with the hardware and to synchronise the communication, one to parse and create communication strings and one to allow the processing of a predefined chain of actions (see Figure 5.1). The connection with the hardware is based on the serial connection framework. Therefore the generic **ElectronicConnection** class extends the **PortServerConnection** class. Additionally the functionalities of both other parts are incorporated into this generic connection class. This allows to use the **ElectronicConnection** class for hardware communication as well as for parsing and creating communication strings and processing a predefined communication protocol. Besides the classes for direct hardware communication the **CommunicationHandler** class is mandatory to communicate with the control electronics. The inter-service-communication was chosen to be synchronous. As the electronics uses an asynchronous communication protocol all commands need to be synchronised to fit into the service communication model. Therefore the responses of commands that are executed in parallel need to be allocated to the corresponding command calls. In addition each handler can store a timeout value. This is necessary to notify and to wake-up the threads that initiated the communication calls. Without such a command synchronisation, parallel execution of commands would not be available and e.g., the *MCU* could only move the motors consecutively.

The second part of the command analysing framework contains all elements to allow parsing and creating command strings. Over the development period of the *LUCIFER* project the syntax, the structure and the semantics of the hardware communication strings changed several times. Therefore a flexible and configurable interface to the hardware was developed. The **Parser** class provides capabilities to translate between a software side and a hardware dependent representation of commands/responses. Two groups of objects exist to interact with the **Parser** class: These are the **ParserObject** and the **ControllerObject** classes. The **ParserObject** class and its descendants are the software-side representation of commands/responses that are sent to/received from the control electronics. These objects can be used to access commands/responses without knowing the actual conversion on the hardware-side. On the other side the **ControllerObject** class and its descendants contain the hardware communication strings. These response and command strings can be transformed into their **ParserObject** counterparts and vice versa by using a **Parser** object. To realise this two-way-parsing the **Parser** object uses a **Tokenizer** object, a **CommunicationLibrary** object and a **Scanner** object. The **Tokenizer** class defines methods to split a string into tokens by using predefined delimiter symbols. The **CommunicationLibrary** class contains methods to manage sets of commands/responses. These sets contain the information how a **ControllerObject** object is composed from the data of a hardware-independent **ParserObject** object. If a **ControllerObject** is passed to the **CommunicationLibrary** all sets that match the represented command/response string are returned. These sets can be used to create the required **ParserObject** representation. In the other direction a passed **ParserObject** object will result in the sets with the information to create a string representation. As the creation of a **ParserObject** is more complex than the creation of a string representation the **Scanner** class provides the required methods. The **Scanner** class allows to do a backward scan of a **CommunicationLibrary** object for the occurrence of a communication string. Thereby it is important to know which parts of the communication string contain variable data and which parts can be used for identification. This information is encoded in the communication sets and needs to be evaluated during scanning. The results of a search are represented by **ScannerResult** objects that contain the **CommunicationSet** object that is required to finally create a **ParserObject** instance.

The last part to interact with the electronics is the processing of a predefined grammar. Every interaction with the hardware consists of at least two communication activities. A command is sent and a response from the hardware is received. The grammar is used to model the individual communication protocols of the commands. Each of these communication protocols is represented by a tree of **ProtocolItem** objects that can be subdivided into **ReceiveItem** objects and **ControlItem** objects. **ControlItem** objects are used for actions like sending a string to the hardware or supervised waiting for a response string to be able to create a time out and prevent from endless sleeping. **ReceiveItem** objects are used to define the further processing of a protocol in dependency of the received hardware responses. All communication protocols are stored in a **Grammar** object. To allow an efficient access to the protocol trees the items are grouped by their command/response types. The processing of a grammar, e.g., the stepwise execution of a protocol tree, is done in the **ElectronicConnection** class. Thereby the complexity of creating/parsing commands/responses and the correct execution of the protocol is hidden from the electronics services.

Both the parser as well as the grammar are configured via *XML* files. This allows fast adaption to changes of the protocol, e.g., to the order of actions, as well as changes in creating/parsing the commands without touching the source code. To add a new command its syntax needs to be put in the communication set description and a protocol has to be assigned. After the configuration has been modified the new functionality can be accessed through the **ElectronicConnection** class.

5.2.2 The MCU Service

Both *MCU Services* are the central element of the *Control Tier* (compare Figure 3.1). They are responsible for accessing the control electronics of all opto-mechanical parts of the *LUCIFER* instrument. A detailed description of the applied electronics is presented in Lehmitz (2008d). As the instrument and *MOS* electronics only differ in their configuration, both services are realised by one implementation. The *MCU Services* grant access to the corresponding control electronics over an *RMI* interface. The **RMIMotionControlUnitImpl** class is the implementation of this interface. It allows to individually or synchronously move the stepper motors, to stop the motion of elements, to power the motors, to control the magnetic locks, to enquire the orientation/position of elements and to read the status of micro-switches. Another important functionality of the **RMIMotionControlUnitImpl** class is the ability to stop and prevent any kind of motion. For this reason the **RMIMotionControlUnitImpl** class keeps track of all ongoing motions in order to be able to issue the required emergency stop commands.

The **RMIMotionControlUnitImpl** class provides simple methods that hide the complexity of the whole control process. E.g., the **RMIMotionControlUnitImpl**.*moveMotor (motorAddress, steps)* method does several tasks automatically. First this method ensures that it is allowed to move the specified motor. Therefore it checks if the *MCU* is approved to move and that the selected motor is currently not moving. While the calling thread is paused the communication with the electronics is handled. If within a pre-calculated time the control electronics does not report the motion as finished, exceptions are thrown and the waiting thread is notified. After a motion is completed the connected micro-switches are evaluated and if necessary further actions are accomplished.

While the **RMIMotionControlUnitImpl** provides the access to the functionalities of the *MCU* electronics, the **MCUConnection** class contains the implementation of the *MCU* interaction. This class is an extension of the **ElectronicConnection** class (see Figure 5.1). To synchronise the execution of motion a hashtable of **CommunicationHandler** objects exists. Each movable unit is synchronised separately with its own **CommunicationHandler** object. Additional **CommunicationHandler** objects exist to synchronise setup, stop and query commands. To use the correct handler, the **MCUConnection** class contains a method for a command-dependent look-up. The timeout values that can be stored in the **CommunicationHandler** objects are calculated by the **MCUConnection** class, too. This is done by using *LuciferVR* (compare Chapter 8). The **MCUConnection** class realises an interface to the real hardware. Hence this class can be assigned to send the commands to the virtual instrument. This allows to switch between a simulated and a real instrument usage.

Both the configuration of the grammar and the communication sets, required by the **ElectronicConnection** class, and the configuration of the electronics is stored in a joined *XML* file. The configuration of the electronics is evaluated on every service start-up, transformed into command strings and sent to the hardware. If a hardware reset or communication failure is detected the setup of the electronics is re-sent automatically. This ensures a functional and reproducible state of the electronics.

5.2.3 The Switchbox Service

The *Switch Box Service* provides access to the switches of the *MOS Unit*, to the orientation sensors of the instrument and to the status of the strain gauges of the mask grabber (see Subsection 1.3.3). As the hardware allows to trace the status of a switch and detect cabling/contact problems of each switch, the service must be able to report both, switch setting and fault status. To support changes in cabling, the service can be configured to change the assignment of the switches. The service is even able to ignore faulty switches to allow an emergency operation of the instrument. To simplify the management of more than 100 switches, each switch is either assigned to a motor or a mask of the *MOS Unit*. This enables the *Switch Box Service* to look-up switches dynamically by an identifier and report the status to the utilising services of the *Instrument Tier*.

The *Switch Box Service* is designed similarly to the *MCU Service*. The **RMISwitchBoxImpl** class is the implementation of the corresponding *RMI* interface, while the **SwitchBoxConnection** class realises all hardware interactions. Only one **CommunicationHandler** object is required for command synchronisation, because the complexity of the interaction with the *Switch Box* electronics is much simpler than the one of the *MCU* electronics. Apart from the initialisation of the strain gauges, the *Switch Box Service* is only retrieving the status of switches/sensors.

Likewise as the **MCUConnection** the **SwitchBoxConnection** class provides the option of interacting with *LuciferVR*. Its communication configuration and switch assignment is stored in an *XML* file, too.

5.2.4 The HIRAMO Service

Of all services that interact with the instrument hardware, the *HIRAMO Service* is the most compact one. It was developed by Volker Knierim (see Knierim, 2009). As this service only sends simple commands and receives responses that represent 16 switches and the orientation of the instrument cabling, the **ElectronicConnection** is not used. No complex communication protocol needed to be implemented and therefore the **Grammar** class and it functionalities is not utilised. No synchronisation of concurrent commands via **CommunicationHandler** objects is required, too. The **HIRAMOConnection** class implements a direct connection to the electronics by using the underlying serial communication framework. This implementation detail was used to integrate an interface to the virtual instrument.

5.3 The Interfacing Services to External Packages

Besides operating the *LUCIFER* instrument the *LCSP* needs to interact with external software packages. In the *Control Tier* these are the telescope software (*TCS*) and the detector readout package (*GEIRS*) (compare Figure 3.1). To include these software packages in the service structure of the *LCSP*, additional services are required that transfer the command calls from the *RMI* interface side to command calls on the external software package side. Both interfacing services that connect to existing software packages have not been developed in the scope of this thesis.

The *Telescope Service* uses an *ICE* interface to send commands to the telescope control software. By wrapping the method calls to the *IIF*, an additional interface layer is introduced. This layer hides the details of executing the *ICE* calls from the *LCSP* services (see Subsection 3.4.4). The detector readout software *GEIRS* has a socket based interface. Tobias Muhlack implemented a service that wraps the corresponding command strings into *RMI* methods of the *Readout Service*. These methods allow to specify all required parameters like readout mode/configuration, integration time or file name and to perform an integration. See Muhlack (2006) for more details on the available functionalities. Due to a missing software interface in the readout software package, all information that is required to create the *FITS* header has to be written to an external, plain text file. After the readout process, this file is scanned by *GEIRS* and the included data is stored together with the detector data (see Subsection 3.4.3). This exchange of meta-data should be part of the *Readout Service* as well. Currently the header information is however generated by the *Journalizer Service*.

5.4 The Environment Supervising Services

Besides actively controlling the opto-mechanical parts of the *LUCIFER* instrument, the *LCSP* has to monitor and control its environment. These services are part of the *Control Tier*, too.

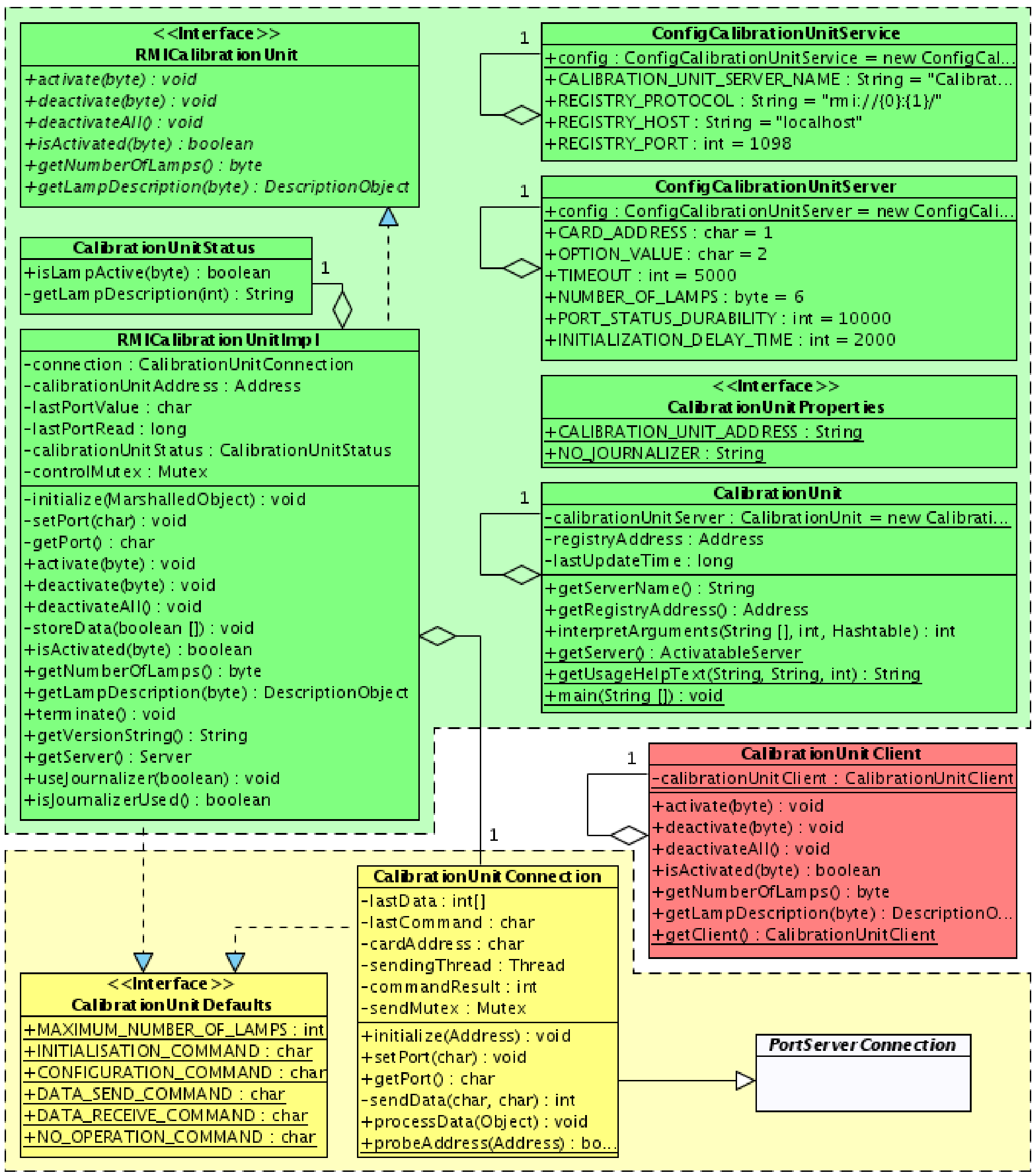


Figure 5.2: Class diagram of the *Calibration Unit Service*. GREEN : Classes that realise the calibration service. YELLOW : Hardware communication via **PortServerConnection** class. RED : Client access to the service.

5.4.1 The Calibration Unit Service

The *Calibration Unit Service* is the service of the *Control Tier* that was developed first. It is responsible for interacting with the electronics that controls the lamps required for spectral calibration of *LUCIFER*. The layout of the electronics is documented in Lehmitz (2008a) while the operation of the mechanical parts is presented in Lehmitz (2008b). The applied electronics board is able to control 8 relays. It has a serial port that is attached to the port server which provides *TCP*/*IP* access (see Subsection 1.3.3). In Figure 5.2 the classes of the *Calibration Unit Service* are presented. The upper part of the *UML* diagram contains the classes that implement the remote service. The **RMICalibrationUnit** interface defines the methods that are accessed via *RMI*. These are the methods to switch calibration lamps on and off as well as the methods to enquire the current lamps status or to retrieve a description of a lamp as a **DescriptionObject** entity. The **RMICalibrationUnitImpl** class contains the core of the service. The implementation of the previously mentioned methods is part of this class. Additionally this class uses a **CalibrationUnitStatus** object to reflect the current status of the lamps. As the **CalibrationUnitStatus** extends the **JournalizerObject** class all changes of the lamp status are logged centrally by the *Journalizer Unit* (see Section 5.5). More details on the logging process are presented in Section 5.5. The structure of a remotely accessible service can be found in Section 4.1.

The communication with the electronics board is based on a proprietary command language that uses non-*ASCII* characters. All commands are composed of bytes that specify the command, the card address, the parameters and a parity check byte. Therefore the **RMICalibrationUnitImpl** class uses a **CalibrationUnitConnection** instance to handle the communication. This class is responsible for translating all incoming and composing all outgoing byte sequences. Analogue to the implementation of the *MCU Service* the **PortServerConnection** class is extended. By implementing the **IntegerListener** interface the underlying serial communication framework is notified to process byte streams instead of *ASCII* streams. The limited number of commands that needs to be exchanged with the electronics does not require the usage of the command analysing framework that is presented in Subsection 5.2.1. Instead the 5 command bytes are constantly defined by the **CalibrationUnitDefaults** interface and the 2 possible parameters are specified in the **ConfigCalibrationUnitServer** class and therefore can be configured via the *Configuration Service* (see Section 4.4.2). The hardware of the calibration unit is only able to process the commands consecutively. For this reason the **CalibrationUnitConnection** class uses just one *mutual exclusion (mutex)* object to synchronise the outgoing commands.

The access to the *Calibration Unit Service*, either by a *GUI* or by other services, is made available by the **CalibrationUnitClient** class. This class handles all tasks that are required to remotely call the methods defined by the **RMICalibrationUnit** interface.

5.4.2 The Services for Temperature Monitoring and Controlling

The *LUCIFER* instrument is cryogenically cooled down to 60 K to reduce its thermal interference with the observed scientific targets. For this reason the temperatures of important elements of the instrument need to be monitored and the detector and its fanout board need to be thermally stabilised. This is done with two electronics from *LakeShore Cryotronics*, the 218 S-model temperature monitor and the 331 S-model temperature controller. Details are summarised in Lehmitz (2006). Both electronics use the same command language which allows to reuse the software of the *Temperature Monitor Service*. This service was developed first. The *Temperature Control Service* was developed later as a copy of the *Temperature Monitor Service*. The required modifications are not part of this thesis.

The class diagram in Figure 5.3 presents the *Temperature Monitor Service*. Its design is similar to the design of the *Calibration Unit Service*. It is divided into the classes that create the service, handle the connection details and provide client access to the service. The **RMITemperatureMonitor** interface and the **RMITemperatureMonitorImpl** class realise the functionalities of the service. These are the ability to measure temperatures, to send initial configuration commands and to set alarm parameters. The temperature retrieval can be done directly or buffered. Direct temperature retrieval forces the **RMITemperatureMonitorImpl** to communicate with the hardware and to return the current values. The temperatures that are received from the hardware are locally stored in an **InstrumentTemperatures** object. The **InstrumentTemperatures** class extends the **Journalize-**

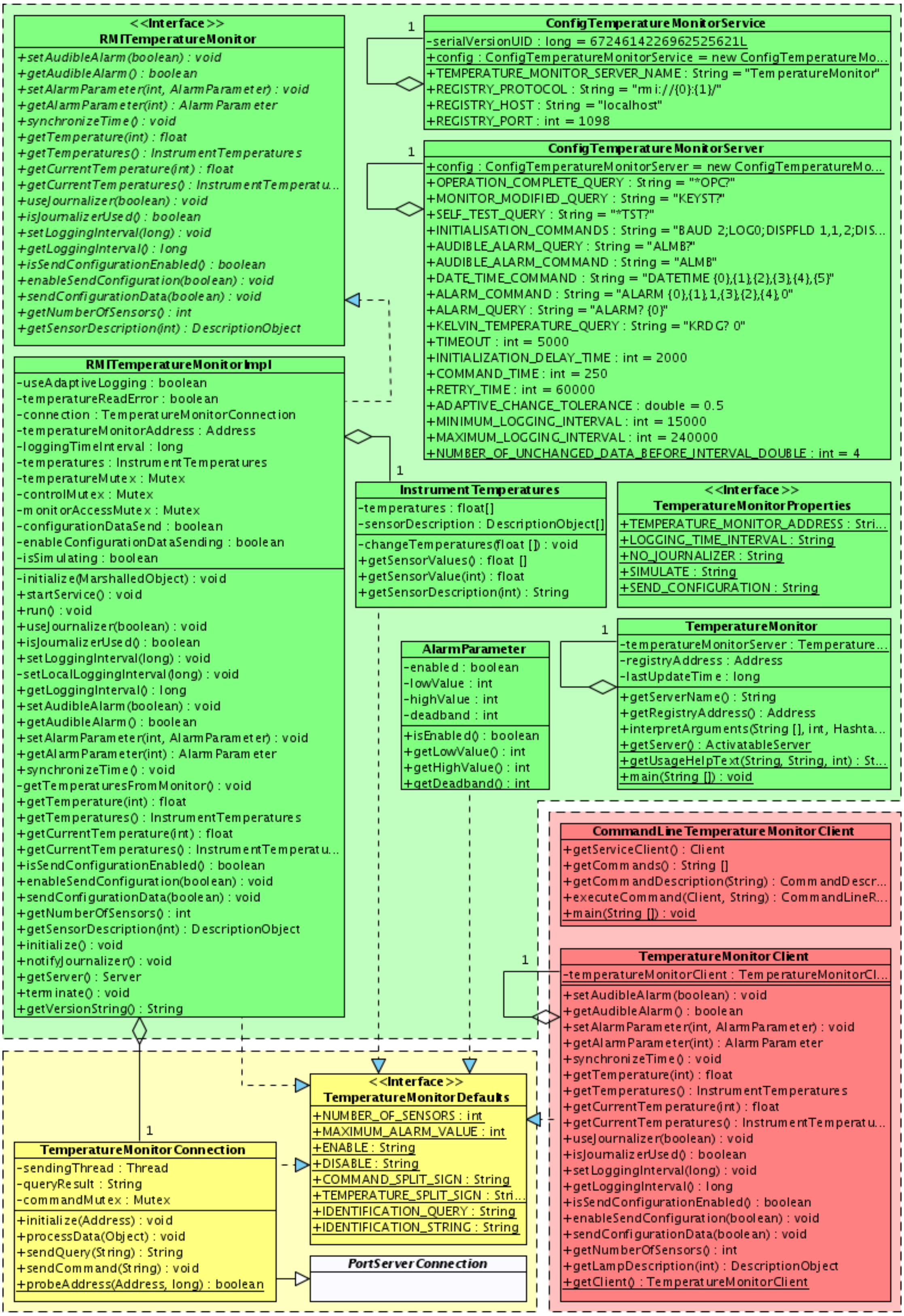


Figure 5.3: Class diagram of the *Temperature Monitor Service*. GREEN : Classes that implement the service. YELLOW : Hardware communication via **PortServerConnection** class. RED : Client access to the service.

rObject class to support processing by the *Journalizer Service* (see Section 5.5). Buffered temperature retrieval returns these stored values instead. This reduces the amount of commands that is sent to the hardware. To minimise the effects of obsolete temperatures, the *Temperature Monitor Service* waits for a predefined time and refreshes its values automatically. The logging interval of the *Temperature Monitor Service* can be changed during operation. If this value is set to a value that is below 1 ms, adaptive logging is used. This logging proceeding monitors the variation of the temperatures. If the temperatures stay constant within a configured tolerance for a defined number of measurements, the length of the measuring interval is doubled. In case the values exceed the allowed tolerance level this interval is halved. The maximum and minimum time between two measurements can be configured, too.

The communication with the electronics is based on an *ASCII* command language that uses mnemonics to increase readability. Both electronics are connected to the port server that is accessed via *TCP*/*IP*. On the software-side the communication with the electronics is done by the **TemperatureMonitorConnection** class that extends the **PortServerConnection** class of the serial communication framework. Only one command can be processed by the hardware in parallel. Therefore only one *mutex* is required to synchronise the command calls.

In contrast to the *Calibration Unit Service*, the client access is done with two classes. The **TemperatureMonitorClient** class implements the general access that is used by other services or a *GUI*. Additionally the **CommandLineTemperatureMonitor Client** class allows a command line interaction with the *Temperature Monitor Service*.

5.4.3 Other Environment Supervising Services

All other environment supervising services have been built following the design of the services presented above. Even though their implementation does not belong to the scope of this thesis, for the sake of completeness their duties and responsibilities is presented hereafter.

The Pressure Monitor Service

To be able to cool *LUCIFER* down to temperatures of ≈60-70 K, the dewar needs to be evacuated to $\approx 10^{-7}$ bar. Therefore the pressure inside of the instrument needs to be monitored. This is done via a TPG 262 pressure monitor that is manufactured by *Pfeiffer*. The description of the pressure monitor hardware is part of Lehmitz (2008a). The *Pressure Monitor Service* provides the functionality of logging the instrument pressure and sending an initial configuration to the hardware.

The Turbo Pump Service

Once the pressure inside of the *LUCIFER* instrument is reduced with a scroll pump it is sustained via a turbo molecular pump. The *Turbo Pump Service* connects to the control electronics of the control hardware, sends an initial configuration and monitors the revolution velocity of the connected turbo pump. This monitoring is done with a TCM 1601 controller from *Pfeiffer* that controls the magnetic bearing of the pump. More details on the controller is specified in Lehmitz (2008e).

The Rack Cooling Control Service

The *Rack Cooling Control Service* interacts with the electronics that controls the temperatures of the electronics racks. Those electronics racks are mounted on the telescope structure right below the instruments. To prevent the thermal interference with the observations the wast heat is dissipated with water pipes. The rack cooling control electronics ensures a stable temperature within the racks. Therefore the water flow must be regulated and for very low ambient temperatures even be switched off. The temperature measurement and regulation is done by the Imago 500 from *Jumo* (see Lehmitz, 2008f). The *Rack Cooling Control Service* initialises the regulation parameters and loggs the rack temperatures. Like all other environment services of the *Control Tier* the retrieved values are stored by the *Journalizer Service*.

5.5 The Journalizer

In an instrument control software, one of the fundamental tasks is to track the instrument status (see Requirement [U6], Section 3.1). In case of the *LCSP* a distributed system controls the instrument. Several services exist that independently control the sub-systems of the instrument. The status of the *LUCIFER* instrument is monitored by the services of environment supervising services of the *Control Tier*. The other services of the *Control Tier* that communicate with the control electronics and with the calibration unit are responsible for changes of the instrument setup. Therefore these services can either be used to move opto-mechanical parts or enquire their position. The motion of the individual optical elements of *LUCIFER* was found to be too complex to be solved monolithically. For this reason individual services for each optical element have been created. These are the services of the *Instrument Tier*. Their interaction with the instrument hardware is based on the services of the *Control Tier*.

To synchronise the amount of services that work in parallel, a central logging service was developed. The *Journalizer Service* accepts incoming status change notifications from the services mentioned above. Besides the logging the *Journalizer Service* provides the instrument status information to the services of the *Operation Tier*. The *Readout Service* was not implemented in a correct manner. It should be changed to retrieve this information, too. This would solve the problem with the temporary solution of creating the *FITS* header in the *Journalizer Service*.

All services that make use of the *Journalizer Service* should implement the **Journalizable** interface. By doing so these services can be triggered to update the part of the instrument setup they monitor. During standard operation the services that observe environmental parameters automatically notify the *Journalizer Service* on a regular basis. The services of the *Instrument Tier* change the setup of the opto-mechanical elements autonomously. Therefore these services know when to notify the *Journalizer Service*. Figure 5.4 presents the class diagram of the *Journalizer Service*. It is divided into three parts. The first part contains all elements that build the service. The **RMIJournalizer** interface and its implementation define and realise the status storage and retrieval methods that are accessed via *RMI*. The other classes are used to start, configure and run the service. The **JournalizerProperties** interface defines a new keyword. This keyword is passed to the created virtual machine that hosts the remote instance of the service and defines the *Hibernate* configuration file. In the second part the client implementation is realised. For the *Journalizer Service* it is important to store all changes of the instrument setup. Therefore the client implementation that is used by the other services of the distributed system needs

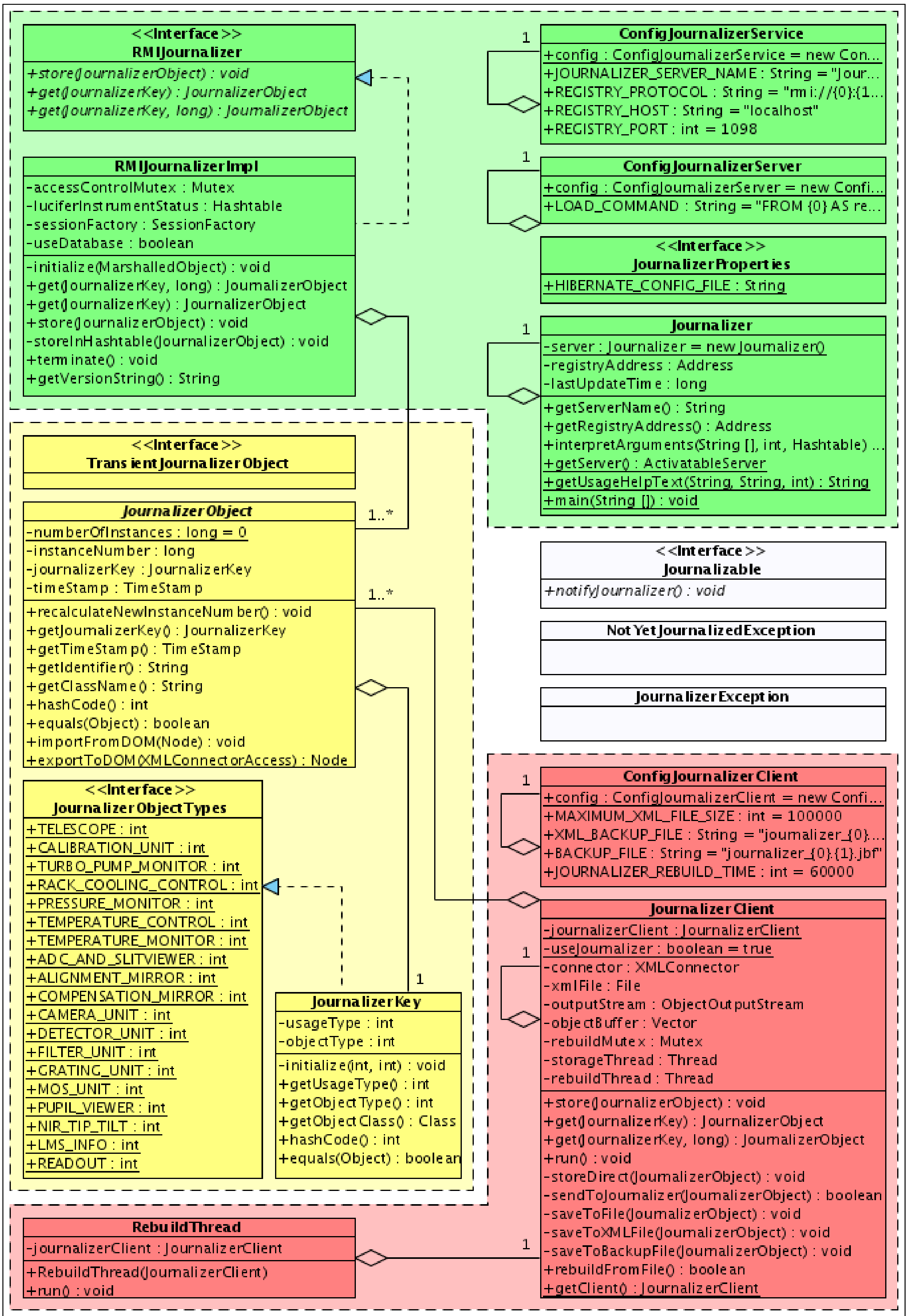


Figure 5.4: Class diagram of the *Journalizer Service*. GREEN : Classes of the instrument status logging service. YELLOW : Abstract data structure to implement instrument status reporting in *LCSP* services. RED : Failsafe client access to the service.

to be fault-tolerant. In case of a broken connection to the *Journalizer Service* all status changes must to be stored locally on disk by the client. If later the connection can be re-established an instance of the **RebuildThread** class uses a **JournalizerClient** instance to rebuild the status logs. This ensures that a status is never lost even though its processing can be delayed. Therefore the **RMIJournalizerImpl** class must evaluate the time stamp of every received **JournalizerObject** entity to store the statuses accordingly in the database and to be able to represent the current instrument status. The last part contains the abstract data structure that has to be extended by all logging services to reflect their part of the instrument status. The **JournalizerObject** class contains all basic data that is required to be processed by the **RMIJournalizerImpl** class. These are the time the object was created and a unique key. A service that sends data to the *Journalizer Service* just needs to extend the **JournalizerObject** class and add its own status information attributes. Then all handling and storing is done by the **RMIJournalizerImpl** class without knowing the precise implementation details. Additionally the **JournalizerObject** class provides the functionality of automatically transforming any instance of an extending implementation in an *XML* representation. This is done by using the **XMLLizer** class of the *XML* transformation framework (see Subsection 4.4.1). As the **JournalizerObject** objects are stored by the **RMIJournalizerImpl** class in a hashtable, a key is required to retrieve them. This key is defined by the **JournalizerKey** class. Such a key consists of a usage and an object information. The usage information specifies where the information belongs to. This can be either an identifier to discriminate between both instruments or it can specify a system or engineering usage. The object type information of a key represents the individual part of the *LUCIFER* instrument. The allowed values are predefined by the **JournalizerObjectTypes** interface.

Besides these three parts the **JournalizerException** class and the **NotYetJournalizedException** class define the exceptions that can occur during status storage or retrieval. These two classes are drawn separately because the defined exceptions are used in both, the service and the client implementation.

A first base release of this service was created by VOLKER KNIERIM similarly to other simple services (see KNIERIM, 2009). The full service functionality including the persisting of the instrument status to an *SQL* database was implemented in the *Journalizer Service* in the scope of this thesis. Additionally the *Journalizer Service* was enabled to rebuild the instrument status from database for any given time. This allows the service to initialise its status during service start-up and it can be used to trace a previous instrument status, too. Even though this service does not know the data structure of the status information of each individual service, the persisting is done centrally at the *Journalizer Service*. The database access is realised with the *Hibernate* framework (see Subsection 3.4.1). This framework allows to directly exchange objects with a database server. The only requirement is the presence of a mapping information that enables *Hibernate* to translate between data stored in tables and a *Java* data structure. In Appendix E such a mapping is presented.

CHAPTER 6

The Instrument Tier

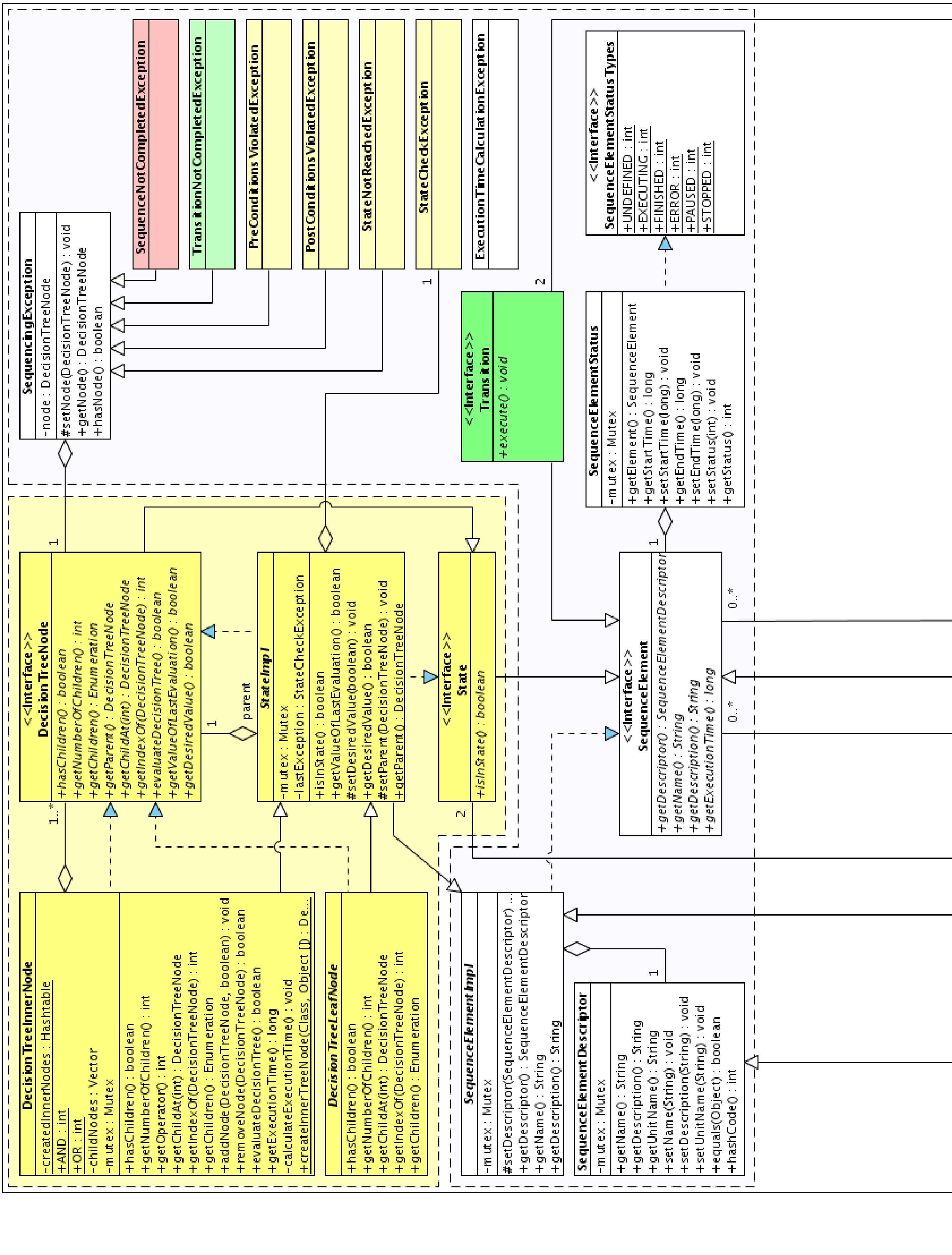
The Instrument Tier contains the services that control the opto-mechanical parts of the LUCIFER instrument. All motions are modelled as finite state transition networks. The framework that handles the execution of these motion sequences is presented first. In addition the representation of states and transitions is described. The service of the most complex part of LUCIFER, the MOS Unit Service, is presented in concepts. To document this service and its functionalities comprehensively its subunits are depicted separately.

To allow observations with the *LUCIFER* instrument, the opto-mechanical parts of the instrument must be brought into specific states. It is necessary to change the optical setup with regard to the scientific objective to achieve. This includes to select a wavelength range by placing filters in the optical beam. These filters are stored in two filter wheels. Additionally the spectral and spatial resolution is changed via the *Grating Unit* and the *Camera Wheel*. The most important part of *LUCIFER* is the *MOS Unit* that allows to select different long-slit masks for spectroscopy and to use custom-made *MOS* masks. The other opto-mechanical parts are used for instrument checks, detector focusing and compensation of disturbing effects like structural distortion or atmospheric effects.

All motions of the optical elements are controlled by the hardware interfacing services of the *Control Tier* (see Section 5.1). The services of the *Instrument Tier* rely on the provided functionalities of these underlying services which are accessed by the corresponding clients. The motion of an opto-mechanical part demands more than a simple motion command that is sent to the electronics. In most cases it is composed of a complex series of actively moving and position probing commands. The task of the *Instrument Tier* is to model the logics of these motion sequences and to implement simple interface functions that can be access by the services of the *Operation Tier*. For the *Filter Wheel Unit* that can rotate limitless the logics are very simple. In contrast, the *MOS Unit* has to exchange a free mask between the cabinet and the focal plane. This exchange involves motions of many parts with possible collision risks that might severely damage the instrument. Therefore a huge set of very complex logics is required (compare Subsection 3.5.2). A dedicated service exists for every opto-mechanical part of the instrument (see Section 3.2). To keep these service implementations simple the common tasks of managing the complex motion sequences are bundled in a framework.

6.1 The Sequencing Framework

The sequencing framework is designed to model motion sequences as finite state transition networks. In such a network actions transfer an element from one state into another. The definition of sequences as a transfer between states allows to compose very complex sequences from other sequences or basic prime actions. This can be used to reduce the


SequencingException
-node : DecisionTreeNode
#setNode(DecisionTreeNode) : void
+getNode() : DecisionTreeNode
+hasNode() : boolean
SequenceNotCompletedException
TransitionNotCompletedException
PreConditionsViolatedException
PostConditionsViolatedException
StateNotReachedException
StateCheckException
ExecutionTimeCalculationException
<<Interface>>
SequenceElementStatusTypes
+UNDEFINED : int
+EXECUTING : int
+FINISHED : int
+ERROR : int
+PAUSED : int
+STOPPED : int
<<Interface>>
Transition
+execute() : void
SequenceElementStatus
-mutex : Mutex
+getElement() : SequenceElement
+getStartTime() : long
+setStartTime(long) : void
+getEndTime() : long
+setEndTime(long) : void
+setStatus(int) : void
+getStatus() : int
<<Interface>>
DecisionTreeNode
+hasChildren() : boolean
+getNumberOfChildren() : int
+getChildren() : Enumeration
+getParent() : DecisionTreeNode
+getChildAt(int) : DecisionTreeNode
+getIndexOf(DecisionTreeNode) : int
+evaluateDecisionTree() : boolean
+getValueOfLastEvaluation() : boolean
+getDesiredValue() : boolean
StateImpl
-mutex : Mutex
-lastException : StateCheckException
+isInState() : boolean
+getValueOfLastEvaluation() : boolean
#setDesiredValue(boolean) : void
+getDesiredValue() : boolean
#setParent(DecisionTreeNode) : void
+getParent() : DecisionTreeNode
<<Interface>>
State
+isInState() : boolean
<<Interface>>
SequenceElement
+getDescriptor() : SequenceElementDescriptor
+getName() : String
+getDescription() : String
+getExecutionTime() : long
parent
DecisionTreeInnerNode
-createdInnerNodes : Hashtable
+AND : int
+OR : int
-childNodes : Vector
-mutex : Mutex
+hasChildren() : boolean
+getNumberOfChildren() : int
+getOperator() : int
+getChildAt(int) : DecisionTreeNode
+getIndexOf(DecisionTreeNode) : int
+getChildren() : Enumeration
+addNode(DecisionTreeNode, boolean) : void
+removeNode(DecisionTreeNode) : boolean
+evaluateDecisionTree() : boolean
+getExecutionTime() : long
-calculateExecutionTime() : void
+createInnerTreeNode(Class, Object [], De...
DecisionTreeLeafNode
+hasChildren() : boolean
+getNumberOfChildren() : int
+getChildAt(int) : DecisionTreeNode
+getIndexOf(DecisionTreeNode) : int
+getChildren() : Enumeration
SequenceElementImpl
-mutex : Mutex
#setDescriptor(SequenceElementDescriptor) ...
+getDescriptor() : SequenceElementDescriptor
+getName() : String
+getDescription() : String
SequenceElementDescriptor
-mutex : Mutex
+getName() : String
+getDescription() : String
+getUnitName() : String
+setName(String) : void
+setDescription(String) : void
+setUnitName(String) : void
+equals(Object) : boolean
+hashCode() : int
1
1..*
2
0..*

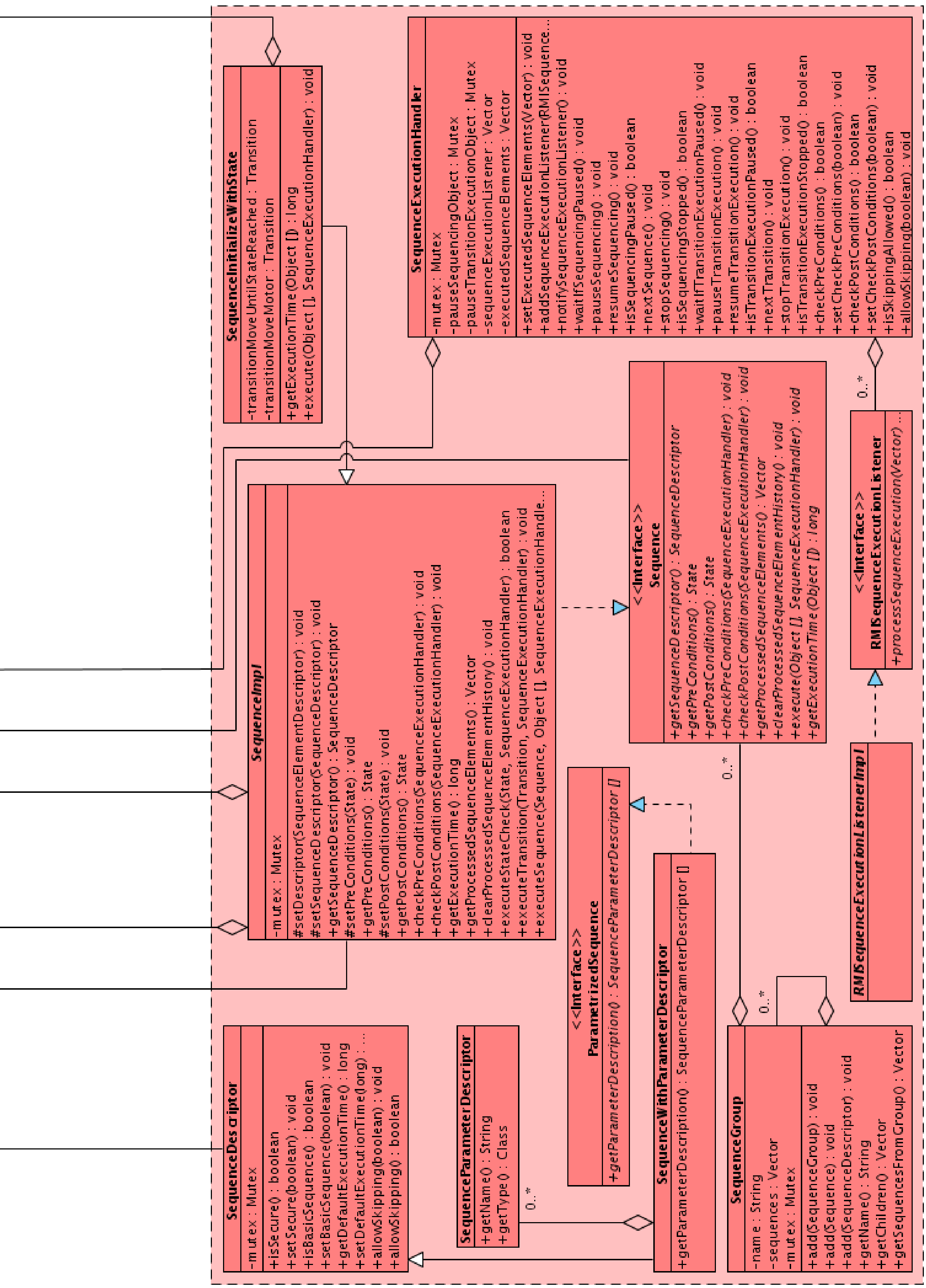


Figure 6.1: Class diagram of the sequencing framework. WHITE: The abstract definition of a sequence element and its sequencing exceptions. GREEN: The interface and exception used to define and execute transitions. YELLOW: The classes and interfaces that allow to define a state as a complex decision tree. RED: Basic sequence definition and execution classes and the **Sequence** interface.

complexity of a sequence and therefore improve their maintainability. Another benefit of using finite state transition networks to model the motion logics is the ability to define start and end states. These states are utilised to execute checks before and after every sequence and therefore increase the execution reliability. No motion is started if an element is not in the predefined state. As soon as a motion is ended malfunctions can be detected. In such a case further motions are prevented and the involved risk of damage is reduced.

