

University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, &
Professional Papers

Graduate School

1997

Space VLBI user assistance software: an object-oriented design implemented in Java

Istvá?n Noszticzius

The University of Montana

Follow this and additional works at: <https://scholarworks.umt.edu/etd>

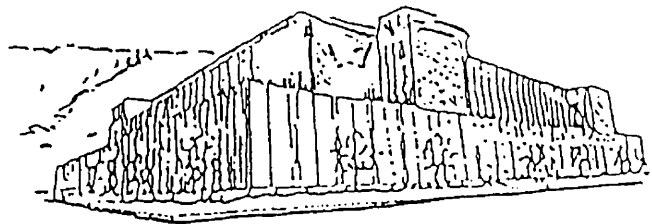
Let us know how access to this document benefits you.

Recommended Citation

Noszticzius, Istvá?n, "Space VLBI user assistance software: an object-oriented design implemented in Java" (1997). *Graduate Student Theses, Dissertations, & Professional Papers*. 8330.

<https://scholarworks.umt.edu/etd/8330>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike
MANSFIELD LIBRARY

The University of **MONTANA**

Permission is granted by the author to reproduce this material in its entirety,
provided that this material is used for scholarly purposes and is properly cited in
published works and reports.

*** Please check "Yes" or "No" and provide signature ***

Yes, I grant permission

No, I do not grant permission

Author's Signature Isiah Wondrus

Date 1997 / July / 31

Any copying for commercial purposes or financial gain may be undertaken only with
the author's explicit consent.

Space VLBI User assistance software
An Object-Oriented design
implemented in Java

by

István Noszticzius, B.S.

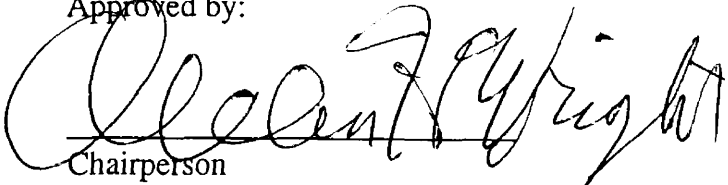
presented in partial fulfillment of the requirements
for the degree of

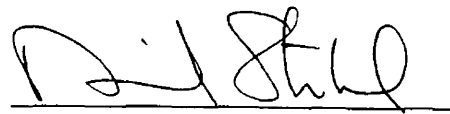
Master of Science

The University of Montana

1997

Approved by:


Chairperson


Dean, Graduate School

8-1-97
Date

UMI Number: EP39131

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP39131

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Space VLBI User Assistance Software: An Object Oriented Design and Implementation in Java
(64 pp.)

Director: Dr. Alden Wright



Very long baseline interferometry (VLBI) is a radio astronomical observing technique that achieves high angular resolution by means of radio telescopes separated by many thousands of kilometers on the surface of the Earth. This resolution can be further enhanced by using satellites for even longer baselines. The geometry of such observations is very complicated due to the orbiting telescope and its constraints that also have to be taken into account with the ground based network.

A simulation software called SPAS (Space VLBI User Assistance Software) was developed for visualizing such observations at the Satellite Geodetic Observatory, Hungary. This software runs on DOS machines only, so the need arose for a version that can be run on other operating systems too. Since the original software is highly complicated with several modules only a partial design and implementation was feasible.

Emphasis was placed on the design of this new SPAS - called JavaSPAS, since it is implemented in Java - which was carried out in an object oriented fashion with the design artifacts generated according to the Booch-method. The architectural design is based on four different class categories - collections of classes - that represent elementary blocks of the software. These include basic astronomical classes, the observational elements and parameters, the main modules and the graphical user interface. On the lower level, the detailed design shows the structure of these categories and also how they behave and interact to produce the simulation output. At the end a very detailed interface of each important class is shown in one of the appendices.

During the implementation part a couple of questions were raised concerning Java, so comments and insights are given on different aspects of the language, in a more general overview and in relevant detail too, including precision requirements, multi-threaded execution and visibility of class variables and methods.

Table of Contents

1. Background	5
1.1 Brief introduction to space VLBI.....	5
1.2 The role of a support software	7
2. The original SPAS.....	8
2.1 Structure and modules	8
2.2 Software engineering - design and implementation.....	9
3. JavaSPAS - an Object Oriented Design	11
3.1 Overview.....	11
3.2 Architectural Design	11
3.3 Detailed Design.....	15
3.3.1 “BasicData” classes	15
3.3.2 “Observation” classes.....	18
3.3.3 “GUI” (Graphical User Interface) Classes	20
3.3.4 “Modules” Classes	25
3.3.5 Scenarios	28
4. JavaSPAS - implementation.....	33
4.1 Why Java.....	33
4.2 Important Java Features	34
4.3 Implementation Details.....	36
4.3.1 How to physically store the code.....	37
4.3.2 Optimizations	38
4.3.3 Other language dependent features	39
5. Appendices	42
A. Pseudocode class definitions & source code	42
B. JavaSPAS database text file format	48
B.1 LEX-type lexical specification	48
B.2 YACC-type grammar specification	49
B.3 A simple example database file	51
C. JavaSPAS setup text file format	53
C.1 LEX-type lexical specification	53
C.2 YACC-type grammar specification	54
C.3 A simple example setup file	55
D. The Booch method artifacts	56
E. Explanation of satellite constraints and their encoding	58
F. Glossary	62
G. Bibliography.....	64

List of Tables

- Table 1* – The main characteristics of currently planned space VLBI satellites (page 6)
Table 2 – Java primitive Data Types (page 35)
Table 3 – Java visibility modifiers and their effect (page 40)
Table E.1 – RadioAstron constraint list (page 58)
Table E.2 – VSOP constraint list (page 58)
Table E.3 – Simplified RadioAstron constraint list (page 61)

List of illustrations

- Figure 1* – Scheme of the space VLBI experiment (Page 6)
Figure 2 – The evolutionary development approach (Page 10)
Figure 3 – JavaSPAS class categories (Page 13)
Figure 4 – JavaSPAS system state transition diagram (Page 14)
Figure 5 – *BasicData* class category inheritance and structure diagram (Page 16)
Figure 6 – Alternative *Position* class hierarchy (Page 17)
Figure 7 – *Observation* class category inheritance diagram (Page 18)
Figure 8 – *Observation* class category structure diagram (Page 19)
Figure 9 – Graphical User Interface class category inheritance diagram (Page 21)
Figure 10 – *SetupDialog* structure design (Page 22)
Figure 11 – *StationSelectDialog* structure design (Page 23)
Figure 12 – *MainWindow* structure design (page 23)
Figure 13 – *SimulationWindow* structure design (page 24)
Figure 14 – *SimulationWindow* state transition diagram (page 24)
Figure 15 – *Modules* class category inheritance diagram (page 25)
Figure 16 – *Modules* class category structure diagram (page 26)
Figure 17 – *UVPlot* state transition diagram (page 27)
Figure 18 – *3DView* state transition diagram (page 28)
Figure 19 – System startup and closing interaction diagram (page 29)
Figure 20 – An opening of a *SimulationWindow*: object diagram (page 30)
Figure 21 – Simulation setup-start-stop interaction diagram (Page 31)
Figure 22 – *UpdateActiveStation()* object diagram (Page 32)
Figure 23 – JavaSPAS files hierarchy (Page 37)

1. Background

1.1 Brief introduction to space VLBI

Very long baseline interferometry (VLBI) is a radio astronomical observing technique that achieves high angular resolution by means of radio telescopes separated by many thousands of kilometers on the surface of the Earth. The signals from artificial or natural radio sources are simultaneously recorded at the telescope sites and later cross-correlated at a central processing facility. In the last 20 years VLBI has been very successful in high resolution imaging of radio sources, radio source position measurements and in determination of geodetic parameters related to Earth rotation and crustal movements. The angular resolution achieved by this technique is presently superior to any other method of astronomical observation.

The resolution of the Earth-based VLBI, however, is limited by the physical dimension of the Earth at any given wavelength. Therefore it has been proposed that the angular resolution be further increased by putting radio telescopes into space. Thus the concept of space VLBI was born.

A schematic diagram of a space VLBI system is illustrated on *Figure 1*. It consists of four basic components:

- the space radio telescope(s),
- the telemetry and control stations,
- ground radio telescope(s),
- processing facility.

Note that the first two points are the concerns of the space agencies, while the last two points are technically covered by the ground based VLBI facilities. Thus in order to operate space VLBI, a very complicated interaction (and understanding) between the two groups is necessary.

Currently two dedicated space VLBI satellites are in an advanced stage of development, for being used in the second half of this decade with radio telescopes of 8-10 m diameter. The VSOP satellite is being developed in Japan, and has already been launched in February of 1997, and currently is undergoing different testing procedures before real observations can begin. The RadioAstron satellite is being constructed by the Russian Space Agency in international collaboration with seven other countries or agencies. The main characteristics are given in *Table 1*. The primary goal of these missions is astrophysical.

Many more space VLBI satellites are under consideration with improved characteristics to be realized after the turn of the century.

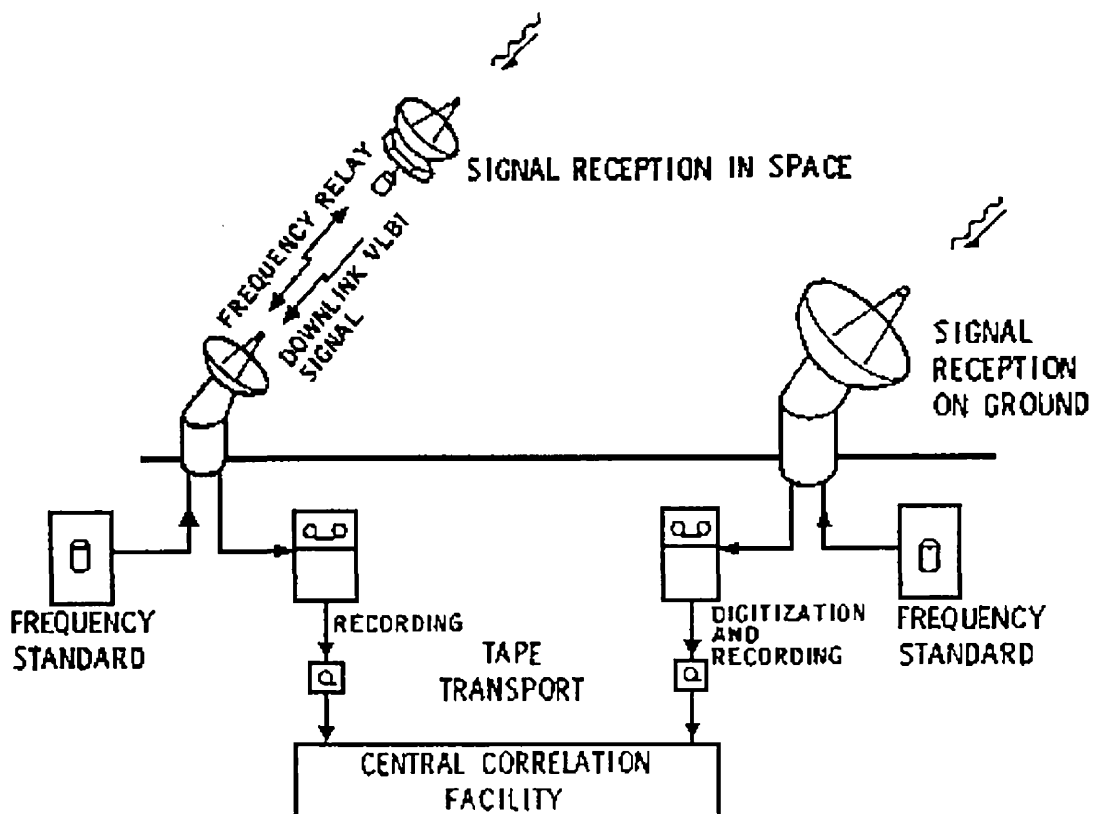


Figure 1
Scheme of the space VLBI experiment.

Space VLBI experiments	VSOP	RadioAstron
Country	Japan	Russia + 7 countries
Launch date	1997	1998 ?
Expected lifetime	5 years	3 years
Height at apogee	20 000 km	75 000 km
Maximum baseline length	30 000 km	85 000 km
Observing frequencies	22 GHz, 5 GHz, 1.6 GHz	22 GHz, 5 GHz, 1.6 GHz, 0.3 GHz
Maximum angular resolution	90 micro arc sec	30 micro arc sec
Telescope diameter	8 m	10 m
Sensitivity	40 mJy	20 mJy

Table 1
The main characteristics of currently planned space VLBI satellites.

1.2 The role of a support software

Space VLBI is a new technique which will be open for a wide community of users. The need for space VLBI assistance software is based on the fact that space VLBI observations are more complicated geometrically and operationally than ground based VLBI. Moreover they are also one or two orders of magnitude more expensive, which justify a very careful preparation, optimization and care for the details.

The complications are due to the addition of an orbiting radio telescope to the network of ground radio telescopes. Consequently the station network geometry is changing fast and continuously in both the terrestrial and celestial reference frame. Observational limitations should be taken into account due to the orbit geometry and technical constraints of the satellite. Although the orbit can be considered fixed in the celestial frame for a short period of time, it may change considerably on the longer term, due to perturbations. Therefore the optimal periods for observing a given source depend on this "orbit evolution". Space VLBI also requires a complicated scheme of tracking, telemetry and control operations. Because there are no hydrogen masers on the satellites, the on-board oscillators are slaved to ground-based frequency standards via two way link from the telemetry stations. The VLBI data from the satellite, during one observing session, may be recorded on magnetic tapes in short sections at several telemetry stations, each with different clock characteristics.

2. The Original SPAS

2.1 Structure and modules

SPAS is a software developed at the Satellite Geodetic Observatory, Hungary, by a dedicated team of software engineers and astronomers. The program runs on IBM PC compatibles under MS-DOS. The software visualizes the complex geometry of space VLBI experiments, checks the technical restrictions of the satellites and ground based equipment. It is useful for approximate scheduling of space VLBI observations with the satellites in preparation (VSOP and RadioAstron). SPAS is used for optimization of ground based tracking and contributing radio telescope networks and also to design future space VLBI experiments. Some operations of the software are also applicable for other space related simulations and demonstrations.

- **SubSat** plots the subsatellite tracks and locations of telemetry stations on Earth's map.
- **SatVis** displays and lists the visibility schedule of satellites from tracking stations. It gives the tracking time coverage of satellites from telemetry stations.
- **ObsInt** displays and lists the observability schedule of radio source from ground VLBI stations and space VLBI satellites. It gives the time coverage of observations.
- **TopoPos** plots tracks of radio source and satellites on the sky in horizontal system as seen from ground stations. It lists the azimuth and elevation values.
- **UVPlot** produces UV-plot for selected baselines.
- **3DView** shows a 3-dimensional movie of the rotating Earth, the ground VLBI stations and the satellites as seen from the direction of radio source.
- **SkySurv** creates a survey for a specified area of the sky deciding if certain parts are or are not observable from ground VLBI stations and satellites at a given time.
- **Access** creates a long term survey for a sky area. It shows how long the certain parts are observable from ground VLBI stations and satellites during a time interval.
- **CheckArc** gives information about the possible attitudes of space VLBI satellite when observing a selected radio source during a time interval. It takes the on-board observing constraints into account.
- **Beam** selects radio sources within the primary beam of space VLBI antenna for phase referencing purposes.
- **Sensitivity** plots the RMS sensitivity of baselines as a function of integration time. It plots the minimum detectable brightness temperature and the signal-to-noise ratio for the baselines as a function of time.

- **Cone** selects radio sources in special geometric configurations with respect to the satellite orbit normal.
- **Calculator** performs basic calculations, unit conversions and coordinate transformations.

Note that in JavaSPAS my goals were to only design and implement two of these modules: notably UVPlot and 3DView. While 3DView gives an overview of a Space VLBI experiment, UVPlot is one of the most common tools used in VLBI radio astronomy.

2.2 Software engineering - Design and implementation

There is really no such thing as “pure” software engineering. Software engineering is an applied science: One must get involved in a field that the software is being written for. This usually includes interacting with the people in that field, trying to understand the problem and translating it into code in an orderly fashion.

The original SPAS package was developed in the Pascal language, following strict software engineering standards set up by the European Space Agency (ESA) [1]. This standard actually describes a “framework” for software development, with some options to choose from concerning the approach to the software life-cycle. The so called “evolutionary” software life-cycle (see *Figure 2*), chosen for SPAS is comparable to the “spiral-model” of development [7], because multiple releases for a program can be created by refining each phase in the model as necessary.

There are several reasons to use these approaches, for example (taken from [1]):

- Some user experience is required to refine and complete the requirements (shown by dashed lines inside OM boxes in *Figure 2*)
- Some parts of the implementation may depend on availability of future technology
- Some new user requirements are anticipated but not yet known
- Some requirements may be significantly more difficult to meet than others, and it is decided not to allow them to delay a usable delivery.