The elements of the sequencing framework class diagram (see Figure 6.1) are divided into three groups. First there are the classes and interfaces that define and execute sequences. Second is the interface that defines access to the transitions. Finally there are the interface and classes for state representation which are used as pre- and post-conditions of the sequences. The elements of all these groups are based on fundamental elements of the sequencing framework. The central element is the **SequenceElement** interface that defines the core functionalities of every component of a sequence. These are the methods to retrieve description data for improving the feedback of information to the user during sequence execution. The description data is encapsulated in a **SequenceElementDescriptor** object and contains a name, a basic description string and additional data that allows to create a link to an opto-mechanical unit. The most important function of every **SequenceElement** implementation is the ability to pre-calculate the time required for execution. This information can be used to estimate the approximate time a complete sequence will last and to inform the user in advance. Combined with functionalities of *LuciferVR* to pre-calculate motion times of active elements an upper limit is presented to the engineer. This feature was heavily used during instrument integration and is important for system recovery measures. If an error during execution time calculation occurs an **ExecutionTimeCalculationException** is thrown. The **SequenceElementImpl** class contains an abstract implementation of the **SequenceElement** interface and is extended by the root elements of the groups previously described. During sequence execution the status of a **SequenceElement** conform entity is reflected by a **SequenceElementStatus** object. Besides the plain status information such an object stores the start and end time of the execution. This data is evaluated to calculate the time actually required for execution.

6.1.1 The State Representation and Evaluation

The definition of states is elemental to model state transition networks. In Figure 6.1 the required classes and interface are grouped together. The central element to define states is the **State** interface that extends the **SequenceElement** interface by a method to execute a state evaluation. The **StateImpl** class provides an abstract implementation of this interface. All descriptive functionalities are inherited from the **SequenceElementImpl** class. Methods to store and retrieve the desired Boolean value as well as the value of the last state check are implemented, too. The access to the methods of the **StateImpl** class is synchronised with a *mutex* object to prevent overlapping evaluation which could result in inconsistent checks. If an evaluation of a state fails a **StateCheckException** is generated. Additionally the last exception that occurred is referenced by the **StateImpl** class.

To reduce the complexity of a concretised state implementation, states can be composed of other states. This nesting of sub-states allows to create large decision trees. The **DecisionTreeNode** interface defines the access to the elements of such a decision tree. The defined methods are used to navigate within the tree structure and to evaluate the Boolean state of a sub-tree. To support the tree structure a method to get the parent node is implemented in the **StateImpl** class. The realisation of the **DecisionTreeNode** interface is split into two versions. One to cover the requirements of a leaf node and another that represents an inner node of a decision tree. Both, the **DecisionTreeLeafNode** and the **DecisionTreeInnerNode** class extend the **StateImpl** class to inherit basic state functionalities. The abstract **DecisionTreeLeafNode** class misses a method to get the time that is required for state evaluation. Therefore the concrete implementation of a state is forced to specify this. In the exact same way every concretised state is compelled to implement the **DecisionTreeNode**.*evaluateDecisionTree()* method to evaluate its Boolean state. The **DecisionTreeInnerNode** class is the composing element of the decision trees. It allows to combine sub-states with a Boolean operator. Therefore the implementation of the **DecisionTreeNode**.*evaluateDecisionTree()* method evaluates the Boolean state of every child recursively and combines their results with the specified operator. The same depth-first search is applied to the evaluation time calculation process. Special for the **DecisionTreeInnerNode** class is that new entities of inner nodes are created with a static factory method. This ensures that only one instance of every inner node exists no matter how many trees make use of this node. Another benefit of this global node creation is to have an overview of all created states and their last evaluation value. This functionality is used to improve the engineering process by providing detailed status information of the instrument.

During sequence execution the check of a state is done in the **SequenceImpl** class. This allows to detect errors during state evaluation and to throw corresponding exceptions. As a sequence transfers an instrument unit from one state to another it uses these two states as its pre- and post-conditions. The check of both states is an integral part of the sequence execution process. Discrepancies between the anticipated state and the detected state of a unit directly result in the creation and throwing of **PreConditionsViolatedException** and **PostConditionsViolatedException** objects.

6.1.2 A Transition as a Prime Action

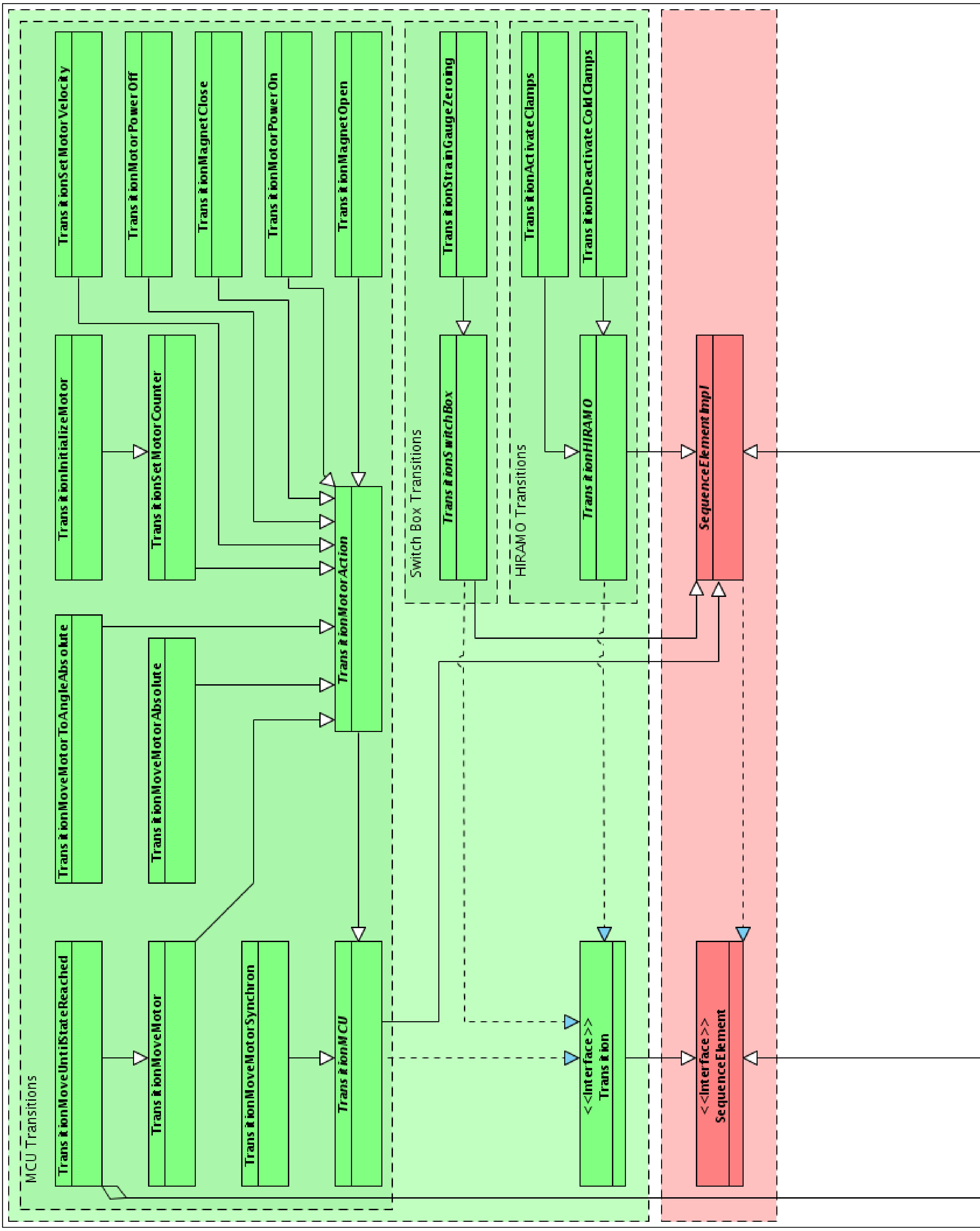
The transitions are the active elements of a finite state transition network. They realise the prime actions that transfer the opto-mechanical parts of *LUCIFER* from one state to another. Therefore the **Transition** interface contains only one method. This method is called to execute an action. As every action requires a different time to execute, the calculation of this time has to be done by the implementing classes

Because the sequences are composed of the transitions, the execution of these prime actions is done by the **SequenceImpl** class. All transitions that are implemented extend the **SequenceElementImpl** class and inherit the ability to reflect their status. The status of a transition is changed accordingly during execution within the **SequenceImpl** class. If such a transition produces an error during the execution, a new **TransitionNotCompletedException** object is created and therefore must be handled by the utilising sequence.

6.1.3 The Basic Transitions and States

The sequencing framework contains several concrete implementations of commonly used transitions and states. These basic transitions and states as well as their inheritance topology are presented in Figure 6.2.

The transitions are used to actively change the state of a unit. To be able to do so the transitions require access to the hardware interacting services of the *Control Tier*. This is done by using the corresponding client implementations of these services. With regard to the three available hardware interfaces, the basic transitions are split into groups, respectively. Every root element of these groups implements the **Transition** interface and


MCU Transitions
TransitionSetMotorVelocity
TransitionMotorPowerOff
TransitionMagnetClose
TransitionMotorPowerOn
TransitionMagnetOpen
TransitionInitializeMotor
TransitionSetMotorCounter
TransitionMotorAction
TransitionMoveMotorToAngleAbsolute
TransitionMoveMotorAbsolute
TransitionMoveUntilStateReached
TransitionMoveMotor
TransitionMoveMotorSynchron
TransitionMCU
Switch Box Transitions
TransitionStrainGaugeZeroing
TransitionSwitchBox
HIRAMO Transitions
TransitionActivateClamps
TransitionDeactivateColdClamps
TransitionHIRAMO
<<Interface>>
Transition
SequenceElementImpl
<<Interface>>
SequenceElement

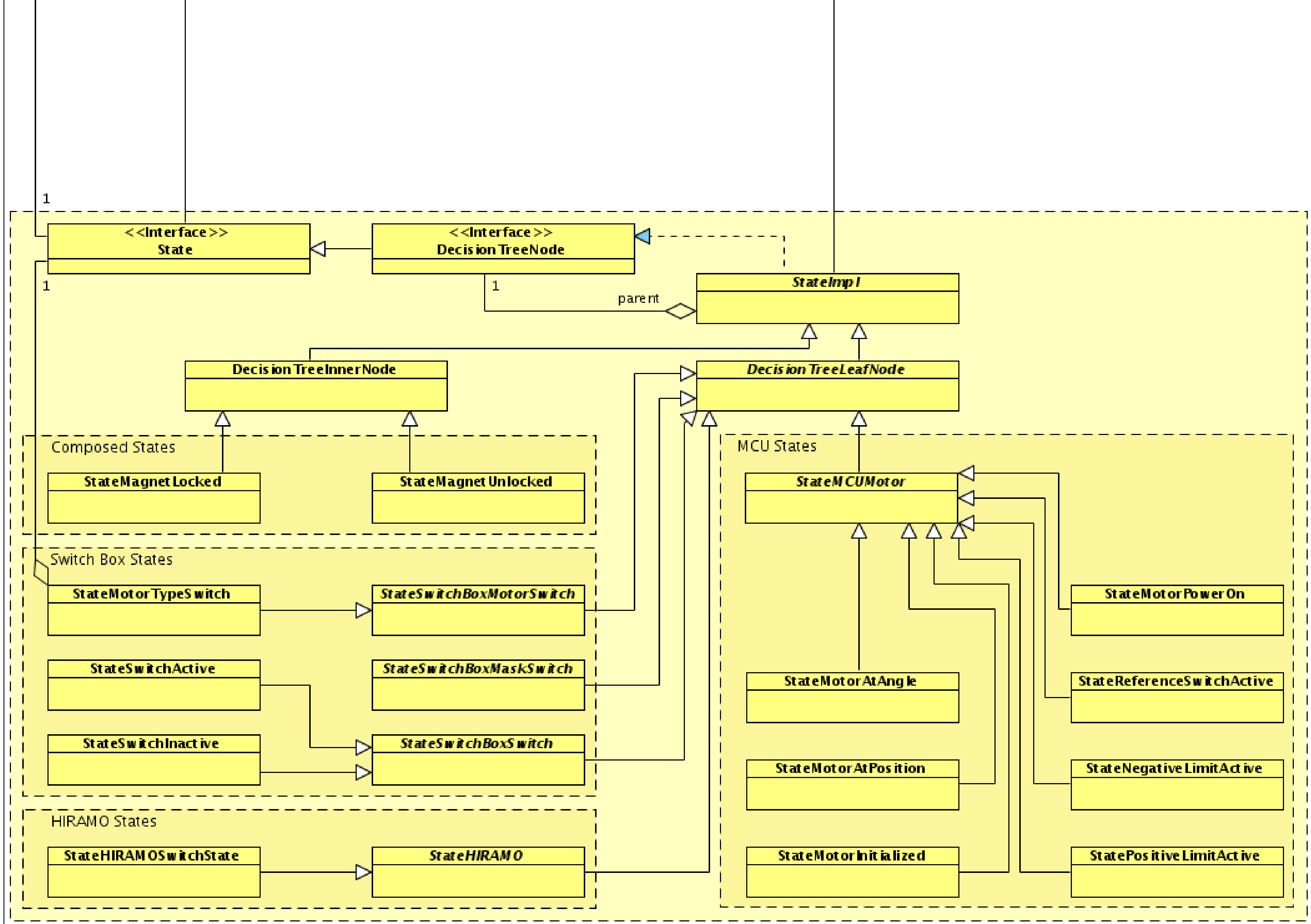


Figure 6.2: Class diagram of the basic transitions and states that are part of the sequencing framework. GREEN : The elementary transition defining interface and the classes of all fundamental transitions combined with their inheritance hierarchy. YELLOW : The classes and interfaces that define the states as hierarchical decision trees. Additionally all basic states and their inheritance relations are presented. RED : Abstract definition of a sequence element as root of all transitions and states.

adds additional methods to keep a reference on the required client. The group of transitions which is used to access the *Switch Box* is the simplest one. Its only concretised implementation allows to zero the strain gauges used for the mask grabber of the *MOS Unit*. The transitions of the *HIRAMO* group implement methods to control the cold clamps that are used to increase the cool-down time for some optical parts. The largest group of transitions is used to access the *MCU*. The **TransitionMoveMotorSynchron** class defines all methods to drive several stepper motors synchronously. Therefore methods to keep the references of the employed motors are part of this implementation. All other transitions of this group can be associated to only one motor. For this reason the abstract **TransitionMotorAction** class specifies the method to store the address of the used motor. The classes that extend the **TransitionMotorAction** class contain concretised implementations of the *MCU Service* functionalities. These are the methods (i) to move a motor by a specified amount of steps, (ii) to an absolute position or (iii) to an absolute angle. The amount of steps can be specified as an array to model more complex successive motions. For both absolute motions an allowed delta can be specified as well as an amount of correction tries. When the position of a motor is found to be outside of the predefined limits the specified number of retries is executed automatically to reach the required value. A similar concept is used by the **TransitionMoveUntilStateReached** class to reach a given state. The array of steps is executed as long as the end state is not reached. Another speciality of the absolute angle motion transition is the ability to specify several angles. When specified the transition chooses the angle that is closest to the current position of the motor. Directly connected to the motion transitions are those transitions that initialise or set the internal position counter of the electronics or set the speed. The remaining basic transitions control the locking magnets and the power characteristics of a motor after a motion. Important for the creation process of a transition object is that all parameters must be specified during instantiation. This parametrisation makes every object unique and allows to execute a transition without passing values.

Corresponding to the definition of basic transitions an implementation of fundamental states exists. These states are grouped similarly to the transitions. Besides the three groups that use the clients of the three hardware services an extra group contains composed states. The states of this group extend the **DecisionTreeInnerNode** class and combine several switches of the *Switch Box* to represent the state of a locking magnet. As these magnets are associated to a motor the corresponding address must be specified during state creation. All other basic states extend the **DecisionTreeLeafNode** class. The concrete state of the *HIRAMO* group is used to access the state of a switch which is monitored by the hardware. The state of switches that are connected to the *Switch Box* is represented by several implementations. The only difference in the implementation is the way of addressing a switch. A switch can be either associated to a motor or to a mask of the *MOS Unit*. Additionally the **StateSwitchBoxSwitch** class and its descendants can be used to evaluate the state of a switch directly. Corresponding to the addressing options, different constructors are provided for the states of the *Switch Box*. The states that utilise the client of the *MCU Service* are used to test the limit and reference switches that can be connected to every stepper motor. The power characteristics as well as the initialisation state of a motor are reflected by state implementation, too. The initialisation state of a motor is important to be able to evaluate the position of a motor correctly. In case of a power loss at the control electronics, the internal position counter is set to zero. Same things happen if the *MCU Service* is restarted. The resetting of the hardware at service start is required to bring the electronics in a predefined state. All changes at the electronics that modify the internal position counter need to be detected. If modifications are detected further absolute motions of an uninitialised motor can be prevented by using the corresponding initialisation transition and state. Then the absolute position information of a motor can be used reliably. The parametrisation of states is done in the constructor of the objects. Therefore the inspection of the states does not require to pass values.

6.1.4 The Execution and Inspection of Sequences

Both, transitions and states are combined to create sequences. The **Sequence** interface defines the methods that have to be implemented in all motion sequences. Besides the default methods of the **SequenceElement** interface there are four other methods to retrieve and check the pre- and post-conditions of a motion sequence (see Figure 6.1). The **PreConditionsViolatedException** and **PostConditionsViolatedException** classes define the exceptions that are used by the check methods. There are additional methods to execute the sequence and calculate the required execution time. To improve the readability and debugging of a sequence, methods have to be implemented that manage the history of the processed sequence elements. All the required methods are implemented by the **SequenceImpl** class. As this class extends the **SequenceElementImpl** class all basic methods are inherited. Only the calculation of the execution time has to be overwritten. An instance of the inherited **SequenceElementDescriptor** class is used to store an object of the extending **SequenceDescriptor** class. This allows to store additional description information for every **SequenceImpl** object. This information is used to configure the execution behaviour of a sequence. The default execution is supplemented by the ability to skip sequences. If skipping is activated the post-condition is evaluated first and if the state is found to be reached the embedded elements of the sequence are not executed. Directly connected to this functionality is the ability to disable the mandatory pre-condition checks for composed sequences. This should not be done for sequences that are basic and contain transitions. A disabled pre-condition check for basic sequences can damage the instrument. Finally the **SequenceDescriptor** entity can be used to declare a sequence as secure. By definition safe means that a sequence can be executed in any configuration of the instrument because the used pre- and post-conditions are restrictive enough to ensure safe operation. For sequences that require parameters for execution the **ParametrizedSequence** interface and the corresponding **SequenceWithParameterDescriptor** class allow to retrieve an array of **SequenceParameterDescriptor** objects. This description of parameters is mainly used for user interaction. It enables the software to present an appropriate description of every sequence parameter. Based on the parameter description, the type conformance of the passed parameters can be checked, too. Another construct to improve user interaction is defined by the **SequenceGroup** class. Such a group is used to bundle sequences with common objectives. Additionally every group can contain sub-groups and therefore realise a whole tree of sequences. For the *MOS Unit* the groups are used to organise the sequences unit-wise.

The abstract **SequenceImpl** class provides methods to execute a state check, a transition and a sequence. Those methods manage the whole complexity of execution, including the status changes of the currently processed objects that are conform with the **SequenceElement** interface. If an error during one of these methods occurs a **SequenceNotCompletedException**, a **TransitionNotCompletedException** or a **StateCheckException** object is thrown, respectively. The last exception of the group of classes that inherit their functionalities from the **SequencingException** class is the **StateNotReachedException** class. This class is used to create exceptions if an additional state check, which is nested in the sequence execution, fails.


MaskData
+getRetainerSelectionAngle() : double
+setRetainerSelectionAngle(double) : void
+getTranslatorPosition() : int
+setTranslatorPosition(int) : void
+getTranslatorAngle() : double
+setTranslatorAngle(double) : void
+isUsed() : boolean
+setUsed(boolean) : void
+validInPlaceSwitch() : boolean
+validInPlaceSwitch(boolean) : void
+validOpenSwitch() : boolean
+validOpenSwitch(boolean) : void
+validFocalPlaneXReedContact() : boolean
+validFocalPlaneXReedContact(boolean) ...
+validFocalPlaneYReedContact() : boolean
+validFocalPlaneYReedContact(boolean) ...
+getId() : String
+setId(String) : void
+getDescription() : String
+setDescription(String) : void
+getLMSFileName() : String
+setLMSFileName(String) : void
33
ConfigMOSUnit
+config : ConfigMOSUnit = new ConfigMOSUnit()
+PASSWORD : String = ""
+MHU_TRANSLATOR_OPEN_POSITION : int = -2916
+MHU_TRANSLATOR_OPEN_ANGLE : double = 186.7
+MHU_TRANSLATOR_HOLD_POSITION : int = -2582
+MHU_TRANSLATOR_HOLD_ANGLE : double = 71.4
+MHU_TRANSLATOR_INITIALIZE_ANGLE : double = 18...
+MHU_TRANSLATOR_PARK_POSITION : int = -6916
+MHU_TRANSLATOR_NEGATIVE_LIMIT : int = -25000
+MHU_TRANSLATOR_INITIALISATION_POSITION : int ...
+MHU_TRANSLATOR_ONE_TURN_OUT_OF_LIMIT : int ...
+MHU_TRANSLATOR_TOLERANCE : int = 5
+MHU_TRANSLATOR_ANGLE_TOLERANCE : double = ...
+RETAINER_SELECTION_ALL_POSITION : double = 13...
+RETAINER_SELECTION_TOLERANCE : double = 0.3
+SEQUENCE_MHU_TRANSLATOR_INITIALIZE_OPEN_ST...
+SEQUENCE_MHU_PICKER_ARMS_OPEN_STEPS : int[] = ...
+SEQUENCE_MHU_PICKER_ARMS_CLOSE_STEPS : int[] ...
+SEQUENCE_MHU_PICKER_ARMS_INITIALIZE_STEPS : i...
1
MaskIdLookupTable
+addElement(LookupTableElement) : void
+getNewElement() : LookupTableElement
+editElement(int, int, int, Object) : void
1
<<Interface>>
MOSUnitStatusTypes
+UNKNOWN_POSITION : int
+MASK_IN_FPU : int
+MASK_IN_TURNOUT : int
+NO_MASK_IN_USE : int
+IN_MOTION : int
+UNKNOWN_MASK : int
MOSUnitStatus
+getPosition() : int
+getMaskNumber() : int
+getMaskID() : String
+getMaskDescription() : String
1
MaskAddress
+NUMBER_OF_MASKS : int = 33
+NUMBER_OF_FIXED_MASKS : int = 10
+getMaskNumber() : int
+getDescription() : String
+hasDescription() : boolean
+hashCode() : int
+equals(Object) : boolean
<<Interface>>
RMIMOSUnit
+isInitialized() : boolean
+getStatus() : MOSUnitStatus
+maskToFPU(int) : void
+maskToCabinet(int) : void
+maskFromFPUToTurnout() : void
+maskFromCabinetToTurnout(int) : void
+exportCabinet() : void
+importCabinet() : void
+getLookupTable() : MaskIdLookupTable
+setLookupTable(MaskIdLookupTable) : void
+getMaskID(int) : String
+getMaskDescription(int) : String
+getSlotNumber(String) : int
+getLastProcessedSequenceTree() : Vector
+isLocked() : boolean
+commitPassword(String) : void
states
sequences
<<use>>
<<use>>
RMIMOSUnitImpl
-mutex : Mutex
-statusMutex : Mutex
-isInitialized : boolean
-sequenceExecutionHandler : SequenceExecutionHandler
-mosUnitStatus : MOSUnitStatus
-isStatusValid : boolean
-maskIdLookupTable : MaskIdLookupTable
-executesSequence : Boolean
-lastProcessedSequenceTree : Vector
-initialize(MarshalledObject) : void
+initialize() : void
-generateStatus() : void
-getCurrentMaskNumber() : int
-initializeSequences() : void
+isInitialized() : boolean
+getStatus() : MOSUnitStatus
+maskToFPU(int) : void
+maskToCabinet(int) : void
+maskFromFPUToTurnout() : void
+maskFromCabinetToTurnout(int) : void
+exportCabinet() : void
+importCabinet() : void
+getLookupTable() : MaskIdLookupTable
+setLookupTable(MaskIdLookupTable) : void
+getMaskID(int) : String
+getMaskDescription(int) : String
+getSlotNumber(String) : int
+getLastProcessedSequenceTree() : Vector
+isLocked() : boolean
+commitPassword(String) : void
+getVersionString() : String
+getServer() : Server
+executeSequence(SequenceDescriptor, Object [], Sequenc...

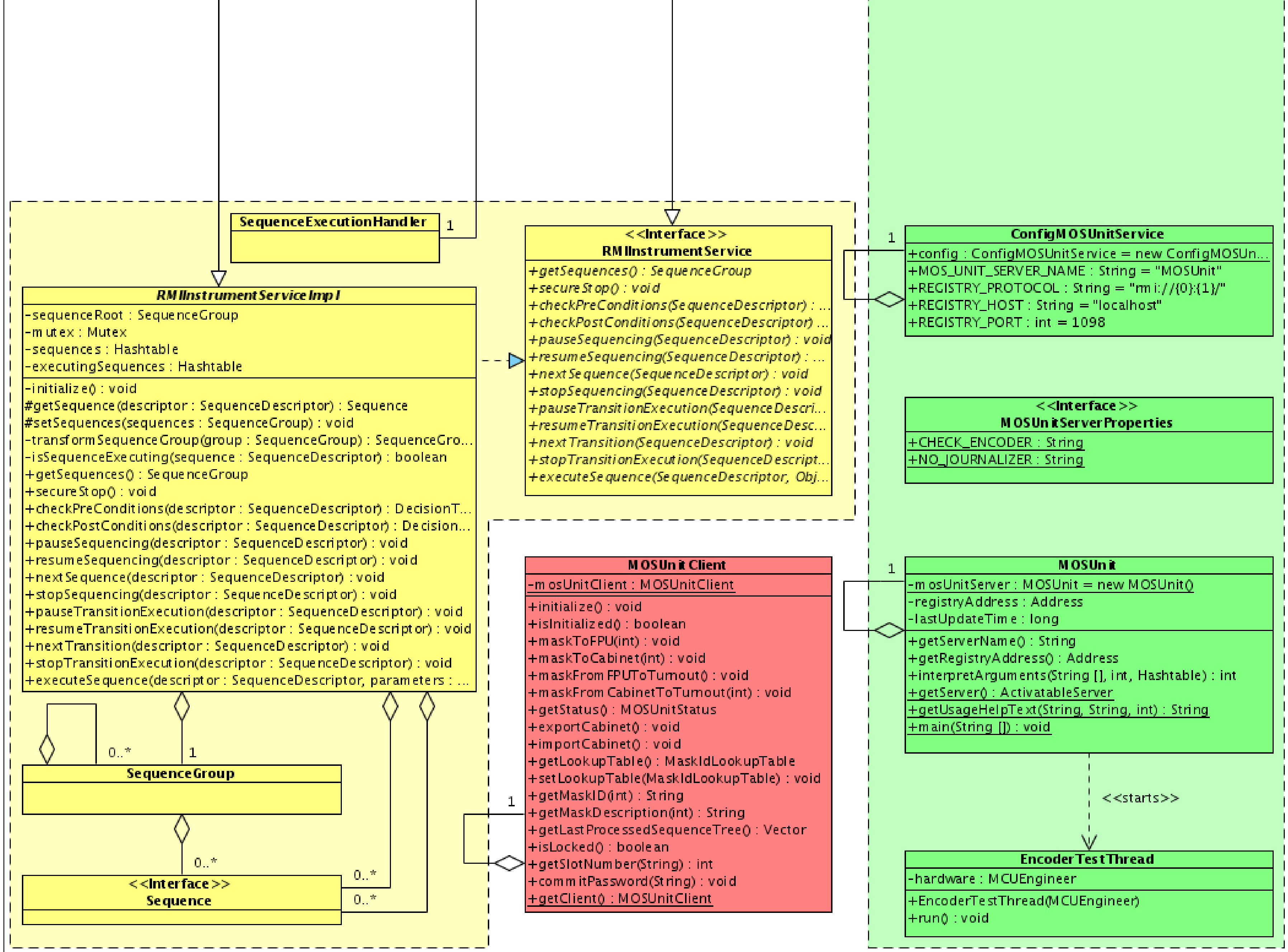


Figure 6.3: Class diagram of the *MOS Unit Service.* GREEN : The classes and interfaces that realise the *MOS Unit Service.* YELLOW : The abstract *Instrument Service* that is used as the base of all services of the *Instrument Tier.* RED : Client access to the service.

The processing in the execution methods is controlled via **SequenceExecutionHandler** objects. This handler is used to pause, step through, resume and stop the execution of sequences and their included transitions separately. This controlling of the execution process allows inspection by the engineer. Sequences can be executed stepwise to find mechanical problems of opto-mechanical parts. After hardware modifications new parameter sets can be tested securely, too. Additionally skipping of sequences can be triggered and pre- and post-conditions can be disabled globally. Another functionality of the **SequenceExecutionHandler** objects is to provide a gateway to the status of the sequence execution process including all encapsulated elements. The **RMISequenceExecutionListener** remote interface and its *RMI* implementation are used to connect to a sequence executing service, to register at the used handler and to display detailed information to the user (see Section 7.3).

6.2 The MOS-Unit Service

The use of individually manufactured masks makes a concurrent spectral analysis of several objects possible. In *LUCIFER* a total of 33 masks can be stored. 10 of them are fix defined and contain long-slit masks as well as a blind mask. The most important service of the *Instrument Tier* is the *MOS Unit Service* which controls the hardware that exchanges these masks. As *LUCIFER* is a cryogenically cooled instrument a mask replacement would usually require a warm-up and a cool-down phase in which the instrument could not be used scientifically. For a dewar of the size of the *LUCIFER* instrument the required time can easily extend to one week. In this calculation the required time of un-mounting the instrument from the telescope is neglected. One big advantage of the design of *LUCIFER* is that an auxiliary cryostat can be attached. Two auxiliary cryostats are utilised, one to remove the old masks and the other to insert the new ones. In the auxiliary cryostat a cabinet with a maximum of 23 user-defined masks is pre-cooled to avoid an interruption in the observation schedule. The use of auxiliary cryostats allows to exchange masks on a weekly basis. Technical details on the *MOS Unit* hardware can be found in Hofmann et al. (2004); Buschkamp et al. (2010).

The successful operation of a complex cryogenic mechanism like the *MOS Unit* highly depends on its control software. To comply with the requirements the hardware must be assisted by the software. If a mechanical function can not be solved by a constructional approach, an intelligent control of sub-components has to realise it. One example for such an interconnection between the hardware and the software is the spindle that moves the cabinet between the instrument and an auxiliary cryostat. It is manufactured as two independent parts. One in the instrument and another in the auxiliary cryostat. Both elements have to move synchronously as if they were joint even though a mechanical coupling would be too complex to realise. Another example is the *MHU*. The robot arm must precisely transport the masks between the storage and the *FPU*. However the limitations of space confine its mechanical stiffness. Therefore for each orientation of the instrument the correct placement of the masks can only be guaranteed by the control software.

The class diagram of the *MOS Unit Service* is presented in Figure 6.3. The functionality to execute sequences which are based on the sequencing framework is integrated into the *MOS Unit Service* by extending the abstract *Instrument Service*. This abstract service is used by every service of the *Instrument Tier*. This enables the engineer to use one common *GUI* to control the sequence execution process (see Subsection 6.1.4) for all services that drive opto-mechanical units. Additionally all common methods to execute and manage sequences are implemented centrally. This increase the maintainability. The remote access to the abstract *Instrument Service* is defined in the **RMIInstrumentService** interface. This interface combines the required methods of the **SequenceExecutionHandler** and the **SequenceImpl** class. In contrast to the methods defined in those elements, only a **SequenceDescriptor** object needs to be passed as a parameter. The implementation of the **RMIInstrumentService** interface is done in the **RMIInstrumentService** class. This implementation keeps a hashtable of all available sequences. Another hashtable is used to keep track of all running sequences. To improve the presentation of sequences to the engineer, the root of the sequence tree is stored additionally. Both hashtables are used by the service implementation to look up the required sequences and to control them. The **RMIInstrumentServiceImpl**.*secureStop()* method does not require any parameters because this method tries to stop all currently executing sequences of a service. In contrast to directly dispatching an emergency stop of all motions at the *MCU Service*, the sequences are continued until the next state is reached.

Even though the *MOS Unit* is the most complex part of the instrument its service has a simple interface that is used by the higher-ranking *Instrument Manager*. There are four methods to transport a mask between *FPU*, turnout position and storage plus additional two methods to perform a cabinet exchange. The other functionalities of the *MOS Unit Service* are just required for status retrieval, engineering access and to administrate the masks. The remote methods of the *MOS Unit Service* are defined in the **RMIMOSUnit** interface, realised in the **RMIMOSUnitImpl** class and accessed via the **RMIMOSUnitClient** class. The motion logics that are required to safely move the *MOS Unit* are realised as sequences. These sequences are created initially at service start time together with the states and a single **SequenceExecutionHandler** object. The parametrisation of the sequences and states is done by using the huge configuration set that is provided by the **ConfigMOSUnit**. Only a small fraction of all parameters is presented in Figure 6.3. This configuration contains a separate parameter set for every mask which is required to compensate slight variances in manufacturing and to store meta data. The handler object is used to control the execution of those sequences that realise the main *MOS Unit* operations. The **RMIMOSUnitImpl** class keeps a direct reference to those six main sequences. If one of the main methods is called, the handler is used to execute the corresponding main sequence. The resulting execution tree is stored in the **RMIMOSUnitImpl** class in order to be analysed in case of a malfunction. This wrapping of sequence execution into methods enables the **RMIMOSUnitImpl** class to change the status of the *MOS Unit Service* accordingly. As the **MOSUnitStatus** class extends the **JournalizerObject** class, the status is transmitted to the *Journalizer Service* and stored in a database (see Section 5.5). Important for status tracking is the overwriting of the method that executes a single sequence. This method is initially implemented in the abstract *Instrument Service* class. Thereby the service is enabled to determine if engineering actions have modified the unit status. Important for engineering access is the ability to lock the *MOS Unit Service* and prevent unwanted usage by others. Only by transmitting the right password, this lock can be disabled. Another tool for engineering is activated at service start and creates an **EncoderTestThread** object that observes the encoder values to detect malfunctions. This feature was introduced in a phase where encoders started to report motions even though no active command was issued.

All complexity of moving the *MOS Unit* is hidden in sequences and states. The resulting finite state transition networks model the complex motions of the individual parts of this unit. The sequences of the *MOS Unit Service* can be divided into two groups. One group that realises the mask transport in the instrument and the other to exchange the mask cabinet. The next subsections present the sequences of the controlled subunits in detail.

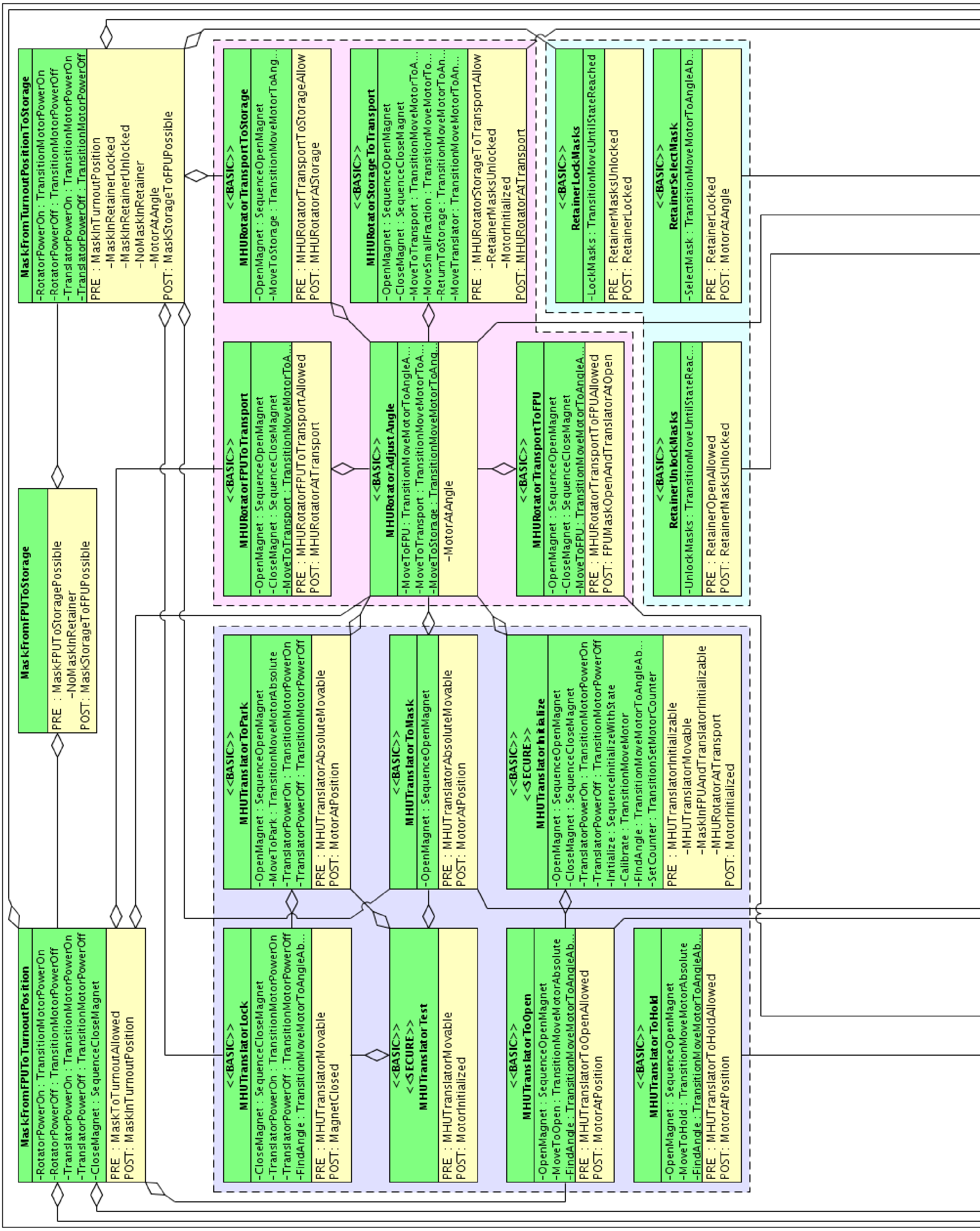
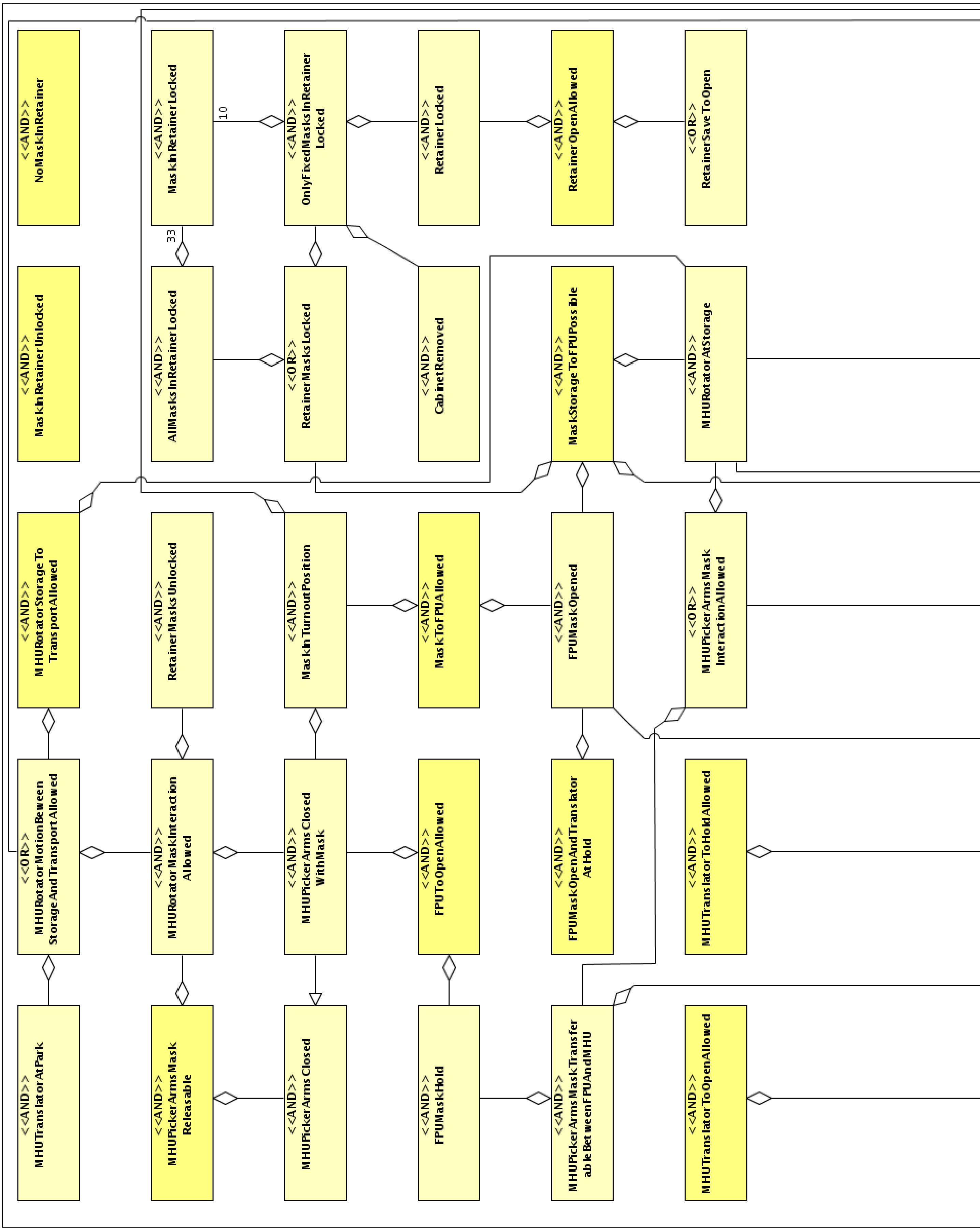

MaskFromTurnoutPositionToStorage
-RotatorPowerOn : TransitionMotorPowerOn
-RotatorPowerOff : TransitionMotorPowerOff
-TranslatorPowerOn : TransitionMotorPowerOn
-TranslatorPowerOff : TransitionMotorPowerOff
PRE : MaskInTurnoutPosition
-MaskInRetainerLocked
-MaskInRetainerUnlocked
-NoMaskInRetainer
-MotorAtAngle
POST: MaskStorageToFPUPossible
MaskFromFPUToStorage
PRE : MaskFPUToStoragePossible
-NoMaskInRetainer
POST: MaskStorageToFPUPossible
MaskFromFPUToTurnoutPosition
-RotatorPowerOn : TransitionMotorPowerOn
-RotatorPowerOff : TransitionMotorPowerOff
-TranslatorPowerOn : TransitionMotorPowerOn
-TranslatorPowerOff : TransitionMotorPowerOff
-CloseMagnet : SequenceCloseMagnet
PRE : MaskToTurnoutAllowed
POST: MaskInTurnoutPosition
<<BASIC>>
MHURotatorTransportToStorage
-OpenMagnet : SequenceOpenMagnet
-MoveToStorage : TransitionMoveMotorToAng...
PRE : MHURotatorTransportToStorageAllow
POST: MHURotatorAtStorage
<<BASIC>>
MHURotatorStorageToTransport
-OpenMagnet : SequenceOpenMagnet
-CloseMagnet : SequenceCloseMagnet
-MoveToTransport : TransitionMoveMotorToA...
-MoveSmallFraction : TransitionMoveMotorTo...
-ReturnToStorage : TransitionMoveMotorToAn...
-MoveTranslator : TransitionMoveMotorToAn...
PRE : MHURotatorStorageToTransportAllow
-RetainerMasksUnlocked
-MotorInitialized
POST: MHURotatorAtTransport
<<BASIC>>
RetainerLockMasks
-LockMasks : TransitionMoveUntilStateReached
PRE : RetainerMasksUnlocked
POST: RetainerLocked
<<BASIC>>
RetainerSelectMask
-SelectMask : TransitionMoveMotorToAngleAb...
PRE : RetainerLocked
POST: MotorAtAngle
<<BASIC>>
MHURotatorFPUToTransport
-OpenMagnet : SequenceOpenMagnet
-CloseMagnet : SequenceCloseMagnet
-MoveToTransport : TransitionMoveMotorToA...
PRE : MHURotatorFPUToTransportAllowed
POST: MHURotatorAtTransport
<<BASIC>>
MHURotatorAdjustAngle
-MoveToFPU : TransitionMoveMotorToAngleA...
-MoveToTransport : TransitionMoveMotorToA...
-MoveToStorage : TransitionMoveMotorToAng...
-MotorAtAngle
<<BASIC>>
MHURotatorTransportToFPU
-OpenMagnet : SequenceOpenMagnet
-CloseMagnet : SequenceCloseMagnet
-MoveToFPU : TransitionMoveMotorToAngleA...
PRE : MHURotatorTransportToFPUAllowed
POST: FPUMaskOpenAndTranslatorAtOpen
<<BASIC>>
RetainerUnlockMasks
-UnlockMasks : TransitionMoveUntilStateReac...
PRE : RetainerOpenAllowed
POST: RetainerMasksUnlocked
<<BASIC>>
MHUTranslatorToPark
-OpenMagnet : SequenceOpenMagnet
-MoveToPark : TransitionMoveMotorAbsolute
-TranslatorPowerOn : TransitionMotorPowerOn
-TranslatorPowerOff : TransitionMotorPowerOff
PRE : MHUTranslatorAbsoluteMovable
POST: MotorAtPosition
<<BASIC>>
MHUTranslatorToMask
-OpenMagnet : SequenceOpenMagnet
PRE : MHUTranslatorAbsoluteMovable
POST: MotorAtPosition
<<BASIC>>
<<SECURE>>
MHUTranslatorInitialize
-OpenMagnet : SequenceOpenMagnet
-CloseMagnet : SequenceCloseMagnet
-TranslatorPowerOn : TransitionMotorPowerOn
-TranslatorPowerOff : TransitionMotorPowerOff
-Initialize : SequenceInitializeWithState
-Calibrate : TransitionMoveMotor
-FindAngle : TransitionMoveMotorToAngleAb...
-SetCounter : TransitionSetMotorCounter
PRE : MHUTranslatorInitializable
-MHUTranslatorMovable
-MaskInFPUAndTranslatorInitializable
-MHURotatorAtTransport
POST: MotorInitialized
<<BASIC>>
MHUTranslatorLock
-CloseMagnet : SequenceCloseMagnet
-TranslatorPowerOn : TransitionMotorPowerOn
-TranslatorPowerOff : TransitionMotorPowerOff
-FindAngle : TransitionMoveMotorToAngleAb...
PRE : MHUTranslatorMovable
POST: MagnetClosed
<<BASIC>>
<<SECURE>>
MHUTranslatorTest
PRE : MHUTranslatorMovable
POST: MotorInitialized
<<BASIC>>
MHUTranslatorToOpen
-OpenMagnet : SequenceOpenMagnet
-MoveToOpen : TransitionMoveMotorAbsolute
-FindAngle : TransitionMoveMotorToAngleAb...
PRE : MHUTranslatorToOpenAllowed
POST: MotorAtPosition
<<BASIC>>
MHUTranslatorToHold
-OpenMagnet : SequenceOpenMagnet
-MoveToHold : TransitionMoveMotorAbsolute
-FindAngle : TransitionMoveMotorToAngleAb...
PRE : MHUTranslatorToHoldAllowed
POST: MotorAtPosition

Figure 6.4: Class diagram of the sequences that realise a mask exchange between the cabinet and the *FPU*. YELLOW: The states that are used as pre-, post- or intermediate conditions in the motion sequence classes. A detailed presentation of the states is given in Figure 6.5. GREEN: The hierarchically structured motion sequences that are required for mask exchange. The sequences are organised by the operated hardware parts: *MHU* Rotator, *MHU* Translator, *MHU* Picker Arms, Mask Retainer and *FPU*.


<<AND>> NoMaskInRetainer
<<AND>> MaskInRetainerUnlocked
<<AND>> MHURotatorStorageToTransportAllowed
<<OR>> MHURotatorMotionBetweenStorageAndTransportAllowed
<<AND>> MHUTranslatorAtPark
<<AND>> MaskInRetainerLocked
10
33
<<AND>> AllMasksInRetainerLocked
<<AND>> RetainerMasksUnlocked
<<AND>> MHURotatorMaskInteractionAllowed
<<AND>> MHUPickerArmsMaskReleasable
<<AND>> OnlyFixedMasksInRetainerLocked
<<OR>> RetainerMasksLocked
<<AND>> MaskInTurnoutPosition
<<AND>> MHUPickerArmsClosedWithMask
<<AND>> MHUPickerArmsClosed
<<AND>> RetainerLocked
<<AND>> CabinetRemoved
<<AND>> MaskToFPUAllowed
<<AND>> FPUToOpenAllowed
<<AND>> FPUMaskHold
<<AND>> RetainerOpenAllowed
<<AND>> MaskStorageToFPUPossible
<<AND>> FPUMaskOpened
<<AND>> FPUMaskOpenAndTranslatorAtHold
<<AND>> MHUPickerArmsMaskTransferableBetweenFPUAndMHU
<<OR>> RetainerSaveToOpen
<<AND>> MHURotatorAtStorage
<<OR>> MHUPickerArmsMaskInteractionAllowed
<<AND>> MHUTranslatorToHoldAllowed
<<AND>> MHUTranslatorToOpenAllowed

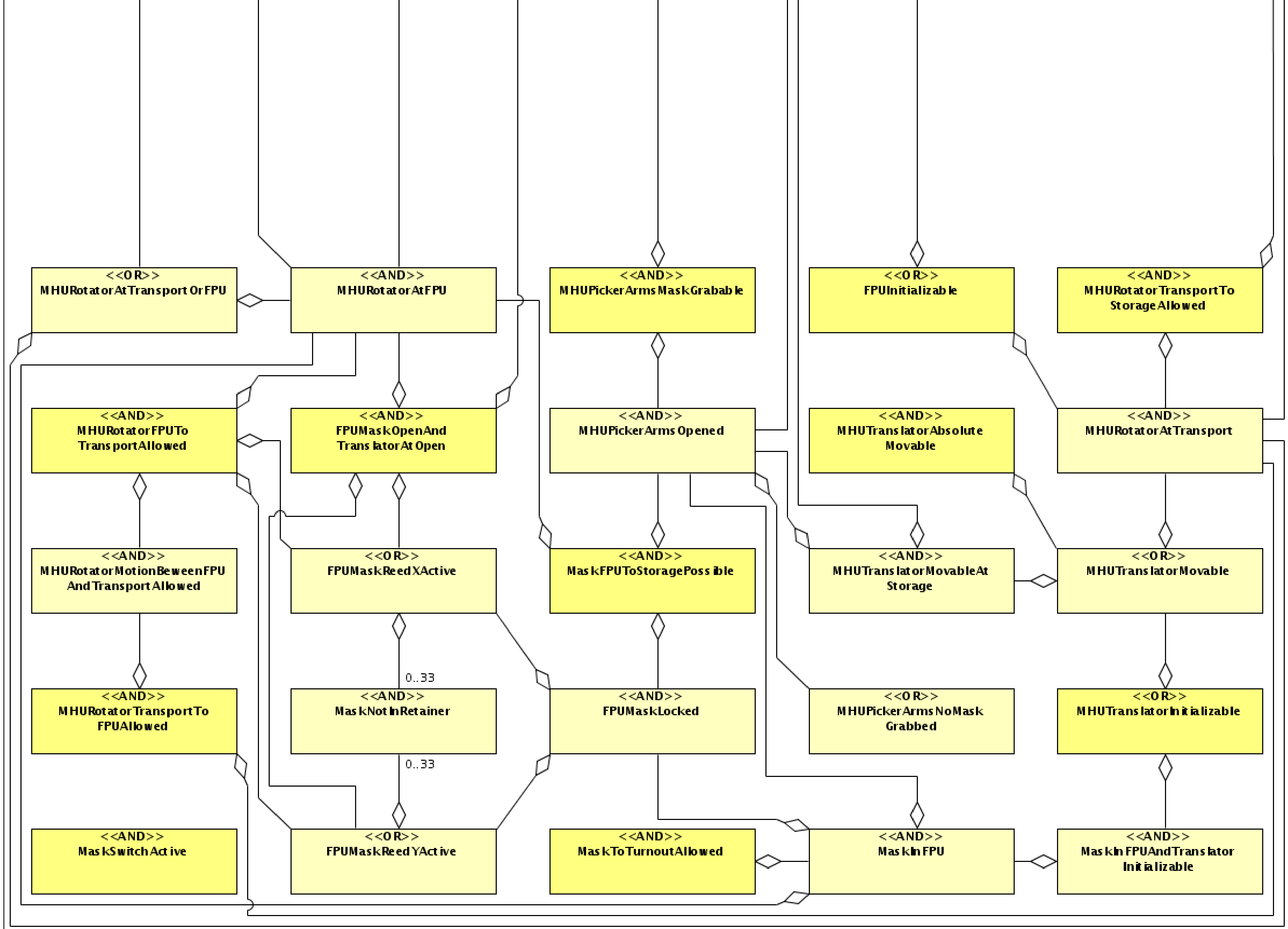


Figure 6.5: Class diagram of the states that are required for a mask exchange between the cabinet and the *FPU*. YELLOW : The states that are used by the mask exchange sequences of Figure 6.4. The complex aggregation hierarchy visualises the composition of the required states. States representing the roots of the largest decision trees are coloured more intense.

6.2.1 The Mask Exchange between the *FPU* and the Cabinet

The sequences that are required to position a mask in the optical beam of *LUCIFER* are presented in Figure 6.4. The corresponding states and their aggregation hierarchy is shown in Figure 6.5. In the class diagrams of Figure 6.4, Figure 6.5 and Figure 6.6 the <`Sequence`> and <`State`> prefix is omitted, respectively. The two main sequences that realise the mask exchange are composed in the **SequenceMaskFromFPUToStorage** and the **SequenceMaskFromStorageToFPU** class. Both classes are the root of large sequence trees. Their direct children model the motion between the end points and the intermediate turnout position. 6 of the 31 required sequences are used only for engineering purposes. To initialise the corresponding subunits there are the **SequenceFPUInitialize**, the **SequenceMHUTranslatorInitialize** and the **SequenceMHUPickerArmsInitialize** classes. Two sequences can be used to open and close the mask grabber manually. The **SequenceMHUTranslatorTest** class combines several sequences to automatically test the successful operation of the *MHU* translator. To organise the sequences they are grouped by the part they operate. The sequences of these groups are described next.

The Mask Handling Unit

The *MHU* is responsible for picking and transporting a mask. It is composed of three subunits. These are the *MHU* picker arms, the *MHU* rotator and the *MHU* translator. The sequences of the *MHU* picker arms are required to securely pick/release a mask. Even though the required motions are simple, the states that are used as pre- and post-conditions are very complex. These states have to ensure that the *MOS Unit* is in a setup to transfer a mask. As a mask is a free element a release without previous fixing would drop the mask in the instrument and result in damage. The same problem exists when a mask is grabbed. If the grabbing fails and the mask is not locked in the picker arms further processing of sequences would lead to a loss of a mask.

The *MHU* rotator allows to rotate the *MHU* picker arms together with a grabbed mask. To be able to move a mask in *LUCIFER* the mask has to be rotated into the transport orientation. The other available orientations are towards the cabinet or the *FPU*. In these three orientations the rotator can be locked magnetically. If the motor is powered off the rotator will keep its orientations permanently. When a grabbed mask is removed from the storage cabinet it can get jammed. In such a case the implementation of the **SequenceMHURotator StorageToTransport** class tries to free the mask automatically: (i) After detecting a jam the mask is brought back to the cabinet. (ii) The *MHU* translator position is slightly modified and a new extraction try is started. (iii) This procedure is repeated as long as the translator position is within a predefined tolerance area. (iv) If the automatic extraction procedure is successful the configuration set of the corresponding mask is modified (see Section 6.2).

The *MHU* translator moves the *MHU* in front of the mask cabinet to the positions of the masks. As this positioning requires high accuracy, the translator needs to be initialised in advance. As described in Subsection 6.1.3 several transitions and states exist to ensure an accurate absolute positioning of elements. After a mask is rotated to the transport position it can be moved between the turnout position and the cabinet. In the turnout position the translator is locked magnetically to prevent the *MHU* translator from drifting in case the motor power is switched off. The same locking has to be done when the translator is at the *FPU* or is currently not used. Although the mechanical manufacturing tolerance is very small absolute positioning is not sufficient enough to activate a lock successfully. An additional angle correction of the translator spindle is required. To insert a mask in the

FPU the translator has to place it on centring ball pins. Therefore the *MHU* translator has to move towards the *FPU* after the mask has been rotated from the transport to the *FPU* orientation.

Because an accurate interaction is required to prevent a collision, the whole *MHU* is as complex as all other units of Section 6.3.

The Mask Retainer

Only a part of the total mask retainer functionality is required to exchange a mask. These are the ability to select a mask and unlock/lock it. The mask selection is mechanically realised with a slotted spindle. Therefore the **SequenceRetainerSelectMask** class has to operate the stepper motor until a required angle is reached. When a mask in the retainer is unlocked the selection spindle is not allowed to move. This requirement is modelled in the corresponding pre-condition. The unlocking/locking of masks itself demands a simple limited motion of the corresponding stepper motor.

The Focal Plane Unit

The *FPU* has to keep the masks in the focal plane with high positioning accuracy. This is mechanically realised with centring ball pins. After a mask has been rotated towards the *FPU* and placed by the translator, both arms of the *FPU* can move to an intermediate position that ensures that the mask can not get lost. Then the picker arms can release the mask to prevent stress during alignment. Finally the picker arms can close and press the mask onto the ball pins. It is important for all motions of both axis of the *FPU* arms that they are executed synchronously to prevent a jam. At the same time the tolerance in the manufacturing of both arms has to be compensated by the 4 *FPU* sequences.

6.2.2 The Cabinet Exchange between LUCIFER and Auxiliary Cryostat

The replacement of 23 masks without the necessity of warming up the whole instrument is one of the advantages of *LUCIFER*. No complications during a cabinet exchange should occur because the auxiliary cryostat is attached to *LUCIFER* while it is mounted to the telescope. If the cabinet is stuck in the middle of the transfer process and can not be recovered to a save state, the instrument needs to be opened. During this warm-up process the telescope is forced to point to the zenith. The sequences and states that realise the cabinet exchange are presented in Figure 6.6. 6 of the 20 sequences are required for engineering tasks. These sequences are either used to initialise the different subunits or to perform a cabinet exchange in the laboratory without having an auxiliary cryostat attached. The main motion sequences of the cabinet exchange are represented by the **SequenceCabinetToLucifer** and the **SequenceCabinetToAuxCryostat** classes. These classes combine several sequences of the different parts of the *MOS Unit* to realise a safe transfer. The sequences of these parts are described next.

The Shield Lock

The shield lock is the simplest unit that takes part in the cabinet exchange procedure. A hardware switch limited motion is required to open the radiation shield. The only complexity of this motion is handled by the pre-condition which ensures that the flexboard that is attached to the *MHU* translator is out of the critical working area of the lock.


<<BASIC>>
<<SECURE>>
ThermalBridgeInitialize
-Open : TransitionMoveUntilStateReached
POST: ThermalBridgeOpened
<<BASIC>>
ThermalBridgeClose
-Close : TransitionMoveUntilStateReached
PRE : ThermalBridgeOpened
POST: ThermalBridgeClosed
<<BASIC>>
ThermalBridgeOpen
-Open : TransitionMoveUntilStateReached
PRE : ThermalBridgeClosed
POST: ThermalBridgeOpened
CabinetToAuxiliaryCryostat
PRE : CabinetToAuxiliaryCryostatAllowed
-AllMasksInRetainerUnlocked
POST: CabinetToLuciferAllowed
<<BASIC>>
CabinetExport
-MoveLuciferCabinet : TransitionMoveMotorAbs...
-AdjustLuciferAngle : TransitionMoveMotorToA...
-AdjustACAngle : TransitionMoveMotorToAngle...
-SetLuciferPosition : TransitionSetMotorCounter
-SetACPosition : TransitionSetMotorCounter
-ExportCabinet : TransitionMoveMotorSynchron[]
-MoveACCabinet : TransitionMoveMotorAbsolut...
-FindACLimit : TransitionMoveMotorUntilStateR...
PRE : CabinetExchangeAllowed
-CabinetReadyToSynchronize
POST: CabinetRemoved
<<BASIC>>
CabinetUnlock
-Unlock : TransitionMoveUntilStateReac...
PRE : CabinetUnlockable
POST: CabinetUnlocked
<<BASIC>>
<<SECURE>>
CabinetInitialize
-Lock : TransitionMoveUntilStateReached
PRE : CabinetSaveToLock
POST: CabinetLocked
<<BASIC>>
CabinetLock
-Lock : TransitionMoveUntilStateReached
PRE : CabinetLockable
POST: CabinetLocked
<<BASIC>>
CabinetRemoveInLab
-RemoveCabinet : TransitionMoveMotor
PRE : CabinetLabExchangeAllowed
-AllMasksInRetainerLocked
-AllMasksInRetainerUnlocked
POST: CabinetRemoved
<<BASIC>>
<<SECURE>>
ShieldShutterInitialize
-Close : TransitionMoveUntilStateReached
PRE : ShieldShutterSaveToMove
POST: ShieldShutterClosed
<<BASIC>>
ShieldShutterClose
-Close : TransitionMoveUntilStateReached
PRE : ShieldShutterToCloseAllowed
POST: ShieldShutterClosed
<<BASIC>>
OpenMagnet
-Close : TransitionMagnetClose
-Open : TransitionMagnetOpen
PRE : MagnetClosed
POST: MagnetOpened
<<BASIC>>
ShieldShutterOpen
-Open : TransitionMoveUntilStateReached
PRE : ShieldShutterToOpenAllowed
POST: ShieldShutterOpened
<<BASIC>>
CabinetImport
-MoveACCabinet : TransitionMoveMotorAbsolut...
-AdjustLuciferAngle : TransitionMoveMotorToA...
-AdjustACAngle : TransitionMoveMotorToAngle...
-SetACPosition : TransitionSetMotorCounter
-SetLuciferPosition : TransitionSetMotorCounter
-ImportCabinet : TransitionMoveMotorSynchron
-MoveLuciferCabinet : TransitionMoveMotorAbs...
-FindLuciferLimit : TransitionMoveMotorUntilStateRea...
PRE : CabinetExchangeAllowed
-CabinetReadyToInsert
POST: CabinetLockable
CabinetToLucifer
PRE : CabinetToLuciferAllowed
-AllMasksInRetainerUnlocked
POST: CabinetToAuxiliaryCryostatAllowed
<<BASIC>>
RetainerUnlockMasks
-Unlock : TransitionMoveUntilStateReached
PRE : RetainerOpenAllowed
POST: RetainerMasksUnlocked
<<BASIC>>
<<SECURE>>
RetainerLockInitialize
-Lock : TransitionMoveUntilStateReached
POST: RetainerLocked
<<BASIC>>
RetainerLockMasks
-Lock : TransitionMoveUntilStateReached
PRE : RetainerMasksUnlocked
POST: RetainerLocked
<<BASIC>>
RetainerSelectAll
-SelectAll : TransitionMoveMotorToA...
PRE : RetainerLocked
POST: MotorAtAngle
<<BASIC>>
CabinetInsertInLab
-InsertCabinet : TransitionMoveMotor
PRE : CabinetLabExchangeAllowed
-AllMasksInRetainerLocked
-AllMasksInRetainerUnlocked
POST: CabinetLocked

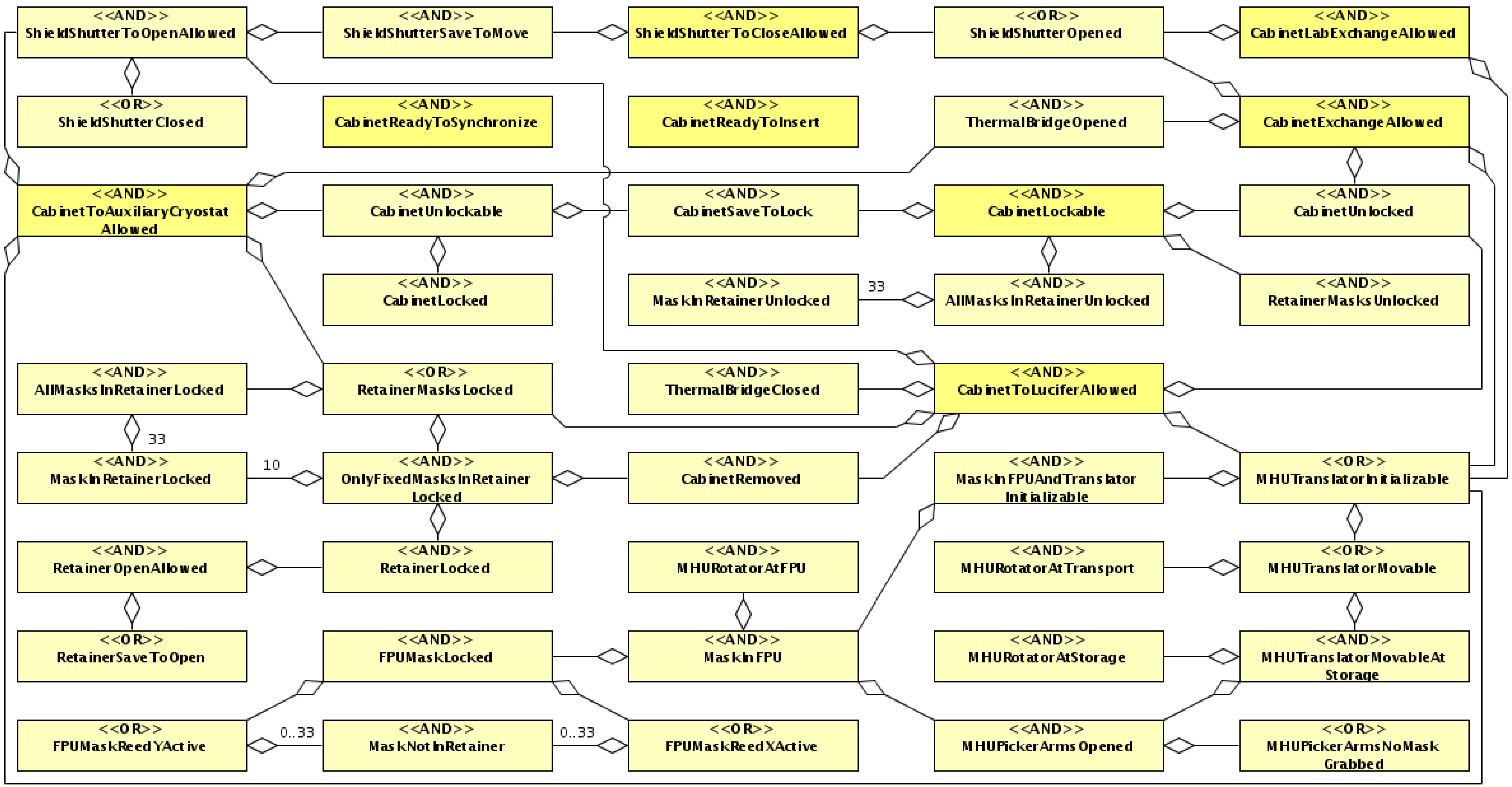


Figure 6.6: Class diagram of the sequences and states that realise the cabinet exchange between *LUCIFER* and an auxiliary cryostat. YELLOW : The states that are used by the motion sequences. The composition of states is presented in the lower part. States representing the roots of the largest decision trees are coloured more intense. GREEN : The hierarchical structure of motion sequences. The sequences are organised by the operated hardware parts: Shield Shutter, Thermal Bridge, Mask Retainer and Cabinet Lock.

The Mask Retainer

The mask retainer is used in a similar way as in the mask exchange sequences. The only difference in the selection process is, that instead of a single mask all masks are selected and unlocked. After a successful exchange all masks are locked again. Besides the mask locking the cabinet locking is part of the retainer structure. Its motion is modelled in dedicated sequences, too. Most important for the cabinet exchange is the transport of the mask cabinet. This motion is the critical element of the whole exchange process . The cabinet translations of both, the instrument and the auxiliary cryostat, have to be operated synchronously. The motion of the mask cabinet is implemented in the **SequenceCabinetImport** and **SequenceCabinetExport** class. Both sequences work in the same way except that the motion direction is reversed. (i) The cabinet is brought stepwise towards the end of the cabinet translation spindle. (ii) Both translation spindles are aligned with respect to their orientation. This is required to move the spindles as if they were mechanically joint. (iii) The synchronous motion that realises the transfer between both spindles is executed stepwise, too. Between the motions steps the absolute angles of both elements are compared to determine whether the last motion was successful. In this case the export sequence can be resumed. (iv) After the cabinet is movable by the spindle of the auxiliary cryostat the motion is continued stepwise until the cabinet is fully inserted.

The Auxiliary Cryostat

Besides the spindle that realises the cabinet motion as described above, the auxiliary cryostat contains a thermal bridge. This part can be pressed on top of the masks to ensure a thermal coupling and speed-up the cool-down process. Additionally the thermal bridge guarantees that the masks stay in position during the handling of the auxiliary cryostat.

6.3 The Other Services

Even though the remaining services of the *Instrument Tier* are very important for the optical setup of *LUCIFER* their implementation is very simple. Due to the constructional realisation of these units a maximum of two motors is used to create a rotational or translational motion. Therefore a very limited amount of sequences and states is required to model the required motions in the services. Both, the *Filter Wheel Unit Service* and the *Camera Unit Service* control a simple rotation of the corresponding opto-mechanical parts. For the *Camera Unit* the rotation is limited by two hardware limit switches. The positioning accuracy of these units is mechanically realised with notches and requires no additional software compensation. The *Grating Unit Service* is the most complex element of the services of this section. It combines two mechanically independent motions. First a rotation that ends in notches to select different gratings for spectroscopy or a plain mirror for imaging. The second motion is the tilting of the gratings. As the tilt angle is actively controlled by the *HIRAMO* the service just needs to look up the corresponding voltage value and send it to the electronics. The *Detector Focus Unit Service* controls the focal stage on which the detector is mounted. The used stepper motor is controlled by the service to create a plain translation of the detector along the optical beam. Depending on the used camera and filters the focus position is adjusted to a predefined value. The *Pupil Viewer Unit Service* moves an additional lens into the optical beam. This motion requires the stepper motor to turn until the required limit is reached. The *Compensation Mirror Unit Service* is responsible for adjusting the optical beam and to compensate the gravitational distortion of the instrument structure. This compensation is realised with two mirrors that are mounted on two translational elements each. For this the two stepper motors are brought to predefined positions that are dependent on the instrument orientation.

Except the services of this section all software of the *Instrument Tier* was developed as a part of this thesis.

Chapter 7

The Operation Tier

All services that are required for operating the LUCIFER instrument are part of the Operation Tier. There are dedicated services to operate and supervise the instrument. Other services exist to manage the interaction with the Telescope and the Readout Services. The applications for user interaction and observation preparation are part of this tier, too. Their graphical client access to the services is realised via separate GUIs for engineers and observers.

The *Operation Tier* combines the services of the lower tiers to use the *LUCIFER* instrument scientifically. Besides the centralised configuration facility the *System Tier* contains the *Message Server*. Its service is used to inform either the engineer or the observer about the activities and status of all services (see Section 4.5). During operation this end-user information is mandatory to notice technical malfunctions and to be able to react accordingly. The services of the *Control Tier* provide direct access to the electronics of the instrument. Therefore a direct engineering access is created with dedicated *GUI* clients (see Section 7.3). The most important service of the *Control Tier* is the *Journalizer Service* which is required to execute observations with high efficiency (see Section 5.5). This service keeps a log of the current instrument status which is required to change the setup fast and present it to the user. Thereby the necessity to query all services individually is avoided. The services of the *Instrument Tier* are used to setup the individual optomechanical units of *LUCIFER*. Their current status is reported to the *Journalizer Service*, too. A data structure to represent the instrument status was developed together with Jan Schimmelmann (see Schimmelmann, 2007). This data structure is used in his scheduler prototype to setup the instrument with a minimum of required service interactions. Currently there is no implementation of a *Telescope Manager* or *Readout Manager* which processes binocular telescope commands or readout processes. It makes no sense to use the scheduler as it is restricted to execute setup changes only. Scientific observations are limited to a simple text-file based script execution (see Knierim, 2009).

In the following sections the instrument control and the graphical client access is described.

7.1 The Services Managing LUCIFER

There are two services that manage the *LUCIFER* instrument. The *Instrument Manager* that combines the functionalities of the services of the *Instrument Tier* and the *Supervisor* that monitors the environmental parameters. As all complexity of controlling the optomechanical parts of *LUCIFER* is hidden in the services of the *Instrument Tier*, simple method calls are combined to build the *Instrument Manager*. The current workaround solution of the instrument manager does not use the *Journalizer* functionality accordingly and therefore increases the required setup time. Depending on the required optical setup, the services autonomously execute their tasks and report their status. The current version of the *Instrument Manager* in combination with the observation *GUI* continuously polls the status of the services either directly or from the *Journalizer Service* and thus produces additional workload. The *Supervisor* has to monitor the environmental parameters of *LUCIFER* and has to notify the observer/engineer if limits are exceeded. For this the *Supervisor* connects to the corresponding services of the *Control Tier* (see Section 5.4). On a regular basis the monitored environmental measurements are retrieved and compared with predefined limits. If the limits are exceeded, new messages are generated and additionally the user is informed visually.

Apart from the functional description, the development of the *Instrument Manager* and the *Supervisor* was not part of this thesis.

7.2 The Observer‘s Access to the Instrument

To support both observers and engineers in an optimised way, the access to the *LUCIFER* instrument is done differently. The access of the observer needs to be restricted to prevent the instrument from damage caused by misusage. The presented information must be filtered to an appropriate level. During observation runs, a presentation of the current status is required. This information must reflect the status of the instrument and the telescope. Additionally the observer needs to inspect the data taken in the readout process. In some cases the observer must interactively take appropriate actions. This is e.g., the case when the *MOS* masks are positioned on a target and fine adjustments to the telescope pointing are required.

7.2.1 The Performing of Observations

To be able to reuse the *GUI* components that are developed for both, engineering and observation purposes, a hierarchical interface structure was developed. By inheriting this structure the *GUI* access is divided into three groups. One to provide basic information, another for observers and one with full engineering capabilities. This approach was presented together with Volker Knierim in Knierim et al. (2006). When the observer wants to take scientific data sets interactively, he must be able to setup the instrument manually and perform integrations. Therefore he requires access to the *Instrument Manager*. The *GUI* client that implements this access has to commit setup changes and present the current status. A first prototype of an observer *GUI* is presented in Knierim et al. (2006), too.

A client access to the readout software *GEIRS* is needed when integrations should be performed manually. *GEIRS* has its own real time display and can be used to control the readout process. A service that interfaces with *GEIRS* and a prototype *GUI* client is presented in Muhlack (2006). To integrate the *SkyCat* tool of *ESO* (see Albrecht et al., 1997) as an alternative real time display, a script based solution was developed in this thesis.

7.2.2 The Preparation of Observations

An automated execution of observations is preferred by the scientist. *NIR* observations have to be done differently to optical ones to compensate the variable sky emission. This variability limits the maximum integration time. Typically a series of readouts is combined to one integration by the readout electronics. With respect to the sky variability and the used wavelength range, such an integration is limited to several seconds up to a few minutes. Depending on the scientific objective it may be required to take exposures at off-target pointings to measure the sky background. Especially extended targets require larger telescope offset. By these restrictions a scientific observation could require many pointings and integrations. It is nearly impossible to perform this manually and achieve good results.

Another benefit of automated observations is that the user is able to prepare everything in advance. Well prepared observations increase the scientific outcome, as the instrument usage can be optimised to reach the required sensitivity. The planning process involves the scheduling of targets during the night, an efficient rearrangement of instrument setups and the minimisation of telescope motions. The planning of observations is affected by the anticipated reduction process of the data sets, too.

In Schimmelmann (2007) a first prototype of an *Observation Preparation Tool (OPT)* is presented. This tool covers all important requirements and provides a parameterised solution for typical observation tasks. It contains pre-implemented code to realise common telescope motion tasks e.g., to point to the sky or dither around a telescope pointing with custom patterns. Besides the automated scheduling of observations over the available nights, the *GUI* of the *OPT* visualises important parameters like the airmass, to allow for a manual planning. The existing prototype is not used for observations yet because of pending modifications. Instead a text file driven scripting was implemented and has to be used by the observers (see Knierim, 2009).

7.3 The Engineer's Access to the Instrument

In contrast to the observer, the engineer requires full access to both the hardware and the software. On the software side the status of the services must be monitored. Additionally the engineer must be able to stop and start subsystems of the control software and modify their configuration parameters. In most cases engineering access to the hardware is required to solve problems. When an observation is interrupted by an error, appropriate actions are required to bring the instrument back to operation. During the integration and commissioning the engineering access was required to improve the performance of the *LUCIFER* instrument by optimising the configuration parameters. After each technical servicing, the mechanical modifications require an adaption of these values, too. Finally engineering access is required to exchange the cabinet and reconfigure the designations of the masks. The next subsections present the engineering access to the services of the different tiers.

7.3.1 The Access to the Services of the *System Tier*

The *System Tier* contains all basic services of the *LCSP*. These are the *Configuration Service*, the *Time Service* and the *Message Service* (see Chapter 4). As all services of the *LCSP* are based on the remote service framework of the *System Tier*, a unified access is realised by the *Start Manager* application (see Section 3.3). This *GUI* provides all required service management and configuration functionalities.

The *Message Service* that centrally processes the incoming data of all *LCSP* services and applications stores the messages in an *SQL* database. The persisted data can be analysed with the message browser afterwards (see Subsection 3.4.2). To keep track of the messages a direct access to the *Message Service* is required instead. This realtime visualisation of messages is done by the *Message Panel* (see Figure 7.1). The central element of this panel is a scrollable list of messages. Each message row contains a time stamp, a type

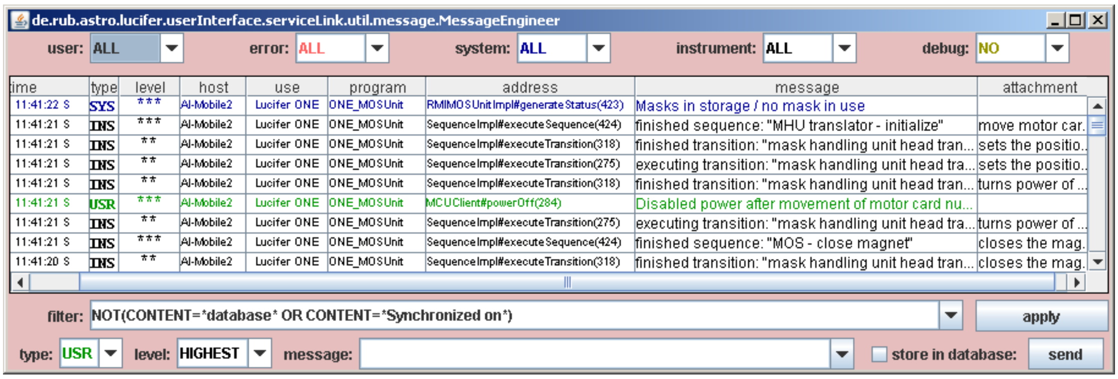


Figure 7.1: The panel used to display notifications of the services of the selected message channels.

and a level, localisation data, the message text and additional information. To minimise the network traffic between the *GUI* client and the service, the user can specify the level of messages to receive. This specification can be done individually for the 5 different message channels. Another filtering functionality is implemented in the *Message Panel* which evaluates a user defined string and reduces the output to matching results. The only difference between an observer and an engineer usage of the *Message Panel* is the capability to send messages manually.

7.3.2 The Access to the Services of the *Control Tier*

Access to the services of the *Control Tier* is fundamental to perform engineering tasks. These services realise the communication with the control electronics and therefore are required for basic operations (see Chapter 5). Besides the monitoring of environmental parameters, the full control over the opto-mechanical parts of *LUCIFER* is available on this level only.

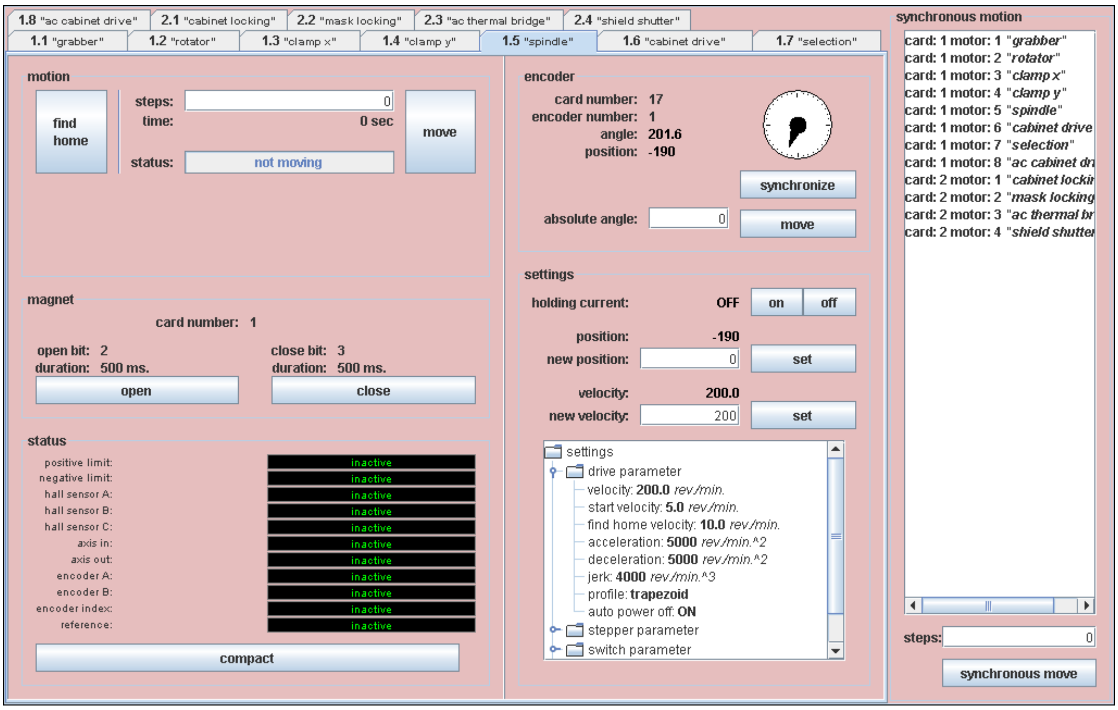


Figure 7.2: The *GUI* client of the *MCU Service*.

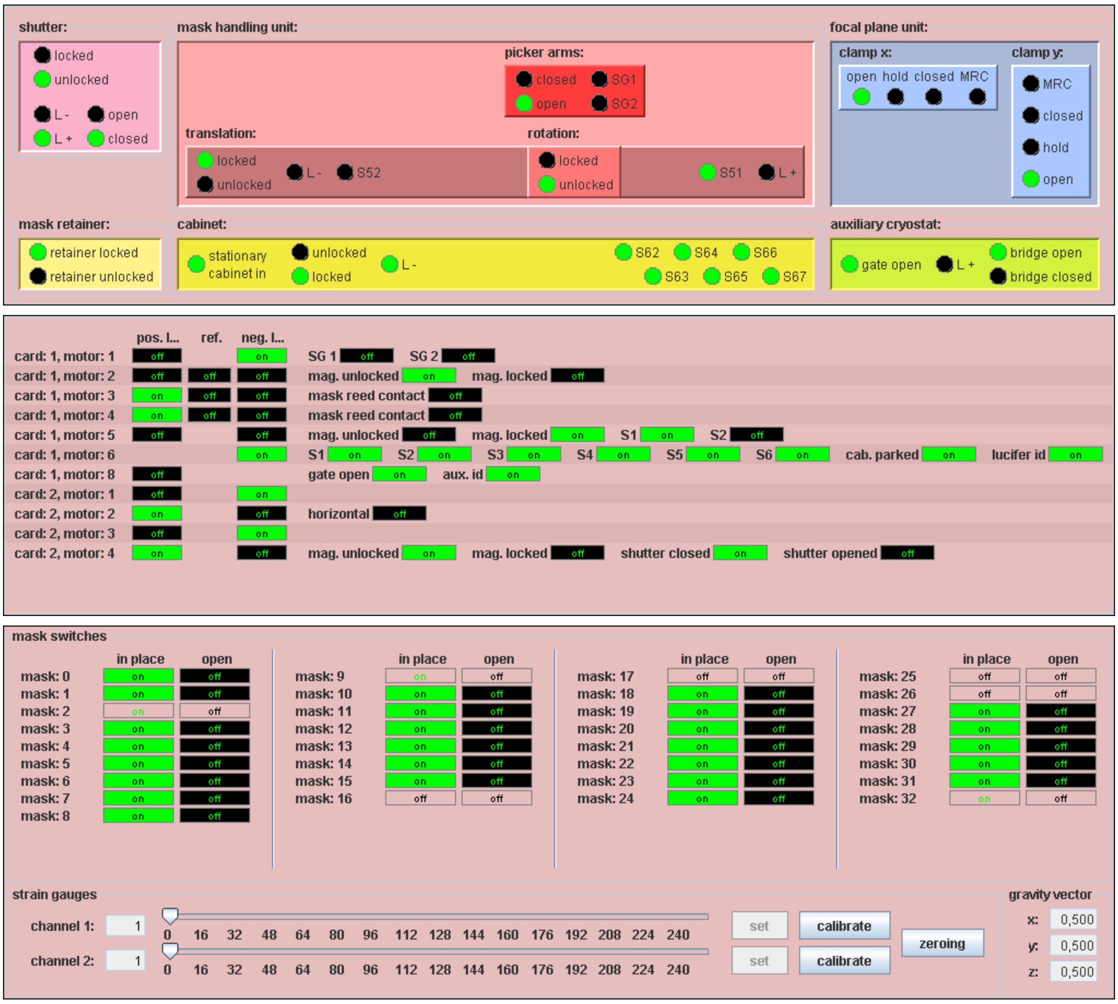


Figure 7.3: The three views of the *Switch Box* panel. **Top:** A schematic status display of the *MOS* switches that are associated to a motor. **Centre:** The status of the same switches in a tabular view. **Bottom:** The mask switches and access to the strain gauges.

To control the motion of the optical elements a *GUI* client to the *MCU Service* is required (see Figure 7.2). In a tabbed view the connected motors can be selected and controlled individually. This includes a manual motion by a specified amount of steps. Thereby the predicted motion time is presented while the steps are entered. This calculation is based on a functionality of the *MCU Service* which uses parts of the simulation code of *LuciferVR* (see Chapter 8). During motion a progress bar visualises the remaining motion time. A display of the used motion parameters is part of the *MCU Panel*. This information is required by the engineer to check the configuration. Related to the motion of a stepper motor is the ability to set the position which is internally managed by the electronics. Another functionality is the direct change of the speed and the power characteristics. Depending on the configuration of a motor, magnetic locks or encoder data are available. The magnetic locks can be used to fix the position of an element. The encoder data provides additional positioning information which is required to compensate steps that have been lost or to drive to an absolute angle. In the *MCU Panel* the angle information delivered by an encoder is graphically visualised, too. Apart from the individual controlling of motors, the right side of the panel hosts elements to move several motors synchronously. For each motor the *MCU* has the ability to probe several status signals

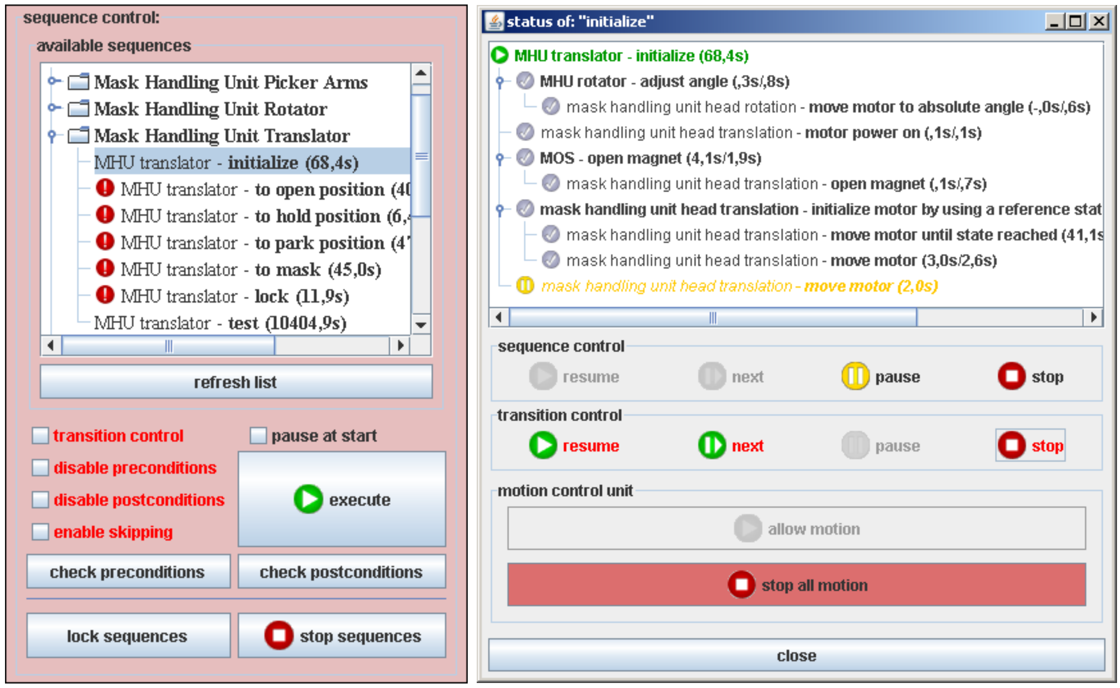


Figure 7.4: Sequence selection panel and execution dialogue. **Left:** The panel that provides engineering access to the services of the *Instrument Tier*. **Right:** The sequence execution dialogue with its interaction capabilities.

which are shown, too. This information can be used by the engineer to check the status of hardware switches which are directly connected to a motor.