The five basic phases in the ESA software life-cycle are:

- User Requirements (UR): This is the “problem definition” phase for the project [2].
- Software Requirements (SR): This is the “analysis” phase for the project [3].
- Architectural Design (AR): Define the architecture of the software [4].
- Detailed Design and Production (DD): Define the detailed design and code the software [5].
- Transfer (TR): Acceptance (installation of the software)

After completing these phases an operations and maintenance (OM) phase is entered, during which the software is carefully monitored to confirm that it meets all the requirements specified.

During and after these phases different levels of testing are to be conducted (unit test, integration tests, system tests and acceptance tests.), in the case of SPAS some were done against other software to assure the validity of the calculations and simulations.

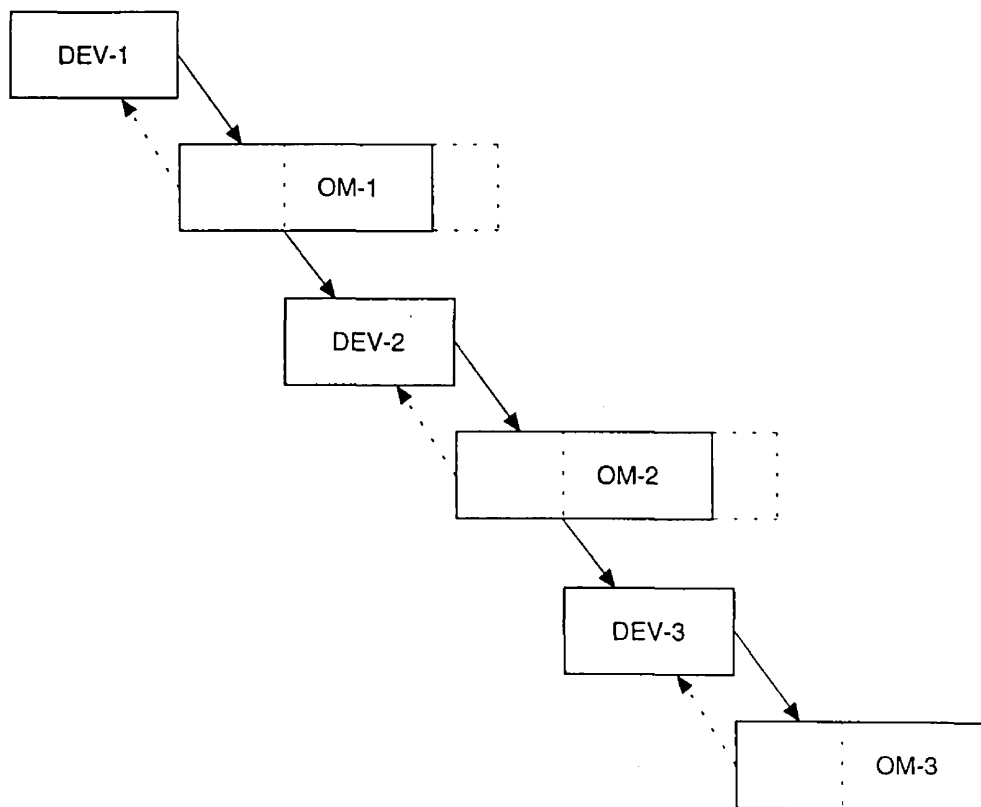


Figure 2 The evolutionary development approach

DEV box is equivalent to UR/SR/AD/DD and TR phases

dashed lines inside OM boxes represent user experience required to refine and complete previous DEV phase, while dashed lines outside the OM boxes represent experience on which next DEV phase can be based (from [1])

3. JavaSPAS - an Object Oriented Design

3.1 Overview

Since it would have obviously been beyond my resources to do all five phases of the ESA software life-cycle (see chapter 2.2) - and those have been already done - I tried to concentrate on a object-oriented version of the Architectural and Detailed design parts. The design incorporates the two planned simulation modules: *UVPlot* and *3DView*, but due to its modularity it would be easily extendible to other ones as necessary. For the design I decided to use the “Booch method” [8], which seems to have notation for most of the “scenarios” that I came across in planning my object oriented version of SPAS. Not all types of the diagrams are used, since the Booch method tries to be prepared for all types of design-cases, not all of which are applicable to SPAS. One such omitted diagram type is the Process Diagram, which is used to show of process/processor allocation in a physical design, not relevant in this case. Another one is the Module Diagrams which are used to show the allocation of classes and objects to modules in the physical design of the system - in the case of JavaSPAS this the same that is shown in the class category diagram (*Figure 3*). On the other hand one thing the Booch notation seems to lack is the notion of classes of which there can be only one object (such as *Database*) - so those are marked with a *.

While the architectural design is an overview of the basic structure, giving the high level building block categories needed for the program, the detailed design is a more thorough examination of how those categories are built and how objects in them interact with one another. To avoid congestion only the relevant features of classes are shown in each diagram - that is, the ones which help one to understand a given part of the design. A full declaration of each class can be found in Appendix A.

Also note that some building block elements are not necessarily used, but they are included in the design to provide a full version which can be extended to more higher level modules (based on the original SPAS module structure).

3.2 Architectural Design

In the architectural (high level) design I divided up the components of JavaSPAS into four basic categories: *BasicData*, *Observation*, *GUI* (Graphical User Interface) and *Modules*. The hierarchy of these “class categories” with the their contained classes is shown in *Figure 3*.

The purpose of these basic categories are as follows:

Modules: This is category of “control classes” . Most of the objects of these classes are active, meaning that they have their own thread of control. *Spas* is the main class - it initializes and starts all activity, including the bringing up of the user interface (*MainWindow*), which in turn then handles events which start other processes (like simulations). Each simulation can have its own setup too, thus there can be more than one simulation of the same type running with different configurations.

Gui: The Graphical User Interface elements of *Spas*. Note that most of these classes can be built on classes available in the Java AWT (Abstract Windowing Toolkit), thus they do not have to be fully designed and implemented from scratch.

These graphical elements interact with other class categories, by accepting things to display (e.g. simulation display in a window) and/or by modifying contents of other classes on input (e.g. modifying an observation setup through a setup window).

Observation: Elements (objects) that make up and contribute to an observation session. These include satellites, stations, the source, etc. A given configuration of all these is stored in a *ObservationSetup* object, which can be modified, loaded/saved and assigned to different simulations.

BasicData: Basic (mostly astronomical) data types which make up the contents of objects taking part in an observation. These data types include the calculations necessary to convert them from one to another (if possible). As a design decision these conversion routines were not put in a separate (utility) class but are embedded in the classes themselves. Nevertheless because of the need for general mathematical routines a separate *Calculator* class is added.

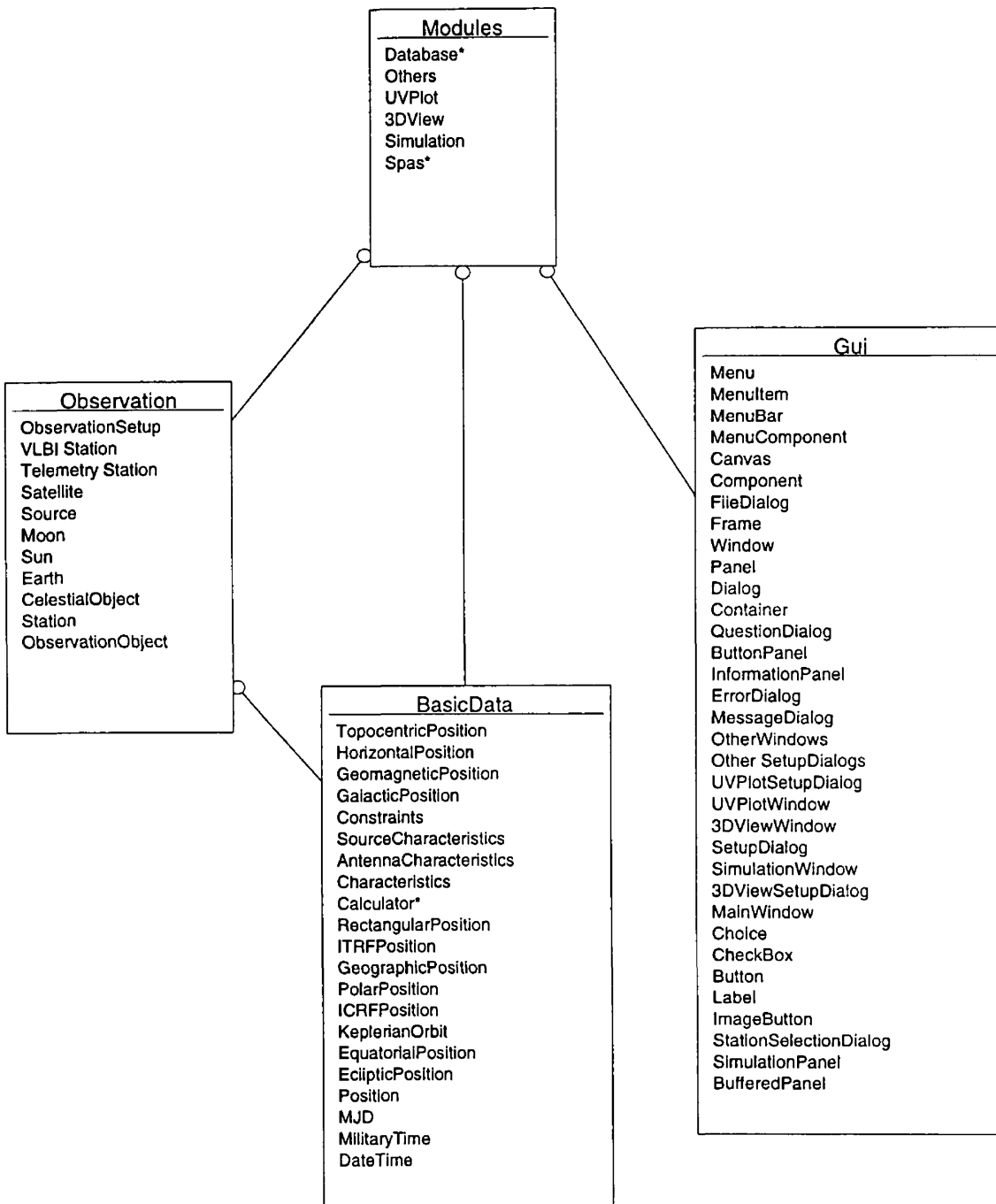


Figure 3
JavaSPAS class categories
(High level module diagram)

A state transition diagram (*Figure 4*) describes how the program works at the system level, which is essentially equivalent to the *MainWindow* class behavior. Other class level state transition diagrams for classes that exhibit interesting event-ordered behavior can be found in the detailed design. Not shown in these diagrams for clarity reasons are “error” events which are handled in “error states”, since nearly all

states can have them. The behavior described by the system state transition diagram shows that each module can be run independently of the main one, or even more than one instance of each module can be started. That is the user could run two *3DView* modules and have them execute at the same time.

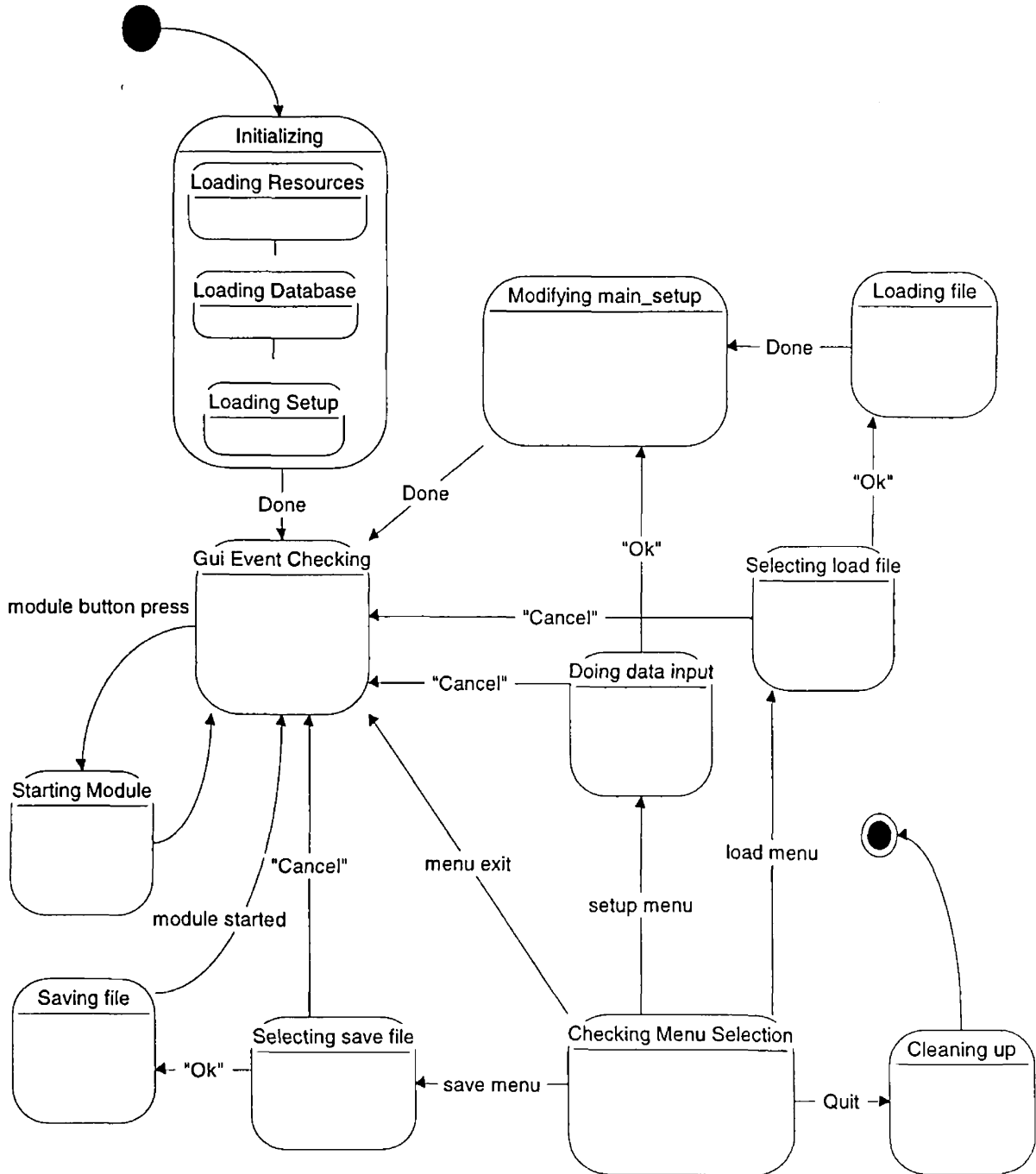


Figure 4
JavaSPAS system state transition diagram

3.3 Detailed Design

The Detailed Design is where all the relationships between the classes in each category are defined from a number of viewpoints (**inheritance, structure**) in class diagrams. In the case of the Graphical User Interface the structure is described via prototype design of the on-screen appearance. Explanation of how classes work (**behavior**) is done with state transition diagrams for classes that exhibit such interesting event oriented behavior, while some important and/or interesting **scenarios** are described with interaction and/or object diagrams.

3.3.1 “BasicData” classes

Figure 5 shows the inheritance hierarchy of the *BasicData* class category and the structure of some of its classes. These classes are the low-level building blocks, from which other components (notably the *Observation* classes) create more advanced, and higher level class types. All classes include a conversion routine which shows the given object state in a string (human readable) format, which is essential for debugging purposes, as well as saving object states into different text files (observation setup, database). The main classes in this category are:

- *Position (abstract) class*. This is a parent class of a fundamental hierarchy, since everything taking part in an observation has some kind of position. These include different systems (e.g. ICRF/ITRF) most of which can be converted to other types, where the conversion routines are embedded into a given class. Some conversion routines need other data types as input (e.g. a *DateTime* class) due to the nature of these position types (some celestial based some terrestrial based). This hierarchy is centered on the way data is represented in each type of position from a calculation point of view. For example both ICRF and ITRF positions have the same parent class (Rectangular), since both are traditionally represented with X, Y and Z coordinates, while geographical and ecliptic positions are both represented with longitude and latitude thus they also have the same parent class (Polar). An alternative approach might be to create a hierarchy, where position data classes are solely structured according to whether they are terrestrial or celestial (*Figure 6*) or even one that combines both hierarchies through multiple inheritance (although this latter one would be quite messy, and in my opinion the problems created by it would outweigh the benefits gained).