Another possibility to check the status of switches is realised in the *Switch Box Service* (see Subsection 5.2.3). The corresponding *GUI* access i done with the *Switch Box Panel* (see Figure 7.3). This panel combines three views. The first two views show the status of switches that are associated to a motor. First of all these are the limit and reference switches. The other switches are for special purposes. The status of a switch is visualised with colour and text. As the *Switch Box Service* is able to determine malfunctions, these errors are displayed, too. In the first view the switches are schematically structured to support the status recognition process performed by the engineer. The second view presents the status of the same switches as a table. The last view visualises the status of the mask switches. In addition to the status display of the mask switches, the strain gauges are operated by this panel.

The last service of the *Control Tier* that needs to be accessed for engineering is the *HIRAMO Service*. Its *GUI* client is presented in Knierim (2009)

7.3.3 The Access to the Services of the *Instrument Tier*

The services of the *Instrument Tier* combine the functionalities of the *Control Tier* to realise complex motion sequences (see Chapter 6). As the generic *Instrument Service* is extended by all these services, basic sequencing functionalities can be provided by a uniform *GUI* (see Figure 7.4). The *Sequence Panel* enables the user to browse the available sequences and to select them for further processing. When the sequences are grouped accordingly, the panel presents them in a tree structure. In case the accessed service was restarted the available list of sequences can be refreshed. After a sequence is selected it can be executed by the user. If necessary a dialogue window will appear and query the required parameters. The execution process can be manipulated by using check boxes. The

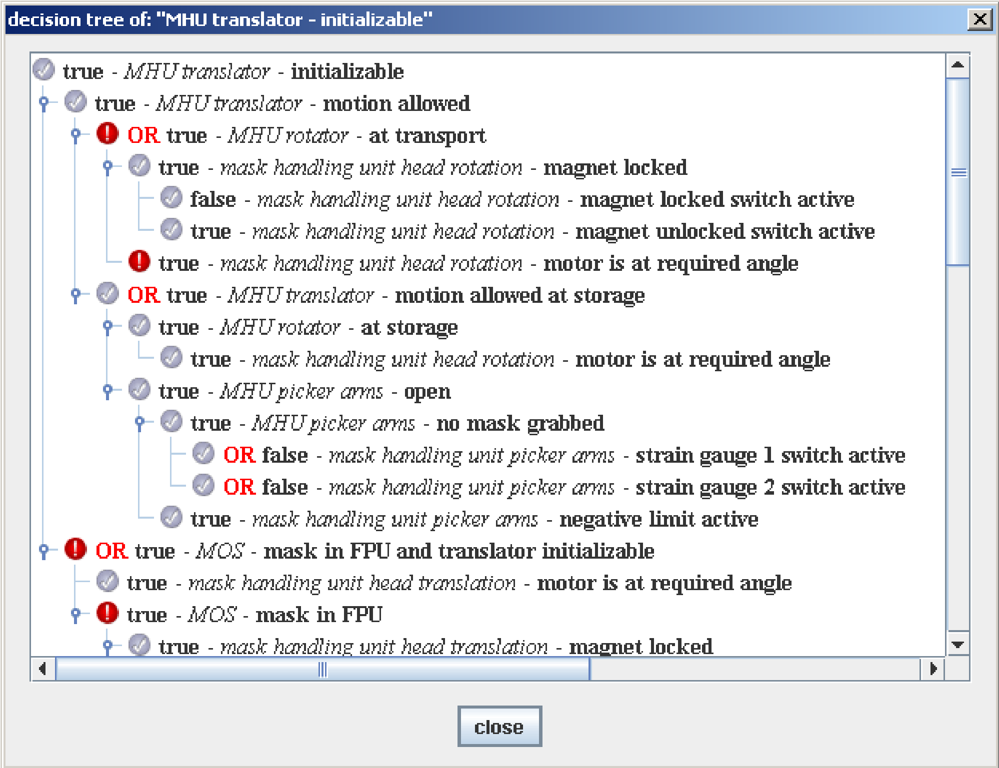


Figure 7.5: The dialogue to inspect decision trees.

user can activate a sequencing control on transition level and pause the sequence execution right after beginning. Additionally the skipping of sequences can be activated. When a sequence is executed its progress is visualised in a separate dialogue (see Figure 7.4). This dialogue enables the engineer to stop, to pause, to resume and to step through the execution on both sequence and transition level. Additionally direct access to the *MCU* is used to provide an emergency stop functionality. The realisation details of the sequence execution and inspection process are described in Subsection 6.1.4. Based on the interfaces defined there, the remote visualisation of the sequence execution is implemented in this control dialogue. Each element is represented by a unit name and a task description. The currently executing sequence element is marked with a green triangle. When a sequence element has been processed successfully it is marked as checked. Paused elements are printed in yellow. When an error occurs during execution the corresponding element is marked with a red cross. In this case more detailed information is presented to the engineer. In most cases the failure of a sequence element is found by checking a state. The result of this state check is presented to the engineer, too.

As sequences model the motion as finite state transition networks each sequence has its specific pre- and post-conditions. The checks of these pre- and post-conditions can be disabled during sequence execution. On the other side the states can be checked manually without the necessity of executing a sequence. The corresponding state check dialogue is shown in Figure 7.5. This dialogue enables the engineer to browse a decision tree. In each row the required value, the Boolean composition operator, the hardware identification string and the brief description of a node are printed. Additionally the nodes reflect the Boolean value of the last status check. A red exclamation mark is used for nodes that do not match the required value.

For the services of the *Instrument Tier* specialised *GUIs* exist. They combine the *Sequence Panel* with the panels of the *Control Tier* to realise an engineering access.

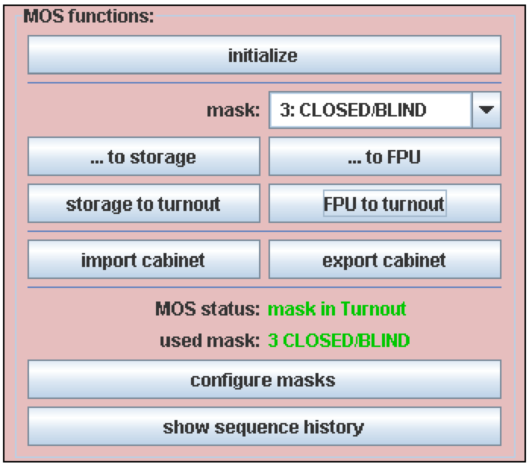


Figure 7.6: The panel to control the *MOS Unit Service*.

7.3.4 The MOS Unit *GUI*

The *MOS Unit Service* can be directly accessed by an engineering panel (see Figure 7.6). First of all this panel allows to initialise the *MOS Unit* by executing the corresponding sequence of the *MHU* translator. The 6 main functionalities of the *MOS Unit Service* (see Section 6.2) can be used to exchange the cabinet or to operate a selected mask. The current status is shown below the buttons that operate the unit. This status is automatically refreshed. The protocol of the last sequence execution can be accessed, too. It is shown in the same dialogue as presented in Figure 7.4. When a cabinet has been replaced, the configuration of the currently used masks can be directly changed. The corresponding dialogue allows to edit the mask identifier, the mask description and the *LMS* file (see Figure 7.7). Furthermore a cabinet position can be enabled or disabled and thus be excluded from normal operation.

7.3.5 The *GUIs* to Exchange the Mask Cabinet

For the convenience of the engineer the cabinet exchange can be done by a separate *GUI* (see Figure 7.8). The cabinet exchange *GUI* was designed to automatically detect the current status of the attached auxiliary cryostat and guide the engineers through the exchange procedure. Step by step the requirements of the cabinet exchange are prepared:

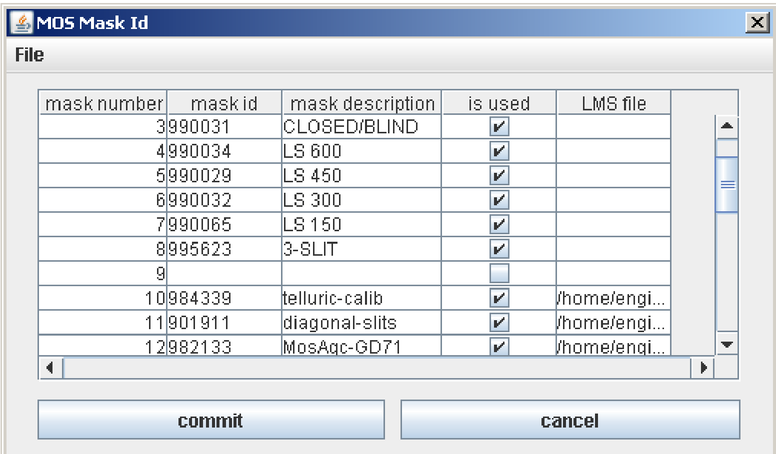


Figure 7.7: The mask configuration dialogue that has to be run after a cabinet exchange.

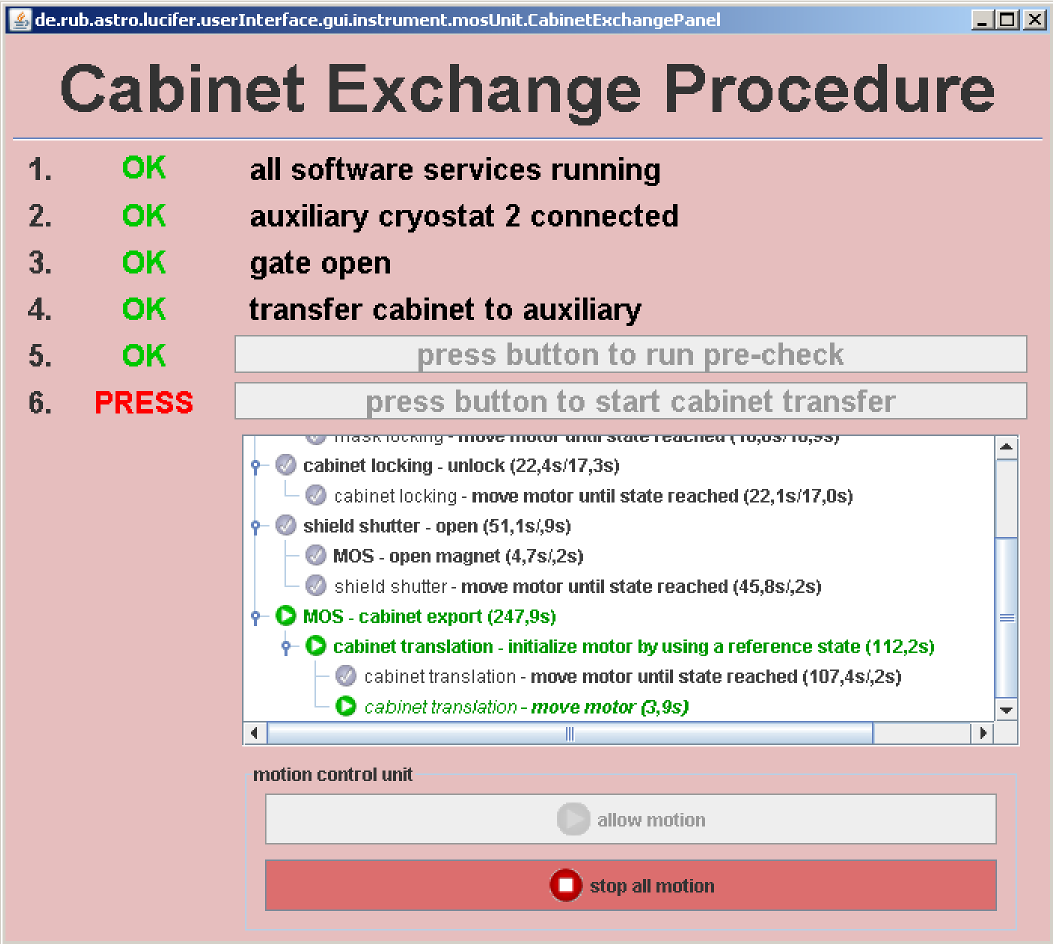


Figure 7.8: The panel to perform a cabinet exchange between *LUCIFER* and an auxiliary cryostat.

(i) A check of all involved services is done. If one of the required services is not alive, the engineer is informed to start it. (ii) The electronics are scanned to check whether an auxiliary cryostat was attached. The hardware identification of this cryostat is used to automatically select the required configuration data. (iii) The software waits until the flange has been evacuated and the gates have been opened. (iv) By analysing the current cabinet status of the *MOS Unit*, the exchange direction is determined. (v) Before executing a cabinet exchange the engineer is forced to check the pre-conditions. This check ensures that all sub-units are initialised and have the required positions. If this check fails the engineer is informed which actions are mandatory to continue. (vi) Finally the exchange is performed and its process is visualised.

A 2-button control of the whole exchange procedure is realised by implementing an automatic probing and a matched processing of the required tasks.

Chapter 8

LuciferVR

LuciferVR is a virtually realised instance of the LUCIFER instrument. It was built to allow improved pre-integration software tests, to train observers and to provide educational access. Therefore the hardware interaction with the real instrument is simulated. This chapter describes the simulation framework that is used to model the LUCIFER hardware. The framework is flexible and extendable to other applications. Further the integration of the simulator into the control software as well as its visualisation components are presented.

When developing software the time to test the software is often insufficient. Especially when developing a control software that interacts with external hardware a discrepancy in available and required test time can be discovered. The tests of the communication with the electronics and the software interaction with movable elements demand the presence of a correctly working hardware. Tests cannot start until hardware development is finished. Nonetheless in many projects everyone expects a fully functional control software at the moment of hardware completion. The physical tests of the hardware[1] and the likewise tests of the software-hardware interaction are often regarded contrarily. Of course software can be created in parallel to the hardware but the tests of the integrated system are still mandatory and the required time needs to be covered by the project schedules. To get out of this time dilemma in the *LUCIFER* project and to reduce the needed post-integration test time a virtual instrument was built.

LuciferVR is the virtual counterpart of the real *LUCIFER* instrument. A first prototype of the virtual instrument was presented in Polsterer (2003). This first prototype was completely re-developed and published in Polsterer et al. (2006). The primary reason to build a simulator is to create a test environment for the control software. On one side each unit of the software is functionally tested and regression tests ensure that changes to a module do not unintentionally interfere with other units. On the other side the integrated software system and its hardware interaction is tested (compare Section 2.1.3). A virtual instrument closes the gap between the regression tests and testing the control software with the integrated instrument. Therefore *LuciferVR* reduced the amount of time necessary to adopt the software to the real hardware.

By modelling the instrument in a simulator the motion times can be calculated. This motion time calculation is integrated into the *MCU* to offer exact time-out calculation for the motion commands. Another benefit is that the positions of all instrument units can be traced. Especially when using complex mechanisms like a *MOS Unit* a virtual instrument makes software development less time consuming. *LuciferVR* was used to test the motion logics in a pre-integration phase of the instrument and helped to reveal fundamental errors in an early stage of the software development process. Of course the detailedness of the simulator defines the grade of accuracy reached during tests. The simulation of the

[1] E.g., by using a telescope simulator to test instrument characteristics for different orientations.

LUCIFER instrument is specialised on the interfaces between the software and the hardware. The aspects of the physical motion of the opto-mechanical parts are incorporated in *LuciferVR*. Randomly generated motion errors are used to increase the robustness of the control software. Furthermore the torque and the speed of the stepper motors can produce simulated variance in expected and actual element positions. Other physical factors like the optical path and errors that occur e.g., by reason of instrument bending or element collisions are currently not simulated.

The testing is not necessarily limited on the tests done for software creation. The simulator is still used to maintain and create new logics. E.g., in December 2009 *LuciferVR* was utilised to write a completely new cabinet exchange mechanism. This new functionality was successfully executed in the cooled down instrument without the possibility to do a test run on the real instrument in advance. A failure during this exchange of the *MOS* masks would have implied a downtime to the whole telescope of at least 4 days.

Since *LuciferVR* simulates the instrument hardware the communication with the readout and telescope control software was tested differently. For readout tests a hardware that used simulators instead of *AD-Cs* was provided by the *MPIA* (see Section 3.4.3). The control software *GEIRS* additionally contains an integrated test mode that allows to run the software even without the readout electronics. To test the communication with the telescope the complete *TCS* needs to be installed. Once a *TCS* instance is running the *IIF* can be used to issue commands. A real simulator of the telescope does not exist however it is planned by the *LBT* software group to build one. Therefore this way of interaction allows to test the interface to the *TCS* only.

Besides using *LuciferVR* for software tests observers can be trained on the simulator. Technicians have been trained to perform a cabinet exchange without moving parts of the real instrument. This way of training allows to repeatedly operate a procedure without mechanically stressing the hardware so that the observers or technicians can gain experience. For future use of *LuciferVR* it is anticipated to have a virtual instance on-line in order to grant training access to observers. This will improve the efficiency of observations because logical errors in their observation runs can be figured out in advance. Additionally the observer will be made familiar with the *GUIs*.

Finally an on-line version of the virtual instrument could be used to provide educational access. By embedding the *Astrophysical Virtual Observatory (AVO)* imaging data could be created. In this case the influences of the instrument on the data would not be considered and only already reduced images could be returned. Noise, sky, filter transmission curves, dispersion and other artifacts would be neglected. Data for the spectroscopic mode would not be available. Thus a pre-recorded observation could be replayed to demonstrate the operation mode of *LUCIFER*. The virtually retrieved data could then be used to exercise the data reduction process on raw data including an astrophysical analysis.

8.1 The Simulation Framework

The simulation framework represents the central element of the virtual instrument. It is fundamental to use the control software stand alone. When designing this framework the prototype as described in Polsterer (2003) was discarded and a more flexible and extendable solution was created. A part of the simulation framework is dedicated to the generation of random numbers. These random numbers are important e.g., to add errors to the motion simulations and to prevent the **RMITimeGeneratorImpl** from generating numerical artifacts. The available random number distributions are based on the number generator of *Java*. Besides the **NormalDistribution** and the **UniformDistribution** com-

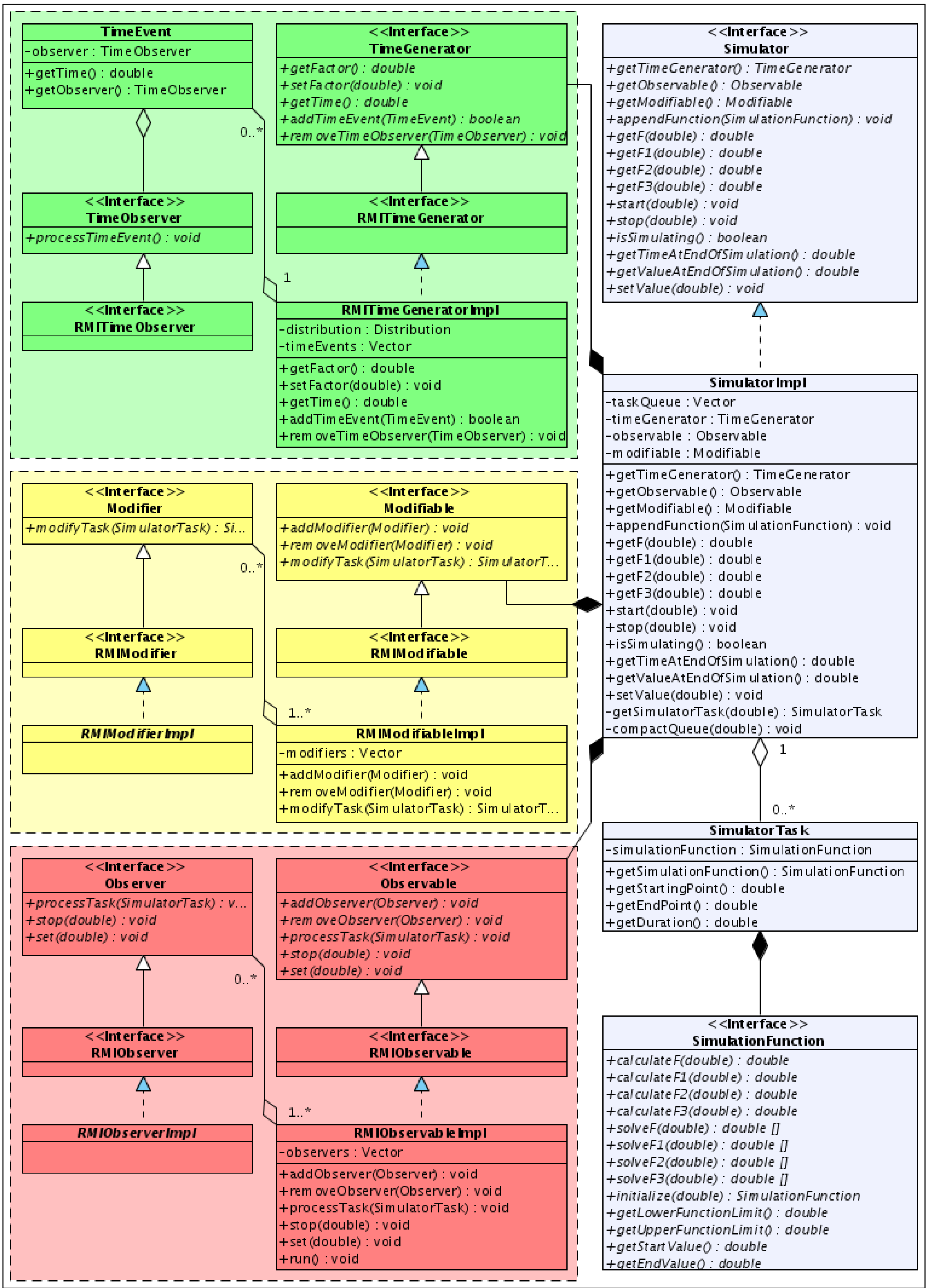


Figure 8.1: Class diagram of the simulation framework. On the right-hand side of the diagram the classes and interfaces are placed that define the simulator core while the left-hand side is split into:

GREEN : event scheduling and model-to-system time concatenation,
YELLOW : modification of simulation functions,
RED : observation of simulation functions.

posed distributions are implemented e.g., in **ExponentialDistribution** and **PoissonDistribution**.

The structure of the simulation framework is presented in Figure 8.1. Besides the simulator core the **simulation.simulator** package can be coarsely divided in 3 sub-domains. One for managing time events, one for modifying simulation functions and one for observing the simulation. The simulation framework is embedded in a distributed environment that provides remote access to observe and modify a simulation. All 6 interfaces starting with <`RMI`> in their name extend the **RemoteObject** interface and all 5 classes that end on <`Impl`> extend the **RemoteObjectImpl** class, both a member of the **de.rub.astro.util.net** package. This is done to incorporate basic *LCSP* remote method invocation capabilities.

By distinguishing between observers and modifiers, the behaviour of a simulated element is modelled. The main function of the **SimulatorImpl** is to manage the **SimulatorTask** objects that are generated by the concrete implementations e.g., of the stepper motors. When the current value that is simulated is polled the simulator processes the queue of tasks up to the current model time that is taken from the central **RMITimeGeneratorImpl** instance. This processing is done by solving the mathematical function that is included in every task. The observers and modifiers that are registered to a simulator will automatically be informed about new tasks. New tasks are first passed to the connected modifiers in the order they have been registered. After the modification the tasks are sent to the registered observers. To allow these modifications and observations remote callable implementations of the **Observable** and **Modifiable** interfaces are connected to the simulator. These implementations manage the registered **RMIObserverImpl** and **RMIModifierImpl** objects in a list, respectively. New objects can be added to these lists and older ones removed accordingly. **RMIModifierImpl** objects can fold the function that is embedded in a **SimulatorTask** with other functions. This is useful e.g., to add noise to a simulated motion or to cancel motions in case of reached limits. **RMIObserverImpl** objects are used to track the state of the simulation. An **Observer** conform object analyses a simulation function and can determine if a given value is reached. In this case the **Observer** can trigger the creation of an event in order to react appropriately. E.g., if the simulated position reaches the limits of the activation range of a switch an action event is scheduled at the pre-calculated simulation time. Moreover the implementation of **Observer** conform objects is outstandingly powerful to visualise and display the status of a simulation. Due to the system design these objects are native probes to the simulation core.

The event-based part of the simulation is realised by the **RMITimeGeneratorImpl** class which is responsible for scheduling the events. New events are created by the observers and sent to the time generator. Therefore this service contains a queue sorted by the model time. A real time simulation is achieved by linking the real system time to the model time. The **RMITimeGeneratorImpl** contains functions to modify the coupling factor between the real and the model time in order to run the virtual instrument e.g., in slow motion.

In this flexible framework everything that can be described at least as a function of time can be simulated. When a **RMIModifierImpl** object additionally implements the **Observer** interface to monitor other simulators multidimensional functions can be used to simulate complex mechanisms. In the next section these compositions of observers and modifiers and the modelling of *LuciferVR* are explained.

8.2 The Virtual Instrument

As already mentioned *LuciferVR* is a simulation of the *LUCIFER* instrument. Therefore the interaction with the motion controlling electronics is reproduced. To emulate all electronics interfaces the environment mapping monitors need to be covered, too. In the first prototype the hardware had been simulated and the corresponding commands and responses were emulated centrally in a single application (compare Polsterer et al., 2006). A value storage functionality was implemented to generate environmental values and to record the state of each monitor. This means that e.g., all parameters that have been sent to a monitor and might be queried later have to be stored. The value storage also provided the functionality of generating random numbers that correspond to specified distribution functions. Several random distributions have been available and could be used to describe e.g., the thermal behaviour of an element in connection with the current simulation time.

As it turned out later the implementation of the simulation capabilities in the environment observing/controlling services was too time-consuming and the emulation of the command interface prevented a parallel use of *LuciferVR* and the real hardware. Additionally the interface emulation based on a socket connection is more complicated and requires an internal system knowledge of each device to reproduce its temporal characteristics. Therefore the integration strategy of the virtual instrument into the *LCSP* was changed (see Section 8.3). The simulation of environmental parameters is not implemented yet in the services even though the interfaces and data structures already exist. Thanks to the central position of the *Journalizer* it is easy to run the system without the environment monitoring and controlling services.

To realise a modelling of the mechanical parts the **simulation.mechanics** package contains generic implementations of hardware simulators. Most important for the simulation of motion is the **MotionFunction** class. This class implements the **SimulationFunction** interface that was introduced in Section 8.1 to describe motion as a function of time. The used Newtonian position description allows to specify start position, start velocity, acceleration and jerk. As demanded by the **SimulationFunction** interface the first three derivations of the motion function are implemented as well. This enables the simulator and all connected modifiers/observers to calculate the position, speed, acceleration and jerk for any simulation time within the specified function limits. The **DisturbedMotionFunction** inherits all functionalities from the **MotionFunction** class and injects additional noise to the motion. This noise is specified by one of the provided random distributions. To model a moving element these simulation functions need to be appropriately created and added to a simulator.

In *LuciferVR* their is a distinction between the mechanical and the electronical position of an element. The mechanical position represents the real position of a hardware element while the electronical position is the expected position based on the calculation of the virtually created motor impulses. E.g., if a motor is stuck the element won't move physically whereas the internal position counter of the electronics continues to count. This discrepancy between real and expected motion needs to be modelled as well. Therefore each moving element of the virtual instrument consists of two simulators. One to simulate the physical motion of the element and the other to simulate the electronics behaviour. Both are necessary to create responses equivalent to those created by the real instrument. These simulators influence each other in that sense that both execute similar and interrelated simulation tasks. Depending on the added modifiers the simulator of the physical motion can be stopped while the simulator of the expected position is still running. This

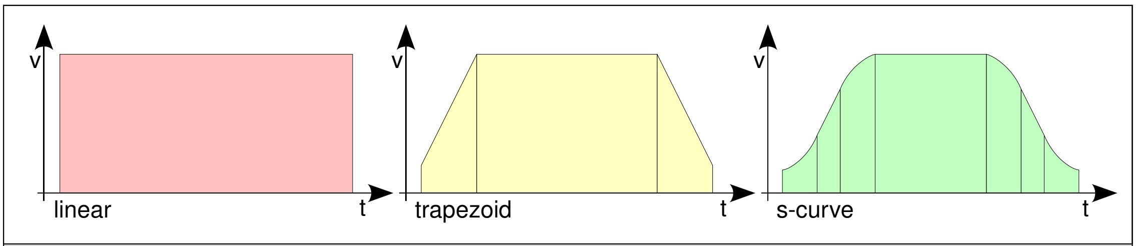


Figure 8.2: Comparison of the linear, trapezoid and s-curve motion profiles of a stepper motor.

will happen when e.g., the element stops because the torque is out of limits. On the other hand a limit switch which is modelled as an observer and is added to the physical simulator can stop both simulators.

Fundamental for motion simulation is the **RMIMotorImpl** class. This abstract class contains basic methods to manage two simulators. This includes observer administration, simulation function adding and starting/stopping of the simulators. The **RMISteppingMotorImpl** class is based on this abstract motor. This class is responsible to model the motion behaviour of a stepper motor. For a specified motion profile and a passed set of parameters the corresponding simulation functions are created and transferred to the simulators. E.g., an s-curve profile is realised by creating 7 simulation functions. The middle one is a simple plateau phase with no variance in speed while the first and last 3 functions are used for an acceleration and deceleration phase. These phases, in turn, are divided into functions with increasing, linear and decreasing acceleration/deceleration (see Figure 8.2). The implementation of a stepping motor is also used by the *MCU* to precisely calculate motion timeouts (see Section 5.2.2). To build a virtual instrument all simulators of the elements must use the same time generator. This is mandatory to synchronise the individual simulators. In contrast to the motors that create simulation functions the **RMIMotionLimit** class is used to stop the motion if a limit is reached by manipulating the simulation functions. These motion limits are realised as modifiers that are added to the simulator of the physical motion.

Besides the generic modelling elements presented above *LUCIFER* specific characteristics must be considered. Some of the motors of the *LUCIFER* instrument are equipped with angular resolvers or position encoders. To reproduce these position tracing capabilities in *LuciferVR* the simulator of the physical element position needs to be coupled with the electronics simulator. Depending on the type of used encoder the absolute angle or the incremental position of an element can be generated. For a correct incremental positioning the virtual electronics must be synchronised with the virtual hardware. This procedure performs identically to the behaviour of the real instrument hardware/electronics.

The motions within the *LUCIFER* instrument are not necessarily restricted to those motions evoked by a motor. Besides the motors the **RMINotchImpl** class can be used to generate motions. If a simulation function is finished within the limits of a notch and the motor is not energised a new motion is created to simulate the latching. Even though this implementation of a notch was designed to emulate hardware characteristics it can be used to simulated free motions of elements in case of disabled motor current.

In *LuciferVR* the stepper motor implementations can be equipped with switches. In the real instrument these switches are used to retrieve positioning information or limit the operating range of an opto-mechanical part. The **RMILimitSwitchImpl** and the **RMIPositionSwitchImpl** class are implementations of the **Switch** interface. These switch classes reproduce the behaviour of their physical counterparts. The **RMILimitSwitchImpl** is both an **Observer** and a **TimeObserver**. By observing a simulator of a physical element the moment a switch will be reached can be pre-calculated. This moment is then used

Listing 8.1: RMILuciferVRImpl.java, line 277 et seq. (Java Source File)

```java
277 motorAddress = new MotorAddress(3,1); // 3,1 grating selection, 200 steps/rev
278 motor = new RMISteppingMotorImpl(this.timeGenerator);
279 limitSwitch = new RMILimitSwitchImpl(this,LUCIFER_ELECTRONICS,motor,
        motorAddress,-0.1,false);
```

```java
281 limitSwitch = new RMILimitSwitchImpl(this,LUCIFER_ELECTRONICS,motor,
        motorAddress,3.85,true);
```

```java
283 new RMINotchImpl(this,LUCIFER_ELECTRONICS,motor,motorAddress,
        0.0,0.001,-0.1,0.1,false,1.0);
```

```java
289 getElectronics(LUCIFER_ELECTRONICS).setSwitch(2,new RMIPositionSwitchImpl(
        motor,motorAddress,new double[][]{{3.75,0.02}}));
290 motor.setAngle(Motor.ACTUAL_ANGLE,0);
291 getElectronics(LUCIFER_ELECTRONICS).getStepperMotors().put(motorAddress,motor);
```

to schedule a **TimeEvent** at the **TimeGenerator**. When the model time of an event is reached the corresponding **TimeObserver** is informed by remotely calling the **TimeObserver**.*processTimeEvent()* method. In case of a **RMILimitSwitchImpl** this call is used to stop the corresponding simulators. Special about the implementation of the **RMIPositionSwitchImpl** class is that it allows to observe the simulated position of a physical element in a cyclic way. This is useful for modelling elements that can freely turn and for this reason can reach positions periodically.

Another feature of the *LUCIFER* instrument is to use magnetically activated locks to prevent elements from moving. This fixing mechanisms are equipped with switches in order to display their current status. To emulate these magnetic locks the **MagnetLock** and **MagnetSwitch** classes exist. An activated lock prevents the physical simulator from moving. As for the real hardware this may lead to an increasing discrepancy between the electronics counter and the positions of the element.

In Listing 8.1 the creation of the *Grating Unit* as a simulated element is described. In line 277 the motor address is created. Here the numbers <`3,1`> equal the address of the motor at the real control electronics. This address object is used to identify the element at the virtual instrument. Next a new instance of a **RMISteppingMotorImpl** is created. As mentioned above the unique time generator is assigned to this object. After the motor with the two simulators is created 2 limit switch objects are instanced. The constructor of these objects automatically registers the limit switches to the corresponding simulators of the specified motor. Note that the position of the limit switch is specified in revolutions of the element instead of steps of the motor. These two switch objects are stored later in the virtual instrument and can used for switch status queries by the services of the control electronics. In case of the *Grating Unit* 4 notches exist to perform a repeatable selection of the gratings/mirror. In line 283 one of these notches is created. Besides the motor association the centre, the size, the lower and the upper limit are specified. Additionally this notch is declared to be non-cyclic. The last parameter is used to set the transmission ratio between the motor and the element. This value is required to create the simulation functions correctly for elements that use a gear between the motor and the element. As for the limit switches the values of a notch are specified in revolutions of the motor. Finally a position switch is assigned to the motor and is stored together with the motor at the corresponding electronics instance. The different electronics instances are necessary to emulate the existence of two autonomous control electronics for both the *MOS* hardware and the other instrument units.

In comparison to the other instrument units it is more complicated to model the *MOS Unit* in *LuciferVR* because the masks are freely movable elements with no restriction in

Listing 8.2: RMILuciferVRImpl.java, line 522 et seq. (Java Source File)

```
522 new RMIMaskGrabWatcher(clampY,-7,false, new RMIComposedSwitchImpl(new Switch[]{
523       new MaskInGrabberSwitch(this), // mask in grabber
524       new RMIPositionSwitchImpl(rotator,rotatorAddress,new double[][]
              {{19.4222222,0.01}}),       // rotator at fpu
525       new RMIPositionSwitchImpl(translator,translatorAddress,new double[][]
              {{-11.8,0.01}})             // translator at hold
526   })) {public void processGrab() {setMaskInFPU(true); }
527 };
```

location. However their current positions affect the status of several hardware switches. E.g., the presence of a mask in the cabinet is inspected with a dedicated hardware switch. For that reason the **MaskInCabinetSwitch**, **MaskInFPUSwitch** and **MaskInGrabberSwitch** can be used to test whether a mask is present in a simulated element. **RMIComposedSwitchImpl** objects can be used to bundle position, limit, magnet and mask switches to create new composed states. This is inevitable to reproduce the switches that are influenced by the positions of the masks. A mask can either be in the grabber, the cabinet or the focal plane of the instrument. If a mask is lost this would cause serious problems in *LuciferVR* as well as in the real *LUCIFER* instrument. To process the grabbing and releasing of a mask the abstract **RMIMaskGrabWatcher** and **RMIMaskReleaseWatcher** class can be concretised. These watchers use a composed state to determine a transition of a mask status. In Listing 8.2 the **RMIMaskGrabWatcher** of the *FPU* y-clamp is presented exemplarily. It is specified that a position of -7 revolutions must be negatively exceeded by the y-clamp to activate the watcher. In addition a composed switch is used to ensure that a mask is in the grabber, the rotator is pointed towards the *FPU* and the translator is at the hold position. If all these criteria are fulfilled the **RMIMaskGrabWatcher**.*processGrab()* method is called. The implementation in Listing 8.2 calls an internal method of *LuciferVR* in order to change the virtual mask status of the instrument. This new mask status will then affect all queries of switches that depend on a mask in the *FPU*.

8.3 Integration into the Control Software

The simulation was projected to cover both the hardware of the instrument and the monitoring devices. *LuciferVR* was planned to be a separately running simulation that emulates the command interfaces of the used electronics (compare Polsterer et al., 2006). For this reason the first prototype used the parser of the *MCU* (compare Section 5.2.2) to handle the bidirectional communication between the electronics services and the simulator. This first approach of integration even allowed tests of the controlling software services itself. The provided test electronics was equipped with just one stepper motor. Therefore the simulation was the only solution to verify the correct handling and the synchronisation of simultaneously executed commands by the *MCU Service*. Especially thread synchronisation bugs could be fixed before using the real instrument.

The intended way of emulating the firmware of the controllers was found inappropriate. Changes to a firmware enforced modifications both to the controlling services and to the firmware emulation of the virtual instrument. The applied flexible two-way parser was capable of translating command and response symbols in both directions. Even though the flexible parser was used for the firmware emulation the internal processing of command sequences had to be reproduced, too.

Instead of running the virtual instrument as a stand alone version in the final realisation it is hard-wired with the services. To integrate *LuciferVR* in the control software a direct connection is created between the simulation and the services that access the control electronics. Therefore probes are implemented in the controlling services for the communication with the virtual instrument. These probes directly call remote methods of the *LuciferVR Service*. This service provides all methods to interact with the virtual instrument. E.g., it issues motion commands that are passed to the appropriate simulators or calculates the status of a specified switch.

The *HIRAMO* controlling service, the switch electronics service of the *MOS Unit* and the service that interacts with the motion control electronics have been modified to use *LuciferVR*. By activating the probes via the **-use_lucifer_vr** command line argument the connection to the virtual instrument is enabled. Thereby the communication layer between the service and the emulated control electronics is bypassed. This way of inter-service communication minimises the complexity to administrate parameters, to parse commands/generate responses and to emulate the behaviour of the firmware. When neglecting the command transmission time of the real electronics *LuciferVR* reacts within milliseconds identically to the real hardware. Another benefit of this kind of integration is that it allows to split the commands. This means that commands that are sent to the real instrument hardware can be sent to virtual instrument as well. Thus *LuciferVR* can run synchronously in order to track the state of the instrument. In the next version of *LuciferVR* it is planned to implement a status synchronisation mechanism that uses the information of the hardware switches to adjust the position of the simulated elements. This comparison of target and actual positions will be helpful for troubleshooting in case of an error.

To run the control software without the real control electronics the **-simulate** command line argument must be specified (compare Appendix B). This enables the evaluation of simulator generated responses. Once activated the user can not distinguish whether a service is communicating with the real electronics or with its virtual counterpart. Instead of waiting for a controller response that is created when a motion is finished the service waits for the simulator to stop its motion. The potential of *Java* to handle exceptions simplifies the interaction with *LuciferVR*. In the first version that used a full emulation of the firmware an activated limit switch caused a simulated motion to end and generated a corresponding response string. In the current version of *LuciferVR* a reached limit switch stops the simulation and the method that started the motion throws a limit-reached exception. This exception is handled by the control service directly. For a parameter query the currently used parameters can be returned without sending a command and analysing the response string. Only the provided functionalities of the services of the *Control Tier* can be used. Thus these services ensure a full transparent use of both the real hardware and the virtual instrument.

8.4 Visualising a Virtual Instrument

Even though most of the time was spent to create the simulation framework and to model the instrument *LuciferVR* has an outstanding visualisation mechanism, too. *LuciferVR* uses *Java3D* to display the status of the simulated instrument in three dimensions. The 3D representation of *LuciferVR* can be turned, translated and zoomed to change the view. For a future version it is intended to have the option of automatically moving and zooming to an active element. Before the parts of the *LUCIFER* instrument could be visualised the *CAD* drawings were pre-processed. First the *CAD* data was converted into a format supported by *Java3D* to build a scene graph. Next for all parts of the *LUCIFER* instrument the number of polygons was reduced by 90%. This was required for a fast reacting display. Figure 8.3 compares the raw *CAD* model with the compressed version and demonstrates

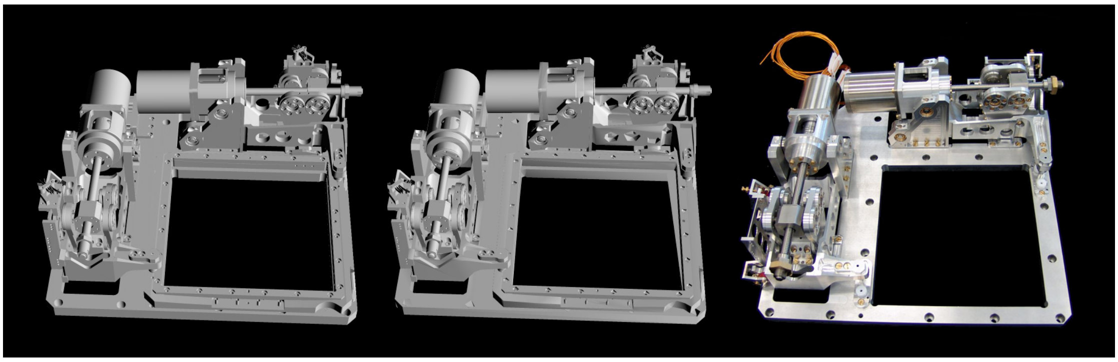


Figure 8.3: Comparison of the unprocessed 3D-model (left) with the reduced version (middle) and a photo of the assembled unit (right).

the similarity between a real unit and its simulated visualisation. To increase the clarity of the display the cryostat is omitted, the inner structure is set transparent and the individual units are painted in different colours (see Figure 8.4).

In order to show the motion of a component the display registers to the simulators of the motors which drive the elements of the instrument. The computational load to do the visualisation is distributed because the current position of an element is not necessarily calculated within the virtual instrument. The visualisation of elements is realised as

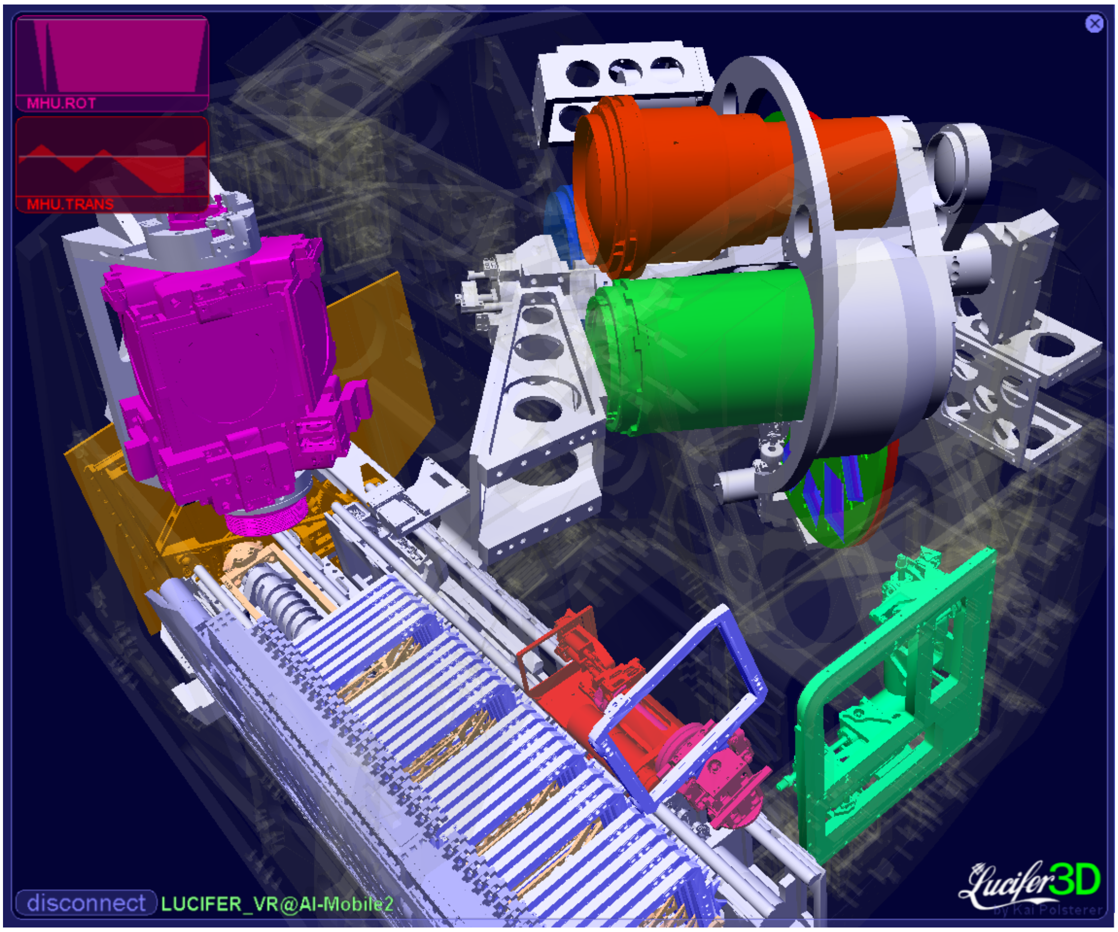


Figure 8.4: 3D representation of *LuciferVR*. Overlay plots visualise the speed of the moving elements (upper-left corner). The connection status of the visualisation panel is indicated (lower-left corner) and can be changed (nearby button).