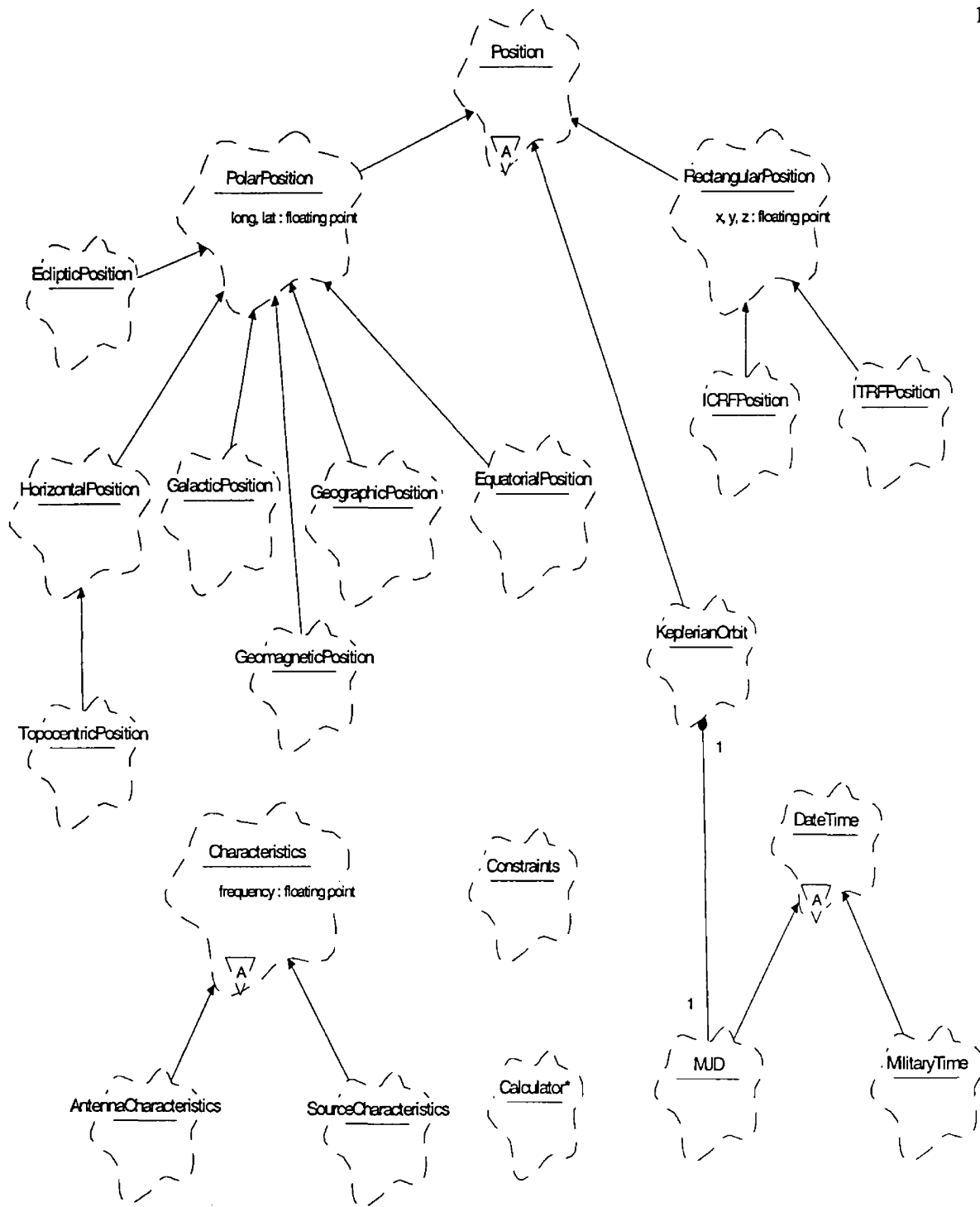


Figure 5
BasicData class category inheritance and structure diagram

- *DateTime (abstract) class.* Subclasses encapsulate the notion of different date and time formats, as well as the routines to convert between them. *MJD* is the Modified Julian Date (see the entry in Appendix F - glossary), which is a very convenient way of calculating time and date differences. Also most astronomical calculations are tailored to the Modified Julian Date or Julian Date. *MilitaryTime* is the way the user inputs time data to the program, i. e. as a 24 hour clock with calendar date.
- *Characteristics class.* Describes various characteristics of objects on different frequencies. For example sources have different flux densities on different frequencies, while antennas (stations or satellites) list their sensitivity values on different frequencies. Thus, usually more than one object of this type is instantiated when used.
- *Constraints class.* A class encapsulating the encoded version of satellite constraints. The method is the same as in the original SPAS. See Appendix E for a detailed explanation and examples of constraints-encoding.
- *Calculator class.* A collection of mathematical routines that are not available in the target language. These could range from very simple (like taking the fractional part of a number) to more complex (like matrix or vector manipulation).

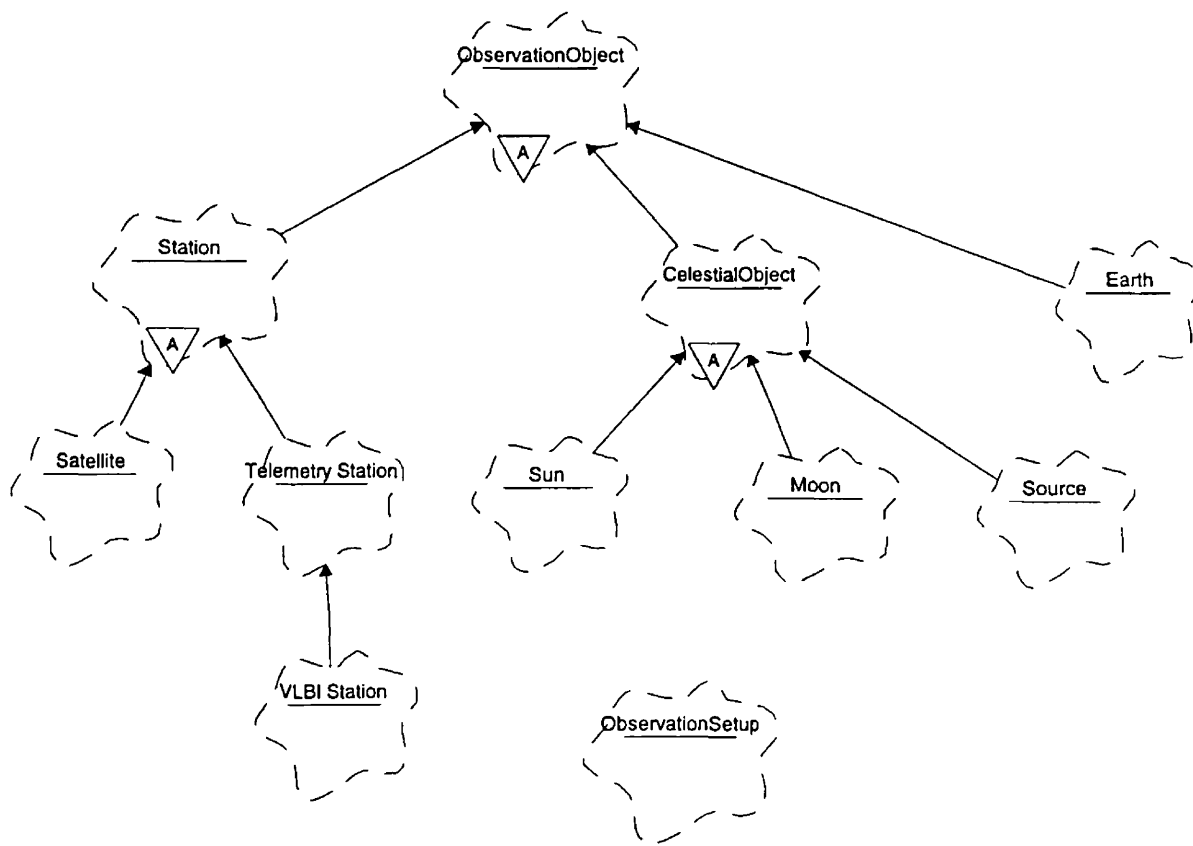


Figure 6
Alternative Position class hierarchy

3.3.2 “Observation” classes

The classes in this category describe all the different object types that can affect or take part in a Space VLBI observation (*ObservationObject*) as well as a collection of these objects with other relevant features (such as start and stop time) that make up a concrete observational setup (*ObservationSetup*). These can each be saved to and loaded from disk - see comment on text files in chapter 3.3.4 (*Database* class). *Figure 7* shows the inheritance hierarchy of the classes in this category, while *Figure 8* shows the fundamental structure of the classes themselves. Note that subclasses inherit all elements of their parent classes that are shown.

All of the observation objects have more than one coordinate (either by type or by number) - the reason being is optimization at the design level: To avoid repeated calculation of different but equivalent position types. For example *Satellite* has two *KeplerianOrbit* classes - one for the database information, and one that is to be evolved (iterated) during simulations. If this weren't the case the original orbital elements would have to be computed iteratively from the base value every time - substantially slowing the simulation (to a point where it is unusable). *Satellite* also inherits *RectangularPosition* classes which describe the same position of the satellite but are used in different calculations.

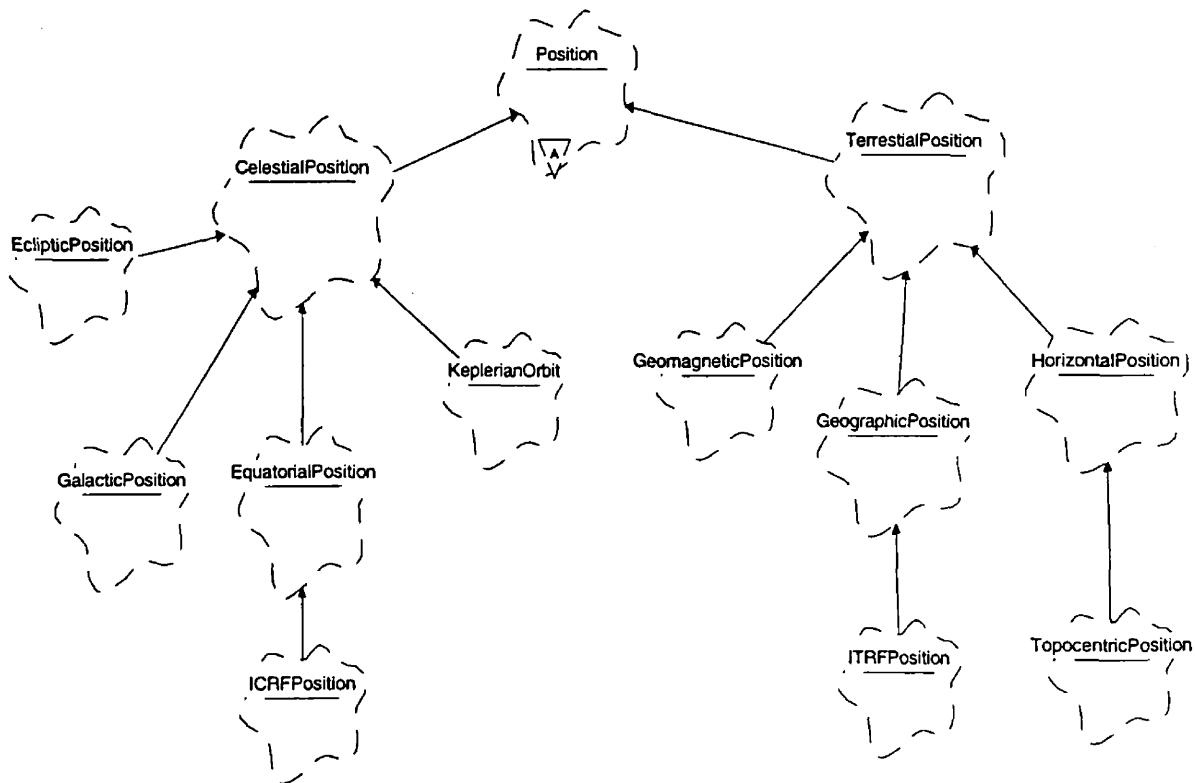


Figure 7
Observation class category inheritance diagram

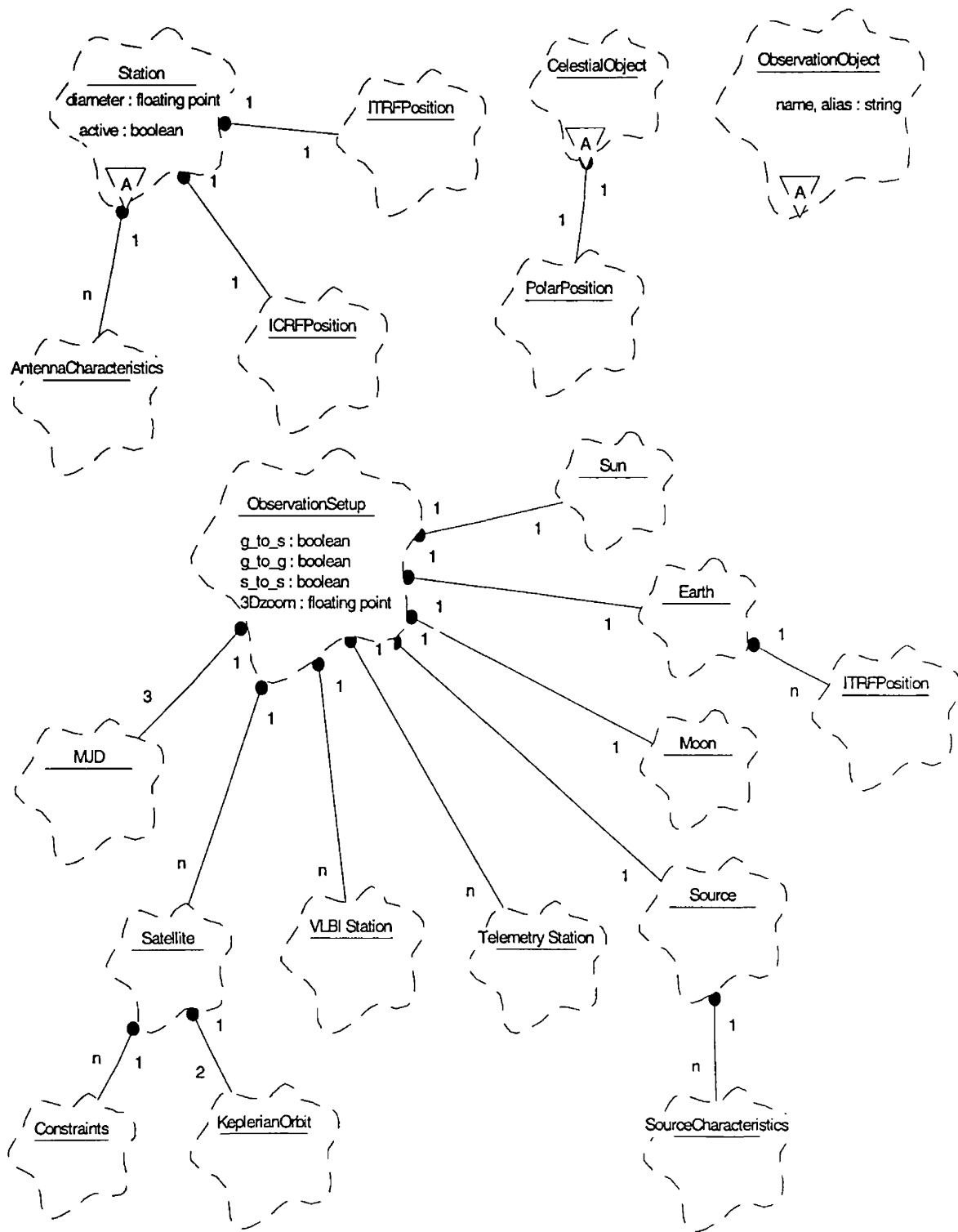


Figure 8
Observation class category structure diagram

3.3.3 “GUI” (Graphical User Interface) classes

The graphical user interface is a separate category, so that the appearance of the program can be easily changed if needed (e.g. by a usability analysis). The classes here include the windows, buttons and message dialogs that appear on the screen as well as the handling the messages generated by those components (e.g. a button press).

Although separate from the *Modules* category, these GUI classes obviously very closely interact with the corresponding classes in that category. The effect of that collaboration is shown on the state transition diagram of the *SimulationWindow* class (Figure 14), where all the events are generated by the user interface, but the class' state also depends on what state the “underlying” *Simulation* class is in (e.g. running or idle). Depending on that, the interface can be in different states (e.g. whether the Start button is enabled or not). Also the *Simulation* class displays its results in the *SimulationPanel* part of the *SimulationWindow*. *SimulationPanel* has two varieties: the simple one, where everything drawn is immediately shown and the *BufferedPanel* version where results are shown when specifically requested. This latter one makes it easier to produce animation with less flicker (e.g. for *3DView*) while the former one is used for modules like *UVPlot* where results are shown as they are calculated.

Note that the hierarchy of the elements in the GUI (Figure 9) is based on the Java 1.0 AWT (Abstract Windowing Toolkit - see [10] page 239) hierarchy, thus if implementing the GUI in Java only the SPAS specific parts need to be programmed.