Observer conform objects. Therefore each visualisation object is notified of new simulation tasks, value changes and stops of the simulator (compare Section 8.1). The included simulation function of a task is utilised to calculate the position, speed and acceleration of an element locally. By describing the translation and rotation of a moveable part as a function of the simulated motor position the three dimensional motion is modelled. The same information is additionally used for the overlay display of the motor speed (see Figure 8.4).

The visualisation capabilities of *LuciferVR* enables the engineer to understand what is going on in the *LUCIFER* cryostat and to track the instrument status in a reduced and human friendly way. Without having the 3D display the engineer would have to gather the instrument status information from the more detailed engineering panels. Another advantage of the 3D display is that an observer will be able to see the instruments mechanics working. This is much more intuitive than looking at a progress bar. Additionally the observer will be able to use the instrument control software prior to an observation visit. This improves the familiarity with the *GUIs* and reduces the amount of logical errors in the prepared observation runs. Therefore it increases the scientific gain of the *LUCIFER* instrument.

Part III

Science with LUCIFER

Chapter 9

NIR Observations of NGC 1156

This chapter presents Ks and narrow band H_2 images of the dwarf galaxy NGC 1156 taken by the LUCIFER instrument. After the data set is described the applied data reduction steps are shown as well as the required special treatment of this observation. Finally the results of the analysis are discussed. A new distance measurement of the galaxy was done and a first direct detection of structured warm molecular hydrogen in the outer parts of a dwarf galaxy is presented.

NGC 1156 is a bright irregular dwarf galaxy similar to the *Large Magellanic Cloud (LMC)*. It is located in the Aries constellation and shows a global starburst. In the *Third Reference Catalogue of Bright Galaxies (RC3)* (de Vaucouleurs et al., 1991) this galaxy is classified as *Magellanic* with a total B band magnitude of 12.32 mag and a size of 3.′31 are specified for this galaxy. This size perfectly fits in the *FOV* of *LUCIFER*. Karachentsev et al. (1996) determined a distance of 7.8 Mpc by using photometry of the three brightest red and blue stars. *NGC 1156* is one of the most isolated galaxies in the local universe with no visual companion within 10°. (Karachentseva, 1973). 21-cm (neutral hydrogen) observations of *NGC 1156* with the *Arecibo* telescope show an undisturbed *HI* distribution and a small dwarf companion at 35′which is equivalent to 80 kpc in projection (Minchin et al., 2010).

9.1 The Data Set of *NGC 1156*

On November 10th, 2010 the dwarf galaxy *NGC 1156* was observed with the *LUCIFER* instrument and its *N*3.75-Camera[1]. During this observation the broadband Ks filter[2] and the narrowband H_2 filter[3] were used. As readout strategy the double correlated readout mode was chosen. In this mode the detector is initially reset before a first readout. To create the final frame the data of this first readout is combined with the data of a second readout which is executed after the specified integration time. In each frame a *Number of Detector Integrations (NDIT)* with a specified *Detector Integration Time (DIT)* are combined. The resulting data set is presented in Table 9.1. Each of these ≈ 300 files has a size of 16 MB.

The data set consists of frames for data reduction and science frames. The data reduction subset contains dark frames and flat fields. The former frames are required to compensate the dark current of the *NIR* array. The latter ones compensates the differential detector sensitivity, instrumental transmission and vignetting characteristics as well

[1] The *N*3.75-Camera has a *FOV* of $4' \times 4'$ and a corresponding pixel scale of 0.″12/pixel.

[2] The $Kshort$ (Ks) filter has a central wavelength of 2.163 μm, a *FWHM* of 0.270 μm and an average transmission rate of 86.8%.

[3] The H_2 filter has a central wavelength of 2.124 μm, a *FWHM* of 0.023 μm and an average transmission rate of 84.9%.

Purpose	Type	Filter	*NDIT* × *DIT*	Files
Science	Object Frame	Ks	12×5 s.	15
		H_2	3×20 s.	60
Calibration and Data Reduction	Sky Frame	Ks	12×5 s.	5
		H_2	3×20 s.	20
	Dark Frame	Blind	12×5 s.	10
			3×20 s.	10
			1×2 s.	11
			1×3 s.	10
	Flat Field	Ks	1×2 s.	94
		H_2	1×3 s.	63

Table 9.1: The *LUCIFER* data set of galaxy *NGC 1156* which consists of 298 *FITS* files.

as illumination effects that are produced by the telescope. Additionally sky frames have been taken which are required to compensate the sky emission. Besides the data files which are required for processing the raw data, 75 science observations exist with an observation time of 75 minutes. 25 minutes of the total observation time of 100 minutes is used to compensate for the fluctuations in sky emission. In total the galaxy *NGC 1156* was observed 15 minutes in Ks and 60 minutes in H_2. From the first scientific exposure to the last it took 156 minutes to obtain the data. The overhead of 56 minutes is related to telescope interactions, changes to the instrument setup, file storage and observation script processing.

The observation layout was intended to have three main pointings on the galaxy and one off pointing on the sky. Unfortunately the dithering around these pointings did not work because of script execution problems. Therefore the resulting science data set has been created with two pointings on the object and only one pointing on the sky without any dithering in position. This limitation in different pointings requires a special treatment of the data set (see Subsection 9.2.2).

9.2 The Reduction of the *NIR* Data Set

In contrast to standard observations in the optical the reduction of a *NIR* data set requires special treatment. A single scientific raw frame does not necessarily show the scientific target (see Figure 9.1 top left). The target is hidden by the disturbing effects of the instrument, the telescope and the emission of the sky. The applied data reduction steps are presented in the following.

9.2.1 The Standard Processing Steps

The presented data reduction steps have been executed by using the *Munich Image Data Analysis System (MIDAS)* (Banse et al., 1983) of *ESO*. This software package provides all required functionalities to process data frames. To improve the data processing all steps of the data reduction of *NGC 1156* have been scripted and were executed automatically. When user interaction was required the retrieved information was stored in tables to allow for a faster reprocessing, if required.

All frames are affected by a detector element specific dark current which is time dependent. A dark frame is required to compensate for this effect. Such a frame is taken by blocking the optical beam with blind filters and a blind mask. It is important to create the dark frames with the same detector readout strategy and integration time as the science

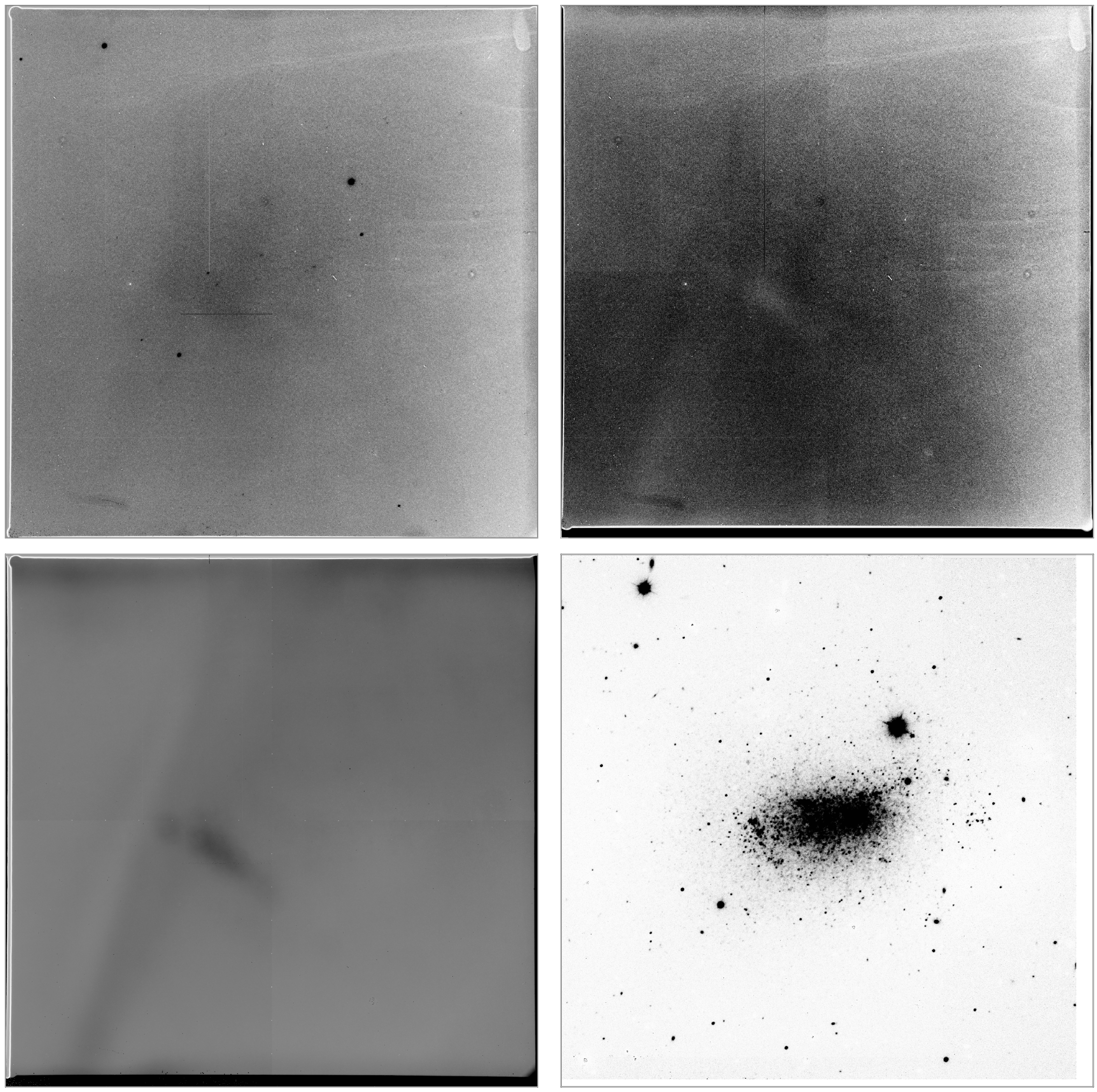


Figure 9.1: A sample of the different types of frames which are combined to a scientific image. A raw exposure of 1 minute of the galaxy *NGC 1156* in the *Ks* band (**top left**). The corresponding master flat field which has been processed and normalised (**top right**). A sky frame in the *Ks* band which has been used for sky reduction (**bottom left**). The final image created by combining 15 processed raw frames (**bottom right**). All frames are presented with an inverted grey scale plot.

observations. In a first step these dark frames are grouped by integration time and combined per time block by using the median value of each pixel. Then the resulting master darks are subtracted from all frames.

In the next step the influence of the pixel dependent detector sensitivity, the illumination and the vignetting characteristics are compensated. For this reason so called flat fields are required. A flat field is an observation of a homogeneously illuminated area. This observation allows to determine the efficiency of each pixel with respect to detector, optical elements and telescope. For the presented data set the flat fields have been created by observing the sky at twilight in both filters. Each flat field was automatically analysed and a map was created which contains the pixels with values that deviate more than 3σ from the mean value of the frame. This map is then used to replace the values of these pixels by linearly interpolating over the values of their neighbours. In each filter, a master flat was created by averaging over the available flat fields. The highest 15 and lowest 15 values of each pixel have been excluded from this calculation. Finally the master flats have been normalised to 1. The master flat of Ks is presented in Figure 9.1, top right. All science frames are divided by the corresponding master flats to create homogeneous and flat data frames.

The observation strategy which was used to compensate the fluctuation in sky emission directly affects the processing of the data. After 3 observations on the target the telescope was pointed to a sky position and a sky frame was acquired. The sky frames which enclose a target block are averaged to retrieve a mean sky frame (see Figure 9.1 bottom left). For each of the observed target blocks a separate sky frame is required. These sky frames are used to subtract the interfering sky and telescope emission.

As the final step the flux of all science frames was normalised to counts per second, the frames have been aligned and co-added to a single Ks and H_2 image (see Figure 9.1 bottom right). The process of aligning the images is the only step where user interaction was required. Coarsely selected stars are used to match the individual frames in position, rotation and scale.

9.2.2 Problems with the Data Set / Special Data Processing

Due to problems during script execution no dithering was applied to the telescope pointings. Additionally the pointing for the sky frames was half on the scientific target. This leads to a contamination of the sky frame with extended diffuse emission of the galaxy. Therefore the described standard data processing steps produced a science image with a sloped background. This leads to errors in the photometric analysis. Additionally the stars which appear in the sky frames can not be removed accordingly, as only data for one pointing exists. To improve the quality of the sky frames and remove the mentioned artefacts, special processing steps have been performed. These steps make use of the sky information which is contained in the science frames of the target, too. The advanced sky frame processing steps are: (i) The sky frames which directly have been taken in advance and after a block of science frames where combined with an average composition, as in the standard processing. (ii) A copy of the enclosed science frames is created and the flux level of their background is adjusted to the same value as of the averaged sky frame. (iii) For each pixel the minimum value of the averaged sky frame and the adjusted science frames is used to create a cleaned sky frame with a minimum of contamination by stars and extended emission of the galaxy. The result of the improved sky frame processing is presented in Figure 9.2.

Detailed analysis of residual images between a standard sky and an advanced sky revealed artificially added faint structures which are remnants of the galaxy emission in the sky frame. These remnants could lead to false detections in the narrowband analysis. Therefore the narrowband analysis was done on basis of the images reduced with the standard processing steps.

9.2.3 The Processing of the Narrowband Image

The narrowband H_2 image requires an additional processing step to ensure that it only contains the flux of the molecular hydrogen. The continuum emission which can be found in the H_2 filter wavelength range is created by other radiation mechanisms and must be subtracted. Therefore the continuum Ks image was scaled to the H_2 image by using several stars as continuum sources. This led to a scaling factor of 1/12.8 which is in the order of the ratio between the filter transmission window sizes. As both images have a similar *PSF*

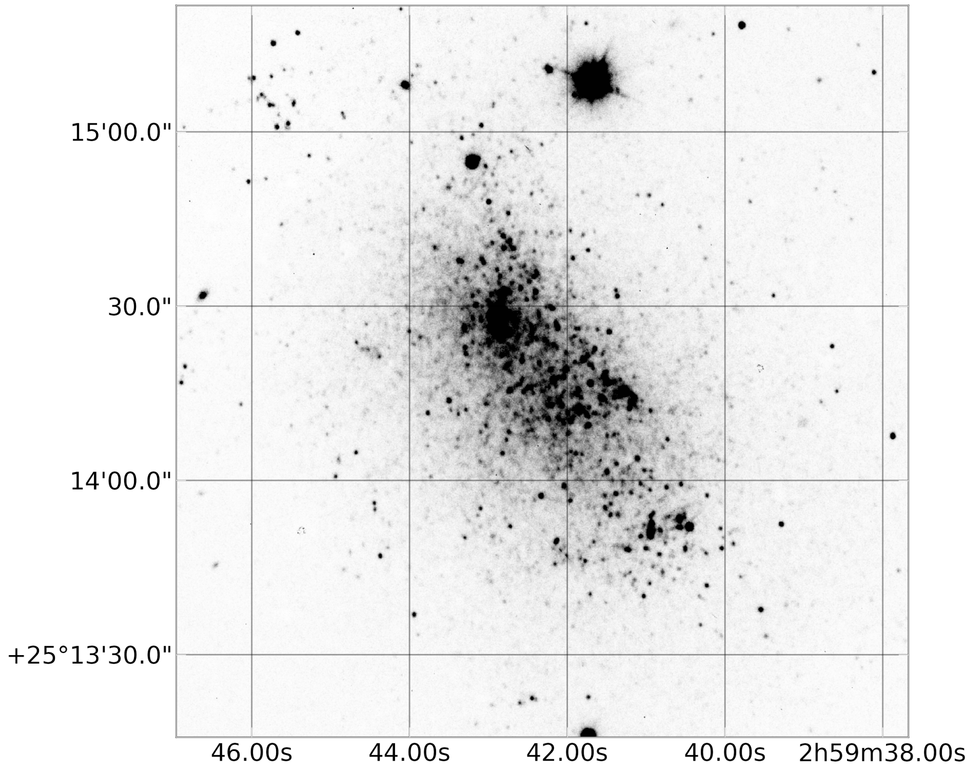


Figure 9.2: *LUCIFER* image of the galaxy *NGC 1156* with the Ks filter. This image is a combination of 15 exposures, 1 minute each.

(*FWHM* $\approx 0\farcs6$) no additional convolution was necessary. The scaled Ks image was used to subtract the continuum of the H_2 image.

9.3 The Analysis of the Data Set

The processed scientific images are analysed differently. The Ks image is used to create a photometric catalogue. This catalogue is combined with the results of *HST* archival data to derive a distance to *NGC 1156*. The H_2 image instead was inspected for structured emission in the outer parts of the galaxy. The results of these analyses are presented in this section.

9.3.1 The Photometry of *NGC 1156*

A photometric data set with 724 objects was extracted from the Ks image by using the *SExtractor* software package (Bertin and Arnouts, 1996). This software determines the magnitude of an object by calculating the flux within an aperture. Thereby the background level of each object is collected automatically and included in the magnitude calculation. An aperture of 10 pixels was chosen. This is equivalent to twice the *FWHM* of the *PSF*. The zero point of the Ks image was determined with 8 reference stars of the *2MASS* catalogue to be 23.95 mag (*Vega*). During the commissioning of the *LUCIFER* instrument the zero point was determined to have a value of 24.00 mag. This value is consistent with the *2MASS* calibrated zero point value within the measurement uncertainties of the photometry of Δmag = 0.1 that has been empirically determined during the zero-point calibration. The extracted magnitudes have been corrected for a foreground extinction of 0.082 mag (Schlegel et al., 1998).

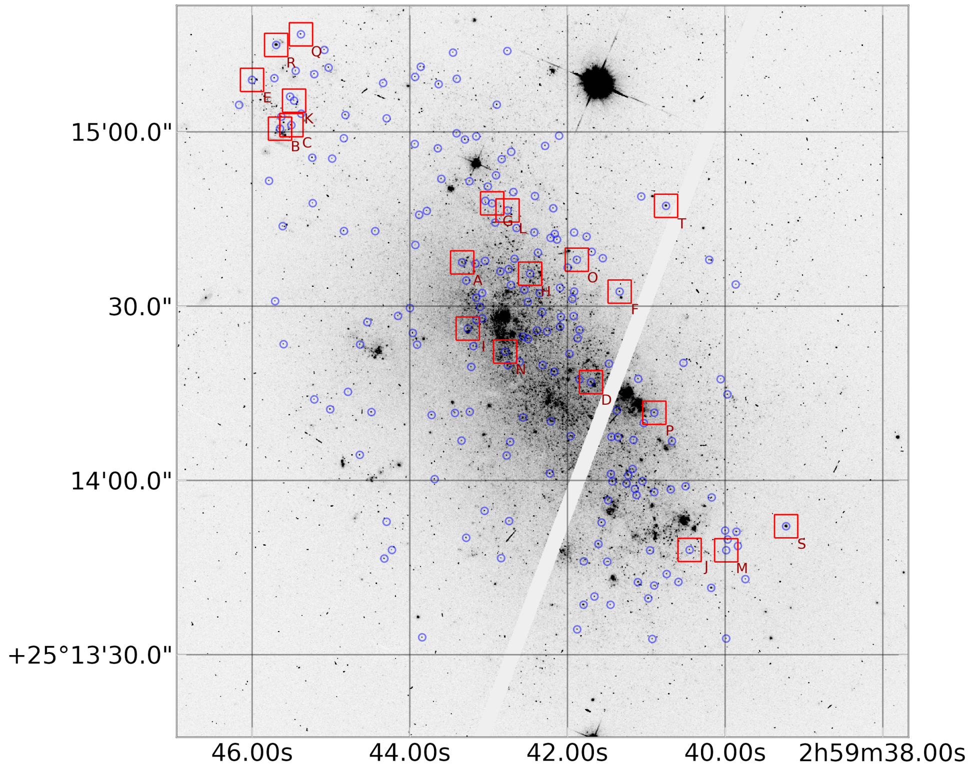


Figure 9.3: *HST* archive image of *NGC 1156*. A single 140-second exposure taken by the *ACS* system and the *F625W* filter (R band). Blue circles mark the objects which have been used for Ks and R band photometry. Red boxes mark objects that have been inspected in detail.

The Ks photometric data set of *NGC 1156* was extended by data of the *Hubble Space Telescope (HST)* legacy archive. In 2004 *NGC 1156* was observed with the *Advanced Camera for Surveys (ACS)* (Proposal: 9892/P.I.: Jansen). The observations were carried out with the *F625W* (R) and the *F658N* filter ($H\alpha$). As the available photometric data set is given in AB magnitudes a conversion to *Vega* magnitudes was applied ($R_{\rm Vega} = R_{\rm AB} - 0.055$ (Frei and Gunn, 1994)). Additionally the R band extinction of 0.599 mag by the *Milky Way* was corrected (Schlegel et al., 1998). After matching both photometric catalogues, rejecting objects with uncertain magnitudes and confining the position on the galaxy, 197 objects remain. In the *HST*/*ACS* image (Figure 9.3) these objects are marked with circles.

9.3.2 The Distance to *NGC 1156*

The distance of *NGC 1156* was determined on basis of the Ks photometry. An analysis of the size of superbubbles identified in the $H\alpha$ image (see Figure 9.9, right) taken by *HST*/*ACS* constrained the distance to 5 Mpc to 10 Mpc. Chu et al. (1995) specified typical sizes of superbubbles in the *LMC*. Their values have been taken as reference for the distance estimation. The covered magnitude range of the data set implies that the resolved stars are intrinsically brighter than red giants and therefore the *Tip of the Red Giant Branch (TRGB)* method (Lee et al., 1993) could not be applied. The objects of the photometric catalogue are either red supergiants, extended objects like globular cluster, background sources or objects with a blended photometry in Ks.

The galaxy *NGC 1156* is similar to the *LMC* in many aspects, especially mass and metallicity (see Minchin et al., 2010). In Oestreicher et al. (1997) a catalogue

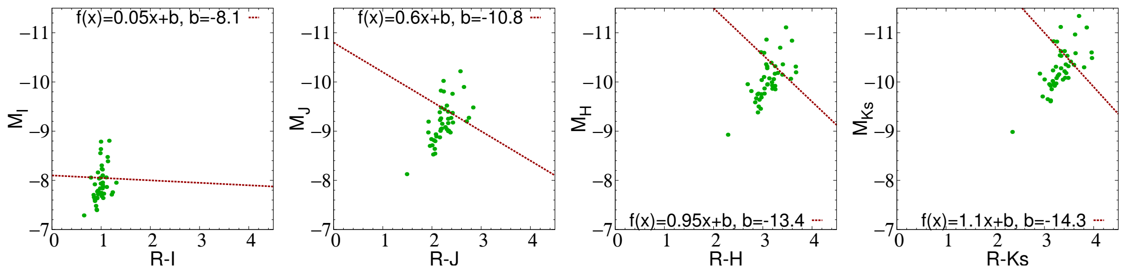


Figure 9.4: Colour magnitude diagrams of red supergiant candidates of the *LMC* (Oestreicher et al., 1997) for different colour terms. In each diagram a linear function is fitted to the upper edge of the main aggregate of stars. The best linear separation between the main aggregate and outliers is achieved with broader colour terms.

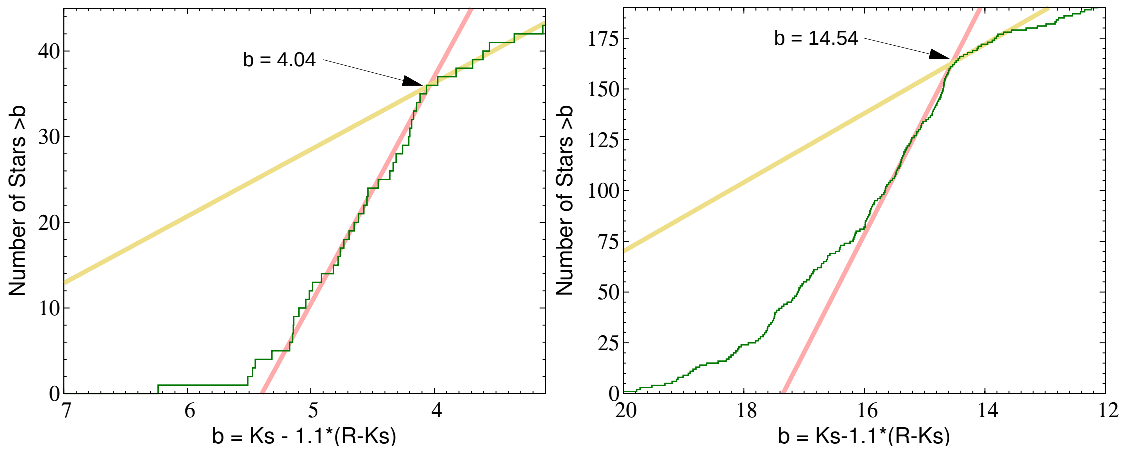


Figure 9.5: Alignment of the upper edge of the main aggregate of the *LMC* (**left**) and *NGC 1156* (**right**). The density distributions perpendicular to the linear function of Figure 9.4, right, are analysed to determine the turning-points.

of red supergiant candidates in the *LMC* is presented. A comparison of different colour magnitude diagrams of this *LMC* data set indicates a main aggregate surrounded by outliers (see Figure 9.4). The upper edges of this main aggregate are fitted with a linear separation function for each diagram. The fit of the $R-Ks/Ks$ colour magnitude diagram was used to determine the actual turning point in the density gradient which proceeds perpendicularly to this linear function (see Figure 9.5). The difference in b-values can be directly transformed into a magnitude offset. The presented approach to measure distances is new. The structure which is aligned is most likely associated to the *Humphreys-Davidson limit* a physical upper luminosity limit for supergiants. Together with a distance modulus of the *LMC* of $m - M = 18.5$ mag a distance modulus of $m - M = 29.0$ mag was determined for *NGC 1156*. This leads to a distance 6.3 ± 0.4 Mpc for this galaxy. The uncertainty of ± 0.4 Mpc (± 0.14 mag) is caused by the measurement precision of the photometry of Δmag $= 0.1$ and the exactness of the density alignment of Δmag $= 0.1$.

In Figure 9.6 the $R - Ks/Ks$ colour magnitude diagram of the galaxy *NGC 1156* is presented. The absolute Ks magnitudes are obtained by using the calculated distance modulus. For comparison the red supergiant candidates in the *LMC* are plotted, too. As a representative for the class of highly luminous red hypergiants *Variable A* in *M 33* is given (Humphreys et al., 2006). This luminous object is situated right below the edge of the main aggregate. Additionally 4 theoretical isochrones which have be improved for asymptotic giant branch stars by Marigo et al. (2008) are plotted. These isochrones

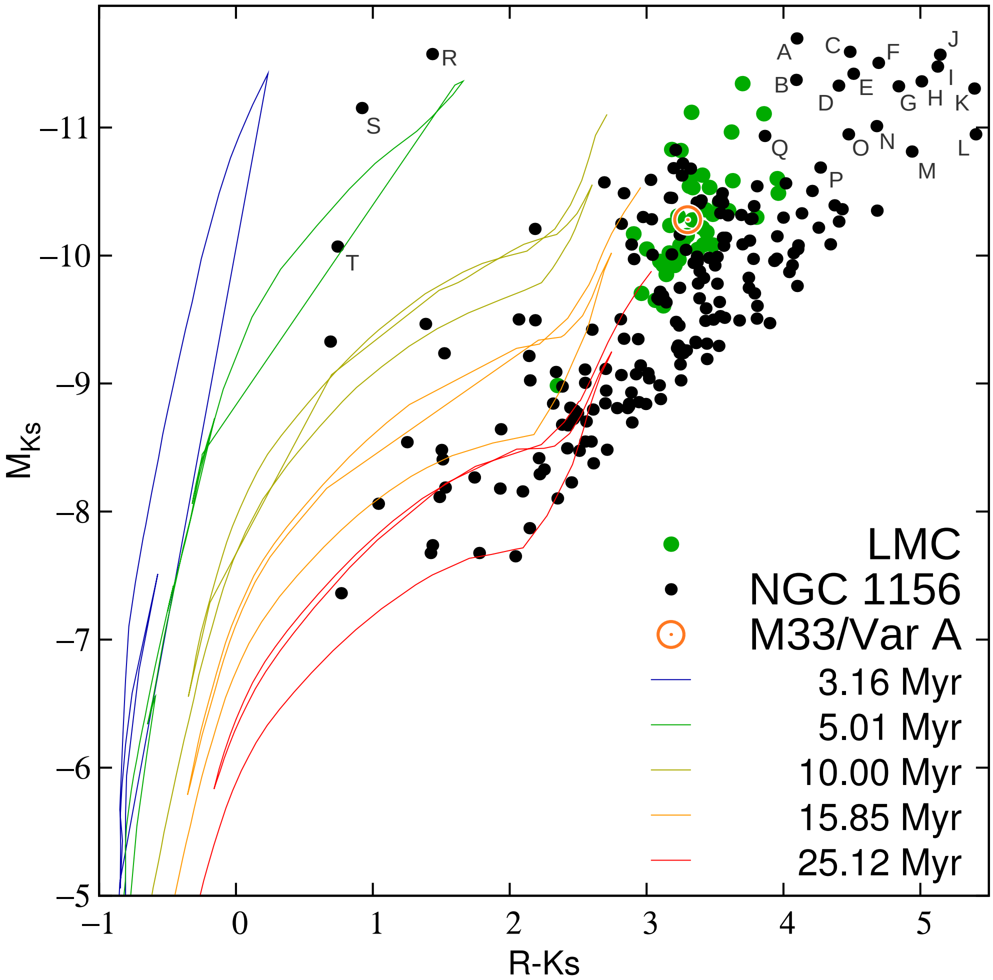


Figure 9.6: Colour magnitude diagram on basis of the *LUCIFER* Ks band and *HST*/*ACS* R band photometry of *NGC 1156*. The best match to *LMC* values of OESTREICHER ET AL. (1997) was achieved for a distance of *NGC 1156* of 6.3 Mpc. *Variable A* in *M 33* is plotted as a reference point for a highly luminous red supergiant. The plotted theoretical isochrones for asymptotic giant branch stars (MARIGO ET AL., 2008) allow to determine the age of the stellar population. Objects with conspicuous positions in the diagram are marked by letters and are presented separately in Figure 9.7.

are plotted for stars with a metallicity of $Z = 0.008$, which is comparable to the *LMC*. They allow to determine the age of the stellar population and to constrain the duration of the starburst to at least 25 Myr. The progression of the isochrones fits well with the found distribution of stars, too. The slight mismatch of the isocrones for very red objects is a known problem of the colour transformation from T_{eff} due to strong absorption features in the spectrum, stellar winds and the circumstellar environment. Objects with conspicuous positions in the diagram are marked by letters. These objects are marked in Figure 9.3 and are compared in Figure 9.7. The more blue objects R, S and T which are located in the halo of the galaxy are extended in the *HST*/*ACS* R band image. These objects are most likely globular clusters or background galaxies. Except for G, M and O all remaining objects are blended in the Ks image and can therefore be ignored in the colour magnitude diagram. The G, M and O objects could be foreground sources or variable stars, because 6 years past since the *HST*/*ACS* observation. Therefore further observations are required to determine their nature.

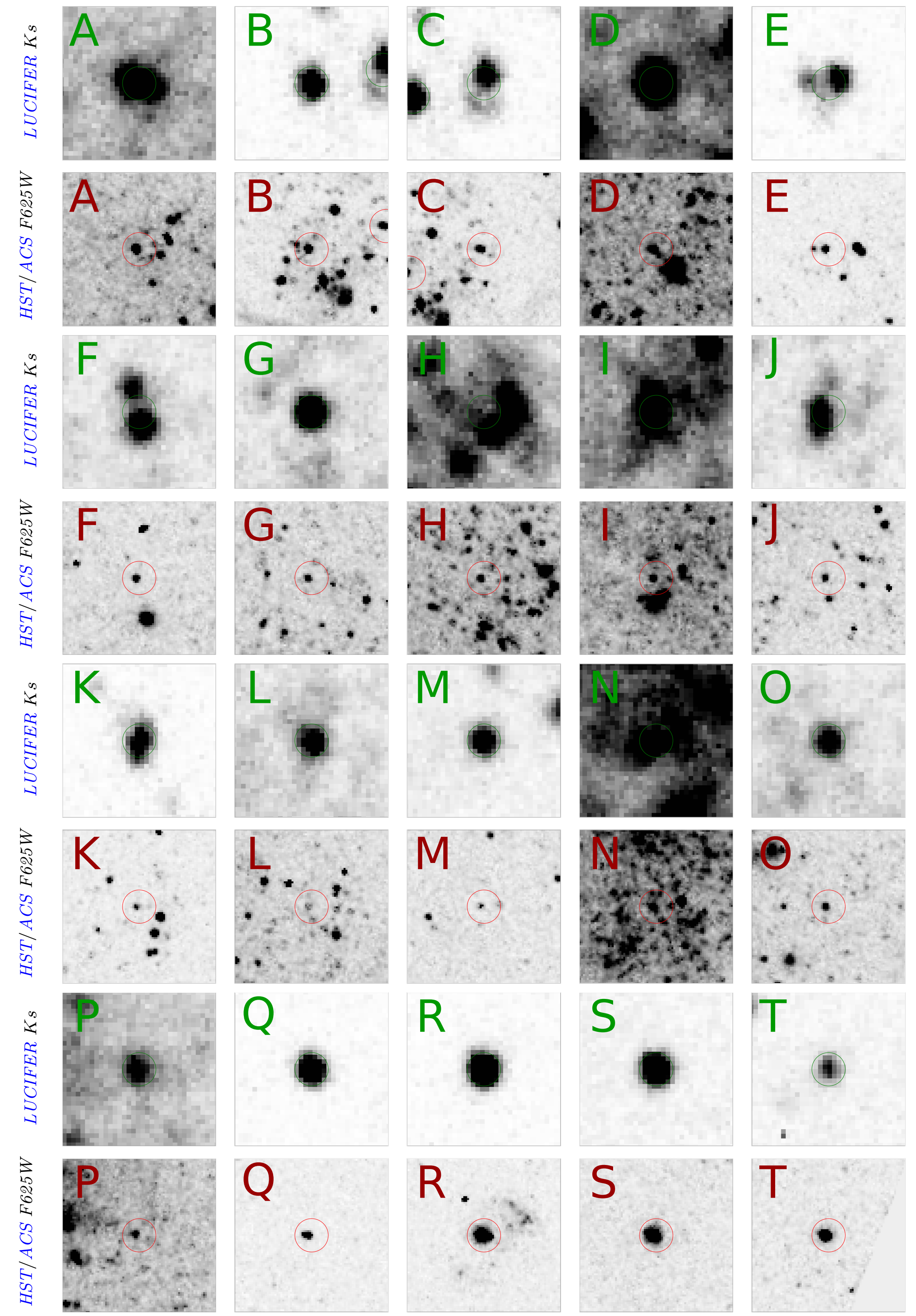


Figure 9.7: A detailed comparison of objects with conspicuous colours/magnitudes of Figure 9.6. The *LUCIFER* *Ks* images are compared with the *HST*/*ACS* *F625W* images.

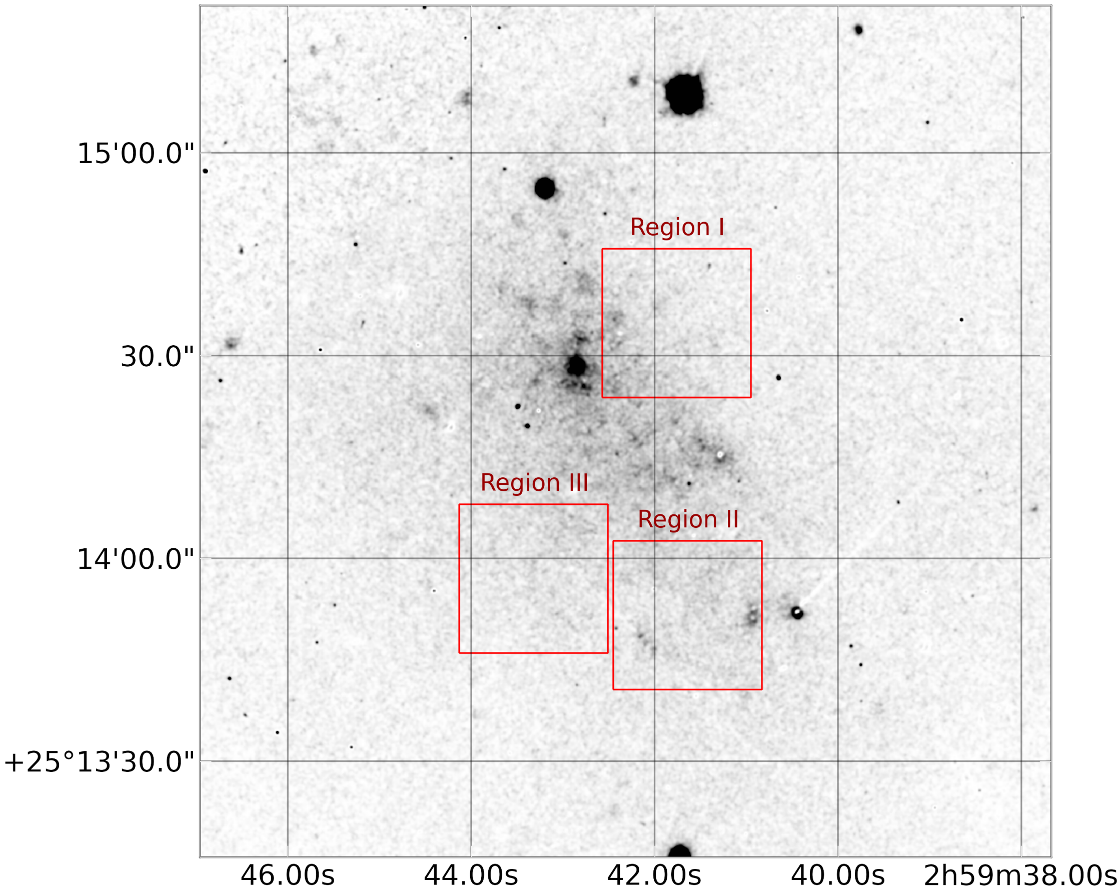


Figure 9.8: Continuum subtracted H_2 image of *NGC 1156* observed with *LUCIFER*. Additionally the regions which exhibit structured H_2 emission are marked and displayed with an increased contrast in Figure 9.10.

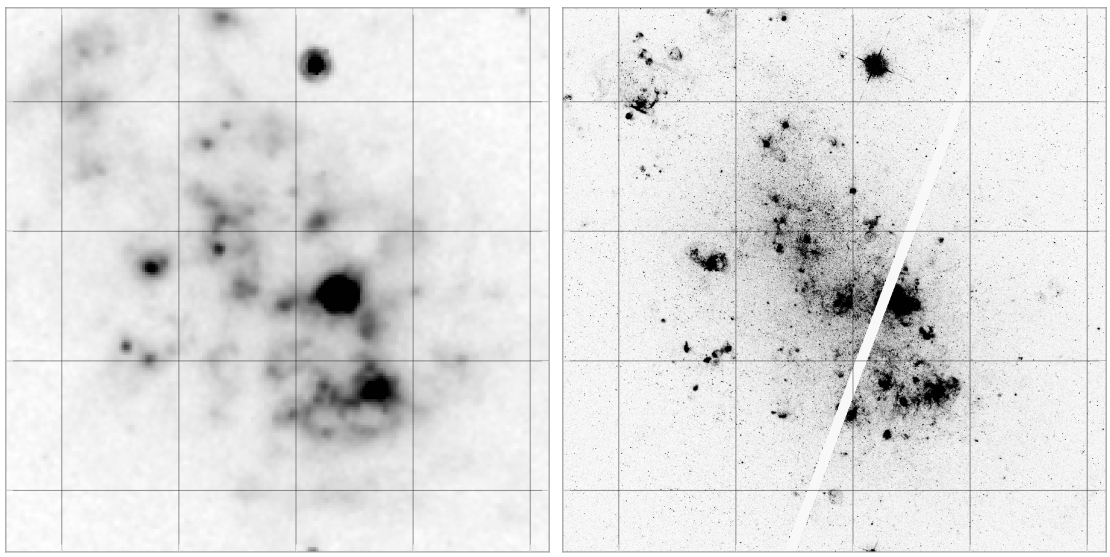


Figure 9.9: Comparison of *Spitzer* 8 μm and *HST*/*ACS* $H\alpha$ image. Both images have the same coordinate grid as in Figure 9.3.3. (**left**) *Spitzer* archive image of *NGC 1156* which was taken in channel 4 (8 μm) of the *IRAC* system. (**right**) *HST* archive image of *NGC 1156*. Two combined 600-second exposures taken by the *ACS* system and the *F658N* filter ($H\alpha$).

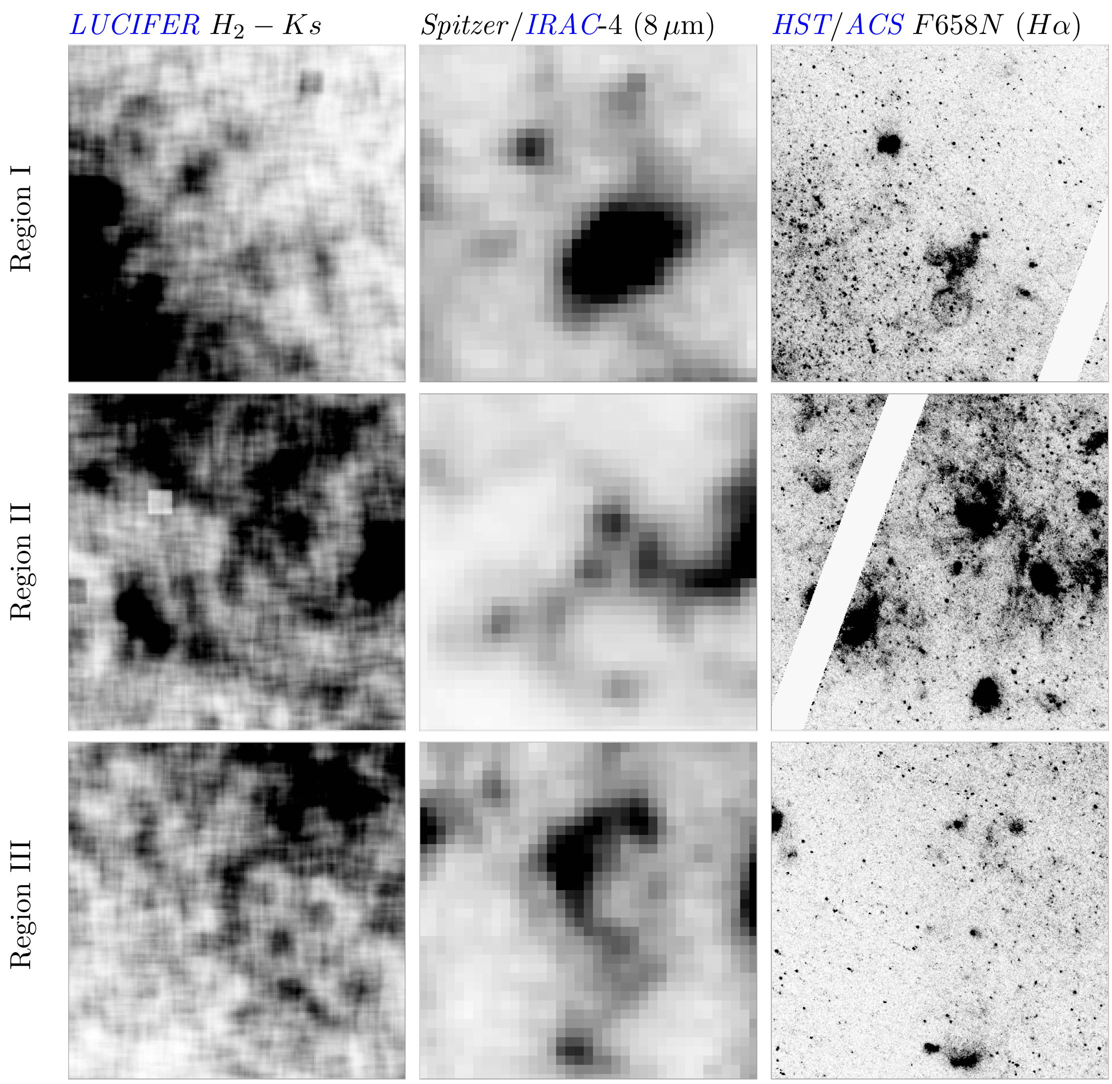


Figure 9.10: Detailed study of selected regions of *NGC 1156*. The continuum subtracted *LUCIFER* H_2 images have been enhanced in contrast and a boxcar smoothing filter was applied. For comparison the same regions are plotted for *Spitzer*/*IRAC* 8 μm and HST/ACS $H\alpha$ data.

Karachentsev et al. (1996) determined for *NGC 1156* a distance modulus of $m - M = 29.46 \pm 0.15$ mag (7.8 Mpc) by using the three brightest red and blue stars. Their result is comparable to the one presented here calculated with the aligned supergiant edge. Bottinelli et al. (1984) calculated a distance modulus of $m - M = 27.89$ mag (3.8 Mpc) with the B band *Tully-Fisher* relation. The difference of 1.1 mag fits well with the standard error of the *Tully-Fisher* relation of 1.2 mag for irregular galaxies (Karachentsev et al., 1996).

9.3.3 The Analysis of the H_2 Image and a Detection of Warm Molecular Hydrogen

In Veilleux et al. (2009) a detection of extraplanar warm molecular hydrogen in *M 82* is presented. *M 82* is the most prominent galaxy with a massive central starburst. They show that knots and filaments can be traced up to a distance of 3 kpc above the disc.

Based on their results *NGC 1156* was observed to test whether or not H_2 can be detected in a dwarf galaxy with a global starburst. The observed H_2 emission line is caused by a rovibrational mode of the molecular hydrogen which is excited by inelastic collisions introduced by shocks and heating by the *UV*/X-ray radiation from the starburst. A better knowledge of the distribution of H_2 would provide a broader understanding of the cooling processes in a starburst galaxy.

In Figure 9.8 the continuum subtracted H_2 image is presented. This observation was visually inspected with different intensity transfer functions to determine regions with structured emission of warm molecular hydrogen. The three marked regions in the outer part of the galaxy have been selected because they show interesting structures and filaments. These regions are presented in Figure 9.10 with an increased contrast. Additionally the data was smoothed with a boxcar filter ($1\farcs6$) to improve the signal-to-noise ratio. For each of the three regions *Spitzer*/*Infrared Array Camera (IRAC)* 8 μm archive images (Program ID: 69/P.I.: Fazio, see Figure 9.9, left) and *HST*/*ACS* $H\alpha$ archive images (Proposal: 9892/P.I.: Jansen, see Figure 9.9, right) are presented.

In Region I two linear structures proceed away from the galaxy towards NW similar to outflow structures described by Cecil et al. (2001) in *NGC 3079*. The lower structure matches with an elongated $H\alpha$ structure which is detected at 8 μm, too. The upper structure instead is probably a misinterpretation of several aligned detections as a connected filament. It is a row of knots of warm molecular hydrogen. Therefore this structure is probably not a galactic chimney. Region II shows several shell-like structures which have counterparts in the 8 μm image and contain regions of $H\alpha$ emission. Additionally a long elongated filament can be found in the lower part of the *LUCIFER* H_2 image. This structure shows counter parts in the 8 μm observation, too. It connects several star forming regions that can be identified in the $H\alpha$ image. In Region III an arc-like structure is detected in both the H_2 *LUCIFER* and the 8 μm *Spitzer* image. This structure has no counterpart in the $H\alpha$ image.

Following the discussion of Veilleux et al. (2009), H_2 observations are an important counterpart to *Polycyclic Aromatic Hydrocarbons (PAH)* observations. They allow to detect and analyse the dusty and the molecular components in the winds of galaxies. The detailed transportation process of the molecular hydrogen without destroying it is not well understood, yet (Veilleux et al., 2009). For some regions both the H_2 and *PAH* detections correlate well. In other regions the ratios between the detections differ significantly. The same phenomenon can be observed in our data. This indicates that other processes than shocks or *UV* radiation are involved and further investigation is needed (Veilleux et al., 2009).

The *LUCIFER* observations demonstrate that it is possible to detect H_2 with an 8-m class telescope in a dwarf galaxy. Further observations with longer integration time are required to provide data with better signal-to-noise ratios which would allow to analyse the structure of the filaments. A quantitative comparison of the detected *PAH* (*Spitzer* 8 μm) and the $H\alpha$ emission (*HST*/*ACS*) would be possible then.

Chapter 10

An Efficient Method for Photometric High Redshift QSO Candidate Selection

The costs of building and operating a 10-m class telescope limit the number of facilities and thus the available observation time. Therefore an efficient selection of scientific targets is mandatory. This chapter presents an approach to select QSO candidates with high redshift (z > 4.8) based on photometric catalogues. These candidates can be spectroscopically verified with the LUCIFER instrument. As part of the candidate selection approach a photometric redshift estimator is presented.

In the late 1950's observations in the radio regime discovered *quasi-stellar radio sources (Quasars)* with no optical counterpart. Early observations of *Quasars* like *3C273* used lunar occultations to determine their position and confine their size with high precision (Hazard, 1962; Hazard et al., 1963). This information on *3C273* allowed to assign an object in the visual wavelength range and spectroscopically determine a redshift of $z = 0.158$ (Greenstein and Schmidt, 1964). Later a *Quasar* was found to be a distant galaxy with an *Active Galactic Nucleus (AGN)*.

An *AGN* is a central super-massive black hole which accretes material and thereby creates radiation. This phenomenon is seen in different ways and therefore creates a large number of of observable *AGN* classes (see Carroll and Ostlie, 2007): (i) Spiral galaxies with a very bright nucleus are called *Seyfert Galaxies*. The objects of this class show weak radio emission. This class is divided into objects with broad as well as narrow emission lines (Type 1) and those showing only narrow lines in their spectra (Type 2). (ii) The class of *quasi-stellar objects (QSOs)* with the radio loud subclass of *Quasars*. The subclass of *Blazars* contains highly variable sources with strong radio emission. (iii) *Radio Galaxies* are elliptical galaxies which show strong radio emission. This type of *AGN* is divided in 2 subclasses by using the same criteria as for *Seyfert Galaxies* in the optical or employing the *Fanaroff-Riley* classification in the radio (Fanaroff and Riley, 1974). (iv) Another class that is associated with *AGNs* is the class of *ULIRGs*. The members of this class are possible dust covered *QSOs* or their radiation is driven by a heavy starburst event.

Antonucci (1993) uniformly describes this zoo of *AGNs*. The different types of *AGNs* that are observed can be explained with different orientations of the strongly non-spherical inner part. The class of *QSOs* is among the most luminous object types in the Universe (Page, 1964). Even though *QSOs* can be very distant their extreme luminosity allows us to observe them and thus to study processes in the early universe. Another benefit of their intrinsically high luminosity is the ability to find these sources even in surveys with low detection levels. A representative sample of high-z *QSOs* would help to understand the formation process of galaxies (White and Frenk, 1991) and the influence of super-massive black holes on galaxy evolution (Cattaneo et al., 2009). The formation of larger galaxies through hierarchical clustering has direct effects on the creation of *AGNs*. Carlberg (1990) found that the birth rate of *QSOs* is proportional to the rate of mergers of gas-rich galaxies. The presence of an *AGN* has direct consequences for the hosting galaxy. As soon as the *AGN* starts to accrete material and produces radiation, the gas content of the bulge is heated and in dependence on the strength of the radiation blown away. This directly leads to a stop of star formation in the bulge (see Cattaneo et al., 2009).

As there are only a few *QSOs* known with a redshift of z > 5, statistics on their number density is not reliable (Cristiani et al., 2004). For *QSOs* with z > 6 the *Lyman* α (*Ly*α) emission line is shifted to wavelengths above 0.85 μm, the lower limit of the wavelength range of *LUCIFER*.

10.1 Panoramic Catalogues as a Base of Target Selection

As larger optical telescopes tend to improve both sensitivity and spatial resolution, the field of view is reduced correspondingly. A reduced field of view requires an appropriate selection of targets based on previous observations. Today, large panoramic catalogues are available which can be used for the target selection process. To operate and guide the *Hubble Space Telescope (HST)* a first digital all-sky catalogue (*GSC*) was required. It had to be created on basis of scans of analogue photographic *Schmidt* plates ($\approx$ 1400 plates $\times$ 400 MB) (see Lasker et al., 1990). For the *NIR* wavelength range (J, H, Ks band) the *Two Micron All Sky Survey (2MASS)* can be used (see Skrutskie et al., 2006). It was created to have an all sky catalogue in the *NIR* wavelength range. In this catalogue the size of the processed data reached 2 TB.

The presented method of photometrically selecting high redshift *QSO* candidates is based on the *Sloan Digital Sky Survey (SDSS)* (see York et al., 2000). A dedicated 2.5-m telescope at the *Apache Point Observatory* in *New Mexico* was used to create an imaging and spectroscopic catalogue of the northern *Galactic Cap* ($9,583\,\mathrm{deg}^2$). Images have been taken in drift-scan mode using 5 broadband filter equipped rows of 6 $2048 \times 2048\,\mathrm{pixel}^2$ *CCDs* each. The resulting u, g, r, i, z band stripes have been processed to create an object catalogue which covers the 300 nm to 1,000 nm wavelength range. For a selected sub-sample a spectroscopic analysis has been done with a fibre-fed spectrograph. Each observation was executed in parallel for 640 objects. The used *Sixth Data Release (DR6)* contains 10 TB of calibrated images and spectra. 287 million objects are stored in the catalogue of which 1.27 million have spectra (see Adelman-McCarthy et al. (2008) for more details on *DR6*). The increasing amount of data which is available in catalogues can no longer be handled manually. It requires an automated processing and selection of scientifically important objects. The spectroscopically observed objects have been automatically classified by correlating them with 33 template spectra. In Schneider et al. (2010) a hand-vetted catalogue of 105,783 spectroscopically confirmed *QSOs* is presented. This catalogue contains 1,248 objects with z > 4 whereof 56 objects have a redshift of z > 5. It is used as a reference set for the photometric selection method presented hereafter.

10.2 A Photometric Redshift Estimator

An efficient creation of high-z *QSO* candidate catalogues requires a precise photometric estimation of the redshift. These estimates allow the rejection of *QSO* candidates with a redshift below a certain threshold. Photometric methods try to detect the position of

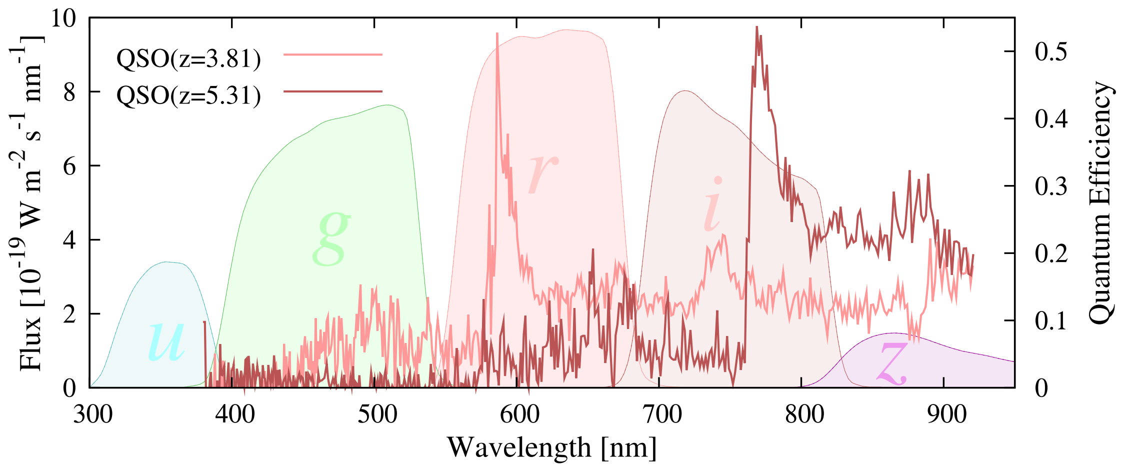


Figure 10.1: Spectra of the high-z *QSOs SDSS J082547.79+332836.9* (z = 3.81) and *SDSS J165902.12+270935.1* (z = 5.31). In the background the wavelength dependent quantum efficiency of the u, g, r, i, z bands of *SDSS* are plotted. The *Ly*α emission line of the *QSOs* is red-shifted to the r and i band, respectively.

strong emission and absorption features which are unresolved in the broadband filters. In Figure 10.1 the spectra of 2 high-z *QSOs* are presented together with the filter bands of *SDSS*. The *Ly*α emission lines of these *QSOs* dominate the flux in the r and i band, respectively.

10.2.1 Previous Photometric Redshift Estimation Methods

There are several methods available for a photometric redshift estimation. A classical way of photometric redshift estimation is the fitting of *Spectral Energy Distributions (SEDs)*. This method is typically limited to a small set of representative model spectra. These spectra can be created based on empirical or simulated *SEDs*. In Bolzonella et al. (2000) an estimation method that uses a χ^2 minimisation is presented. The quality of the fit is highly dependent on the applied template spectra. When the used photometric features do not allow to distinguish between the different *SEDs* the fit can create catastrophic outliers. The big advantage of the *SED* fitting approach is the ability to predict good values even for objects with spectroscopically yet unobserved redshifts.

Another way of redshift estimation is based on empirical reference data. In O'Mill et al. (2011) such an approach is presented for the *SDSS* data. 80% of the *Main Galaxy Sample (MGS)* (see Strauss et al., 2002), 10% of the *Luminous Red Galaxy Sample (LRGS)* (see Eisenstein et al., 2001) and 10% of the *Active Galactic Nucleus Sample (AGNS)* (see Kauffmann et al., 2003) have been combined to a set of 550,000 objects with spectroscopic redshifts. Half of this reference set was used for training a 9:14:14:14:1 *Artificial Neural Network (ANN)* and half for testing. The 9 input nodes represent the *SDSS* magnitudes, the concentration index and the *Petrosian* radii in the g, r band. 3 hidden layers of 14 neurons each have been used to calculate the redshift output. They limited their tests to a redshift range of z < 0.4. In this range their estimates deviate with an rms ≈ 0.03 from the spectroscopically determined redshifts.

Wu and Jia (2010) present a photometric redshift estimator that combines *SDSS* and *UKIRT Infrared Deep Sky Survey (UKIDSS)* data. A reference sample of 7,400 *QSOs* with 0.5 < z < 5.2 was divided into 91 redshift bins. Each bin was analysed and a median colour was calculated for the 8 directly neighbouring bands. In their approach to find the most probable redshift they applied a χ^2 minimisation of colours that take the photometric

SDSS PhotoObjID	z	Reference	*SDSS* PhotoObjID	z	Reference
588023045868553340	5.79	Fan et al. (2006)	587741421098303812	6.00	Fan et al. (2006)
587740525079167786	5.80	Fan et al. (2004)	587738615416554219	6.01	Fan et al. (2006)
587727942951109703	5.82	Fan et al. (2001)	587729751132603659	6.05	Fan et al. (2003)
587733411521299389	5.83	Fan et al. (2006)	587735666926158831	6.07	Fan et al. (2004)
587731186204541926	5.85	Fan et al. (2004)	587739608093491422	6.13	Fan et al. (2006)
587737808499572747	5.85	Fan et al. (2006)	587736783608677304	6.22	Fan et al. (2004)
587736914601902923	5.93	Fan et al. (2004)	587732482206139341	6.23	Fan et al. (2003)
587738951494075293	5.93	Fan et al. (2006)	587728881415553909	6.28	Fan et al. (2001)
587729157893456734	5.99	Fan et al. (2001)	588013383815791587	6.43	Fan et al. (2003)

Table 10.1: High-z *QSOs* which have been found with the i dropout method. These *QSOs* have been used to extend the reference sample.

errors into account. As a result of their tests of the redshift estimator 71.8% of their reference sample have a redshift estimation error of $|\Delta z| < 0.1$.

The presented redshift estimation methods either rely on physical assumptions or the quality of the used reference set. When the size of the reference set is increased to improve the estimation quality the processing speed drops. As part of the *QSO* selection process an estimator was developed which provides both, the ability to process large reference sets without requiring physical assumptions or models.

10.2.2 A kNN Regression Model for Redshift Estimation

A new redshift estimator was developed which is based on empirical data to support the *QSO* selection process (see Section 10.3. This estimator realises the important step of rejecting candidates with low redshifts. It uses a *k-Nearest Neighbours (kNN)* regression model to predict redshifts (see Hastie et al., 2009). This is similar to the approach presented in Csabai et al. (2003) to estimate redshifts of galaxies with $z \lesssim 0.5$. The predicted redshifts $\hat{Y}$ are calculated from the redshift values y_i of the k *Euclidean* closest objects. Thereby the neighbourhood $N(\overrightarrow{x})$ is determined on basis of the object representation $\overrightarrow{x}_i$ in the feature space.

$$\hat{Y}(\overrightarrow{x}) = \frac{1}{k} \sum_{\overrightarrow{x}_i \epsilon N_k(\overrightarrow{x})} y_i$$

To retrieve the k neighbours N of $\overrightarrow{x}$ efficiently, a *k-Dimensional Tree (kd-Tree)* is used (Bentley, 1975). This data structure uses a binary search tree to allow a spacial look-up in $\mathcal{O}(log_2 n)$ instead of $\mathcal{O}(n)$ where n is the number of stored multidimensional values. Besides the redshift value the standard deviation of the k nearest y_i is calculated as a quality measure. High standard deviations indicate a bad coverage of the target space. To analyse the distribution of reference objects in the feature space the length of the average distance vector to the k neighbours can be calculated. Large values indicate that the requested object lies out of the reference sample and therefore might have a very high/low redshift. The disadvantage of the used *k-Nearest Neighbours (kNN)* regression model is its limitation to predict only values that are covered by the reference sample.

A reference sample was created to support the detection process of high redshift *QSOs*. This sample is required to populate the feature space and was created on basis of the *SDSS Quasar* catalogue (see Schneider et al., 2010). To increase the processing speed of estimating redshifts, the size of the reference sample was reduced. For this reason the

Reference	Redshift	Test Set Size	$\langle\frac{\Delta z}{1+z}\rangle$	$\sigma_{\frac{\Delta z}{1+z}}$	$\langle\Delta z\rangle$	$\sigma_{\Delta z}$
complete	$1.0 \leq z < 6.5$	77,096 *QSOs*	0.003	0.033	-0.007	0.091
complete	$4.0 \leq z < 6.5$	1,258 *QSOs*	0.008	0.024	-0.065	0.126
complete	$4.5 \leq z < 6.5$	406 *QSOs*	0.002	0.021	-0.011	0.117
reduced	$1.0 \leq z < 6.5$	77,096 *QSOs*	-0.023	0.095	-0.024	0.119
reduced	$4.0 \leq z < 6.5$	1,258 *QSOs*	0.010	0.025	0.036	0.133
reduced	$4.5 \leq z < 6.5$	406 *QSOs*	0.002	0.016	0.001	0.087

Table 10.2: Results of fitting *Gaussian* distributions to the redshift estimation errors. The values are given for both $\Delta z/(1+z)$ and Δz. Additionally the errors have been determined on both the reduced and the complete reference sample.

input catalogue was split into 3 subsets: (i) The low redshift ($z < 2$) set with 81,238 objects, (ii) the medium redshift ($2 \leq z < 4$) set with 22,696 objects and (iii) the high redshift ($z \geq 4$) set with 1,258 objects. As the high redshift set contains only a few *QSOs* with $z \geq 5$ and is limited to $z_{max} = 5.5$, 18 additional objects with *SDSS* features have been added (see Table 10.1). To create a homogeneously distributed sample, all quasars have been assigned to 120 bins for redshift between $z = 1$ and $z = 7$. For those bins below $z = 4.8$ the size was limited to 10 reference objects and supernumerous objects have been randomly extracted. Above a redshift value of $z = 4.8$ all *QSOs* have been included. Due to missing high-z references the high-z bins are not filled equally. The resulting reference sample contains 1,106 *QSOs* with spectroscopically determined redshifts and *SDSS* magnitudes.

During tests with the *kNN* regression model it turned out that the best results are achieved when using colours instead of magnitudes. This may be caused by distribution effects of the reference objects in the *Euclidean* feature space which are induced by intrinsic object characteristics like their luminosity. These effects are minimised by a kind of normalisation which is obtained by the dimension reduction from filter band to colour space. The k value was set to 8, based on tests of the redshift estimation performance. With smaller k values the standard deviation of the estimation errors increased. Slightly larger values (up to 20) did not significantly improve the results.

10.2.3 The Evaluation of the Redshift Estimation

The quality of the *kNN* regression redshift estimation approach was tested on the *QSOs* with spectroscopic redshifts taken from Schneider et al. (2010). As the reference sample is limited to redshifts above $z = 1$, *QSOs* with lower redshifts have been excluded from the tests. There are only a few high red-shifted *QSOs* known. Therefore all objects are required as reference as well as for testing. To prevent any bias from objects that are part of both the test and reference sample, objects are not considered as reference when their value is estimated. For each object of the test sample a deviation $\Delta z = z_{spectroscopy} - z_{estimation}$ and a redshift independent deviation $\Delta z/(1+z_{spectroscopy})$ was determined. In Figure 10.2 the results of the estimation approach are presented. The area below $z = 4.8$ is marked grey as those *QSOs* candidates lie out of the target redshift range. The comparison of the results of the reduced reference sample (Figure 10.2, top left) with the results of the complete reference sample (Figure 10.2, bottom left) demonstrate that both sets perform equally for $z > 4.8$. With the reduced reference sample lower red-shifted *QSOs* can still be processed with an appropriate estimation quality, even though the size of the reference set was dramatically reduced from 77,096 to 1,106 objects. As the reduced reference sample is homogeneously distributed in redshift, no catastrophic outliers are produced by an over-representation of low redshifts. In Figure 10.2, top and bottom left, a redshift dependent

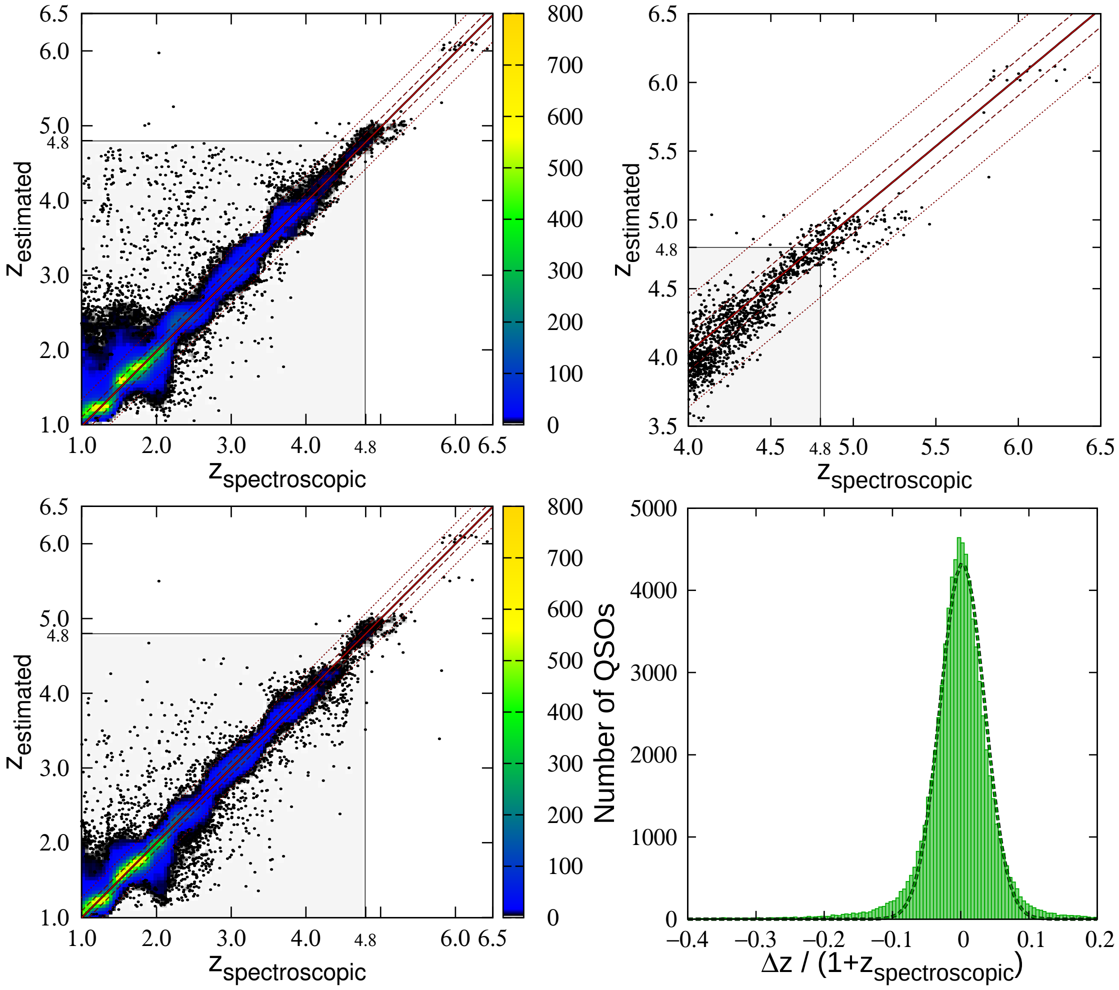


Figure 10.2: Comparison of estimated and spectroscopic redshifts. The results which are based on the reduced reference set (**top left**) are shown with a magnified $z > 4$ sector (**top right**). Additionally the estimation results that are created with the complete reference sample (**bottom left**) and the corresponding error distribution (**bottom right**) are presented. The mean value, the 1σ and the 3σ values of the fitted *Gaussian* distribution are plotted as a solid, a dashed and a dotted line, respectively.

fluctuation between estimated and spectroscopic values can be observed. This is directly connected to the passage of the *Lyα* features though the broadband filters.

The deviation Δz as well as the redshift independent deviation $\Delta z/(1 + z)$ between the estimated and spectroscopic redshifts are fitted with *Gaussian* distributions. This enables one to quantitatively distinguish between the reduced and the complete reference sample. Additionally the estimation quality is determined for different redshift ranges. The results of these fits are presented in Table 10.2. The distribution of the estimation errors for the full redshift range $1 < z < 6.5$ is presented in Figure 10.3, left. In comparison to the tests on the full redshift range, better results are achieved for the higher red-shifted *QSOs* (see Figure 10.3, right). As it was intended, the reduced reference sample performs better on *QSOs* in the targeted selection range with $z > 4.8$. This is caused by the better representation per redshift bin of $z > 4.8$ *QSOs* in the feature space.

The performance of the redshift estimation is comparable to the results of Wu and Jia (2010), but: (i) Their results are based on a test sample of only 8,498 *QSOs* which was

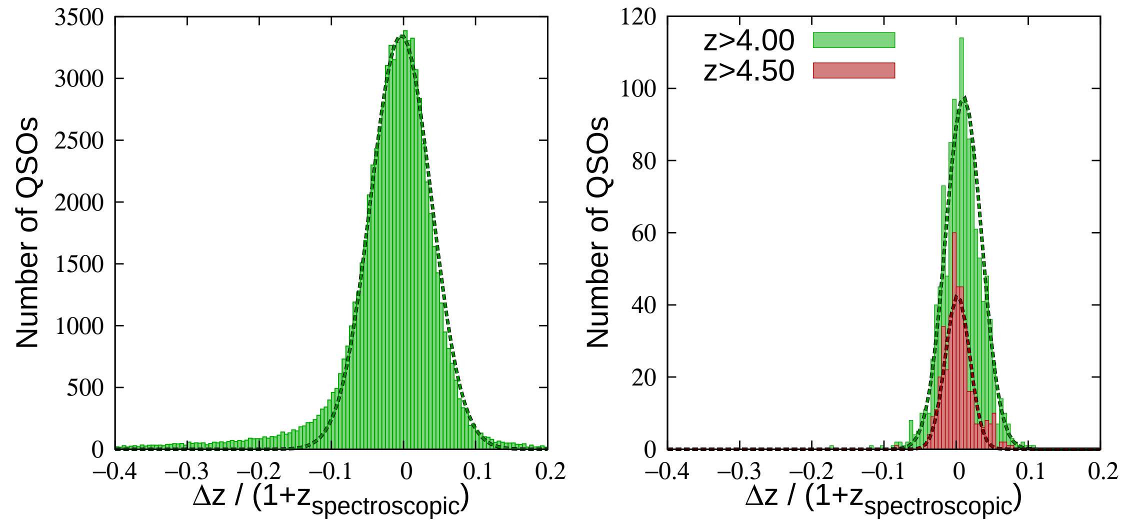


Figure 10.3: Histogram of the redshift estimation errors calculated with all $z > 1$ (**left**) and the high red-shifted $z > 4$ (**right**) *QSOs* which have spectroscopically determined redshifts. The dashed lines are fits to the *Gaussian* distributions.

partially (87%) used to create the median colours. This creates a bias on the test results. (ii) *SDSS* and *UKIDSS* data was used. (iii) The presented results are dependent on the redshift range and therefore can not be compared directly.

Mortlock et al. (2011) present a *Bayesian* redshift estimator which is used to assign observation priorities to their high redshift *QSO* candidates. Based on photometric data of *SDSS* and *UKIDSS* an accuracy of $\Delta z \simeq 0.1$ is presented for a redshift range of $5.8 < z < 7.2$. This deviation is comparable to the results achieved with the presented *kNN* regression model. As there are no *QSOs* known with $z > 6.5$ in both *SDSS* and *UKIDSS*, the results have been computed with simulated *SEDs* and are based on a modelled high-z *QSO* population.

In Cardamone et al. (2010) a 32-band data set is used to calculate photometric redshifts. They present a 1σ scatter in $\Delta z/(1 + z)$ of 0.008 for $0.1 < z < 1.2$, 0.027 for $1.2 < z < 3.7$ and 0.016 for $z > 3.7$, respectively. For the high redshift range these results are as good as the results of the *kNN* regression model presented here.

The developed redshift estimator performs equal to the other presented approaches. Its main benefit is to be able to predict photometric redshifts with comparable quality even with less photometric bands. Additionally the processing speed on large reference samples is increased by using special data structures.

10.3 A Photometric QSO Selection Approach

The described estimation of redshifts is dependent on a reliable pre-selection of *QSOs*. The main problem in selecting these candidates is that broad band observations of the high red-shifted *SEDs* of *QSOs* become similar to the *SEDs* of cool stars. Before describing the *kNN*-based *QSO* selection approach and the creation of the required reference samples, an overview of currently available methods is given.

10.3.1 Previous Selection Approaches

In the *SDSS*, the selection of *QSO* candidates for spectroscopic observations is based on photometric data (see Richards et al., 2002). The magnitudes which are extracted on basis of fits to a *Point Spread Function (PSF)* are inspected for unresolved objects in distinct 3D colour spaces to separate *QSOs* from stars. Several decision trees have been created that reflect relations in band flux and thereby define regions in the feature space.

The z > 5.8 *QSOs* that are presented in Fan et al. (2001, 2003, 2004, 2006) have been detected by using an i band dropout technique in combination with *2MASS* magnitudes. This technique assumes no detection in the u, g, r band, a weak detection in the i band and a detection in the z band. The principle behind this is that the strong $Ly\alpha$ forest absorption enters the i band at z > 5.5. The resulting constraints are: $mag_z < 20.2$ with $\Delta mag_z < 0.1$, $mag_i - mag_z > 2.2$ and $mag_z - mag_J < 1.5$. This method turned out to have a high false positive rate. Only ≈ 3% of the candidates that had follow-up observations were verified as *QSOs*.

Wu and Jia (2010) present a *QSO* selection approach that uses colour-colour relations in *SDSS* and *UKIDSS* bands. The best solution to separate *QSOs* from stars was empirically found in the $Y - K$, $g - z$ colour space with the linear relation $Y - K > 0.46 \times (g - z) + 0.53$. This relation correctly separates 97.7% of both the 8,498 *QSOs* and 8,996 stars which have been used as reference. Unfortunately this simple linear separation fails for *QSOs* with z > 4. For this reason the relation $J - K > 0.45 \times (i - Y) + 0.64$ has been created to the find 99% of the 101 reference *QSOs* with z > 4. The downside of this solution is that the contamination with stars is more than doubled. Another problem of optimising the separating relation is that the equal cardinalities of the reference samples do not necessarily reflect the real distribution.

Mortlock et al. (2011) presented a probabilistic candidate selection approach which uses *SDSS* and *UKIDSS* to find the most probable high-z *QSOs*. Their approach is comparable to their redshift estimator. It uses a *Bayesian* model to separate *QSOs* from stars. Their targeted redshift range is z > 5.8. With their approach a reduction of the primary data set by a factor of $\approx 2.5 \times 10^4$ to 893 candidates was realised. Thereby the probabilistic evaluation of each object took 0.1 s to 0.01 s. Only 88 of these 893 candidates turned out to be real detections of astronomical sources with 3 previously known high-z *QSOs*. Follow-up observations left 7 photometric candidates of which 4 have an estimated redshift $z \simeq 6$. The results of these candidates are not published yet.

10.3.2 A kNN Classifier for QSO Selection

To avoid the problems of the methods mentioned in the previous section a new classifier was created. Its main purpose is to create *QSO* candidates with a high probability for follow-up observations. For this reason the number of recovered *QSOs* is set to a lower percentage than in the other presented approaches. Similar to the presented redshift estimation approach, the classification is done with the *kNN* algorithm (see Hastie et al., 2009). For the classification the types t_i of the k nearest objects N_k in the feature space are evaluated for each $\overrightarrow{x}$. The corresponding ratios $\hat{R}$ reflect the number of the $6k$ neighbouring reference objects that are of a certain type t_n. These ratios are calculated for each type in T.

$$\forall t_n \epsilon T, \hat{R}_{t(\overrightarrow{x})=t_n} = \frac{1}{k} \sum_{\overrightarrow{x}_i \epsilon N_k(\overrightarrow{x})} \begin{cases} 1, & t_i = t_n \\ 0, & \text{otherwise} \end{cases}$$

The *QSO* candidate selection is realised by combining the redshift estimator and 3 classifiers: (i) The first classifier should realise a coarse *QSO* pre-selection. This is done by using a more general reference set with several object types. An *Euclidean* distance in the *PSF* magnitude feature space is used to identify the *kNNs*. (ii) In the next step the redshift is estimated and low red-shifted objects are rejected (see Section 10.2). (iii) A classifier which rejects cool stars is used to decrease the contamination by stellar objects. This is realised by a comprehensive reference sample which contains cool stars and *QSOs* only. The same feature space as for the coarse pre-selection is used. (iv) An alternative distance measure d is used to run a classification with respect to the photometric errors. The first part of this function reflects the similarity of two feature vectors $\overrightarrow{u}, \overrightarrow{v}$ in feature space with respect to the measurement errors $\overrightarrow{\Delta u}, \overrightarrow{\Delta v}$. The second part ensures that objects with similar errors become closer. When two objects with severely deviating measurement errors are compared the first distance component decreases due to the dominant error term. This is compensated by the second component.

$$d(\overrightarrow{u}, \overrightarrow{\Delta u}, \overrightarrow{v}, \overrightarrow{\Delta v}) = \sum_{i=1}^{N} \frac{(u_i - v_i)^2}{\Delta u_i^2 + \Delta v_i^2} + (|\Delta u_i| - |\Delta v_i|)^2$$

In each step of the selection process objects that do not match the ratio criteria are rejected. The ratios have been created on basis of the *SDSS* objects with spectroscopic classifications. They have been optimised to find as many high-z *QSOs* as possible while simultaneously minimising the contamination by other objects. For the coarse pre-selection step a ratio of 10 high-z *QSOs* of the 12 nearest neighbours was determined. The redshift estimator is used to exclude objects with z ≤ 4.8. The standard deviation that is calculated on the *kNNs* is used to allow an undershooting of this z value by 3σ. For the cool stars rejecting classifier a ratio of 8 *QSOs* out of 17 nearest neighbours was empirically found to produce the best results. This means that 10 or more cool stars are required to reject a candidate. In the last selection step the 19 nearest neighbours are inspected. 12 of these neighbours must be high-z *QSOs*. The objects that pass all 4 selection steps are written to a result file. Instead of simple ratios it contains the types of the 20 nearest neighbours for each step. This allows a post-processing to reduce the candidate list for follow-up observations. The likelihood of a candidate being a high-z *QSO* can be estimated by calculating different ratios afterwards. Thereby constraints can be combined like e.g., 10 of the 12 closest references are *QSOs* + only 1 cool star is allowed for the 8 nearest neighbours + a maximum of 2 cool stars for all 20 neighbours. These constraints can be specified for each of the classifiers separately.

As a pre-filtering of all 287 million *SDSS* objects a limiting i band magnitude of 16.5 is used. Brighter objects or objects with an i band error > 0.2 mag are ignored. Additionally only point-like or slightly extended objects are selected. To separate point sources from extended ones *SDSS* uses the *PSF* and model magnitudes: When an object complies with $\mathrm{PSF_{mag}} - \mathrm{model_{mag}} > 0.145$ it is labelled as galaxy in the *SDSS* catalogue. Here, a value of 0.3 is used instead to include slightly extended sources of e.g., possible lensed *QSOs*. The corresponding database query created a sample of 122 million objects with u, g, r, i, z band *PSF* magnitudes and errors.

The classifiers require reference samples similar to the presented redshift estimator sample. The first sample was created to detect *QSOs* and represents 5 different types: (i) All 1,258 high red-shifted *QSOs* that have been composed for the redshift estimation sample. (ii) A random selection of 1,000 medium red-shifted *QSOs*. A sample with randomly selected (iii) 1,000 galaxies, (iv) 1,000 stars and (v) 1,500 cool stars with spectroscopic classifications. The sample to reject cool stars was created from all 1,258 high red-shifted *QSOs* and all spectroscopically determined cool stars that match the criteria described above. The resulting reference sample contains 10,928 objects. By using spectroscopically classified objects and objects detected by the i band dropout method as reference, the applied sample selection criteria (see Richards et al., 2002; Fan et al., 2001, 2003, 2004, 2006) are reproduced by the presented reference samples. Furthermore the objects selected as *QSO* candidates by Richards et al. (2002) that turned out to be cool stars improve the separation capabilities of the samples.

10.3.3 The Evaluation of the Selection Approach

To evaluate the performance of the presented selection approach all 1.2 million *SDSS* objects with spectra have been photometrically analysed. When objects are evaluated that are part of one of the reference sets, they are excluded from the neighbourhood query to prevent any bias. A list of 242 objects was obtained which contains the resulting candidates. The spectra of all of these objects have been visually inspected and their types have been checked. 147 of these candidates are *QSOs* of the high-z reference sample with $z > 4.0$. 75 of the known 147 *QSOs* with $z > 4.8$ are recovered by the photometric selection. The 18 *QSOs* of the high-z extension sample have been tested separately as they are not part of the *SDSS* spectroscopic catalogue. 17 of these *QSOs* can be detected with the photometric selection approach. For these objects that have been found with an i band dropout method the coarse classifier calculates ratios of ≥ 19 out of 20. Of all 32,210 known cool stars only 34 have been falsely classified as *QSOs*. The spectra of the remaining 61 objects are of other types or are not assignable to a type.

As the execution of the individual classification steps is not order-dependent they have been arranged by their processing speed. This ensures that computing intensive steps are only executed when previous classifications have been successful. The developed software scans lists of photometric features. Therefore it can easily be parallelised to increase the speed of processing lists. A single instance is able to process 1,000 objects in $4 - 8\,\mathrm{s}$ on a standard *PC*. For the creation of the final candidate list the software was running on 8 cores and processed the 122 million photometric data sets in a day.

The resulting list contains 121,715 objects that match the specified ratio criteria of the selection steps. This sample contains redshift estimations for 82,258 candidates with $z < 5$, 34,264 with $5 \leq z < 6$ and 5,193 with $z \geq 6$. The detection performance was calculated on the spectroscopic sample to be $\approx 50\%$. Under the assumption that the classifier performs comparably on the photometric sample, $\approx 60,000$ high-z *QSOs* are part of the candidate list.

Cristiani et al. (2004) presented a space density for *QSOs* with $4 < z < 5.2$. A separate classification run for this redshift range created a list of 102,825 candidates. In Figure 10.4 these candidates are plotted in comparison with the results of Cristiani et al. (2004). The red line is a cumulative plot of all candidates while the blue line assumes a *QSO* detection performance of 50%. When the ratio of the first selection step is set to highest possible ratio (i.e. 12 *QSOs* under the 12 nearest neighbours) only 12,586 candidates remain. These candidates with the highest probability of being a *QSO* are plotted as a green line.

The presented results are consistent with the results of Cristiani et al. (2004) and the number of found *QSO* candidates fits well with the presented models. A model which connects *QSOs* and dark matter halos with a *minimal set of assumptions (MIN)* is presented in (Haiman and Hui, 2001). This model assumes: (i) an un-evolved halo, (ii) a constant black hole to dark matter halo mass ratio (iii) and a maximum accretion at the *Eddington* limit (see Cristiani et al., 2004). As this model overpredicts high-z *QSOs* (Haiman et al., 1999) it defines an upper limit for the presented candidate selection approach. Monaco et al. (2000) present a model with a *delayed QSO shining (DEL)* in which the *AGN* activity starts after the formation of the dark matter halo. In their model

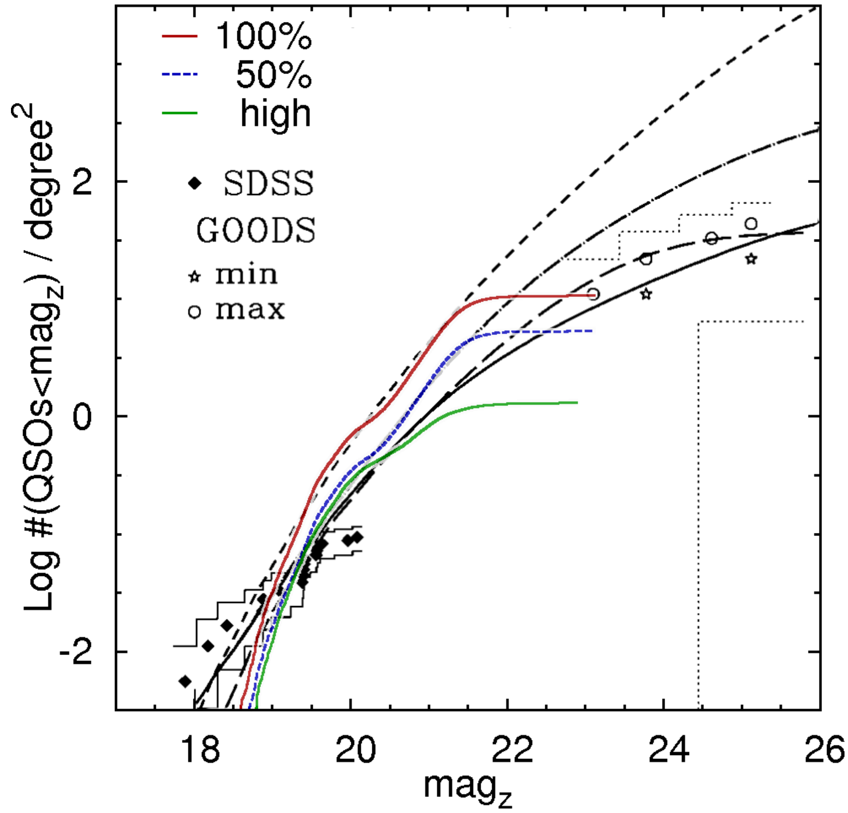


Figure 10.4: Comparison of the space density plot of *QSOs* ($4 < z < 5.2$) from Cristiani et al. (2004) and the parameters derived from the created candidate sample. Circles and stars show the *GOODS* based estimates of Cristiani et al. (2004) while diamonds are used for *SDSS* results. The tiny dashed lines represents the upper and lower 1σ confidence level of the *GOODS* data. The models that have been used by Cristiani et al. (2004) are: *PLE* (dot-dashed), *PDE* (continuous), *MIN* (short-dashed) and *DEL* (long-dashed). The 3 coloured lines represent the entire candidate sample (100%), an assumed detection performance (50%) and the candidates with the highest ratios $\hat{R}$.

AGNs which are hosted in smaller halos are longer delayed than those *AGNs* in larger halos. This allows brighter *QSOs* to appear before the fainter ones. The *Pure Luminosity Evolution (PLE)* model (brighter objects in the past) and the *Pure Density Evolution (PDE)* model (higher object density in the past) are used in Cristiani et al. (2004) to extrapolate the results of (Boyle et al., 2000). Both, the high ratio results as well as the results of the predicted detection performance fit well with these models. In *DR6* of *SDSS* the 95% detection repeatability for point sources in the z band is 20.5 mag. For this reason the results with the highest probability deviate from the model fits for higher z band magnitudes. The results of the other candidate lists fit well for z band magnitudes below 21.5 mag. In comparison to the *QSOs* detected in *SDSS*, an appropriate amount of candidates can be found even for z band magnitudes fainter than 19.5 mag (compare Figure 10.4).

The presented *QSO* candidate selection approach is highly dependent on the coverage of the feature space by the reference samples. Efficient data structures are mandatory to provide good scanning performance when using large reference samples. Except of the last step the *kNN* retrieval of all other classifiers is accelerated by *kd-Trees* (see Bentley, 1975). With the availability of larger spectroscopic surveys and larger sets of known high-z *QSOs* as reference, better photometric selection results will be achievable.

Even though not all of the known *QSOs* are recovered by the presented approach, the resulting candidates have higher probabilities to be a *QSO* than those found with other approaches. Follow-up observations with instruments like *LUCIFER* will help to determine the real detection performance of the candidate selection. The objects that are found not to be a *QSO* will directly improve the reference sets and thereby the detection performance.