Rather than showing the structure of the GUI classes in a traditional Booch diagram with “has” relationships, the planned appearance is designed with various tools and prototyping. Figure 12 shows the *MainWindow*, created in Java as a prototype. It includes *ImageButtons* for all of the original SPAS modules, though only *3DView* and *UVPlot* are shown to be active. The *Menu* includes options to load/save the main setup (of type *ObservationSetup*), as well as to modify it. Figure 10 shows how *SetupDialog* would look on-screen (as designed by the Visual J++ Resource Wizard [9]), while Figure 11 shows the *StationSelection* dialog window that can be reached through pressing the “Satellites”, “VLBI Stations” or “Telemetry Stations” buttons (obviously different entries are shown depending on which button was pressed). Each module can have its own set of parameters above these, reachable through the “More” button (like a zoom factor for *3DView*) - which brings up a dialog window for the extra parameters. Also some modules might disable parts of the setup dialog show in Figure 10, in case not all parameters of it are needed (e.g. for a module that plots subsatellite tracks, there would be no need to select VLBI stations - thus the button would be “grayed out”). For a more detailed explanation of what each parameter does see the description of the setup text file format (in Appendix C).

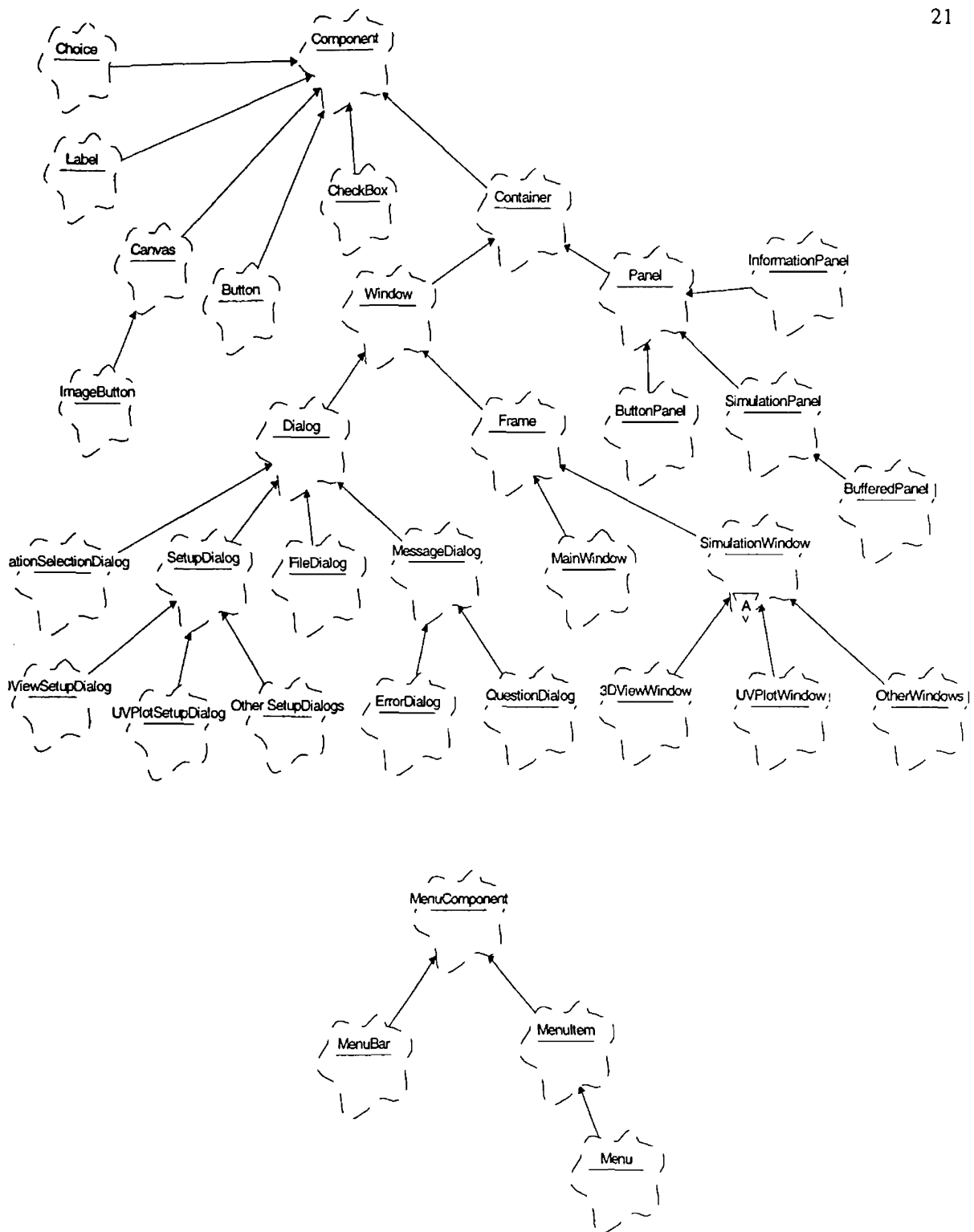


Figure 9
Graphical User Interface class category inheritance diagram

SimulationWindow structure design is shown in *Figure 13*. This is the general design on which subclasses of *SimulationWindow* build on. There are three main parts: an *InformationPanel* (for showing the necessary data and legend to understand the simulation display), a *ButtonPanel* (with all the necessary buttons to control the simulation) and a *SimulationPanel* (where the output of the simulation goes).

The behavior of *MainWindow* is essentially the same as the behavior of the whole system (see *Figure 3.3.2*), while the behavior of each *SimulationWindow* is also similar, as described in *Figure 14*.

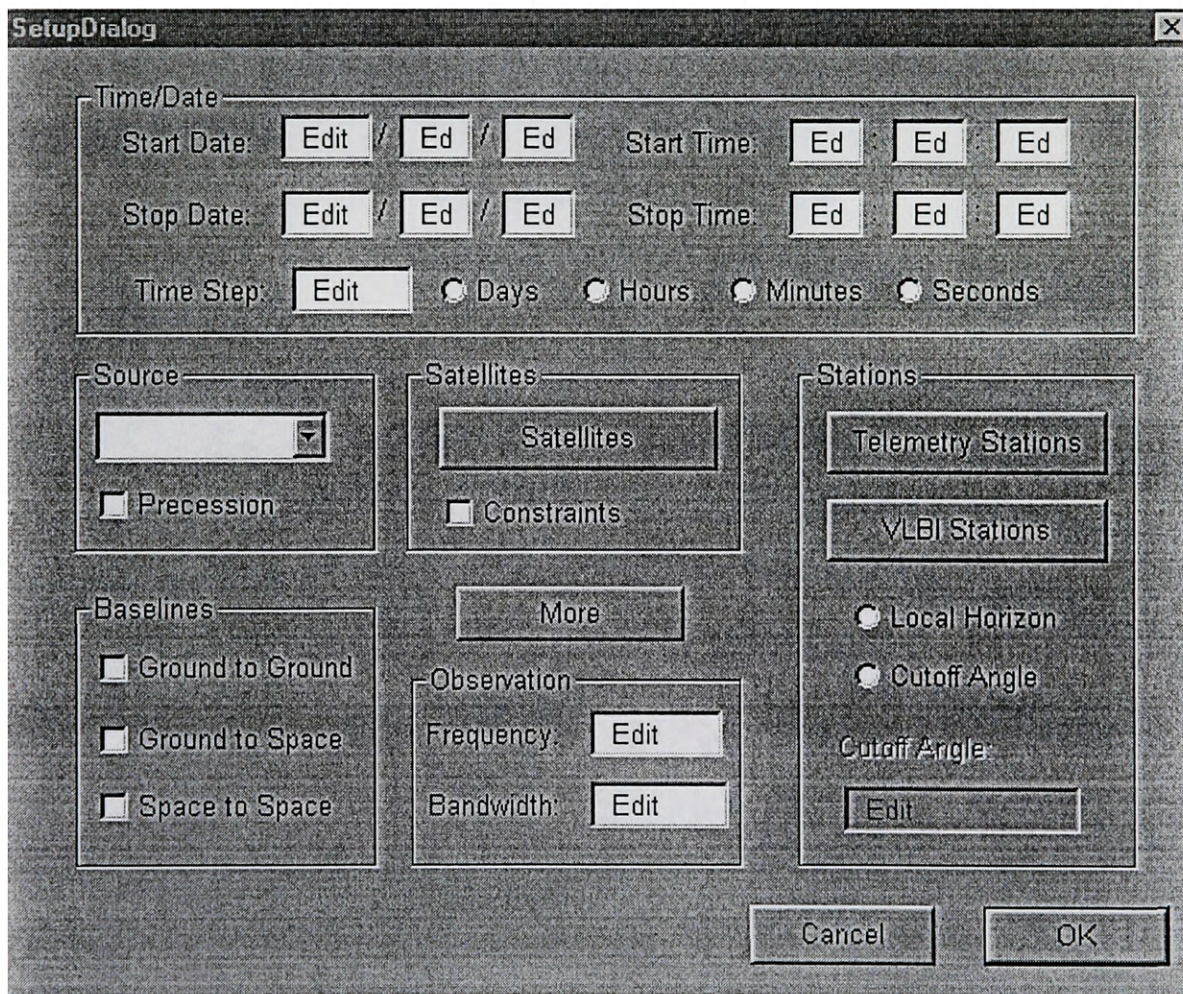


Figure 10
SetupDialog structure design

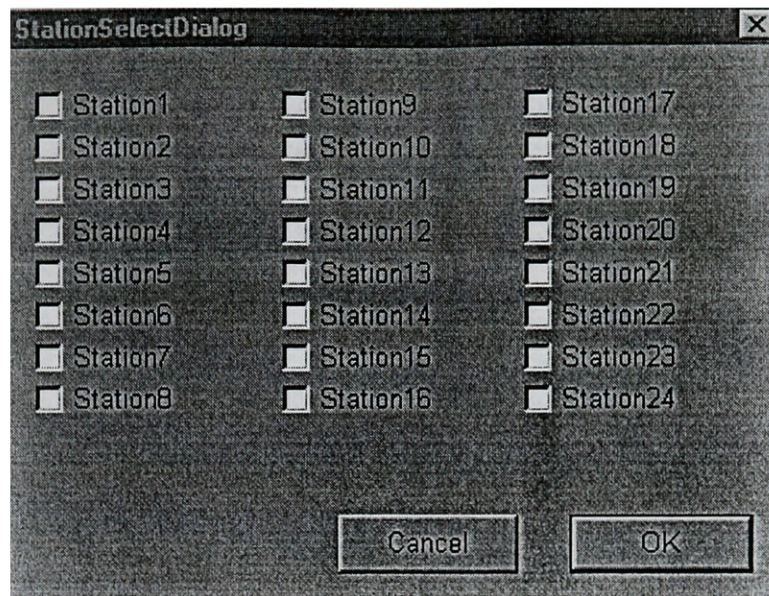


Figure 11
StationSelectDialog structure design

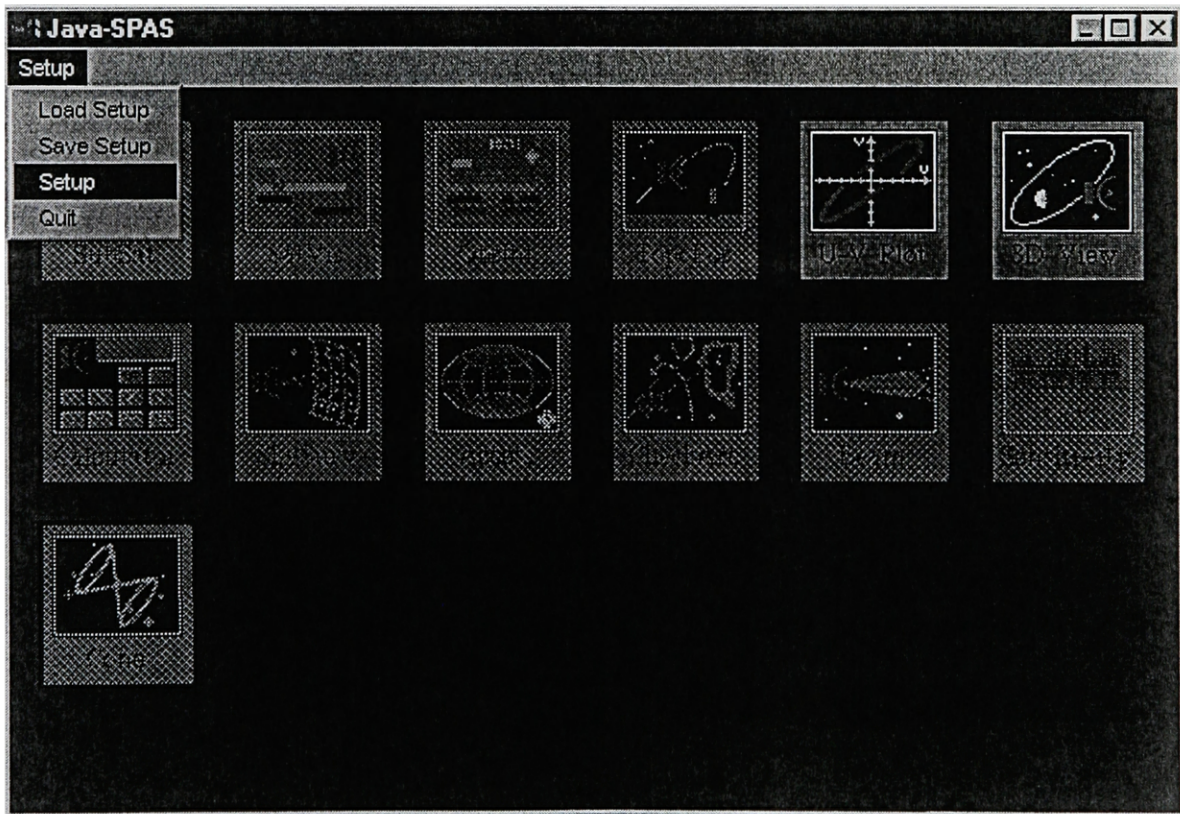


Figure 12
MainWindow structure design

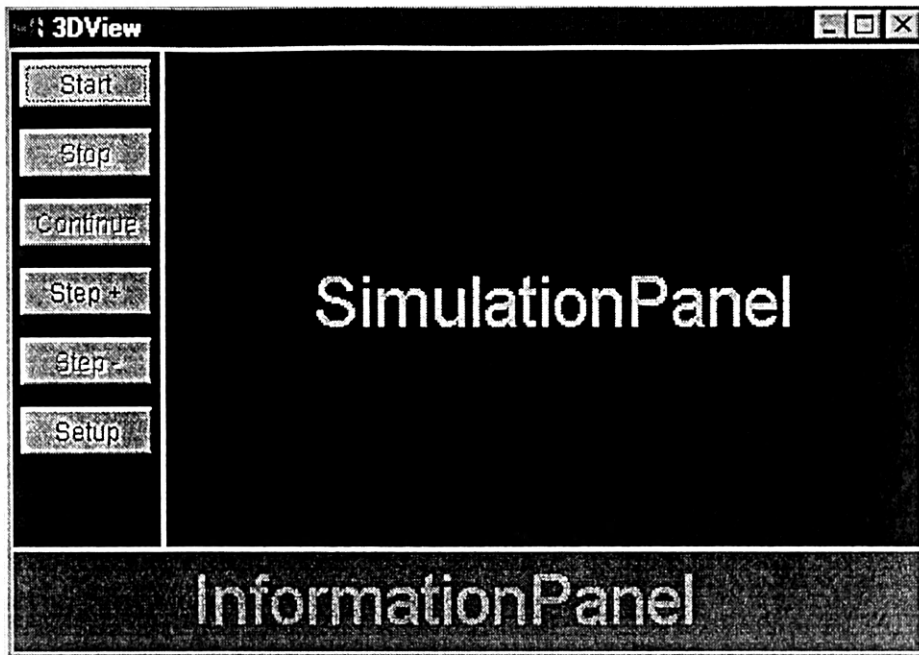


Figure 13
SimulationWindow structure design

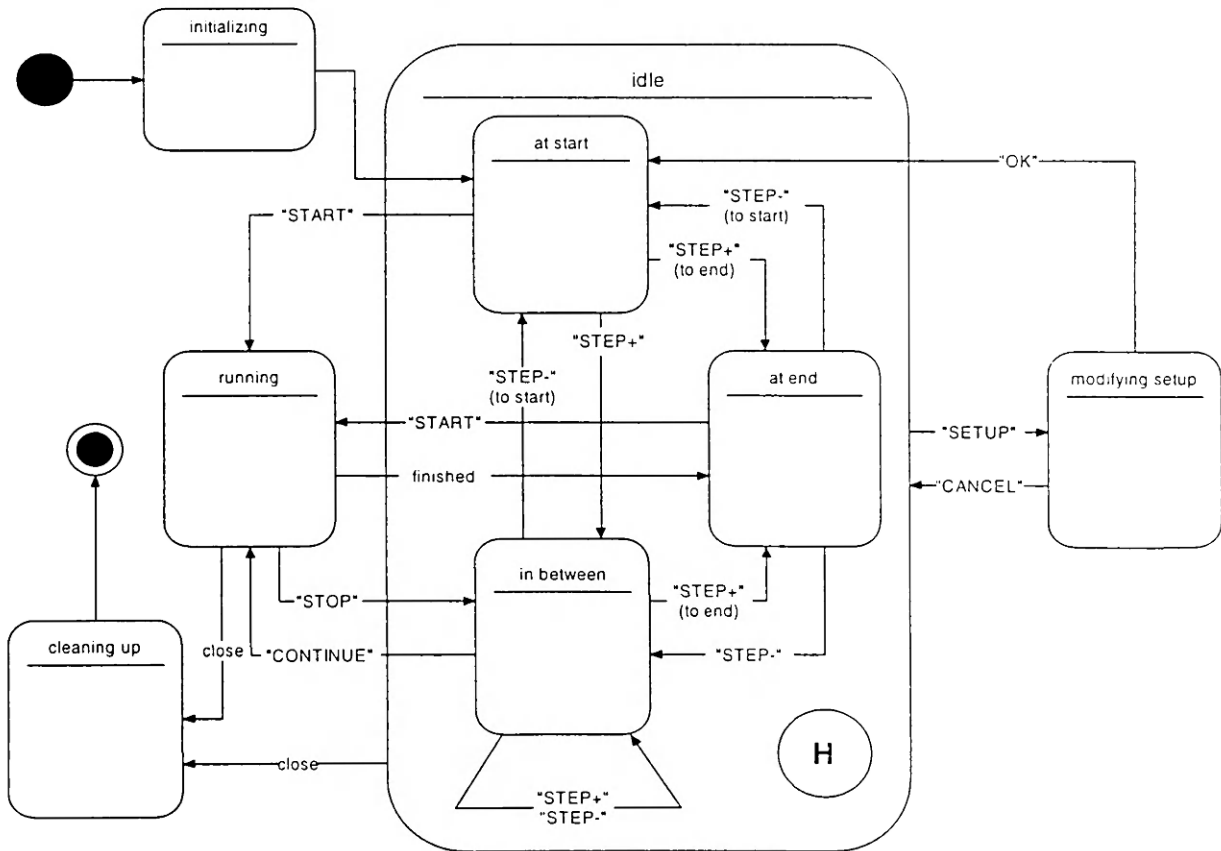


Figure 14
SimulationWindow state transition diagram
capital letters denote button press events by user

The *SimulationWindow* state transition diagram (*Figure 14*) shows that the class can be essentially in two major states: either idle or running and in either of those states it is still active checking GUI events (e.g. button press actions, which are in shown capital letters) that can lead to state changes. Error states are not shown to avoid congestion of the diagram. Also note that buttons that are not valid in a given state are “grayed out”, as in *Figure 13*, which shows *SimulationWindow* in the “idle - at start” state.

The buttons (in capital letters) evoke the following actions

- **START** Starts running the simulation
- **STOP** Stops the simulation
- **CONTINUE** Continues the simulation from the point where it was aborted
- **STEP+** Steps one timestep forward in the simulation
- **STEP-** Steps one timestep backward in the simulation
- **SETUP** Brings up a *SetupDialog* window to modify the observational setup parameters for the current simulation

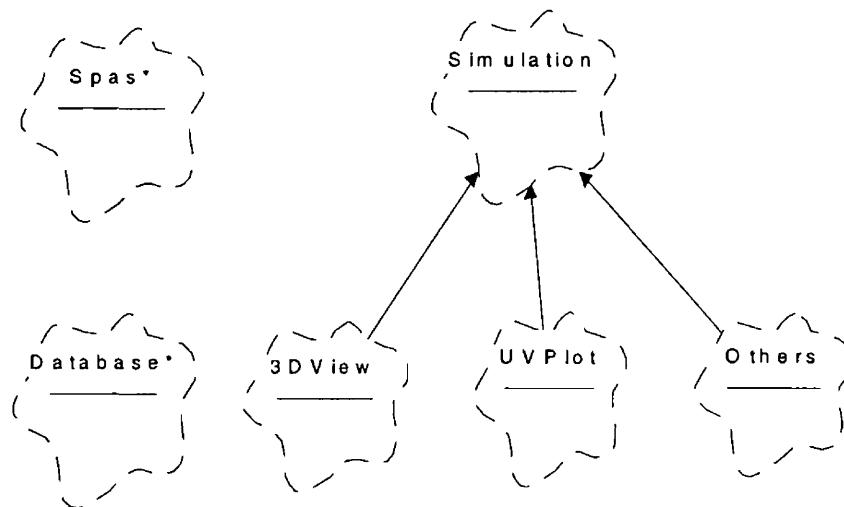


Figure 15
Modules class category inheritance diagram

3.3.4 “Modules” classes

The classes in the *Modules* class category are at the highest level. Most of them are the ones actually performing the simulations (*Simulation* class) and displaying the results in the appropriate *SimulationPanel* object. All objects of type *Simulation* are active - that is they have their own thread of control providing the possibility of more than one simulation (even of the same type) running in parallel with others.

Simple description of each basic class in this category:

- *Spas*. The “startup” class - initializes all necessary objects including *MainWindow* - which is then responsible for handling all events (see *Figure 3.3.2*)
- *DataBase*. Encapsulates all of the available data (stations, sources, satellites, etc.) and the methods to access them. It is initialized once, at startup, and after that it provides services to other classes (mostly to *ObservationSetup*, see *Figure 19*) As a low level design decision all data and setup parameters are saved and loaded from human readable text files. This not only makes it easier to debug the program, but also provides the possibility of skipping the design/implementation of otherwise essential parts (like a user interface for the database) of the software, without crippling it in any way. Yet the option of adding these parts later is not lost either, and until then users can create/modify setup and database files (after understanding the text file structure – see Appendix B) with their favorite text editor.

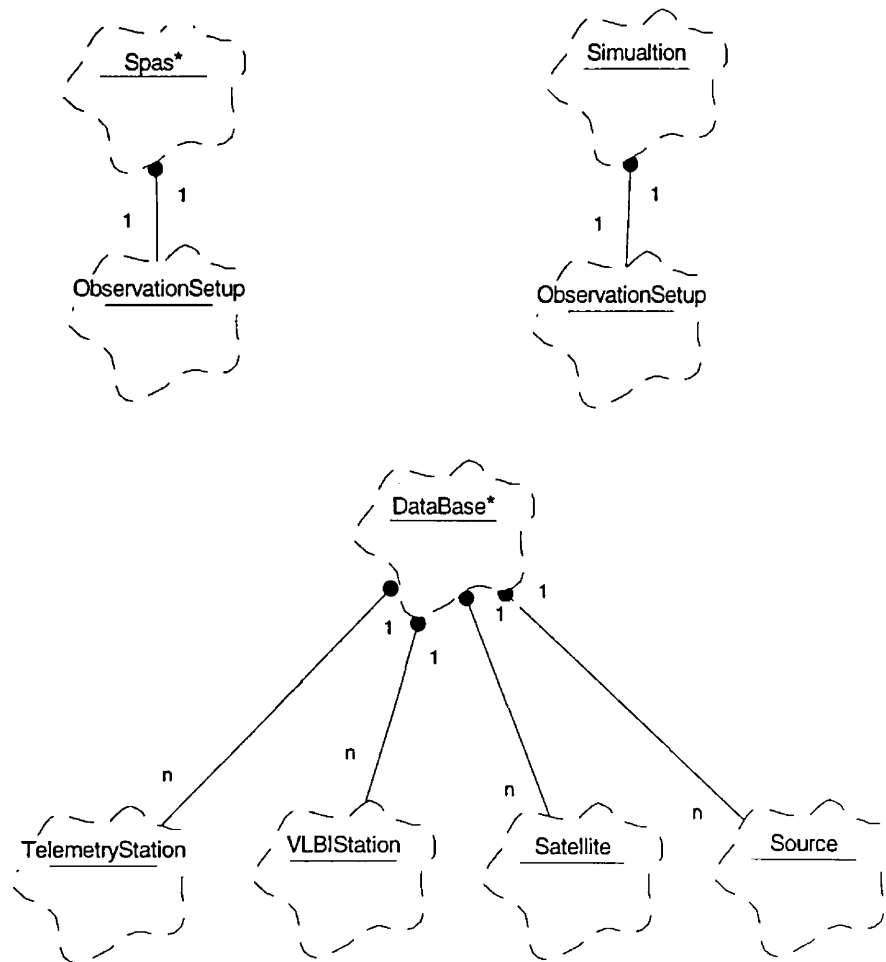


Figure 16
Modules class category structure diagram

- 3DView.** This operation produces a three dimensional movie of the rotating Earth and the orbiting satellites. The viewing direction is the radio source direction. The viewing distance remains the same during a run. The ground *VLBI stations* are also indicated (however *Telemetry stations* are not, it would make the output overly confusing). The operation shows the active ground-to-ground, ground-to-space and/or space-to-space *baselines* step by step. A baseline is active when geometric and technical restrictions don't obstruct the observation at its both ends at the current time. See *Figure 18* for this module's state transition diagram.
- UVPlot.** This operation produces UV-plot of VLBI baselines. You can select ground-to-ground, ground-to-space and/or space-to-space baselines to plot. A UV-plot is the projection of the baselines on a plane perpendicular to the direction of radio source to be observed. The axes are scaled conventionally with the observing wavelength. The projection of a baseline is plotted on the diagram when geometric and technical restrictions don't obstruct the observation at its both ends at the current time. See *Figure 17* for this module's state transition diagram.

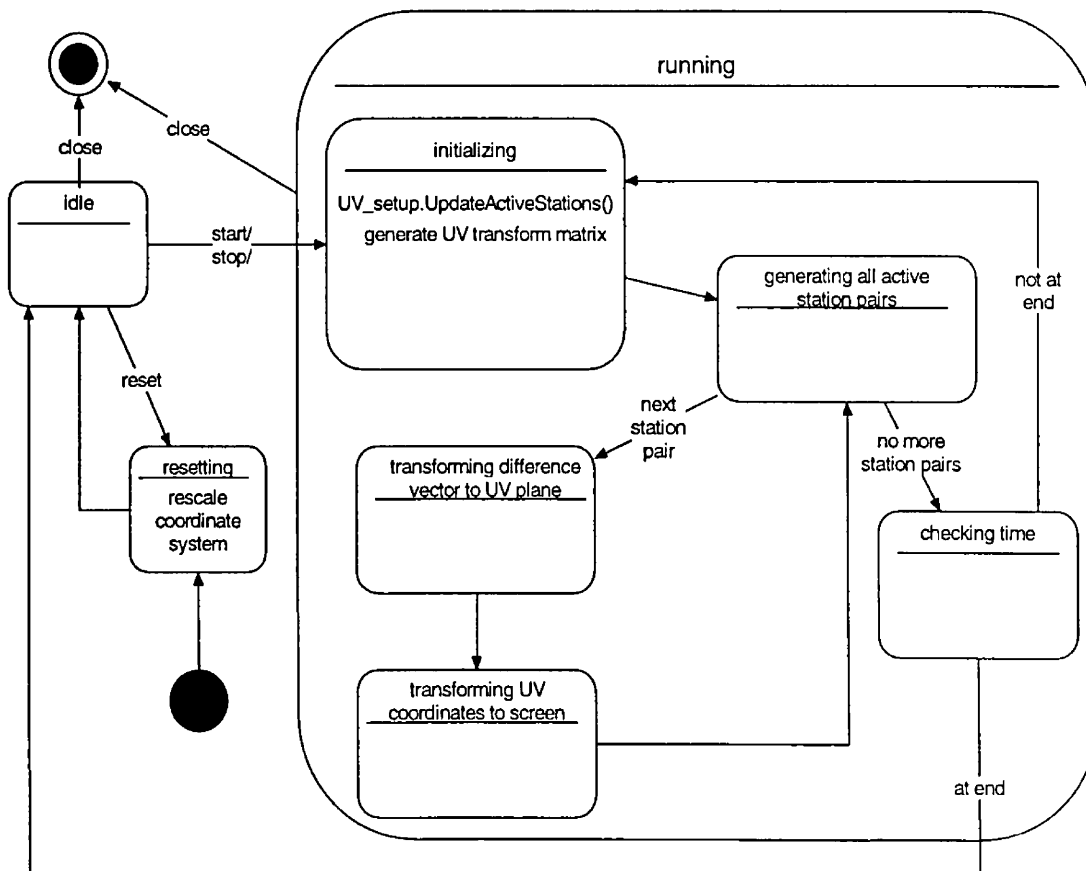


Figure 17
UVPlot state transition diagram

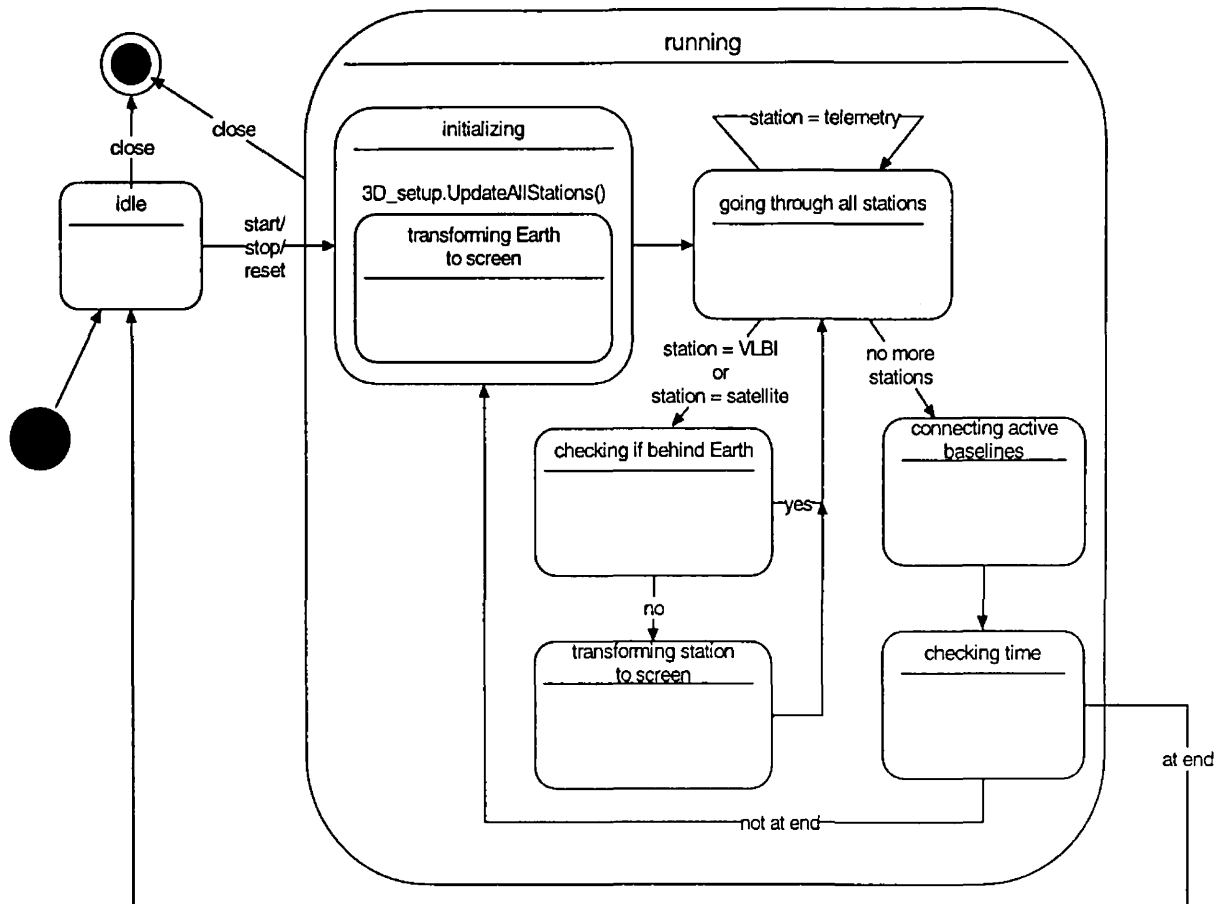


Figure 18
3DView state transition diagram

3.3.5 Scenarios

Scenarios are described in two ways: either with object diagrams or with interaction diagrams (see Appendix D). While as Booch [8] puts it “Object and interaction diagrams are sufficiently close in terms of their semantics that it is possible for tools to generate one diagram from the other with minimal loss of information”, they provide different style views. With interaction diagrams the emphasis is on the relative order of the events/actions, while object diagrams permit the inclusion of other information about the objects’ relationships.