Conclusions And Outlook

The first *LUCIFER* instrument is now scientifically used for more than 18 months. The *LUCIFER Control Software Package* is as important as the instrument hardware in performing observations. All software that was developed as part of this thesis never caused an interruption of an observation run. The decision to be the first large astronomical instrument which uses *JAVA* as programming language with its build-in capabilities to determine exceptions and handle them is one reason for this excellent statistic. Another reason is the choice to use a multi-tier service architecture which allows to independently control the subsystems without interfering with the complete system. The services and frameworks of the *System Tier* are the core of the control software. They provide all functionalities to run a distributed system including the automatic restart of unintentionally ended or failed services. As the services of the *Control Tier* realise the complex communication between the software the electronics the complexity is split in individual services and therefore can be handled independently. The *MOS Unit* is the most complex cryogenic mechanism ever used in an astronomical instrument (Richard Green, *LBTO* director). The operation of this instrument unit is realised by the corresponding service and sequencing framework of the *Instrument Tier*. The use of finite state transition networks to define the motion sequences of the individual sub-units of the *MOS Unit* prevented several severe damages to the instrument. Every malfunction of the hardware was detected successfully and the software stopped all motions to prevent physical damage to the instrument. These hardware failure could be manually recovered by an engineer using the provided engineering access. A simulator was created to test the motion sequences prior to running them on the real instrument. This virtual instrument was used to implement a new and full-automatic cryogenic cabinet exchange procedure which can be controlled with a two button *GUI*. Without any tests at the real hardware the cabinet exchange procedure could be performed without any errors. Additionally the virtual instrument was several times used to train staff in using the *LUCIFER* instrument.

With the operational *LUCIFER* instrument, *NIR* observations in Ks and H_2 of the dwarf galaxy *NGC 1156* have been carried out. The raw data of these observations was reduced and analysed. Due to errors during script execution the observation required a special treatment of the raw data. To determine the distance of this galaxy a new distance measurement method was developed which aligns the upper edge of the red supergiants in a colour magnitude diagram. With this method a distance of 6.3 ± 0.4 Mpc was determined for this galaxy. The method does not require observations with integration times as long as required by the traditional methods. The narrowband H_2 image was analysed and structured emission of warm molecular hydrogen was detected in the outer parts of the galaxy. This is the first direct detection of warm H_2 in the outer parts of a dwarf galaxy.

To provide targets for further observations with *LUCIFER* an approach to select candidates of high-z *QSOs* ($z > 4.8$) was developed. This approach uses a regression technique to estimate redshifts on basis of photometric data sets as well as data mining algorithms to discriminate between *QSOs* and cool stars, which have a similar *SEDs*. The most important part to create candidates with a high probability is the composition of the reference samples. The applied data mining techniques have been optimised to provide a high processing performance in order to handle large catalogues like the *SDSS* in a reasonable amount of time. Tests on basis of a spectroscopically classified data set of 1.2 million objects show a true positive rate of $\approx 50\%$ with a contamination of false positive candidates $\approx 50\%$. This is a significant progress in comparison to the the common techniques.

With the second *LUCIFER* instrument being installed at the *LBT* at the beginning of 2012 a completely new *Operation Tier* will be required that is able to handle the parallel operation of two instruments. Additionally new functionalities to plan and automatically execute a binocular observation are required. This includes the creation of new *GUIs* which improve the usability of the instrument.

To improve the reliability of the newly developed distance measurement method, observations of other objects are required. Additionally the error of the method could be improved by increasing the precision of the extraction of the photometric values. This method allows to determine precise distance measurements even for fainter objects. These results can be used to analyse the spatial distribution of galaxies in the local universe. With deeper observations in the H_2 quantitative analysis of the detected structures will be possible. Therefore new observations of *NGC 1156* are already planned.

The candidates in the high-z *QSO* sample need follow-up observations to determine their true nature and the performance of the selection process. The required observation time has already been granted. By extending the wavelength range, which is covered by the reference samples, to the *NIR* *QSOs* with even higher redshifts can be found. This will be done in the near future.

Acknowledgements

Parts of the virtual instrument which is presented in Chapter 8 have been published in POLSTERER ET AL. (2006).

Preliminary results of the quasar selection approach of Chapter 10 has been presented as a poster at AstroInformatics 2010 in Pasadena (Caltech), USA. Non of the presented results have been published yet.

This research made use of Montage, funded by the National Aeronautic and Space Administration's Earth Science Technology Office, Computation Technologies Project, under Cooperative Agreement Number NCC5-626 between NASA and the California Institute of Technology. Montage is maintained by the NASA/IPAC Infrared Science Archive.

Based on observations made with the NASA/ESA Hubble Space Telescope, and obtained from the Hubble Legacy Archive, which is a collaboration between the Space Telescope Science Institute (STScI/NASA), the Space Telescope European Coordinating Facility (ST-ECF/ESA) and the Canadian Astronomy Data Centre (CADC/NRC/CSA).

This research has made use of the NASA/ IPAC Infrared Science Archive, which is operated by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

This research has made use of the NASA/IPAC Extragalactic Database (NED) which is operated by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

This research has made use of NASA's Astrophysics Data System Bibliographic Services

Based on data of Sloan Digital Sky Survey (SDSS). Funding for the SDSS and SDSS-II has been provided by the Alfred P. Sloan Foundation, the Participating Institutions, the National Science Foundation, the U.S. Department of Energy, the National Aeronautics and Space Administration, the Japanese Monbukagakusho, and the Max Planck Society, and the Higher Education Funding Council for England. The SDSS Web site is http://www.sdss.org/. The SDSS is managed by the Astrophysical Research Consortium (ARC) for the Participating Institutions. The Participating Institutions are the American Museum of Natural History, Astrophysical Institute Potsdam, University of Basel, University of Cambridge, Case Western Reserve University, The University of Chicago, Drexel University, Fermilab, the Institute for Advanced Study, the Japan Participation Group, The Johns Hopkins University, the Joint Institute for Nuclear Astrophysics, the Kavli Institute for Particle Astrophysics and Cosmology, the Korean Scientist Group, the Chinese Academy of Sciences (LAMOST), Los Alamos National Laboratory, the Max-Planck-Institute for Astronomy (MPIA), the Max-Planck-Institute for Astrophysics (MPA), New Mexico State University, Ohio State University, University of Pittsburgh, University of Portsmouth, Princeton University, the United States Naval Observatory, and the University of Washington.

This research has made use of Aladin and Topcat.

I would like to thank Ralf-Jürgen Dettmar for supervising this thesis. The good working atmosphere in his institute enabled me to develop the control software very independently, realising my own ideas. He constantly supported me and realised earlier than I did the importance of the combination between astronomy and computer science. Dominik J. Bomans, I thank for being my second referee. His broad knowledge of the literature was often used for this thesis. Additionally he continuously motivated me to continue with the difficult data set of *NGC 1156* which finally led to a new distance measurement method and an H_2 detection.

In the starting of the *LUCIFER* project Thomas Luks introduced me to the secrets of administrating UNIX operation systems. Additionally he gave great comments to improve the presentation of this thesis. The discussion with him improved my scientific work. Thanks for all.

Peter Buschkamp, Hans Gemperlein and Reiner Hofmann, I would like to thank for the great cooperation throughout the last years. Without this cooperation and their work the *MOS Unit* would not be working. The same great cooperation, I experienced with Michael Lehmitz, Walter Seifert and the rest of the *LUCIFER* team. Without their help and input the control software would have never come to an operational state.

I thank Peter Zinn for being my roommate and the great discussions with him especially those concerning the search for *QSO* candidates.

Fabian Gieseke introduced me to new ideas in applying data mining techniques to astronomical data. The cooperation with him was great throughout the years. Thanks for all the help and discussions.

Thanks to Peter Kamphuis. He provided great comments on the science chapters. With his comments the presentation of my results became more structured.

I would like to thank Enno Middelberg for providing the computing power to create the *QSO* candidates. Discussions with him improved my regression approach, too. All members of the institute which have not been mentioned yet, especially Giuseppe Aronica, Birgitta Burggraf, Tim Falkenbach, Susanne Hüttemeister, Olaf Schmithüsen, Kerstin Weis and Klaus Weißbauer, I thank for all the years of great colleagueship. All contributed directly or indirectly to this thesis with their work, scientific discussions and help.

In all the years Günter Hoppe was a contact person to improve my technical knowledge. He taught me everything that was required to be able to realise a save motion of the opto-mechanical parts. As an amateur astronomer we spend endless nights together observing the sky. This brought me to astronomy. Thanks for doing so.

Finally I would like to thank Klaus Tschira for supporting this thesis with a dissertation grant. By getting this grant I was convinced to start a PhD in astronomy.

Appendix

Appendix A

The LMC Configuration

The *LUCIFER Management Console (LMC)* is the central element to start the *LCSP* (see Section 3.3). Elemental to run the *LMC* effectively is to know how to configure the software start-up. Therefore new services, *Java* application and other commands can be added to an existing configuration. The start descriptors can be manipulated by using the start configuration dialogue. This dialogue provides direct access to all attributes of a descriptor (see Figure A.1). The intent of these attributes is described in Section 3.3, too. A configuration can be saved to and loaded from an *XML* file. When starting the *LMC* in the command shell such a configuration file can be specified by appending the file name e.g., `java -jar lcsp.jar start.xml`. The *JAR*-file that is included in this command is created by the build process (compare Appendix C) whereat the main class of the *LMC* is specified in the manifest file.

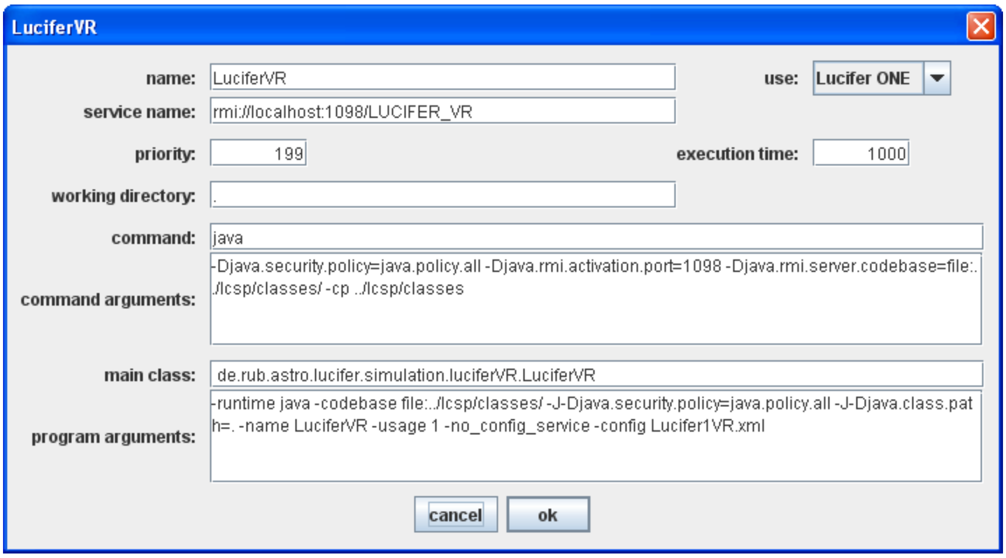


Figure A.1: The *LMC* application start configuration window that is used to view and edit start configurations.

Besides the option to manipulate the start descriptors graphically the configuration file can be accessed with any kind of text editor. Listing A.1 presents in extracts an *LMC* configuration file. To add new entries the specified size of the vector needs to be adjusted. Each of the entries must contain a unique identifier without the need of being sorted. An entry starts with the declaration of the start descriptor type. In the following one can specify the attributes directly (see Table 3.1 for available types and attributes). The exemplarily presented descriptors of Listing A.1 can be used as a base for new descriptors. For that reason an *RMID*, a service and a *Java* start descriptor have been selected to cover the full range of available descriptor types. A more general program start descriptor can be derived from a generic *Java* start descriptor by removing the `mainClass` and `programArguments` attributes.

Listing A.1: start.xml (LMC Start Descriptor File)

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<data>
  <vector size="51">
    <object identifier="0" type="de.rub.astro.lucifer.userInterface.tool.service.start.RMIDStartDescriptor">
      <value name="serviceName" type="String">rmi://localhost:1098/java.rmi.activation.ActivationSystem</value>
      <value name="name" type="String">rmid 1098</value>
      <value name="commandArguments" type="String">-J-Djava.security.policy=java.policy.all -J-Dsun.rmi.server.activation.debugExec=true -port 1098 -C-client</value>
      <value name="command" type="String">nohup rmid</value>
      <value name="workingDir" type="String">.</value>
      <value name="autoStart" type="boolean">true</value>
      <value name="priority" type="int">0</value>
      <value name="executionTime" type="int">5000</value>
      <value name="usage" type="int">3</value>
    </object>
```

```xml
    <object identifier="3" type="de.rub.astro.lucifer.userInterface.tool.service.start.ServiceStartDescriptor">
      <value name="serviceName" type="String">rmi://localhost:1098/MessageServer</value>
      <value name="mainClass" type="String">de.rub.astro.util.message.MessageServer</value>
      <value name="programArguments" type="String">-runtime java -codebase file:lcsp.jar -J-Djava.security.policy=java.policy.all -enable_file_storage</value>
      <value name="name" type="String">MessageServer</value>
      <value name="commandArguments" type="String">-Djava.security.policy=java.policy.all -Djava.rmi.activation.port=1098 -Djava.rmi.server.codebase=file:lcsp.jar -cp lcsp.jar</value>
      <value name="command" type="String">java</value>
      <value name="workingDir" type="String">.</value>
      <value name="autoStart" type="boolean">true</value>
      <value name="priority" type="int">30</value>
      <value name="executionTime" type="int">10000</value>
      <value name="usage" type="int">3</value>
    </object>
```

```xml
    <object identifier="31" type="de.rub.astro.lucifer.userInterface.tool.service.start.JavaStartDescriptor">
      <value name="mainClass" type="String">de.rub.astro.lucifer.userInterface.gui.instrument.mosUnit.MOSPanel</value>
      <value name="programArguments" type="String"/>
      <value name="name" type="String">GUI MOS engineering Client</value>
      <value name="commandArguments" type="String">-cp ./lcsp.jar</value>
      <value name="command" type="String">java</value>
      <value name="workingDir" type="String">.</value>
      <value name="autoStart" type="boolean">false</value>
      <value name="priority" type="int">999</value>
      <value name="executionTime" type="int">1000</value>
      <value name="usage" type="int">4</value>
    </object>
```

Appendix B

The Service Program Arguments

Each service of the *LCSP* is based on the remote service framework (compare Section 4.1). Therefore fundamental command line arguments are passed to each service by its server skeleton. As the server applications initialise the services the command line arguments are the best and often only way to control their operating mode during start-up. The available arguments of the servers are presented in this appendix. This description starts with the general arguments that can be specified for every service as their processing is realised by the remote service framework.

-? -help displays a short list of available program arguments. The use of each of the available arguments of a specific service is explained briefly in this listing. If the arguments that are passed to the server are malformed or invalid the same help screen is displayed.

-name together with this argument the name of a service is specified. This name is used by e.g., the messaging system to uniquely identify a service.

-usage as the name argument this parameter is used to assign a usage to a service. This usage is an additional information to distinguish between services. E.g., the services of the *Instrument Tier* of both *LUCIFER* instruments are tagged as a part of the left or right instrument. Available parameters are <1> for *LUCIFER* 1, <2> for *LUCIFER* 2 and <3> for system services that are used by both instruments.

-codebase specifies the code base where the java classes of the service can be found. This is required by the activation system daemon to create the service correctly. In the other case the code base of the activation system must be modified to locate the classes of each service. The correct specification of the code base is also required by the clients to find the *stub* classes and load them on demand from the specified location.

-runtime defines the runtime that should be used by the activation system daemon to host the remote service. This is helpful if services require a specific *JRE* for their execution.

-J is used to pass runtime arguments to the created virtual machine. This is required to e.g., modify the disposable memory size. In the *LMC* <`-J-Djava.security.policy=java.policy.all`> is specified to use the permissive *LCSP* security policy file for the *JRE* and <`-J-Djava.class.path=`> sets the project specific class path.

-local prevents a service from being started on demand by the activation system daemon. The service is started as a local remote service that is executed within the *JRE* of the servers skeleton instead. This allows for local debugging of a service which is impossible for a dynamically activated one.

-suspend suspends the activated version of a service by sending a message to the corresponding remote object. A grace time can be specified to wait before ending the service.

-stop stops the service completely. In comparison to the suspend argument the service is not only ended. In addition it is ensured that the service is de-registered from the activation system daemon and unbound from the registry. This is necessary to prevent the service from being restarted automatically.

-config sets the configuration file that should be used. As every service uses a basic and shared configuration file to store the access parameter of the centralised configuration service the default value <`localConfig.xml`> can be changed.

-no_config_service disables the usage of the centralised *Configuration Service*. This means that all configuration values are taken from a local file. It is advised to use this argument together with the **-config** argument to have a dedicated file.

-no_time_service disables the usage of the *Time Service* for service synchronisation.

-no_message_service disables the usage of the *Message Service*. This means that messages are processed locally without being send to the *Message Service*. A disabled *Message Service* usage forces the remote service to print the messages out directly.

-ignore_messages defines that all messages that are generated within the subsystems are rejected and that no message is neither stored nor printed out.

Because the following program arguments are service dependent either the group or the specific service is included in the description.

-unit specifies the address of the unit to connect to. This parameter is applied to all services that interact with hardware via the *RS232* framework. Without specifying a certain unit address the hardware socket is transparently retrieved from the central port server look-up service by the underlying socket communication. All services of the electronics and environment allow the usage of this parameter (see Figure 3.1).

-server specifies the socket address of the *GEIRS Server*. This address is used by the *Readout Service* to open a stream to the command server of the *GEIRS* software to send and receive commands and responses.

-log_transmissions activates the persistent logging of data transmissions between a service of the *Control Tier* and its hardware counterpart. By default the data that is transmitted is stored in a database. If the database storage is deactivated all transmission are dumped to a file. This argument is very important to enable the engineers to debug the hardware-software communication. All service that can specify a socket address and therefore use the communication framework support this argument.

-no_database deactivates the database storage of a service. This is either possible for the services mentioned above that use a socket connection to communicate with the electronics and allow persistent database storage of transmissions or those services that have their own database connection to store elements. These are the *Message Service* that is responsible for storing messages and the *Journalizer* that maps the instrument status to the database.

-MOS tells the *MCU Service* which version of hardware is connected. Since both the instrument units and the *MOS Unit* use comparable electronics this argument enables special functions for the *MOS Unit*. This differentiation is also required for the correct integration of *LuciferVR*.

-check_encoder enables the encoder check capabilities of the *MOS Unit Service*. By continuously observing the values of the *MOS* hardware encoder drift errors can be traced.

-logging_interval specifies a fixed logging interval for all services of the *Control Tier* that control the environment of the instrument (except the *Calibration Unit*). If no fixed logging interval is specified those units use by default an adaptive logging mechanism that increases its rate when changes are measured and vice versa.

-check_hardware enables a continuous testing of the switches that are access via the *Switch Box Service*. This is necessary to test the stability of the switches and find randomly jumping outputs.

-send_configuration activates the sending of initial configuration data to the *Pressure Monitor* and *Temperature Controler* electronics.

-no_journalizer disables the status reporting to the *Journalizer*. This argument is applicable to all services of the *Instrument Tier* as well as the environment services, *Readout Service* and *Telescope Service* of the *Control Tier*.

-hibernate allows to specify an alternative *Hibernate* configuration file.

-use_lucifer_vr activates the native probes in the electronics services to send commands directly to the virtual instrument (see Chapter 8).

-simulate activates the integration of *LuciferVR* into the electronics services or the temperature monitor. This means that communication with the real hardware is disabled and all responses are created on the basis of the simulated instrument status.

-reload tells the *Configuration Service* to reload the configuration file. This can be required if the configuration was changed manually instead of using the *LMC*.

-enable_file_storage enables the file storage of messages in the *Message Service*. This argument was used to analyse messages off-line before the *Message Database Browser* provided a user-friendly visualisation of stored messages.

Appendix C

The LCSP Ant Build File

The *LUCIFER Control Software Package (LCSP)* is build with the *Ant* tool. This tool allows to control the calls to individual tasks that compile, package and assemble the software product. In the *LUCIFER* project an automated build process was introduced in a very early state, to allow all developers a unified and easy way of software generation and deployment. The following Listing C.1 presents the used *Ant* configuration file.

The configuration file starts with an *Ant* preamble where the project name and basic parameter are defined. This preamble is followed by lines that specify the name of the application and the different directories of the project. This directories are used to separate sources, resource files (e.g., images), external libraries, configuration data and internationalisation files. The next specified directories are generated by the build process and contain the results of the individual build tasks (e.g., compiled classes, *API*-documentation, bundled software package). Finally the used external packages are defined.

Listing C.1: build.xml (LCSP Build File)

```xml
<project name="lucifer control software package"
         default="build_distribution"
         basedir="..">
  <description>Buildfile of the lucifer control software package</description>
  <property name="APPLICATION_NAME" value="lcsp"/>
  <property name="APPLICATION_ARCHIVE" value="lcsp.jar"/>

  <property name="SOURCE_DIR" location="${basedir}/source"/>
  <property name="RESOURCE_DIR" location="${basedir}/resource"/>
  <property name="LIBRARY_DIR" location="${basedir}/lib"/>
  <property name="CONFIG_DIR" location="${basedir}/config"/>
  <property name="LOOKUPTABLE_DIR" location="${basedir}/config/lookupTables"/>
  <property name="I18N_DIR" location="${basedir}/I18N"/>
  <property name="TARGET_DIR" location="${basedir}/classes"/>

  <property name="DOCUMENTATION_DIR" location="${basedir}/docs"/>
  <property name="DISTRIBUTION_DIR" location="${basedir}/dist"/>
  <property name="EXPORT_DIR" location="${DISTRIBUTION_DIR}/export"/>
  <property name="EXPORT_LIB_DIR" value="lib"/>
  <property name="EXPORT_LOOKUPTABLE_DIR" value="lookupTables"/>

  <property name="MYSQL_CONNECTOR_NAME"
            value="mysql-connector-java-3.1.12-bin.jar"/>
  <property name="HIBERNATE_CONFIG_FILE" value="hibernate.cfg.xml"/>
  <property name="LOG4J_CONFIG_FILE" value="log4j.properties"/>
  <property name="PDF_DOCLET.JAR" value="pdfdoclet-1.0.2-all.jar"/>
  <property name="TOOLS.JAR" value="tools.jar"/>
  <property name="JAR_CLASS_PATH"
            value="lib/mysql-connector-java-3.1.12-bin.jar lib/hibernate3.jar 
                lib/jta.jar lib/log4j-1.2.11.jar lib/dom4j-1.6.1.jar lib/
                commons-logging-1.0.4.jar lib/commons-collections-2.1.1.jar lib
                /cglib-2.1.3.jar lib/asm-attrs.jar lib/asm.jar lib/antlr-2.7.6
                rc1.jar"/>
```

The next section of the build file is used to set some basic path variables. These path information are used in later tasks.

```
<path id="JAVADOC_CLASS_PATH">
  <fileset dir="${LIBRARY_DIR}">
    <include name="${TOOLS.JAR}"/>
  </fileset>
</path>
<path id="DOCLET_PATH">
  <fileset dir="${LIBRARY_DIR}">
    <include name="${PDF_DOCLET.JAR}"/>
  </fileset>
</path>
<path id="COMPILATION_LIBRARIES">
  <fileset dir="${LIBRARY_DIR}">
    <include name="*.jar"/>
  </fileset>
</path>
```

The *Ant* build file syntax allows to define individual targets. These targets can be executed individually. By defining dependencies the *Ant* tool ensures to execute the specified and required targets previously. The next target is used to clean the output directory.

```
<target description="create a clean target directory"
        name="clean">
  <delete dir="${TARGET_DIR}" failonerror="false"/>
  <mkdir dir="${TARGET_DIR}"/>
</target>
```

The following target specifies how to copy the resources to the output directory. Therefore the needed images, audio files, hibernate mapping and hibernate configuration files are duplicated. Additionally the needed internationalisation property files are copied.

```
<target description="copy all needed resources to the target folder"
        name="copy-resources">
  <copy todir="${TARGET_DIR}">
    <fileset dir="${RESOURCE_DIR}">   <!-- copy images -->
      <include name="**/*.gif"/>
      <include name="**/*.jpg"/>
      <include name="**/*.au"/>
    </fileset>
    <fileset dir="${SOURCE_DIR}">     <!-- copy hibernate mappings -->
      <include name="**/*.hbm.xml"/>
    </fileset>
    <fileset dir="${CONFIG_DIR}">     <!-- copy hibernate configuration -->
      <include name="${HIBERNATE_CONFIG_FILE}"/>
      <include name="${LOG4J_CONFIG_FILE}"/>
    </fileset>
    <fileset dir="${I18N_DIR}">       <!-- copy I18N -->
      <exclude name="Root"/>
      <exclude name="Entries"/>
      <exclude name="Repository"/>
      <exclude name="**/*de_DE.properties"/>
    </fileset>
  </copy>
</target>
```

The next target contains all information that is necessary to call the *JavaDoc* tool and generate the *API*-documentation. This targets depends on a successful compilation of the source files. Together with the *HTML* pages a *PDF* document is generated. Besides the default parameters a specialisation is needed to support the custom-made and project specific annotation tags. More Information on the project specific tags can be found in Appendix D.

```
<target description="create the javaDoc documentation"
        name="javaDoc"
        depends="compile">
  <mkdir dir="${DOCUMENTATION_DIR}"/>
  <delete dir="${DOCUMENTATION_DIR}/api"/>
  <tstamp />   <!-- Create the time stamp -->
  <javadoc access="private"
           destdir="${DOCUMENTATION_DIR}/api"
           sourcepath="${SOURCE_DIR}"
           packagenames="de.astro.rub.*"
           windowtitle="Lucifer Control Software Package"
           link="http://java.sun.com/j2se/1.5.0/docs/api"
           verbose="true"
           linksource="true"
           author="true"
           version="true"
           breakiterator="true"
           maxmemory="512m">
    <header>
      <![CDATA[<em><b>L</b>ucifer <b>C</b>ontrol S</b>oftware <b>P</b>ackage<
        br><font size=-2>by <a href='mailto:polsterer@astro.rub.de'>Kai
        Polsterer</a>, <a href='mailto:juette@astro.rub.de'>Marcus J&uuml;
        tte</a> and <a href='mailto:volker.knierim@astro.rub.de'>Volker
        Knierim</a></font></em>]]>
    </header>
    <footer>
      <![CDATA[<font size=-1>please report errors to <a href='
        mailto:juette@astro.rub.de'>Marcus Juette</a> or <a href='
        mailto:polsterer@astro.rub.de'>Kai Polsterer</a></font>]]>
    </footer>
    <doctitle>Lucifer Control Software Package Documentation</doctitle>
    <packageset dir="${SOURCE_DIR}">
      <include name="de/rub/astro/**"/>
    </packageset>
    <tag name="pre"
         description="Preconditions:"
         scope="constructors,methods"/>
    <tag name="post"
         description="Postconditions:"
         scope="constructors,methods"/>
    <taglet name="de.rub.astro.util.taglet.ToDoTaglet"/>
    <taglet name="de.rub.astro.util.taglet.ChangesTaglet"/>
    <taglet name="de.rub.astro.util.taglet.JUnitTaglet"/>
    <taglet name="de.rub.astro.util.taglet.ExampleTaglet"
            path="${TARGET_DIR}"/>
  </javadoc>
  <javadoc classpathref="JAVADOC_CLASS_PATH"
           access="private"
           sourcepath="${SOURCE_DIR}"
           additionalparam="-pdf lcsp_api_documentation.${DSTAMP}_${TSTAMP}.
             pdf"
           maxmemory="512m">
    <packageset dir="${SOURCE_DIR}">
      <include name="de/rub/astro/**"/>
    </packageset>
    <doclet name="com.tarsec.javadoc.pdfdoclet.PDFDoclet"
            pathref="DOCLET_PATH">
    </doclet>
  </javadoc>
</target>
```

All information needed to compile the software is specified in the listing of the following target.

```
<target description="compile the source"
        name="compile"
        depends="clean">
  <tstamp />   <!-- Create the time stamp -->
    <javac srcdir="${SOURCE_DIR}"
           destdir="${TARGET_DIR}"
           classpathref="COMPILATION_LIBRARIES"
           target="1.6"
           source="1.6"
           optimize="on"
           deprecation="on"
           debug="true"/>
</target>
```

To allow remote method access the stubs of the remote objects need to be generated. With *Java* 1.5 the creation of skeletons is no longer required. The next target specifies the parameters of the *RMI*-compiler (*rmic*). This generation of stubs is depending on existing compiled classes.

```
<target description="create skeletons and stubs"
        name="rmic"
        depends="compile">
  <rmic base="${TARGET_DIR}" includes="**/RMI*Impl.class"/>
</target>
```

The next target is used to generate a deployable version of the control software. Therefore the software is compiled, the stubs are generated and the resource files are copied to the target folder. After the depending tasks have been executed the export directory is prepared and the distribution is bundled in a *JAR*-file.

```
<target description="generate the distribution"
        name="build_distribution"
        depends="compile,rmic,copy-resources">
  <mkdir dir="${DISTRIBUTION_DIR}"/>
  <delete dir="${EXPORT_DIR}" failonerror="false"/>
  <mkdir dir="${EXPORT_DIR}"/>
  <mkdir dir="${EXPORT_DIR}/${EXPORT_LIB_DIR}"/>
  <mkdir dir="${EXPORT_DIR}/log"/>
  <jar jarfile="${DISTRIBUTION_DIR}/${APPLICATION_NAME}.${DSTAMP}_${TSTAMP}.
      jar"
       basedir="${TARGET_DIR}">
    <manifest>
      <attribute name="Built-By"
                 value="${user.name}"/>
      <attribute name="Specification-Title"
                 value="Lucifer Control Software Package"/>
      <attribute name="Specification-Version"
                 value="Saturn"/>
      <attribute name="Specification-Vendor"
                 value="Astronomisches Institut Ruhr-Universitaet Bochum"/>
      <attribute name="Implementation-Title"
                 value="Lucifer Control Software Package"/>
      <attribute name="Implementation-Version"
                 value="${DSTAMP}/${TSTAMP}"/>
      <attribute name="Implementation-Vendor"
                 value="Astronomisches Institut Ruhr-Universitaet Bochum"/>
      <attribute name="Main-Class"
           value="de.rub.astro.lucifer.userInterface.gui.tool.start.Start"/>
      <attribute name="Class-Path"
                 value="${JAR_CLASS_PATH}"/>
    </manifest>
  </jar>
```

Together with all basic configuration files this *JAR*-file is compressed to a *ZIP* file with a representative time stamp in its name. This file contains all data to run the software and can be easily distributed within the *LUCIFER* team.

```
    <copy tofile="${EXPORT_DIR}/${APPLICATION_ARCHIVE}"
      file="${DISTRIBUTION_DIR}/${APPLICATION_NAME}.${DSTAMP}_${TSTAMP}.jar"/>
    <copy todir="${EXPORT_DIR}">
      <fileset dir="${CONFIG_DIR}">
        <include name="start.xml"/>
        <include name="config.xml"/>
        <include name="java.policy.all"/>
        <exclude name="${HIBERNATE_CONFIG_FILE}"/>
        <exclude name="${LOG4J_CONFIG_FILE}}"/>
      </fileset>
    </copy>
    <copy todir="${EXPORT_DIR}">
        <fileset dir="${basedir}">
          <include name="update.*"/>
        </fileset>
    </copy>
    <copy todir="${EXPORT_DIR}/${EXPORT_LIB_DIR}" >
      <fileset dir="${LIBRARY_DIR}">
        <include name="**.jar"/>
        <exclude name="${TOOLS.JAR}"/>
        <exclude name="${PDF_DOCLET.JAR}"/>
      </fileset>
    </copy>
    <copy todir="${EXPORT_DIR}/${EXPORT_LOOKUPTABLE_DIR}" >
      <fileset dir="${LOOKUPTABLE_DIR}">
        <include name="**.xml"/>
     </fileset>
    </copy>
    <jar basedir="${EXPORT_DIR}"
        jarfile="${DISTRIBUTION_DIR}/${APPLICATION_NAME}.${DSTAMP}_${TSTAMP}.
            zip"/>
  </target>
```

The last target builds a deployable version of the software and the corresponding *API*-documentations files. This is done by simply calling the target that build the software and generate the documentation.

```
  <target description="generates the distribution and the documentation"
          name="build_all"
          depends="build_distribution,javaDoc"/>
</project>
```

With merely 200 lines of code this *Ant* build file controls the whole build process of the *LCSP*. This demonstrates how powerful the *Ant* tool is and how easy a configuration file can be composed. The power of the *Ant* tool lies in its predefined tasks that just need to be configured accordingly. Instead of writing platform dependent shell scripts that initiate the individual command calls *Ant* provides an independent build process specification approach.

Appendix D

The LCSP JavaDoc Annotations

Documenting software is one of the most important tasks in software development projects. Thus the *LUCIFER* project started with the development of an appropriate documentation procedure in advance of writing source code. It was decided to use the *JavaDoc* tool for the task of generating the *API*-documentation. The *JavaDoc* tool is part of the *Java Software Development Kit (SDK)* of *SUN*. This tool provides functionalities to generate a comprehensive *API*-documentation out of the source code itself. This documentation is by default stored as browsable *HTML* pages. To comply with user-defined requirements the *JavaDoc* tool can be extended by writing individual *Taglets* to interpret additional documentation annotations. For the *LUCIFER* project several *Taglets* have been written.

The *JavaDoc* tool automatically scans the source code of a project and analyses the individual class structures and their dependencies. Additionally documentation tags that are embedded in the source code are evaluated. These tags can be used e.g., to describe the parameter of a method or its return value. See Listing D.1 and the generated *HTML* documentation (Figure D.1) for an example of using *JavaDoc* annotations. By using special *Doclets*, *JavaDoc* can generate e.g., a *PDF* documentation with several thousand pages.

In the *LUCIFER* project the following tags with their project specific purpose has been used. They can be divided into project specific and and default *JavaDoc* annotations.

@author A general tag that is used to assign a person to a class or interface. In the *LUCIFER* project the authors of a source file are sorted by their responsibility. This allows to identify the persons to ask for bug-fixing or additional information on using the code.

@version Another default *Java* tag to record the current version of a source file. This version tag contains the date of the last major change instead of a common *Java* version number. Minor changes to the source are documented by the version management system (see Section 2.3).

@since This tag is used to store the date a class, interface or method was created first, instead of a common *Java* version number.

@changes This tag is a very project specific *JavaDoc* annotation. A special *Taglet* class has been written to support the logging of a change history and export it in a formatted way to the *API*-documentation. This reduces the necessity of accessing the version management system to retrieve a change history. Another benefit is that the source itself contains its history and minimises the possibility of loosing it when changing the version management system. This sepcialised tag can be used for classes, interfaces and methods.

@see A default *JavaDoc* annotations to refer onto another class, interface or method. This references may be used to point to helpful descriptions of other source elements.

@deprecated Deprecated classes, interfaces or methods can be marked with this tag. Further use of elements with this annotation is discouraged. This elements still exist to ensure interoperability with other source code and may me removed in the future. In the *LUCIFER* project this tag that belongs to the default *JavaDoc* annotations is rarely used at the moment. With future version and major changes this tag will be used more often. Finally, after a major change has been applied the deprecated elements may be removed from the source including the tag itself.

@param This default tag is used to describe the parameters of a constructor or a method. It is used to specify addition information concerning the use of a parameter.

@return A standard annotation to describe the return value of a method. Especially for complex data structures an extensive description may be very helpful.

@throws The exception handling is one of the remarkable benefits of *Java*. To add comprehensive information to the *API*-documentation the error behaviour of a method can be described by this default tag. All possible exception that can occur can be described including the reason of their appearance. To know the exception declaration is essential to use a method and react accordingly onto an error.

@pre This project internal tag is used to document any kind of pre-conditions to be considered when calling a method. This may be e.g., the initialisation of a data structure in advance or the existence other objects to communicate with. Together with the exception handling information the pre-conditions are essential for calling a method accordingly.

@post Another project internal tag is available to document the post-conditions of a method call. This information may be helpful to understand the internal actions of a method that do not belong to its primary task (e.g., a data structure being initialised/modified or a motion command being transmitted).

@testcase This project specific *JavaDoc* annotation is based on an individual *Taglet* class, written to allow the coupling between a class and its *JUnit* test case.

@todo An individual *Taglet* class allows to add missing programming tasks to the documentation. This tasks are most often nice to have but not essential for a functional software. This annotation is project specific and does not belong to the default *JavaDoc* annotations.

@example The most complex tag can be used to specify examples on how to use a class/method. This project specific *Taglet* was written to add formatted source code examples to the *API*-documentation. This examples can be very helpful to understand how to integrate functionalities of existing code. Especially in projects with several developers that may change within time these examples reduce the period of vocational adjustment. There is no need to spend time on searching for corresponding source parts in other files.

Listing D.1: TimeClient.java (JavaDoc Annotation Example)

```java
/**
 * This class provides a client to the time server. It is used from the
 * TimeStamp class to retrieve a synchronized time. Its get-method returns the
 * current time in milliseconds since 1.1.1970 corrected by the correction term
 * @author Kai Polsterer & Marcus J&uuml;tte
 * @since 30.04.2003
 * @changes 30.04.2003 adding {@link #getTime()}
 * @changes 31.10.2003 changing {@link #getTime()} to use time server for
 *   synchronisation.
 * @changes 31.10.2003 adding {@link #correctionTerm}, {@link #synchron}
 *   {@link #run()}, {@link #getCorrectionTerm()}, {@link #isSynchron()}
```

```java
/* @version 31.08.2004
 * @example Using the TimeClient.
 * import de.rub.astro.util.time.TimeClient;
 *
 * public class Tester {
 *   public static void main (String[] args) {
 *     TimeClient.getClient().sync(); // synchronize with the server
 *     long currentTime = TimeClient.getClient().getTime(); // gets the time
 *   }
 * }
 */
public class TimeClient extends ActivatableRemoteServiceClientImpl
                        implements Runnable, Debug, TimeObserver {
```

```java
  /**
   * Initialises a new <code>TimeClient</code> object by setting the
   * attributes to their initial values.
   * @since 11.12.2003
   * @param registryAddress address of the registry hosting the service this
   *   client is used to connect to.
   * @param serviceName the name of the service at the registry.
   * @param timeService the <code>RMITimeService</code> the client should be
   *   used for. If this parameter is not specified the <code>RMITimeService
   *   </code> is retrieved by using the service information.
   * @throws IllegalArgumentException if the <code>serviceName</code> is null
   *   or empty or the <code>registryAddress</code> is null, has no port
   *   specified or has no protocol.
   */
  public TimeClient(Address registryAddress, String serviceName, RMITimeService
      timeService) throws IllegalArgumentException {
```

```java
  /**
   * This method returns the current system time in milliseconds since 1.1.1970
   * GMT 0:00 corrected by the <code>correctionTerm</code>. The <code>
   * correctionTerm</code> represents the time difference between the time
   * server and this client.<br>
   * If the time client could not synchronize with the time server the value
   * returned is negative.
   * @return current time in milliseconds since 1.1.1970 GMT 0:00. If not
   *   synchronized with the time server this value is multiplied with -1.
   * @see java.lang.System#currentTimeMillis()
   * @since 30.04.2003
   */
  public long getTime()
```

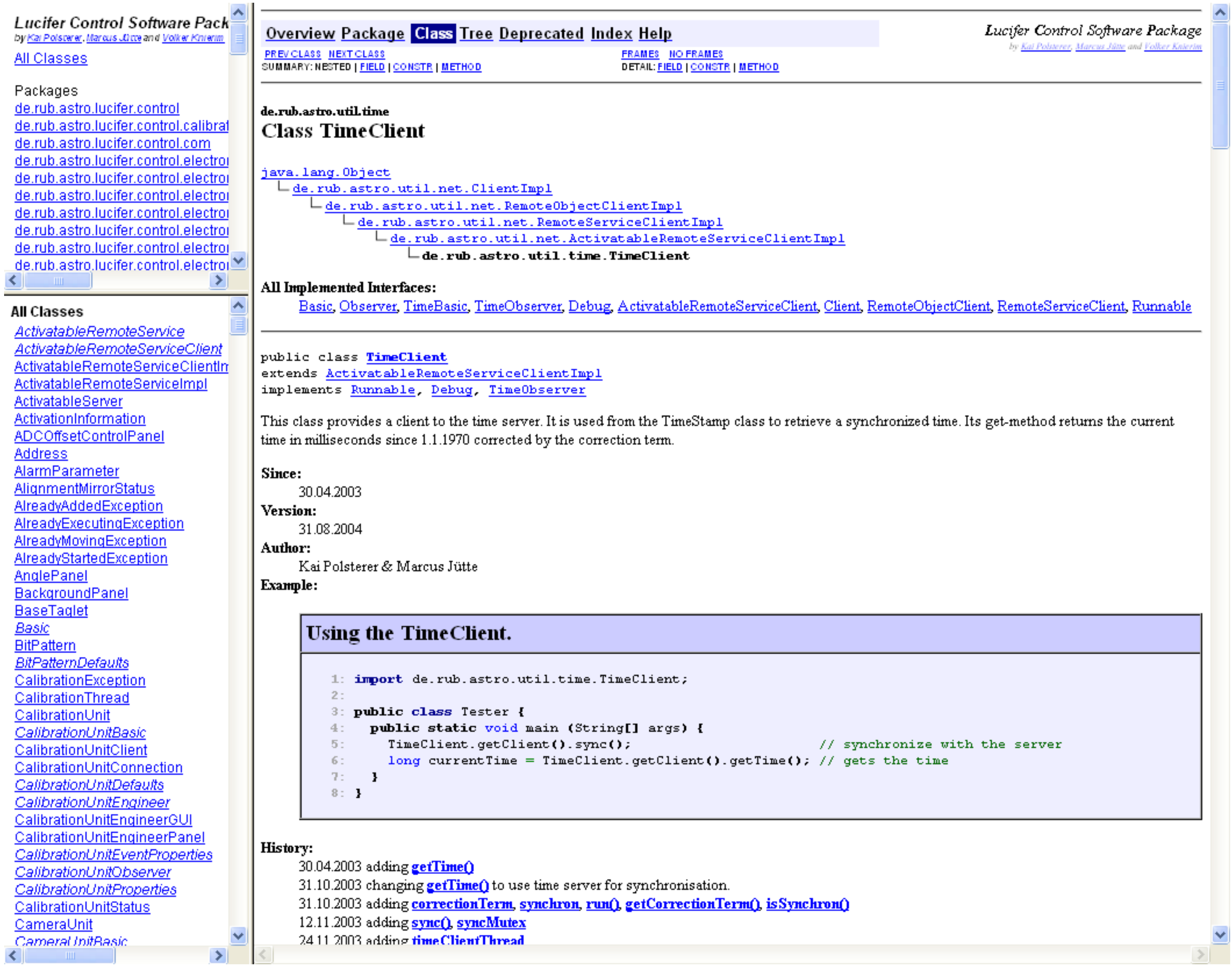


Figure D.1: *HTML API*-documentation of *TimeClient.java* generated with the *JavaDoc* tool.

Appendix E

The LCSP Hibernate Configuration and Mapping File Example

To use the *Hibernate* framework to persist data of objects into a relational database a configuration and the mappings of each class need to be specified. The following sources provide a short example on how to do this. The presented mapping and configuration are taken from the *LCSP*. This mapping was actually the first mapping created. All other mappings of system status objects have the same complexity and are comparably assembled by using this mapping as a draft. Before starting to describe a mapping and the mapped class, the *Hibernate* configuration is discussed.

The *Hibernate* configuration file (see Listing E.1) is needed to configure the behaviour of the session factory. It starts with the specification of the database connection. These four properties define the database driver/address, the database user and the password to use. The next property is used to limit the size of connections that are managed by *Hibernate* to one. This parameter can be used to optimise the systems performance. Because *Hibernate* is able to communicate with different relational database implementations the next property is needed to specify the database dialect. The next property binds the execution of **SessionFactory**.*getCurrentSession()* onto the calling thread. After disabling the second level object cache and the *SQL* statement display the table creation policy is specified. By using create instead of update each software restart would empty the database. Finally the *Hibernate* configuration file contains the links to the mapping files of the individual classes. To improve the readability of the listing the path information has been truncated. The full path points to the same location where the relevant source code files can be found e.g., <`de/rub/astro/lucifer/instrument/mosUnit`>. The first mapping that was created is presented as an example in Listing E.2.

To fully understand this mapping the corresponding *Java* class needs to be known. Listing E.3 shows the essentials of the class that belong to the mapping presented in Listing E.2. The extended class **JournalizerObject** inherits a simple **TimeStamp** and **JournalizerKey** attribute. These attribute are needed to identify logged instrument parameters (see Section 5.5).

A *Hibernate* mapping contains entries for each class that should be mapped. In our example the mapping specifies just one class. Besides the fully qualified class name the table where the **InstrumentTemperatures** are stored is specified. Inside this class definition first of all the unique identifier that is used to reference an object in the instrument temperature table is defined. Therefore the type and the method of generation is entered. In this case a numeric identifier that is created by the database is chosen. The attributes inherited from **JournalizerObject** are sub-components of the **InstrumentTemperatures** class. The access of *Hibernate* onto this attributes is specified by using the access value of either a property or component tag. A direct field access allows to directly interact with the attributes without get and set methods. Additionally to this direct access *Hibernate* provides two other possibilities. These are property access with get and set methods as well as custom definable access. For each sub-component of a class their type and name needs to be declared. Attributes of a sub-component are defined equal to direct attribute. For each of these properties their *Java* source name, access style and the name of the database table column to store the data in should be given. In some cases it is useful to specify the database data type.

In the **InstrumentTemperatures** class (see Listing E.3) all important data is stored within an array of floating point numbers. This primitive array is reproduced in a separate table. The primary identifier of this extra table is used in the main table to find the corresponding array, while the index column references the individual entries. In the *LUCIFER* project relatively simple data structures needed to be mapped. Nonetheless the complexity to map larger data structures with *Hibernate* is still negligible in comparison to compose and integrate plain *SQL* statements into software.

Listing E.1: hibernate.cfg.xml (Hibernate Configuration File)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration 
    DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.
    dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">
      jdbc:mysql://127.0.0.1/instrument</property>
    <property name="connection.username">guessWho</property>
    <property name="connection.password">secret</property>
    <property name="connection.pool_size">1</property>
    <property name="dialect">
      org.hibernate.dialect.MySQLInnoDBDialect</property>
    <property name="current_session_context_class">thread</property>
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider</property>
    <property name="show_sql">false</property>
    <property name="hbm2ddl.auto">update</property>
    <mapping resource=".../InstrumentTemperatures.hbm.xml"/>
    <mapping resource=".../ReadOutControlTemperatures.hbm.xml"/>
    <mapping resource=".../RackControlValues.hbm.xml"/>
    <mapping resource=".../Pressures.hbm.xml"/>
    <mapping resource=".../Pressure.hbm.xml"/>
    <mapping resource=".../TurboPumpValues.hbm.xml"/>
    <mapping resource=".../CalibrationUnitStatus.hbm.xml"/>
    <mapping resource=".../GratingUnitStatus.hbm.xml"/>
    <mapping resource=".../FilterUnitStatus.hbm.xml"/>
    <mapping resource=".../CameraUnitStatus.hbm.xml"/>
    <mapping resource=".../DetectorUnitStatus.hbm.xml"/>
    <mapping resource=".../CompensationMirrorStatus.hbm.xml"/>
    <mapping resource=".../AlignmentMirrorStatus.hbm.xml"/>
    <mapping resource=".../PupilViewerStatus.hbm.xml"/>
    <mapping resource=".../MOSUnitStatus.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Listing E.2: InstrumentTemperatures.hbm.xml (Hibernate Mapping File)

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate␣Mapping␣DTD␣3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class table="Instrument_Temperatures"
         name="de.rub.astro.lucifer.control.temperatureMonitor.
             InstrumentTemperatures">
    <id column="ID"
        type="long">
      <generator class="native"/>
    </id>
    <component name="timeStamp"
               class="de.rub.astro.util.time.TimeStamp"
               access="field">
      <property name="time"
                column="TIME"
                type="long"
                access="field"/>
      <property name="synchron"
                column="SYNCHRON"
                type="true_false"
                access="field"/>
    </component>
    <component name="journalizerKey"
               class="de.rub.astro.lucifer.journalizer.JournalizerKey"
               access="field">
      <property name="usageType"
                column="USAGE_TYPE"
                type="int"
                access="field"/>
    </component>
    <primitive-array name="temperatures"
                     table="Instrument_Temperatures_Array"
                     access="field">
      <key column="TEMP_ID"/>
      <index column="SENSOR_NUMBER"/>
      <element column="TEMPERATURE" type="float"/>
    </primitive-array>
  </class>
</hibernate-mapping>
```

Listing E.3: InstrumentTemperatures.java (Java Source File)

```java
public class InstrumentTemperatures extends JournalizerObject implements
    TemperatureMonitorDefaults, Serializable, UsageTypes,
    JournalizerObjectTypes {
```

```java
  /*
   * Stores the temperature values.
   * @since 24.09.2004
   */
  private float[] temperatures;
```

Appendix F

Spectra of the Earth's telluric features at Kitt Peak Observatory

The following high resolution (R = 40.000) spectra (Figure F.1) of the Earth's atmosphere show telluric features in the SZ, J, H and K band. These spectra of the sky at zenith were taken with the *Fourier Transform Spectrometer (FTS)* at the *National Solar Observatory (NSO)*/*Kitt Peak Observatory* and processed by the *National Science Foundation (NSF)*/*National Optical Astronomy Observatory (NOAO)*. The original data is hosted at `ftp://ftp.noao.edu/catalogs/atmospheric_transmission/` (2009). To improve the quality of presentation the spectra have been binned to contain ≈2,000 transparencies per spectrum. The spectra clearly visualise the atmospheric windows for ground based *NIR* observations.