The first interaction diagram (*Figure 19*) shows how a general system startup happens (what objects are initialized, in what order and how). Also shown is the “shutdown” event for ending the program, which is actually a very simple close event initiated by the user, e.g. selecting the “Quit” menu in *MainWindow*. Notice that *DataBase* initializes itself during the first call (from *ObservationSetup*) it has to handle, since there is no need for its existence before that. Also it creates all the *Sources*, *Satellites*, *VLBIStations* and *TelemetryStations* as it reads them from the database text file. These can be later used in different setups. The scenario here has the option of source precession set in the setup file, so the source position is precessed at this point (and whenever the user changes the start or stop time from the *SetupDialog* later).

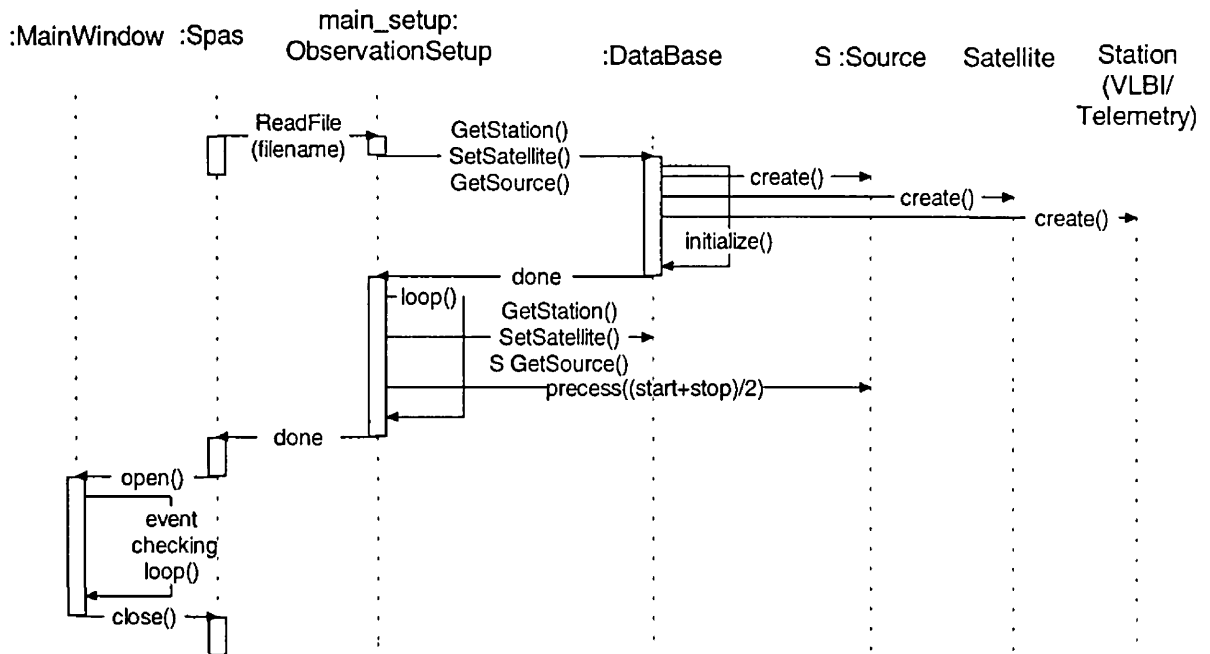


Figure 19
System startup and closing interaction diagram

Once *MainWindow* has started its main loop it acts upon receiving such events as an *ImageButton* press by opening the appropriate *SimulationWindow*. This is not a simple process (shown in *Figure 20*), since the newly opened *SimulationWindow* has to set up all of its panels : *ButtonPanel* , *InformationPanel*, and *SimulationPanel*. For *ButtonPanel* this is done by the *SimulationWindow* itself, but the other two has to be filled with information by the underlying *Simulation* module. This is done by calling that module with the appropriate parameters so that it can draw the default output into the given *SimulationPanel*, and print the relevant setup information into the given *InformationPanel*.

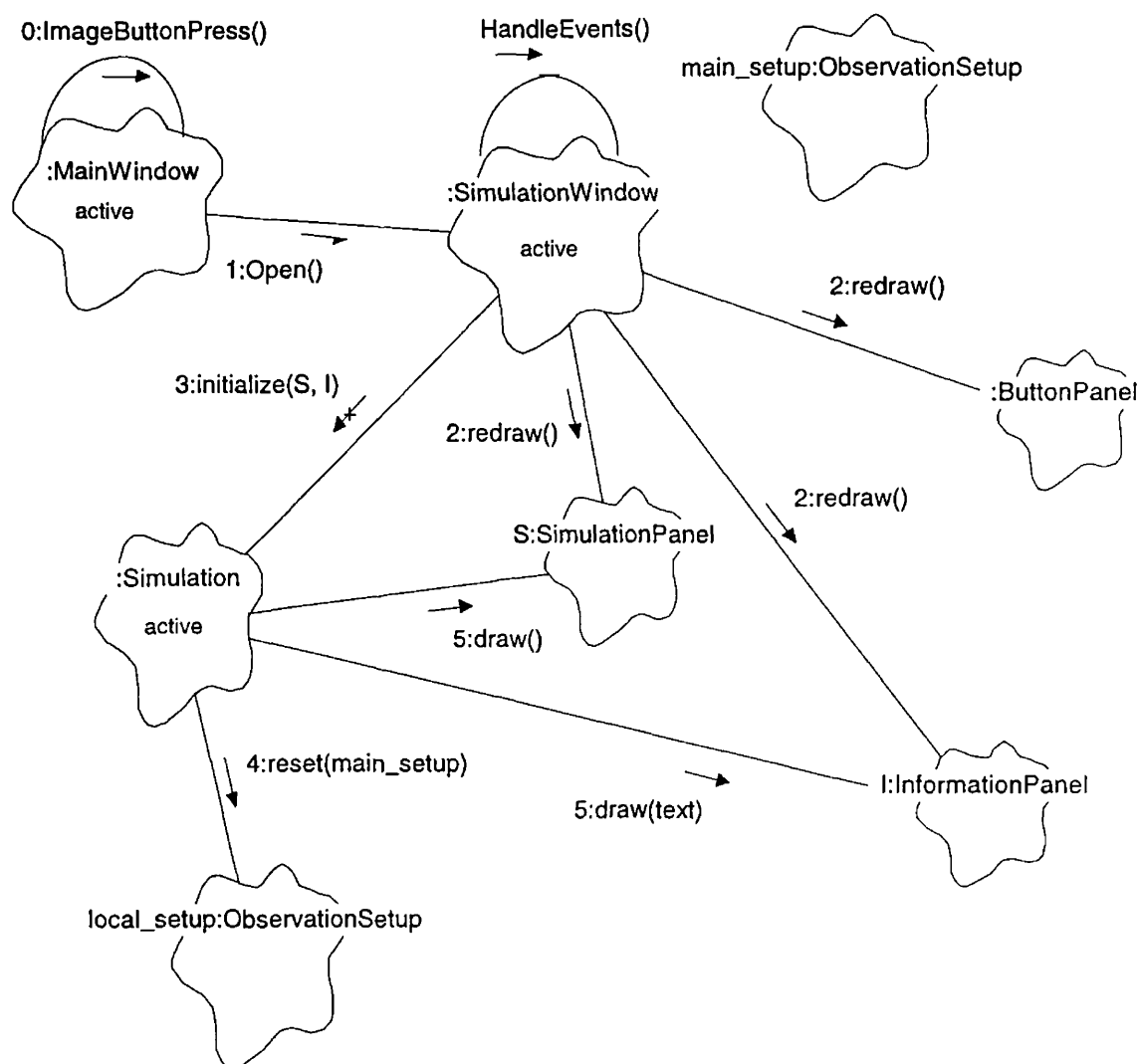


Figure 20
An opening of a *SimulationWindow*: object diagram

For example in the case of *UVPlot* the default output would be the scaled UV coordinates, while for *3DView* it would be the view of the earth-satellites-stations configuration as seen at the starting time of the observation from the source's point view.

Once a *SimulationWindow* is open, the user can change the local setup parameters, start/stop running the simulation, or have it step forward or backward in time. A setup-start-stop scenario is shown in *Figure 21* where the user first changes the local setup, and accepts it (with the "OK" button), then starts running the simulation (with "START" button) and before it ends he or she stops it (with the "STOP" button).

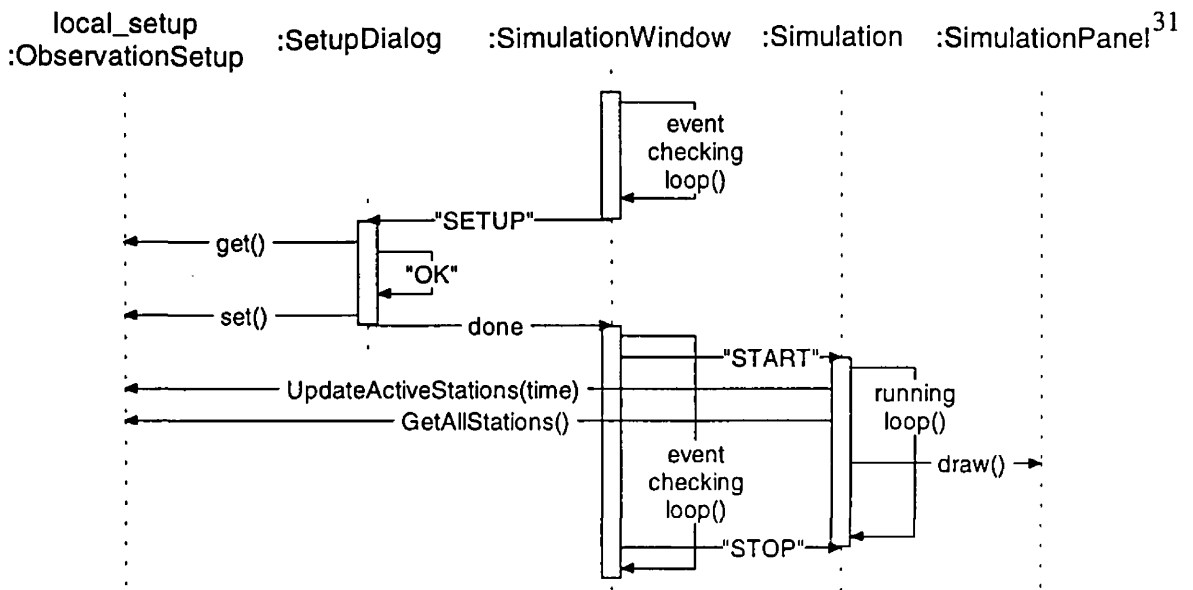


Figure 21
Simulation setup-start-stop interaction diagram

Notice that the *SetupDialog* is modal to *SimulationWindow* - that is the user can do nothing with the *Simulation* until he/she has exited the *SetupDialog* (by clicking on the "OK" or "CANCEL" buttons). On the other hand we can see that the *Simulation* - once started by the user - has an independent thread of execution which either ceases to exist when it finishes (reaching the StopTime of the setup) or when the user generates an asynchronous stop message by clicking on the "STOP" button.

The key point in the simulation is the running loop. This is where each of the simulation modules differ, although there are common parts. Such parts include increasing the time by a time step value each pass and the calls made to the local *ObservationSetup* to update all the active stations for that given time (position, observing status). Which of those stations are updated and will be actually used can differ on the selected baseline type parameter, and the simulation module too. For example *3DView* still displays a satellite even if it can not observe the source (although with a different color) while *UVPlot* strictly uses the active stations only for plotting.

Figure 22 shows a very simple - and highly unlikely - scenario, where the updating of the active stations initiated by a *Simulation* in a local *ObservationSetup* takes place. In this setup, there is only one *TelemetryStation*, one *VLBIStation* and one *Satellite*, the *GroundToSpace* baseline type is selected and the use of *Satellite* constraints are on. Note that *Satellite* Sa is only considered active if all of the following are true (short circuit boolean evaluation is to be used to optimize the calculations):

- a) *TelemetryStation* T can see Sa, that is *ToSatellite*(Sa) is true.
- b) Sa sees *Source* S, that is *ToSource*(T, S, time) is true. The reason T is passed as an argument is that constraints checking includes whether the satellite can see T (should be only checked if T could see the satellite in the first place). Constraints checking also includes a lot of other things - see Appendix E for a list.

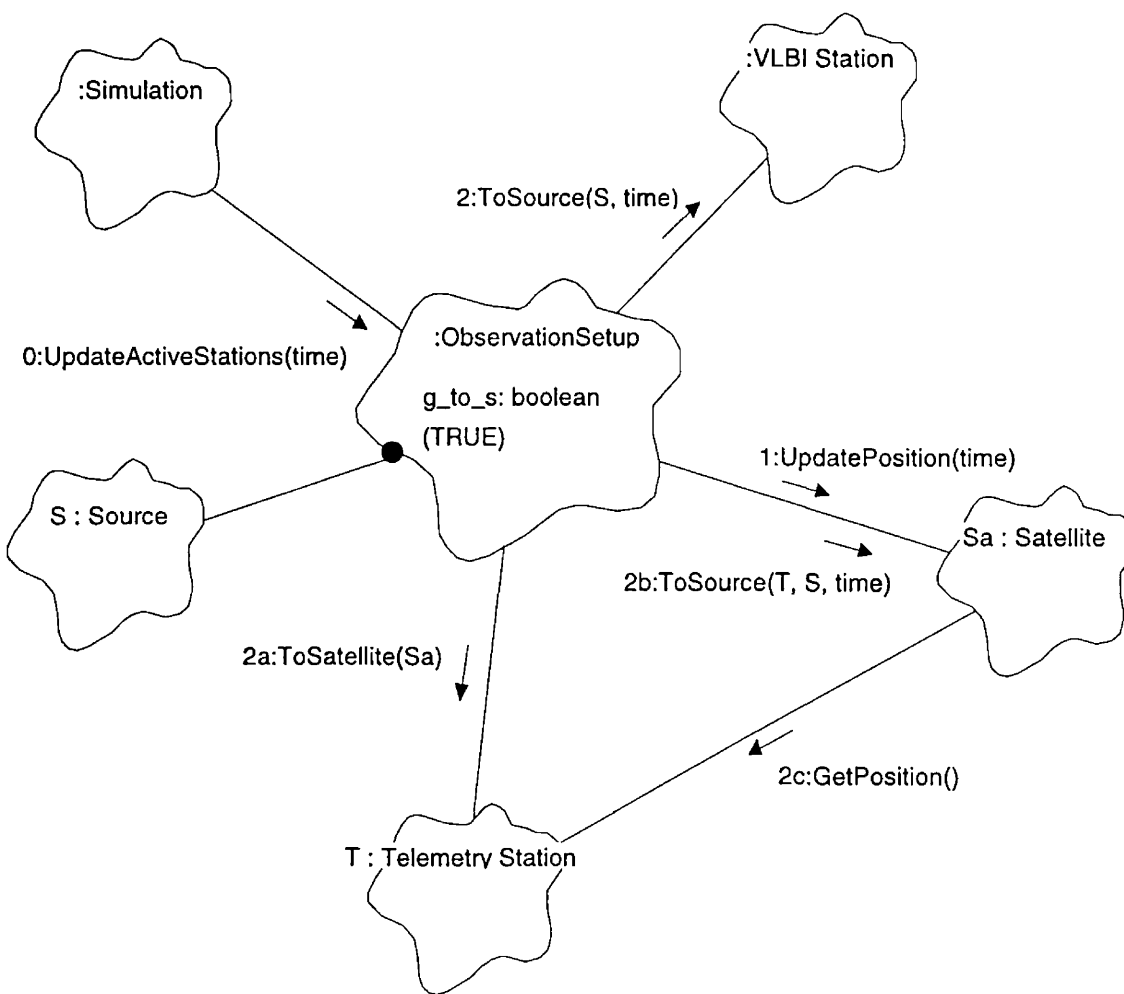


Figure 22
UpdateActiveStations() object diagram

4. JavaSPAS - implementation

In implementing JavaSPAS I found class diagrams and object diagrams especially useful. While the former guided in creating the structure of the system (see *Figure 23*), the latter made clear of the interfaces and variables each class/object needs. Not all of the design is implemented in code yet, but the framework for running simulations is done, as well as simple versions of the *3DView* and *UVPlot* modules themselves. This includes a great part of the user interface and windows, and the underlying class hierarchies and necessary methods to support those simulations.

4.1 Why Java?

The main purpose of my project was to design and partially implement the original SPAS so that it would run under Unix type operating systems. My first approach was to use C and C++ with Motif[®] libraries. This would have required little or no re-design of the software, mostly just re-coding the original version - apart from the user interface that would have to be redone in X-Window/Motif.

There would have been a couple of disadvantages of this method, including:

- Software would have had to be recompiled (left to the user) in cases where the binary version is not available for the given system.
- Complicated makefiles (and possible #ifdefs in the code) would be needed to generalize code (and that still does not guarantee complete portability)
- The program would only run on Unix systems (with Motif libraries needed for recompilation)
- Graphical User Interface can not be fully object oriented, unless C++ wrapper classes are used for the Motif[®] libraries.

Because of all the above possible problems and the availability of the new programming language Java, which includes features that would solve these problems, I have decided to design and partially implement SPAS in that language. Since Java is object oriented and since the original design of SPAS is procedural oriented, implementation implied re-designing SPAS from the basics in an object oriented fashion. Also, Java is a language whose specification is still evolving. What this means is that a given implementation might have to be modified (e.g. a transition from Java 1.0 to 1.1) to accommodate changes, but if the design is solid and well established the effect should be minimal. Thus I concentrated on the object oriented design of the software rather than the implementation.

Apart from the above mentioned problems Java has numerous advantages:

- Software needs no recompilation on different systems (same “binary” under different OS’s)
- No need for huge extra libraries - Java includes most of the user interface components needed. This also makes it possible to distribute the source code without having to worry about proprietary libraries’ copyrighted code.
- Program runs not only on Unix systems but on any other that supports the Java virtual machine (notably Windows 95/NT, thus extending the “user domain” of the program by a large amount)

4.2 Important Java features

Based on Sun’s “The Java Language: A White Paper” [11], I’ll try to summarize here the features of Java I find relevant and important to the implementation of the Object-Oriented SPAS design. According to that paper Java is “a simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.” Even Sun admits that this is a set of so-called “buzzwords”, so we have to go into more detail what each of those mean.

Simple and Robust

Java syntax is based mostly on C/C++, but without the “confusing” parts. What this means that things like multiple inheritance, operator (not method!) overloading are eliminated. While I fully agree with the elimination of multiple inheritance (although there is a similar feature in Java accomplished with the use of *Interfaces*) I found operator overloading useful in C++ and would have used it in Java too. An other thing simplified compared to C/C++ is storage management. Java uses automatic garbage collection, which makes it very easy to do dynamic memory allocation from the heap, without “memory leaks”, which are a very common and hard-to-debug problem found in mid- to large size C/C++ software packages. On the other hand this feature can cause quite a problem for people who start with Java and then try to move on to some other language, most of which does not include automatic garbage collection.

Java also implements a simplified pointer model that eliminates the possibility of overwriting memory and corrupting data. Other runtime checkings (like array subscript checking) and strong typing (e.g. no implicit integer to pointer casting) add to the robustness of the language.

Object Oriented

The object oriented technique is a way to design and implement software that is easily extendible and maintainable. Its benefits show mostly when applied to large complex systems (like SPAS). The basic (very

simplified) idea is that one packs data and the functions that act on that data into one entity (encapsulation). Other important object oriented features include modularity, hierarchy, typing and concurrency all of which Java implements.

Portable and Architecture neutral

This is one of Java's most important features, since most of today's widely used computer languages do not support portability and architecture independence as Java does. And surprisingly this not only applies to the source code, but the also the byte code, which is the object file format executed by the virtual Java machine on any platform. Java also defines the sizes of the primitive data types (see *Table 3*), as well as the behavior of arithmetic on them. Interestingly enough in my experience printing results can be a little bit different depending on the environment, but this is certainly not the problem of the language, but the result of different floating point output routines of Java 1.0/1.1 implementations. These differences are due to the fact that floating point to string conversion was not specified in Java 1.0 (see the section on precision requirements in chapter 4.3.3).

Type	Contains	Size (bit)
boolean	true or false	1
char	Unicode character	16 (!)
byte	signed (!) integer	8
short	signed integer	16
int	signed integer	32
long	signed integer	64
float	IEEE 754 floating point	32
double	IEEE 754 floating point	64

Table 2
Java primitive Data Types
(unconventional values/sizes are marked with a '!')

Interpreted and High-performance

Java compilers generate architecture-independent byte code, which is executed by the "Java Virtual Machine" by translating (interpreting) it to machine code on-the fly. This seems to contradict the possibility that Java is high performance, since interpreted code runs much-much slower than compiled, but in fact just-in-time compilers already exist, which take the byte code, and translate it to native machine code at run time. The execution time of such Java programs is nearly indistinguishable from a comparable program written in C. Yet the overall idea here is that for the sake of portability, one always should be willing to sacrifice some speed.

Secure and Network-savvy

Java grew out of a networking environment so it is obvious that it has all the necessary libraries (classes) to support protocols like TCP/IP and anything on top of that (like FTP or HTTP), handling connections with the same ease as reading or writing local files. Also it supports all the necessary features to keep it safe, for example it makes the distinction between Applets (which are Java programs that one can run in a browser like Netscape coming directly from the net) and Applications that are stand-alone Java programs (run by a separate Java virtual machine). To prevent any security problems Applets are generally not allowed to read, write, create, execute, rename or delete any local files.

Multi-threaded

Java supports a multi-threaded programming paradigm, which makes it possible for well-written programs to gain speed on multi-processor systems. Even on single-processor systems there can be an advantage, for example in better interactive responsiveness. In JavaSPAS this means that one can start multiple simulations at the same time, and still be able to access graphical user interface features (menus/buttons) to control those running simulations.

Dynamic

Java was designed to adapt to an evolving environment - among many things to make it dynamic are: It creates objects only when they are needed (conserves resources), classes have a run-time representation, so unlike in C or C++ a running program can find out which class a given object belongs to by checking this information. This information also makes it possible to dynamically link classes into a running system.

4.3 Implementation details

As for now JavaSPAS is implemented as an application. The main reason for this decision is that applets are not allowed to read any local files. Thus if the user modified the default observation setup, he/she could not save the modified file on the local system. Providing JavaSPAS as an applet in the future may nevertheless be a good idea, since a lot more people have a Java-capable browser installed than the number that are willing to install a Java runtime-environment just to try out a new piece of software. In other words it would make JavaSPAS even more accessible.

Some basic questions have to be resolved when putting the above design into code no matter what language is used. Here is the list I found important - with actions I took in the JavaSPAS implementation:

4.3.1 How to physically store the code

When coding the design I had to decide on a couple of conventions (like naming), regarding the placement of files (which more or less Java “decides for itself”) and some features which I would have liked to use, but were not available (like separate class specifications and bodies).

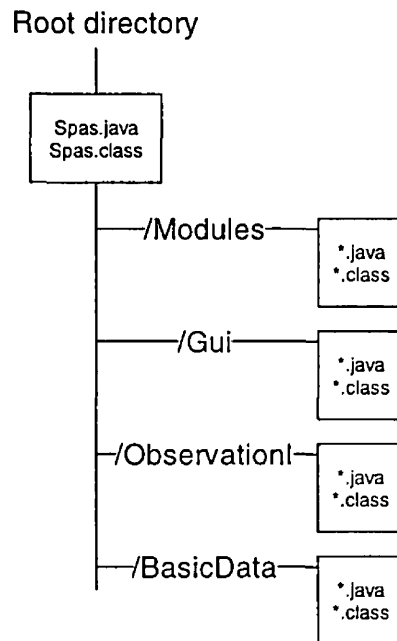


Figure 23
JavaSPAS files hierarchy

Where do we put classes related to their category

Java’s basic class hierarchy is two-leveled: On the higher level there are the collection of closely related classes, called *packages*, and on the lower level there are the *classes* themselves. When a class is a part of a given package it **must** be placed in a directory named exactly as that package and the source code file name must be the name of one of the classes in that file with a *.java* extension. After compiling the *.java* the byte-code is stored in *.class* files in the same directory - with one *.class* file for every class in that package.

In the case of JavaSPAS this means that the source (*.java*) and byte-code (*.class*) class files are put in a hierarchy as in *Figure 23*. Notice that the “startup” module *Spas* is put outside of this hierarchy - yet logically it belongs to the *Modules* package. Also note that supporting packages (e.g. that are part of Java) are not shown here.

Do we separate class specifications from class bodies

Unfortunately Java does not support separate specifications and bodies, thus both of those go into the same *.java* source file. Whatever is visible to the outside world from that file is the specification of that package (as seen in Appendix A). Note that different visibility rules apply to classes in the same package and classes in different packages (see *Table 3*).

What naming conventions to use

The convention I followed originates from the Java in a Nutshell book [10]: Class and method names are with capitalized letters (similar to the “Hungarian” notation [11] - e.g. like *ToSource()*) while actual objects and variables are with all lowercase letters (like *g_to_s*). Following this simple rule it is easy to decide whether an entity I am using in the code is a class or an instantiation of a class, and the code becomes much more readable. Note that one has to be careful when using predefined Java basic types: *Float* is not the same as *float* (see explanation in first section of chapter 4.3.2).

Also the code includes quite a lot of commenting (especially in critical parts) for the ease of later modification and/or enhancement.

4.3.2 Optimizations

Apart from possible optimizations in the calculations, Java language features should be used to provide the best results in running time.

Where to use pass-by-reference instead of pass-by-value

Java “decides” for the programmer where to use pass-by-value and where pass-by-reference. The rule is simple: Only basic data types (see *Table 2*) are passed by value (that is if passed to a method, their value is copied and any modifications to it inside the method will not affect the original variable’s value) anything else is passed by reference (that is if passed to a method, any modification to it inside the method will affect the original object too). This is optimized for speed, since more complex objects would take more time to be copied for pass-by-value, never the less the programmer can emulate this type of behavior if needed. On the other hand if one would like to pass variables of basic data types by reference, a wrapper class can be used (like *Float* for *float*).

Memory allocation/deallocation

As mentioned previously Java has automatic garbage collection - which makes it very easy to program but harder to optimize. This means not only optimization for memory usage - that we would be able to free memory as soon as it is not needed - but also for speed, since the automatic garbage collector can free up and compact memory at any time. This can be especially problematic in real-time applications, fortunately JavaSPAS is not one of them. There are various “tricks” to help the garbage collector, e.g. by setting an object reference to *null*, whenever it is not needed. An other - more radical approach - could be to disable the garbage collector, and call it explicitly when needed, or even to do away with it completely and implement our own.

Multi-threaded execution

As mentioned before, writing a program to take advantage of Java’s multi-threaded features (through the *Runnable* interface and the *Thread* class) can not only speed up its execution on multi-processor systems, but it can also increase a program’s user interface responsiveness (even on single CPU systems) by allocating a separate thread to the calculations and the GUI (Graphical User Interface). This is exactly what JavaSPAS does - all the simulations run as separate threads, thus the user can still interact with buttons/menus in the program after starting a simulation.

4.3.3 Other language dependent features

Multiple inheritance

Although the design for JavaSPAS does not contain explicit multiple inheritance, it is there implicitly. For example *3DView* is not only a type of *Simulation* but it also has its own thread meaning that in some ways it is a type of *Thread* too. Java handles this with interfaces - in the case of *3DView* the module inherits from *Simulation* **and** implements the *Runnable* interface (although this is done through *Simulation* too). The difference between real multiple inheritance and the Java interface implementation is that Java interfaces can only contain abstract methods which have to be implemented by the class that uses the interface, while in real inheritance one can use already implemented methods from any super-class.

Visibility

According to one of the basic paradigm’s of object oriented programming (information hiding), classes should only have controlled access to methods and variables to other classes. Java defines four levels of such control as shown in *Table 3*. Unfortunately the “*private protected*” visibility modifier is not implemented in Java, although it would be quite useful in my opinion. The reason is that given a class

variable usually one does not want any other class in the same package to be able to access it (as done with *private*) but wants it to be accessible to sub-classes (as done with *protected*). The solution (shown “grayed out”) would be the *private protected* visibility modifier. So why is it not in the language? According to the comp.lang.java.programmer FAQ [13]:

3.6 What happened to "private protected"?

- A. It first appeared in JDK 1.0 FCS (it had not been in the Beta's). Then it was removed in JDK 1.0.1. It was complicated to explain, it was an ugly hack syntax-wise, and it didn't fit consistently with the other access modifiers. More people disliked it than liked it, and it added very little capability to the language. It's always a bad idea to reuse existing keywords with a different meaning. Using two of them together only compounds the sin.

The official story is that it was a bug. That's not the full story. Private protected was put in because it was championed by a strong advocate. It was pulled out when he was overruled by popular acclamation.

Situation	default (package)	public	protected	private	private protected
accessible to non-subclass from same package?	yes	yes	yes	no	no
accessible to subclass from same package?	yes	yes	yes	no	no
accessible to non-subclass from different package?	no	yes	no	no	no
accessible to subclass from different package?	no	yes	no	no	no
Inherited by subclass in same package?	yes	yes	yes	no	yes
Inherited by subclass in different package?	no	yes	yes	no	yes

Table 3

Java visibility modifiers and their effect. Based on [10].
Note that the last column is not available from Java 1.0.1 on

Precision requirements

JavaSPAS does a lot of calculation - including iteration - which are sensitive to even small differences. While the precision requirements are not huge, consistency of results across different platforms is desirable. Java provides just that - it has all the floating point types defined in detail (see Table 2), and also the behavior of calculations on them. What was not specified precisely in Java 1.0 is the printing of the results (that is the conversion from *double* or *float* to *string*). As put in the comp.lang.java.programmer FAQ [13]:

There is a limitation of FP in JDK 1.0 (fixed in JDK 1.1). Namely, when you output a floating point number in Java 1.0, the result is system-dependent and contains no more than six digits after the decimal point. This bug is fixed in Java 1.1.

This can lead to strange results in the output when JavaSPAS is run on different Java versions. Consider the following example, where a date in *MilitaryTime* format is converted to *MJD* and back:

```
Compiled with
Microsoft (R) Visual J++ Compiler Version 1.00.6229

1) Viewed with
Microsoft (R) Command-line Loader for Java (tm) Version 1.00.6211

Results of MilitaryTime to MJD and back conversion:
MilitaryTime = 1999/12/31, 23:59:59.999
MJD = 51544 (4677316967000963520)
MilitaryTime = 1999/12/31, 23:59:59.999
MJD = 51544 (4677316967000963520)

2) Viewed with
Sun Microsystems Java(tm) Runtime Loader Version 1.1.2

MilitaryTime = 1999/12/31, 23:59:59.999
MJD = 51543.99999998836 (4677316967000963520)
MilitaryTime = 1999/12/31, 23:59:59.998994171619415
MJD = 51543.99999998836 (4677316967000963520)
```

In brackets after the *MJD* values are the long integer (*Double.doubleToLongBits()*) representations of the double value, which shows that the results are the same in both cases, but with the Sun 1.1.x runtime loader the full precision is shown, while the Microsoft one (1.0.x) rounds up the values for displaying.

4. Appendices

A. Pseudocode class definitions & Source code

This appendix shows the interface of each class in detail (public variables and methods, as well as some important private ones.) The pseudocode definitions are given in simple English, with keywords shown in bold. The preliminary version's commented source code [14] might give more insight on the behavior of each class. Mathematical models of the methods can be found in [5].

Note that maintaining this pseudocode definition-list is not an easy task, and although helpful in the implementation of the software, should be later maintained from the source code through some automated tool.