The *Kitt Peak Observatory* is located close to the *Mount Graham International Observatory (MGIO)*. Therefore the telluric features in the presented spectra can be used for the wavelength calibration of *LUCIFER* observations. The relation between atmospheric parameters like the amount of water vapour or the airmass and the strength of the telluric features demands an attentive calibration.

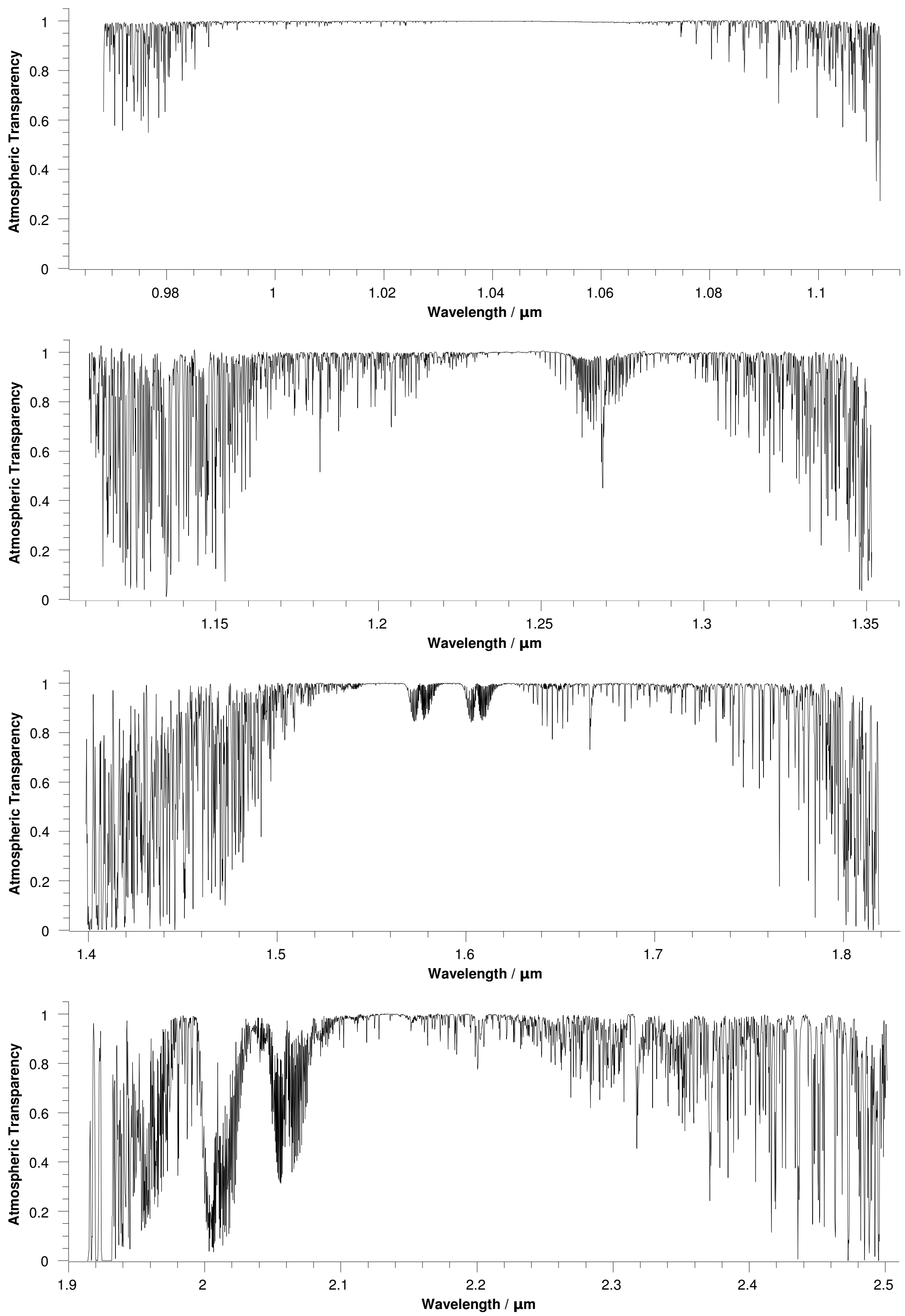


Figure F.1: Spectra of the Earth's telluric features at Kitt Peak Observatory. From top to bottom: SZ band, J band, H band, K band

List of Figures

List of Tables

List of Listings

Acronyms

2MASS	Two Micron All Sky Survey
AC	Auxiliary Cryostat
ACS	Advanced Camera for Surveys
AD-C	Analogue/Digital Converter
ADASS	Astronomical Data Analysis Software & Systems
ADC	Atmospheric Dispersion Corrector
AG	Astronomische Gesellschaft
AGN	Active Galactic Nucleus
AGNS	Active Galactic Nucleus Sample
AGW	Acquisition, Guiding and Wavefront Sensing Unit
AIP	Astrophysikalisches Institut Potsdam
AIRUB	Astronomisches Institut der Ruhr-Universität Bochum
ANN	Artificial Neural Network
AO	Adaptive Optics
AOS	Adaptive Optics System
API	Application Programming Interface
APLpy	Astronomical Plotting Library in Python
ASCII	American Standard Code for Information Interchange
AU	Astronomical Unit
AVO	Astrophysical Virtual Observatory
BASE-T	Baseband over Twisted Pair
CA	afferent coupling
CAD	Computer Aided Design
CAHA	Centro Astronómico Hispano-Alemán
CCD	Charge-Coupled Device
CE	efferent coupling
CMB	Cosmic Microwave Background
CORBA	Common Object Request Broker Architecture
CR	comment ratio
CSQ	Command Sequencer
CVS	Concurrent Versions System

DEL	delayed QSO shining
DIN	Deutsches Institut für Normung
DIT	Detector Integration Time
DMA	Direct Memory Access
DOM	Document Object Model
DR6	Sixth Data Release
ECC	Error Correcting Code
ECS	Enclosure Control System
edt	Engineering Design Team
ERC	Enclosure Rotation Control
ESO	European Southern Observatory
f	focal length
FC-AL	Fibre Channel - Arbitrated Loop
FHTG	Fachhochschule für Technik und Gestaltung
FIR	far-infrared
FITS	Flexible Image Transport System
FOV	Field of View
FPU	Focal Plane Unit
FTS	Fourier Transform Spectrometer
FWHM	full width at half maximum
GCS	Guiding Control System
GEIRS	Generic Infrared Software
Gflop/s	Giga Floating Point Operations Per Second
GOODS	Great Observatories Origins Deep Survey
GPU	Graphics Processing Unit
GSC	Guide Star Catalog
GUI	Graphical User Interface
HAWAII-2	HgCdTe Astronomical Wide Area Infrared Imager-2
HDD	Hard Disc Drive
HIRAMO	High Resolution Analog Measurement and Output Board
HST	Hubble Space Telescope
HTML	Hypertext Markup Language
I18N	Internationalisation
IBM	International Business Machines
ICE	Internet Communications Engine
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IIF	Instrument Interface
INAF	Istituto Nazionale di Astrofisica
IO	Input/Output
IP	Internet Protocol

IR	infrared
IRAC	Infrared Array Camera
ISO	International Organisation for Standardisation
JAR	Java Archive
JDBC	Java Database Connectivity
JRE	Java Runtime Environment
kd-Tree	k-Dimensional Tree
kNN	k-Nearest Neighbours
KTS	Klaus Tschira Stiftung
LBC	Large Binocular Camera
LBT	Large Binocular Telescope
LBTB	LBT Beteiligungsgesellschaft
LBTI	LBT Interferometer
LBTO	LBT Observatory
LCSP	LUCIFER Control Software Package
LHC	Large Hadron Collider
LINC-NIRVANA	LBT Interferometric Camera - NIR / Visible Adaptive Interferometer for Astronomy
LLOC	logical lines of code
LMC	Large Magellanic Cloud
LMC	LUCIFER Management Console
LMS	LUCIFER Mask Making Software
LOC	lines of code
LRGS	Luminous Red Galaxy Sample
LSW	Landessternwarte
LUCIFER	LBT NIR Spectroscopic Utility with Camera and Integral Field Unit for Extragalactic Research
LuciferVR	LUCIFER virtual reality
mas	milliarcsecond
MCS	Mounting Control System
MCU	Motion Control Unit
MGIO	Mount Graham International Observatory
MGS	Main Galaxy Sample
MHU	Mask Handling Unit
MIDAS	Munich Image Data Analysis System
MIDI	MID Infrared Interferometer
MIN	minimal set of assumptions
MIR	mid-infrared
MIT	Massachusetts Institute of Technology
MLOC	method lines of code
MODS	Multi-Object Double Spectrographs
MOS	Multi-Object Spectroscopy

MPE	Max-Planck-Institut für Extraterrestrische Physik
MPIA	Max-Planck-Institut für Astronomie
MPIfR	Max-Planck-Institut für Radioastronomie
mutex	mutual exclusion
N	inverse focal ratio
NASA	National Aeronautics and Space Administration
NDIT	Number of Detector Integrations
NGC	New General Catalogue
NIR	near-infrared
NOA	number of attributes
NOAO	National Optical Astronomy Observatory
NOC	number of classes
NOI	number of interfaces
NOM	number of methods
NRM	Non Redundant Masking
NSF	National Science Foundation
NSO	National Solar Observatory
NTP	Network Time Protocol
NTT	New Technology Telescope
OMG	Object Management Group
OO	Object Oriented
OPT	Observation Preparation Tool
OS	Operating System
OSS	Optics Support Structure Control System
PAH	Polycyclic Aromatic Hydrocarbons
PC	Personal Computer
pc	parsec
PCI	Peripheral Component Interconnect
PCS	Pointing Control System
PDE	Pure Density Evolution
PDF	Portable Document Format
PEPSI	Potsdam Echelle Polarimetric and Spectroscopic Instrument
PHP	PHP: Hypertext Pre-processor
PI	Principal Investigator
PLE	Pure Luminosity Evolution
PLOC	physical lines of code
PMC	Primary Mirror Control System
ppm	parts per million
PSF	Point Spread Function
PSFO	Point Spread Function Optimiser
QSO	quasi-stellar object
Quasar	quasi-stellar radio source

R7S	Resources
RAID	Redundant Array of Inexpensive Disks
RAM	Random Access Memory
RC3	Third Reference Catalogue of Bright Galaxies
RCS	Revision Control System
RMI	Remote Method Invocation
RMID	RMI Activation System Daemon
RPC	Remote Procedure Call
SDK	Software Development Kit
SDSS	Sloan Digital Sky Survey
SED	Spectral Energy Distribution
SMT	Heinrich Hertz Submillimeter Telescope
SOA	Service-Oriented Architecture
SPARC	Scalable Processor Architecture
SPIE	Society of Photo-optical Instrumentation Engineers
SQL	Structured Query Language
SUN	Stanford University Network
TCP	Transmission Control Protocol
TCS	Telescope Control Software
TRGB	Tip of the Red Giant Branch
TT	Tip-Tilt
UKIDSS	UKIRT Infrared Deep Sky Survey
ULIRGs	ultra-luminous infrared galaxies
UML	Unified Modelling Language
US	United States
USA	United States of America
UV	ultraviolet
VATT	Vatican Advanced Technology Telescope
VLBI	Very Long Baseline Interferometry
VLT	Very Large Telescope
VLTI	Very Large Telescope Interferometer
XML	Extensible Markup Language

Index

D

E

Bibliography

Adelman-McCarthy, J.K., Agüeros, M.A., Allam, S.S., Allende Prieto, C., Anderson, K.S.J., Anderson, S.F., Annis, J., Bahcall, N.A., Bailer-Jones, C.A.L., Baldry, I.K. et al. **The Sixth Data Release of the Sloan Digital Sky Survey**. **ApJS**, 175:297–313 (2008).

Albrecht, M.A., Brighton, A., Herlin, T., Biereichel, P. and Durand, D. **Access to Data Sources and the ESO SkyCat Tool**. In G.H..H. Payne, editor, **Astronomical Data Analysis Software and Systems VI**, volume 125, pp. 333–+ (1997).

Alhir, S.S. **UML in a Nutshell**. O'Reilly, Sebastopol, CA, USA (1998).

Antonucci, R. **Unified models for active galactic nuclei and quasars**. **ARA&A**, 31:473–521 (1993).

Axelrod, T. **LBT Software System Definition (LBT CAN 481s001d**. Technical report, LBT Observatory, Tucson, AZ, USA (2005).

Balzert, H. **Lehrbuch der Software-Technik**. Spektrum, Akademischer Verlag, Heidelberg, Germany (1998).

Banse, K., Crane, P., Grosbol, P., Middleburg, F., Ounnas, C., Ponz, D. and Waldthausen, H. **MIDAS - ESO's new image processing system**. **The Messenger**, 31:26–28 (1983).

Bauer, C. and King, G. **Java Persistence with Hibernate**. Manning Publications, Greenwich, CT, USA (2006).

Beck, K. **JUnit kurz & gut**. O'Reilly, Cologne, Germany (2005).

Bentley, J.L. **Multidimensional Binary Search Tress Used for Associative Searching**. **Communications of the ACM**, 18(9):509–517 (1975).

Bertin, E. and Arnouts, S. **SExtractor: Software for source extraction. A&AS**, 117:393–404 (1996).

Boehm, B.W. **A Spiral Model of Software Development and Enhancement**. **IEEE Computer**, 21:61–72 (1988).

Bolzonella, M., Miralles, J. and Pelló, R. **Photometric redshifts based on standard SED fitting procedures**. **A&A**, 363:476–492 (2000).

Borelli, J.L. **InfraRed Test Camera User's Manual (LBT CAN 609g012e**. Technical report, LBT Observatory, Tucson, AZ, USA (2008).

Bottinelli, L., Gouguenheim, L., Paturel, G. and de Vaucouleurs, G. **H I line studies of galaxies. III - Distance moduli of 822 disk galaxies**. **A&A**, 56:381–413 (1984).

Boyle, B.J., Shanks, T., Croom, S.M., Smith, R.J., Miller, L., Loaring, N. and Heymans, C. **The 2dF QSO Redshift Survey - I. The optical luminosity function of quasi-stellar objects**. **MNRAS**, 317:1014–1022 (2000).

Bröhl, A.P. and Dröschel, W. **Das V-Modell. Der Standard für die Softwareentwicklung mit Praxisleitfaden**. Oldenbourg, Munich, Germany (1993).

Buschkamp, P., Hofmann, R., Gemperlein, H., Polsterer, K., Ageorges, N., Eisenhauer, F., Lederer, R., Honsberg, M., Haug, M., Eibl, J. et al. **The LUCIFER MOS: a full cryogenic mask handling unit for a near-infrared multi-object spectrograph**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7735 (2010).

Cardamone, C.N., van Dokkum, P.G., Urry, C.M., Taniguchi, Y., Gawiser, E., Brammer, G., Taylor, E., Damen, M., Treister, E., Cobb, B.E. et al. **The Multiwavelength Survey by Yale-Chile (MUSYC): Deep Medium-band Optical Imaging and High-quality 32-band Photometric Redshifts in the ECDF-S**. **ApJS**, 189:270–285 (2010).

Carlberg, R.G. **Quasar evolution via galaxy mergers**. **ApJ**, 350:505–511 (1990).

Carroll, B.W. and Ostlie, D.A. **An Introduction to Modern Astrophysics**. Pearson Education, San Francisco, CA, USA, second edition (2007).

Cattaneo, A., Faber, S.M., Binney, J., Dekel, A., Kormendy, J., Mushotzky, R., Babul, A., Best, P.N., Brüggen, M., Fabian, A.C. et al. **The role of black holes in galaxy formation and evolution**. **Nature**, 460:213–219 (2009).

Cecil, G., Bland-Hawthorn, J., Veilleux, S. and Filippenko, A.V. **Jet- and Wind-driven Ionized Outflows in the Superbubble and Star-forming Disk of NGC 3079**. **ApJ**, 555:338–355 (2001).

Chu, Y.H., Chang, H.W., Su, Y.L. and Mac Low, M.M. **X-Rays from Superbubbles in the Large Magellanic Cloud. III. X-Ray–dim Superbubbles**. **ApJ**, 450:157–+ (1995).

Cristiani, S., Alexander, D.M., Bauer, F., Brandt, W.N., Chatzichristou, E.T., Fontanot, F., Grazian, A., Koekemoer, A., Lucas, R.A., Monaco, P. et al. **The Space Density of High-redshift QSOs in the Great Observatories Origins Deep Survey**. **ApJL**, 600:L119–L122 (2004).

Cromwell, R.H., Haemmerle, V.R. and Woolf, N.J. **Site testing telescope on Mt. Hopkins**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 1236 (1990).

Csabai, I., Budavári, T., Connolly, A.J., Szalay, A.S., Győry, Z., Benítez, N., Annis, J., Brinkmann, J., Eisenstein, D., Fukugita, M. et al. **The Application of Photometric Redshifts to the SDSS Early Data Release**. **AJ**, 125:580–592 (2003).

Dahl, O.J., Myhrhaug, B. and Nygaard, K. **SIMULA 67: common base language**. Technical report, Norwegian Computing Center, Oslo, Norway (1968).

Dasyra, K.M., Tacconi, L.J., Davies, R.I., Naab, T., Genzel, R., Lutz, D., Sturm, E., Baker, A.J., Veilleux, S., Sanders, D.B. et al. **Dynamical Properties of Ultraluminous Infrared Galaxies. II. Traces of Dynamical Evolution and End Products of Local Ultraluminous Mergers**. **ApJ**, 651:835–852 (2006).

de Vaucouleurs, G., de Vaucouleurs, A., Corwin, H.G., Buta, R.J., Paturel, G. and Fouque, P. **Third Reference Cat. of Bright Galaxies (RC3)**. Springer, New York, USA (1991).

Dijkstra, E.W. **Turing Award Lecture: The Humble Programmer** (1972). `http://awards.acm.org/images/awards/140/articles/4860551.pdf`.

Eisenstein, D.J., Annis, J., Gunn, J.E., Szalay, A.S., Connolly, A.J., Nichol, R.C., Bahcall, N.A., Bernardi, M., Burles, S., Castander, F.J. et al. **Spectroscopic Target Selection for the Sloan Digital Sky Survey: The Luminous Red Galaxy Sample**. **AJ**, 122:2267–2280 (2001).

Fan, X., Hennawi, J.F., Richards, G.T., Strauss, M.A., Schneider, D.P., Donley, J.L., Young, J.E., Annis, J., Lin, H., Lampeitl, H. et al. **A Survey of z>5.7 Quasars in the Sloan Digital Sky Survey. III. Discovery of Five Additional Quasars**. **AJ**, 128:515–522 (2004).

Fan, X., Narayanan, V.K., Lupton, R.H., Strauss, M.A., Knapp, G.R., Becker, R.H., White, R.L., Pentericci, L., Leggett, S.K., Haiman, Z. et al. **A Survey of z>5.8 Quasars in the Sloan Digital Sky Survey. I. Discovery of Three New Quasars and the Spatial Density of Luminous Quasars at z≈6**. **AJ**, 122:2833–2849 (2001).

Fan, X., Strauss, M.A., Richards, G.T., Hennawi, J.F., Becker, R.H., White, R.L., Diamond-Stanic, A.M., Donley, J.L., Jiang, L., Kim, J.S. et al. **A Survey of z>5.7 Quasars in the Sloan Digital Sky Survey. IV. Discovery of Seven Additional Quasars**. **AJ**, 131:1203–1209 (2006).

Fan, X., Strauss, M.A., Schneider, D.P., Becker, R.H., White, R.L., Haiman, Z., Gregg, M., Pentericci, L., Grebel, E.K., Narayanan, V.K. et al. **A Survey of z>5.7 Quasars in the Sloan Digital Sky Survey. II. Discovery of Three Additional Quasars at z>6**. **AJ**, 125:1649–1659 (2003).

Fanaroff, B.L. and Riley, J.M. **The morphology of extragalactic radio sources of high and low luminosity**. **MNRAS**, 167:31P–36P (1974).

Frei, Z. and Gunn, J.E. **Generating colors and K corrections from existing catalog data**. **AJ**, 108:1476–1485 (1994).

`ftp://ftp.noao.edu/catalogs/atmospheric_transmission/`. **FTS spectra of atmosphere at NSO/Kitt Peak Observatory** (2009). National Solar Observatory/National Science Foundation.

Genzel, R. and Karas, V. **The Galactic Center**. In **IAU Symposium**, volume 238, pp. 173–180 (2007).

Greenstein, J.L. and Schmidt, M. **The Quasi-Stellar Radio Sources 3c 48 and 3c 273. ApJ**, 140:1–+ (1964).

Grosso, W. **Java RMI**. O'Reilly, Sebastopol, CA, USA (2002).

Haiman, Z. and Hui, L. **Constraining the Lifetime of Quasars from Their Spatial Clustering**. **ApJ**, 547:27–38 (2001).

Haiman, Z., Madau, P. and Loeb, A. **Constraints from the Hubble Deep Field on High-Redshift Quasar Models**. **ApJ**, 514:535–543 (1999).

Harold, E.R. and Means, S.W. **XML in a Nutshell**. O'Reilly, Sebastopol, CA, USA (2004).

Hastie, T., Tibshirani, R. and Friedman, J.H. **The Elements of Statistical Learning: Data Mining, Inference, and Prediction**. Springer, New York, USA, second edition (2009).

Hazard, C. **The method of lunar occultations and its application to a survey of the radio sources 3C 212**. **MNRAS**, 124:343–+ (1962).

Hazard, C., Mackey, M.B. and Shimmins, A.J. **Investigation of the Radio Source 3C 273 By The Method of Lunar Occultations**. **Nature**, 197:1037–1039 (1963).

Herbst, T.M. and Hinz, P.M. **Interferometry on the Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5491, pp. 383–+ (2004).

Herbst, T.M., Ragazzoni, R., Eckart, A. and Weigelt, G. **The LINC-NIRVANA interferometric imager for the Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5492, pp. 1045–1052 (2004).

Herbst, T.M., Ragazzoni, R., Eckart, A. and Weigelt, G. **LINC-NIRVANA: the Fizeau interferometer for the Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7013 (2008).

Hill, J.M., Green, R.F. and Slagle, J.H. **The Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6267 (2006).

Hill, J.M., Green, R.F., Slagle, J.H., Ashby, D.S., Brusa-Zappellini, G., Brynnel, J.G., Cushing, N.J., Little, J. and Wagner, R.M. **The Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7012 (2008).

Hill, J.M. and Salinari, P. **The Large Binocular Telescope project**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5489, pp. 603–614 (2004).

Hinz, P.M., Solheid, E., Durney, O. and Hoffmann, W.F. **NIC: LBTI's nulling and imaging camera**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7013 (2008).

Hofmann, R., Gemperlein, H., Grimm, B., Jütte, M., Mandel, H., Polsterer, K.L. and Weisz, H. **The cryogenic MOS unit for LUCIFER**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5492 (2004).

http://medusa.as.arizona.edu/lbto/. **The LBTO Homepage** (2009). Large Binocular Telescope Observatory.

http://www.aip.de/groups/pepsi/. **The PEPSI Homepage** (2009). Astrophysikalisches Institut Potsdam.

http://www.dwheeler.com/sloc/. **More than a Gigabuck: Estimating GNU/Linux's Size** (2001). David A. Wheeler.

http://www.hibernate.org. **The Hibernate Homepage** (2009). JBoss Enterprise Middleware System Suite/Red Hat.

http://www.v-modell.iabg.de. **The V-Model Homepage** (2009). Industrieanlagen-Betriebsgesellschaft mbH.

Humphreys, R.M., Jones, T.J., Polomski, E., Koppelman, M., Helton, A., McQuinn, K., Gehrz, R.D., Woodward, C.E., Wagner, R.M., Gordon, K. et al. **M33's Variable A: A Hypergiant Star More Than 35 YEARS in Eruption**. **AJ**, 131:2105–2113 (2006).

Jacobson, I., Booch, G. and Rumbaugh, J. **The Unified Software Development Process**. Addison-Wesley, Reading, Ma, USA (1999).

Jütte, M., Polsterer, K.L., Knierim, V., Luks, T., Schimmelmann, J., Muhlack, T., Mandel, H. and Lehmitz, M. **The Java based control software of the LUCIFER instrument**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6274 (2006).

Jütte, M., Polsterer, K.L. and Lehmitz, M. **The Development Process of the LUCIFER Control Software**. In F. Ochsenbein, M.G. Allen and D. Egret, editors, **Astronomical Data Analysis Software and Systems (ADASS) XIII**, volume 314, pp. 712–+ (2004a).

Jütte, M., Polsterer, K.L., Lehmitz, M. and Dettmar, R.J. **LUCIFER control software: an OO approach using CORBA technology**. In H. Lewis, editor, **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 4848, pp. 387–393 (2002).

Jütte, M., Polsterer, K.L., Lehmitz, M. and Knierim, V. **The development process of the LUCIFER control software**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5496, pp. 469–476 (2004b).

Kalas, P., Graham, J.R., Chiang, E., Fitzgerald, M.P., Clampin, M., Kite, E.S., Stapelfeldt, K., Marois, C. and Krist, J. **Optical Images of an Exosolar Planet 25 Light-Years from Earth**. **Science**, 322:1345– (2008).

Karachentsev, I., Musella, I. and Grimaldi, A. **The distance of the isolated Magellanic-type galaxy NGC 1156. A&A**, 310:722–724 (1996).

Karachentseva, V.E. **The Catalogue of Isolated Galaxies**. **Astrofizicheskie Issledovaniia Izvestiya Spetsial'noj Astrofizicheskoj Observatorii**, 8:3–49 (1973).

Kauffmann, G., Heckman, T.M., Tremonti, C., Brinchmann, J., Charlot, S., White, S.D.M., Ridgway, S.E., Brinkmann, J., Fukugita, M., Hall, P.B. et al. **The host galaxies of active galactic nuclei**. **MNRAS**, 346:1055–1077 (2003).

Knierim, V., Jütte, M., Polsterer, K.L. and Schimmelmann, J. **User interaction with the LUCIFER control software**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6274 (2006).

Knierim, V. **The Lucifer Control Software: Operational and Observational Characteristics and NIR Spectroscopy of Galaxy Centers**. Ph.D. thesis, Fakultät für Physik und Astronomie der Ruhr-Universität Bochum (2009).

Kruchten, P. **The Rational Unified Process: An Introduction**. Addison-Wesley, Reading, Ma, USA (1999).

Lasker, B.M., Sturch, C.R., McLean, B.J., Russell, J.L., Jenkner, H. and Shara, M.M. **The Guide Star Catalog. I - Astronomical foundations and image processing**. **AJ**, 99:2019–2058 (1990).

Lee, M.G., Freedman, W.L. and Madore, B.F. **The Tip of the Red Giant Branch as a Distance Indicator for Resolved Galaxies**. **ApJ**, 417:553–+ (1993).

Lehmitz, M. **Temperature control unit**. Technical report, MPIA Heidelberg (2006).

Lehmitz, M. **Calibration unit and heater control unit**. Technical report, MPIA Heidelberg (2008a).

Lehmitz, M. **Calibration unit drive control**. Technical report, MPIA Heidelberg (2008b).

Lehmitz, M. **Communication unit**. Technical report, MPIA Heidelberg (2008c).

Lehmitz, M. **Motion control electronics**. Technical report, MPIA Heidelberg (2008d).

Lehmitz, M. **Pump and ln2 control unit**. Technical report, MPIA Heidelberg (2008e).

Lehmitz, M. **Rack cooling control unit**. Technical report, MPIA Heidelberg (2008f).

Levine, M., Soummer, R., Arenberg, J., Belikov, R., Bierden, P., Boccaletti, A., Brown, R., Burrows, A., Burrows, C., Cady, E. et al. **Overview of Technologies for Direct Optical Imaging of Exoplanets**. **Astronomy**, 2010:37–+ (2009).

Link, J. **Unit Test mit Java**. dpunkt.verlag, Heidelberg, Germany (2003).

Madsen, O.L., Møller-Pedersen, B. and Nygaard, K. **Object-Oriented Programming in the BETA Programming Language**. Addison-Wesley, Reading, Ma, USA (1993).

Mainzer, A.K., Young, E., Hong, J., Hinz, P., Werner, M., Gorjian, V. and Ressler, M.E. **MegaMIR: a Fizeau thermal infrared camera for the LBTI**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6269 (2006).

Mandel, H., Appenzeller, I., Seifert, W., Baumeister, H., Bizenberger, P., Dettmar, R.J., Gemperlein, H., Grimm, B., Herbst, T.M., Polsterer, K.L. et al. **LUCIFER status report, summer 2004**. In A.F.M. Moorwood and M. Iye, editors, **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5492, pp. 1208–1217 (2004).

Mandel, H., Appenzeller, I., Seifert, W., Xu, W., Herbst, T., Lenzen, R., Thatte, N., Eisenhauer, F., Lemke, R., Bomans, D. et al. **LUCIFER - a NIR spectrograph and imager for the LBT.** In **Astronomische Gesellschaft Abstract Series**, volume 15, pp. 144–+ (1999).

Mandel, H., Seifert, W., Hofmann, R., Jütte, M., Lenzen, R., Ageorges, N., Bomans, D., Buschkamp, P., Dettmar, R.J., Polsterer, K.L. et al. **LUCIFER status report: summer 2008**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7014 (2008).

Mandel, H., Seifert, W., Lenzen, R., Hofmann, R., Jütte, M., Weiser, P., Appenzeller, I., Bomans, D., Buschkamp, P., Polsterer, K.L. et al. **LUCIFER: a NIR Spectrograph and Imager for the LBT**. **Astronomische Nachrichten**, 328:626–+ (2007).

Mandel, H.G., Appenzeller, I., Seifert, W., Baumeister, H., Dettmar, R.J., Feiz, C., Gemperlein, H., Germeroth, A., Grimm, B., Polsterer, K.L. et al. **LUCIFER status report: Summer 2006**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6269 (2006).

Marigo, P., Girardi, L., Bressan, A., Groenewegen, M.A.T., Silva, L. and Granato, G.L. **Evolution of asymptotic giant branch stars. II. Optical to far-infrared isochrones with improved TP-AGB models**. **A&A**, 482:883–905 (2008).

Martin, H.M., Cuerden, B., Dettmann, L.R. and Hill, J.M. **Active optics and force optimization for the first 8.4-m LBT mirror**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 5489, pp. 826–837 (2004).

Martin, R.C. **OO Design Quality Metrics: An Analysis of Dependencies** (1994).

McLean, I. **Electronic Imaging in Astronomy: Detectors and Instrumentation**. John Wiley & Sons, Chichester, England (1997).

Minchin, R.F., Momjian, E., Auld, R., Davies, J.I., Valls-Gabaud, D., Karachentsev, I.D., Henning, P.A., O'Neil, K.L., Schneider, S., Smith, M.W.L. et al. **The Arecibo Galaxy Environment Survey. III. Observations**

Toward the Galaxy Pair NGC 7332/7339 and the Isolated Galaxy NGC 1156. **AJ**, 140:1093–1118 (2010).

Monaco, P., Salucci, P. and Danese, L. **Joint cosmological formation of QSOs and bulge-dominated galaxies**. **MNRAS**, 311:279–296 (2000).

Mortlock, D.J., Patel, M., Warren, S.J., Hewett, P.C., Venemans, B.P., McMahon, R.G. and Simpson, C.J. **Probabilistic selection of high-redshift quasars**. **ArXiv e-prints** (2011).

Muhlack, T. **Der LUCIFER NIR Spektrograph: Detektor und Ausleseprozess** (2006).

Neuhäuser, R., Mugrauer, M., Fukagawa, M., Torres, G. and Schmidt, T. **Direct detection of exoplanet host star companion γ Cep B and revised masses for both stars and the sub-stellar object**. **A&A**, 462:777–780 (2007).

Oestreicher, M.O., Schmidt-Kaler, T. and Wargau, W. **Red supergiants in the LMC - I. BVRIJHK photometry, magnitudes and intrinsic colours**. **MNRAS**, 289:729–744 (1997).

O'Mill, A.L., Duplancic, F., García Lambas, D. and Sodré, Jr., L. **Photometric redshifts and k-corrections for the Sloan Digital Sky Survey Data Release 7**. **MNRAS**, pp. 201–+ (2011).

Page, T. **QSO's, the Brightest Things in the Universe (Quasi-Stellar Objects)**. **ASPL**, 9:161–+ (1964).

Perry, W.E. **Effective Methods for Software Testing**. John Wiley & Sons, New York, NY, USA (2006).

Pilone, D. and Pitman, N. **UML 2.0 in a Nutshell**. O'Reilly, Sebastopol, CA, USA (2005).

Pogge, R.W., Atwood, B., Belville, S.R., Brewer, D.F., Byard, P.L., DePoy, D.L., Derwent, M.A., Eastwood, J., Gonzalez, R., Krygier, A. et al. **The multi-object double spectrographs for the Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6269 (2006).

Polsterer, K.L., Jütte, M., Knierim, V., Lehmitz, M. and Mandel, H. **Lucifer VR: a virtual instrument for the LBT**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6274 (2006).

Polsterer, K. **Entwicklung und Visualisierung eines virtuellen astronomischen Instruments** (2003).

Pomberger, G. and Blaschek, G. **Software Engineering**. Carl Hanser, Munich, Germany (1993).

Purdy, G.N. **CVS kurz & gut**. O'Reilly, Cologne, Germany (2001).

Riccardi, A., Brusa, G., Salinari, P., Busoni, S., Lardiere, O., Ranfagni, P., Gallieni, D., Biasi, R., Andrighettoni, M., Miller, S. et al. **Adaptive secondary mirrors for the Large binocular telescope**. In **Society of Photo-Optical**

Instrumentation Engineers (SPIE) Conference Series, volume 5169, pp. 159–168 (2003).

Riccardi, A., Xompero, M., Zanotti, D., Busoni, L., Del Vecchio, C., Salinari, P., Ranfagni, P., Brusa Zappellini, G., Biasi, R., Andrighettoni, M. et al. **The adaptive secondary mirror for the Large Binocular Telescope: results of acceptance laboratory test**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7015 (2008).

Richards, G.T., Fan, X., Newberg, H.J., Strauss, M.A., Vanden Berk, D.E., Schneider, D.P., Yanny, B., Boucher, A., Burles, S., Frieman, J.A. et al. **Spectroscopic Target Selection in the Sloan Digital Sky Survey: The Quasar Sample**. **AJ**, 123:2945–2975 (2002).

Royce, W. **Managing the Development in Large Software Systems**. In **Proceedings of IEEE WESCON** (1970).

Schimmelmann, J. **Planung und Durchführung astronomischer nah-infrarot Beobachtungen unter Berücksichtigung spezieller Optimierungsprobleme des LUCIFER Instruments** (2007).

Schlegel, D.J., Finkbeiner, D.P. and Davis, M. **Maps of Dust Infrared Emission for Use in Estimation of Reddening and Cosmic Microwave Background Radiation Foregrounds**. **ApJ**, 500:525–+ (1998).

Schneider, D.P., Richards, G.T., Hall, P.B., Strauss, M.A., Anderson, S.F., Boroson, T.A., Ross, N.P., Shen, Y., Brandt, W.N., Fan, X. et al. **The Sloan Digital Sky Survey Quasar Catalog. V. Seventh Data Release**. **AJ**, 139:2360–2373 (2010).

Seifert, W., Appenzeller, I., Baumeister, H., Bizenberger, P., Bomans, D., Dettmar, R.J., Grimm, B., Herbst, T., Hofmann, R., Polsterer, K.L. et al. **LUCIFER: a Multi-Mode NIR Instrument for the LBT**. In M. Iye and A.F.M. Moorwood, editors, **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 4841, pp. 962–973 (2003).

Seifert, W. and Xu, W. **LUCIFER, Final Design Report Optics**. Technical report, Landessternwarte, Heidelberg, Germany (2002).

Shaw, M. and Clements, P. **The Golden Age of Software Architecture**. **IEEE Software**, pp. 31–39 (2006).

Shaw, M. and Garlan, D. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice Hall, Upper Saddle River, NJ, USA (1996).

Sivaramakrishnan, A., Tuthill, P., Martinache, F., Ireland, M., Lloyd, J.P., Perrin, M.D., Soummer, R., McKernan, B. and Ford, S. **Planetary system, star formation, and black hole science with non-redundant masking on space telescopes**. **Astronomy**, 2010:40–+ (2009).

Skrutskie, M.F., Cutri, R.M., Stiening, R., Weinberg, M.D., Schneider, S., Carpenter, J.M., Beichman, C., Capps, R., Chester, T., Elias, J. et al. **The Two Micron All Sky Survey (2MASS)**. **AJ**, 131:1163–1183 (2006).

Speziali, R., Di Paola, A., Giallongo, E., Pedichini, F., Ragazzoni, R., Testa, V., Baruffolo, A., De Santis, C., Diolaiti, E., Farinato, J. et al. **The Large Binocular Camera: description and performances of the first binocular imager**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7014 (2008).

Storz, C. **MIDI Detector Control Software (VLT-TRE-MID-15823-0129)**. Technical report, MPIA, Heidelberg, Germany (2001).

Strassmeier, K.G., Hofmann, A., Woche, M.F., Rice, J.B., Keller, C.U., Piskunov, N.E. and Pallavicini, R. **PEPSI spectro-polarimeter for the LBT**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 4843, pp. 180–189 (2003).

Strauss, M.A., Weinberg, D.H., Lupton, R.H., Narayanan, V.K., Annis, J., Bernardi, M., Blanton, M., Burles, S., Connolly, A.J., Dalcanton, J. et al. **Spectroscopic Target Selection in the Sloan Digital Sky Survey: The Main Galaxy Sample**. **AJ**, 124:1810–1824 (2002).

Stroustrup, B. **The C++ Programming Language**. Addison-Wesley, Reading, Ma, USA (1986).

Tate, B.A. and Gehtland, J. **Better, Faster, Lighter Java**. O'Reilly, Sebastopol, CA, USA (2004).

Terrett, D.L. **Pointing algorithms for binocular telescopes**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 6274 (2006).

Tilly, J. and Burke, E.M. **Ant: The Definitive Guide**. O'Reilly, Sebastopol, CA, USA (2002).

Veilleux, S., Rupke, D.S.N. and Swaters, R. **Warm Molecular Hydrogen in the Galactic Wind of M82**. **ApJL**, 700:L149–L153 (2009).

Wagner, R.M. **The Multi-Object Double Spectrograph and Nulling Interferometer for the Large Binocular Telescope**. **Astronomische Nachrichten**, 328:631–+ (2007).

Wagner, R.M. **An overview of instrumentation for the Large Binocular Telescope**. In **Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series**, volume 7014 (2008).

Watson, F.G. **The zenithal blind SPOT of a computer-controlled altazimuth telescope**. **Monthly Notices of the Royal Astronomical Society**, 183:277–284 (1978).

White, S.D.M. and Frenk, C.S. **Galaxy formation through hierarchical clustering**. **ApJ**, 379:52–79 (1991).

Wilson, R.N. **The NTT (New Technology Telescope) - Predecessor of the VLT**. In **European Southern Observatory Astrophysics Symposia**, volume 17 (1983).

Wu, X. and Jia, Z. **Quasar candidate selection and photometric redshift estimation based on SDSS and UKIDSS data**. **MNRAS**, 406:1583–1594 (2010).

York, D.G., Adelman, J., Anderson, Jr., J.E., Anderson, S.F., Annis, J., Bahcall, N.A., Bakken, J.A., Barkhouser, R., Bastian, S., Berman, E. et al. **The Sloan Digital Sky Survey: Technical Summary**. **AJ**, 120:1579–1587 (2000).

Zeh, A. **Entwicklung einer Steuerungssoftware zum Betrieb eines Motorcontroller eines astronomischen Instruments** (2005).

Zuser, W., Grechening, T. and Köhle, M. **Software Engineering**. Pearson Education, Munich, Germany, second edition (2004).

Curriculum Vitae

Personal Details:	
Name:	Kai Lars Polsterer
Date/Place of birth:	1976 / Hamm(Westf.)
Education:	
1982-1986	GGS-Lierberg, Mülheim a.d. Ruhr
1986-1995	Elsa-Brändström-Gymnasium, Oberhausen
1996-2003	Study of Computer Science at the TU Dortmund
06/2002 - 06/2003	Diploma thesis at Lehrstuhl 1, TU Dortmund and *Astronomisches Institut der Ruhr-Universität Bochum*. Title: “Entwicklung und Visualisierung eines virtuellen astronomischen Instruments”
06/2003	Diploma in Computer Science, grade A (with distinction)
since 2003	PhD thesis at the *AIRUB*
Conferences:	
10/2003	*ADASS* XIII, Strasbourg, France
06/2004	*SPIE* Astronomical Instrumentation, Glasgow, Scotland
05/2006	*SPIE* Astronomical Instrumentation, Orlando, USA
09/2007	*AG*-Meeting on Cosmic Matter, Würzburg, Germany
06/2010	AstroInformatics 2010, Pasadena, USA
09/2010	*AG*-Meeting the Cosmos at High Resolution, Bonn, Germany
09/2011	*AG*-Meeting Surveys & Simulations, Heidelberg, Germany
Working Stays:	
2004 - 2010	*MPE* (Garching) 12× 3-5 days
2003 - 2011	*LSW* (Heidelberg) 7× 2-4 days and several one-day visits
05/2005	Steward Observatory and *LBT* (Arizona, USA) (3 weeks)
08/2008	*LBT* (Arizona, USA) (5 weeks)
Grants:	
11/2003 - 10/2006	Dissertation grant by the *Klaus Tschira Stiftung (KTS)*