Modules class category

Class Spas

```
constants : File names, Resource names
classes   : MainWindow, ObservationSetup
variables : -
methods   : Initialize()
           /* called upon startup */
```

Class Database

```
constants : Resource names
classes   : array of Source, array of VLBIStation
           array of TelemetryStation, array of Satellite
variables : -
methods   : Initialize()
           /* called upon startup: reads database from file - see Appendix B */
           GetVLBIStation(string) returns VLBIStation
           /* Returns a VLBIStation with the given name */
           GetTelemetryStation(string) returns TelemetryStation
           /* Returns a TelemetryStation with the given name */
           GetSatellite(string) returns Satellite
           /* Returns a Satellite with the given name */
           GetSource(string) returns Source
           /* Returns a Source with the given name */
```

Class Simulation

```
constants : -
classes   : SimulationPanel, ObservationSetup
variables : boolean active
methods   : Initialize(SimulationPanel, ObservationSetup)
           /* called upon opening a SimulationWindow */
           Start(), Stop(), Continue()
           /* Starts/restarts, stops, continues running the simulation */
           Reset(ObservationSetup)
           /* Resets the simulation (e.g. with new observation parameters) */
           /* actually called from Initialize() */
```

Class 3DView inherits Simulation

```
constants : -
classes   : -
variables : -
methods   : -
```

Class UVPlot inherits Simulation

```
constants : -
classes   : -
variables : -
methods   : -
```

BasicData class category

Note that most classes in this category have at least two types of initializers:

- one that accepts the basic components of a class to create a new object
e.g. for *PolarPosition* : Initialize(double α_1 , double α_2)
- one that accepts an object of the given class to copy it and create a new object
e.g. for *PolarPosition* : Initialize(*PolarPosition*)

Also note that most member classes and variables can be reached (read and set) through class methods.

```
Class Position
constants : -
classes   : -
variables : -
methods   : -
```

```
Class PolarPosition inherits Position
constants : -
classes   : -
variables : double  $\alpha_1$ ,  $\alpha_2$  [rad]
methods   : ToRectangular() returns RectangularPosition
            /* converts to rectangular coordinates */
```

```
Class EclipticPosition inherits PolarPosition
constants : -
classes   : -
variables : -
methods   : -
```

```
Class EquatorialPosition inherits PolarPosition
constants : -
classes   : -
variables : -
methods   : ToEcliptic(double) returns EclipticPosition
            /* converts to ecliptic coordinates, needs obliquity of ecliptic */
            ToGalactic() returns GalacticPosition
            /* converts to galactic coordinates */
            ToHorizontal(MJDTime, GeographicPosition) returns HorizontalPosition
            /* converts to galactic coordinates */
```

```
Class HorizontalPosition inherits PolarPosition
constants : -
classes   : -
variables : -
methods   : ToEquatorial(MJDTime, GeographicPosition) returns EquatorialPosition
            /* converts to equatorial coordinates */
```

```
Class TopocentricPosition inherits HorizontalPosition
constants : -
classes   : -
variables : -
methods   : -
```

```
Class GalacticPosition inherits PolarPosition
constants : -
classes   : -
variables : -
methods   : ToEquatorial() returns EquatorialPosition
            /* converts to equatorial coordinates */
```

```
Class GeomagneticPosition inherits PolarPosition
constants : -
classes   : -
variables : double height
methods   : ToGeographic() returns GeographicPosition
            /* converts to geographic coordinates */
```

```

Class GeographicPosition inherits PolarPosition
constants : -
classes   : -
variables : double height
methods   : ToITRF() returns ITRFPosition
            /* converts to ITRF coordinates */
            ToGeomagnetic() returns GeomagneticPosition
            /* converts to geomagnetic coordinates */

Class RectangularPosition inherits Position
constants : -
classes   : -
variables : double x, y, z
methods   : Arc(RectangularPosition) returns double
            /* returns arc between two unit(!) vectors [rad] */
            ToPolar() returns PolarPosition
            /* converts to polar coordinates */

Class ITRFPosition inherits RectangularPosition
constants : -
classes   : -
variables : -
methods   : ToICRF(MJDTime) returns ICRFPosition
            /* converts to ICRF coordinates */
            ToHorizontal(GeographicPosition) returns HorizontalPosition
            /* converts to horizontal coordinates */
            ToGeographic() returns GeographicPosition
            /* converts to geographic coordinates */

Class ICRFPosition inherits RectangularPosition
constants : -
classes   : -
variables : -
methods   : ToITRF(MJDTime) returns ITRFPosition
            /* converts to ITRF coordinates */
            ToKeplerian() returns KeplerianOrbit
            /* converts to Keplerian orbital elements */

Class KeplerianOrbit inherits Position
constants : -
classes   : MJDTime
variables : -
methods   : ToICRF() returns ICRFPosition
            /* converts to ICRF coordinates */

Class DateTime
constants : -
classes   : -
variables : -
methods   : EclipticObliquity() returns double
            /* returns the obliquity of ecliptic */
            GAST() returns double
            /* Returns the Greenwich Apparent Sidereal Time [rad] */

Class MJDTime inherits DateTime
constants : -
classes   : -
variables : -
methods   : ToMilitaryTime() returns MilitaryTime
            /* converts into military time format */

Class MilitaryTime inherits DateTime
constants : -
classes   : -
variables : -
methods   : ToMJDTime() returns MJDTime
            /* converts into Modified Julian Date time format */

```

```

Class Characteristics
constants : -
classes   : -
variables : double frequency [GHz]
methods   : -

Class AntennaCharacteristics inherits Characteristics
constants : -
classes   : -
variables : double system_temperature [K], double efficiency [0..1]
methods   : -

Class SourceCharacteristics inherits Characteristics
constants : -
classes   : -
variables : double flux_density [Jy], double flux_density_error [Jy]
methods   : -

Class Constraint
constants : -
classes   : -
variables : /* See Appendix E for explanation of these parameters */
           int type,
           double rekdir_x, rekdir_y, rekdir_z,
           double refnor_x, refnor_y, refnor_z,
           double angle, int object, double earthdistance
methods   : -

Class Calculator
constants : double ToDeg, double ToRad
classes   : -
variables : -
methods   : IntPart(double) returns double
           /* returns the integer part of a number */
           FracPart(double) returns double
           /* returns the fractional part of a number */
           RotMatR(double) returns double[3][3]
           /* Creates a 3x3 rotation matrix (array) - needs rotation angle [rad] */
           /* R can be 1, 2 or 3 (for X, Y, Z) rotation matrix) */
           MatMul(double[3][3], double[3]) returns double[3]
           /* Multiplies two a 3x1 vector and a 3x3 matrix (arrays) */

```

Observation class category

Note that most classes in this category have at least two types of initializers:

- one that accepts the basic components of a class to create a new object
e.g. for *VLBIStation* : `Initialize(string name, string alias, double antenna_diameter,
string mounting, ITRFPosition it_coord,
array of string terminal_types,
array of AntennaCharacteristics characteristics,
array of double local_horizon)`
- one that accepts an object of the given class to copy it and create a new object
e.g. for *VLBIStation* : `Initialize(VLBIStation v)`

Also note that most member classes and variables can be reached (read and set) through class methods.

```

Class ObservationObject
constants : -
classes   : -
variables : string name, string alias
methods   : -

```



```

Class Earth inherits ObservationObject
  constants : array of ITRFPosition /* map of continents */
              radius /* Earth radius [km] */
              flattening, J2, GCG /* other Earth features */
  classes    : -
  variables  : -
  methods    : GetEclipticVector(ICRFPosition, double) returns double[3]
              /* returns a ICRF-centric ecliptic unit vector of Earth's center */
              /* also needs obliquity of ecliptic */

Class CelestialObject inherits ObservationObject
  constants : -
  classes   : PolarPosition
  variables : -
  methods   :

Class Sun inherits CelestialObject
  constants : radius /* Sun radius [km] */
  classes   : -
  variables : -
  methods   : GetEclipticLong(MJDTime) returns double
              /* returns geocentric ecliptic longitude of the Sun */

Class Moon inherits CelestialObject
  constants : radius /* Moon radius [km] */
  classes   : -
  variables : -
  methods   : GetEclipticLongDist(MJDTime) returns double[3]
              /* returns geocentric ecliptic longitude, latitude, distance of the Moon */

Class Source inherits CelestialObject
  constants : -
  classes   : array of SourceCharacteristics, EquatorialPosition
  variables : -
  methods   : Precess(MJDTime, MJDTime)
              /* Converts source mean coordinates from epoch MJD1 to MJD2 */

Class Station inherits ObservationObject
  constants : -
  classes   : array of AntennaCharacteristics, ITRFPosition, ICRFPosition
  variables : double antenna_diameter [m]
  methods   : ToSource(Source, MJDTime)
              /* returns whether given source is visible from station */

Class Satellite inherits Station
  constants : -
  classes   : KeplerianOrbit (2), array of Constraint
  variables : double mass [kg], double cross_section [m^2],
              double solar_cell_area [m^2], double atmospheric_coeff
              double radiation_coeff
  methods   : UpdatePosition(MJDTime)
              /* Updates Satellite Orbital elements + its ICRF position*/
              /* Evolves orbital elements to given date taking first */
              /* order secular perturbations into account */
              ToSource(array of TelemetryStation, Source, MJDTime) returns boolean
              /* returns whether given source is visible from satellite */

Class TelemetryStation inherits Station
  constants : -
  classes   : -
  variables : string mounting, array of double local_horizon
  methods   : ToSatellite(Satellite) returns boolean
              /* returns whether given satellite is visible from station */
              /* note: Satellite ITRF position must be updated !!! */

```

```

Class VLBIStation inherits TelemetryStation
constants : -
classes   : -
variables : array of string terminal_types
methods   : ToSource(MJDTime, Source) returns boolean
            /* returns whether given source is visible from station */

Class ObservationSetup
constants : Resource names
classes   : array of Satellite, array of VLBIStation, array of TelemetryStation
            Source, MJDTime (3), Sun, Moon, Earth
variables : boolean ground_to_ground, ground_to_space, space_to_space,
            double observing_frequency [GHz], double bandwidth [MHz],
            int cutoff_mode, double cutoff_angle [rad],
            boolean satellite_constraints, boolean precession, double d3zoom
methods   : UpdateActiveStations(MJDTime)
            /* Updates active ("can observe") stations'/satellites' status/position */
            UpdateAllStations(MJDTime)
            /* Updates all stations'/satellites' status/position */
            ToSource(MJDTime) returns boolean
            /* returns whether observation is feasible with given configuration */
            ReadFile(string)
            /* Reads observation setup parameters from given file - see Appendix C */
            WriteFile(string)
            /* Writes observation setup parameters to given file - see Appendix C */

```

GUI (Graphical User Interface) class category

Since the classes in this category are based mainly on an implementation of a given windowing system (the Java Abstract Windowing Toolkit 1.0 in this case) the pseudocode definitions are not shown. The relevant information can be found in [10].

B. JavaSPAS database text file format

This appendix describes the format of the text file (Using LEX and YACC), which is used as the database - from which the user can select stations, sources and satellites through the setup dialog window. Note that this file (*database.txt* in the *Resources* directory) is only read once - at startup - and it is never written from within JavaSPAS. To edit it use any text editor.

B.1 LEX-type lexical specification

```

/* ----- */
/* JavaSPAS database LEX file      */
/* ----- */

%{
%}

creturn      [\r]
newline      [\n]
tab          [\t]
space        [ ]
colon        [:]
comma        [,]
slash        [/]
semicolon    [;]
pound        [#]
quote        [""]
dot          [.]
letter       [a-zA-Z]
digit        [0-9]
sign         (( "+" | "-" ))
digits       (( digit )+)
opt_fract    (( dot ) ( digits ))
opt_exp      (( "E" | "e" ) (( sign )?) (( digits )))
number       (( ( sign )? ) ( digits ) (( opt_fract )?) (( opt_exp )?))
string       ( quote ) { ^ " } * ( quote )
comment      ( semicolon ) { ^ \r \n } * ( newline )
separator    ( creturn ) | ( newline ) | ( tab ) | ( space ) | ( colon ) | ( slash ) | ( comma )+
startlist    [<]
endlist      [>]

/* keywords */
VLBI_header      "[VLBI_Stations]"
Telemetry_header "[Telemetry_Stations]"
Satellite_header "[Satellites]"
Source_header    "[Sources]"
Mount_1          "Azimutal"
Mount_2          "Equatorial"
Mount_3          "XYEW"
Mounting         "{ [Mount1] | [Mount2] | [Mount3] }"
Terminal_1       "Mk2"
Terminal_2       "Mk3"
Terminal_3       "Mk3A"
Terminal_4       "Mk4"
Terminal_5       "S2"
Terminal_6       "K3"
Terminal_7       "K4"
Terminal_8       "VLBA"
Terminal_9       "VSOP"
Terminals        "{ [Terminal_1] | [Terminal_2] | [Terminal_3] |
                  [Terminal_4] | [Terminal_5] | [Terminal_6] |
                  [Terminal_7] | [Terminal_8] | [Terminal_9]
                  }"

%%
(newline)      { /* increase line number */ }
(comment)      { /* do nothing */ }
(separator)    { /* do nothing */ }
.              { /* unrecognized character */ }
%%

```

B.2 YACC-type grammar specification

```

/* ----- */
/* JavaSPAS database YACC file */
/* ----- */

%{
/* capitalized identifiers are tokens from LEX */
%}

%%
database      :      vlbi_station_section
                |      telemetry_station_section
                |      satellite_section
                |      source_section

vlbi_sect     :      VLBI_HEADER vlbi_list

telemetry_sect :      TELEMETRY_HEADER telemetry_list

satellite_sect :      SATELLITE_HEADER satellite_list

source_sect   :      SOURCE_HEADER source_list

vlbi_list     :      vlbi_def vlbi_list
                |      /* epsilon */

telemetry_list :      telemetry_def telemetry_list
                |      /* epsilon */

satellite_list :      satellite_def satellite_list
                |      /* epsilon */

source_list   :      source_def source_list
                |      /* epsilon */

vlbi_def      :      STRING          /* name */
                |      STRING          /* alias */
                |      MOUNTING       /* antenna mounting */
                |      NUMBER         /* antenna diameter [m] */
                |      coordinates   /* station ITRF coordinates */
                |      STARTLIST
                |      lochoriz_list /* list of station local horizon values [deg] */
                |      ENDLIST
                |      STARTLIST
                |      terminal_list  /* list of antenna terminal types */
                |      ENDLIST
                |      STARTLIST
                |      character_list /* list of antenna characteristics */
                |      ENDLIST

telemetry_def :      STRING          /* name */
                |      STRING          /* alias */
                |      MOUNTING       /* antenna mounting */
                |      NUMBER         /* antenna diameter [m] */
                |      coordinates   /* station ITRF coordinates */
                |      STARTLIST
                |      lochoriz_list /* list of station local horizon values [deg] */
                |      ENDLIST

```

```

satellite_def :   STRING          /* name */
                  STRING          /* alias */
                  NUMBER          /* antenna diameter [m] */
orbit            :   /* orbital elements */
NUMBER          /* mass [kg] */
NUMBER          /* cross section [m2] */
NUMBER          /* solar cell area [m2] */
NUMBER          /* atmospheric coefficient [] */
NUMBER          /* radiation coefficient [] */
STARTLIST
character_list /* list of satellite antenna characteristics */
ENDLIST
STARTLIST
constr_list   /* list of satellite constraints */
ENDLIST

source_def :   STRING          /* name */
              STRING          /* alias */
              NUMBER NUMBER NUMBER /* Right ascension (RA)[hour:min:sec] */
              NUMBER NUMBER NUMBER /* Declination [deg:min:sec] */
flux_list    /* list of source fluxes */

coordinates :   NUMBER          /* X coordinate [km] */
              NUMBER          /* Y coordinate [km] */
              NUMBER          /* Z coordinate [km] */

terminal_list :   TERMINALS     /* see LEX definition of terminal types */
                |   terminal_list
                |   /* epsilon */

character_list : character_def character_list
                |   /* epsilon */

character_def :   NUMBER          /* frequency [GHz] */
              NUMBER          /* system temperature [K] */
              NUMBER          /* efficiency [%] */

lochoriz_list :   NUMBER          /* local horizon angle, max. 360 numbers */
                |   /* last value is "held" [deg] */
                |   lochoriz_list
                |   /* epsilon */

orbit :   NUMBER          /* semi-major axis (a) [km] */
         NUMBER          /* numerical eccentricity (e) [] */
         NUMBER          /* orbit inclination (i) [deg] */
         NUMBER          /* argument of perigee (w)[deg] */
         NUMBER          /* longitude of ascending node (loan) [deg] */
         NUMBER          /* mean anomaly (m) [deg] */
         date time      /* epoch of orbital elements */

constr_list :   constr_def constr_list
              |   /* epsilon */

constr_def :   /* for a detailed explanation see Appendix E */
              NUMBER          /* type */
              normal_vector /* direction */
              normal_vector /* plane */
              NUMBER          /* reference angle [deg] */
              NUMBER          /* reference object */
              NUMBER          /* earth distance */

date :   NUMBER          /* year */
        NUMBER          /* month */
        NUMBER          /* day */

time :   NUMBER          /* hour */
        NUMBER          /* minute */
        NUMBER          /* second */

normal_vector :   NUMBER          /* [0..1] */
                NUMBER          /* [0..1] */
                NUMBER          /* [0..1] */

```

```

flux_list      :      flux_def flux_list
                |      /* epsilon */

flux_def       :      NUMBER          /* frequency [GHz] */
                NUMBER          /* flux [Jy] */
                NUMBER          /* error [Jy] */

```

B.3 A simple example database file

```
[VLBI_Stations]
```

```
"Arecibo" "ARC" Azimutal 305.00
2390.46300 -5564.83770 1994.66970
```

```
< 70 >
```

```
< S2 >
```

```
< 1.60000    35.000    50.00
    5.00000    45.000    50.00 >
```

```
"Effelsberg" "Bonn" Azimutal 100.00
4033.94750 486.99045 4900.43074
```

```
< 10 >
```

```
< VLBA Mk4 >
```

```
< 1.60000    30.000    50.00
    5.00000    65.000    50.00
    22.00000   180.000    50.00 >
```

```
;
```

```
[Telemetry_Stations]
```

```
"Usuda" "Usuda" Azimutal 10.00
-3855.36000 3427.43000 3740.97000
```

```
< 10 >
```

```
;
```

```
[Satellites]
```

```
"RadioAstron" "RadioAstron" 10
46812.900 0.820000 51.000 285.000 255.000 280.000 1997/01/01 00:00:00
5000.000 80.000 50.000 0.500000 0.500000
```

```
<
```

```
    0.32700    100.000    50.00
    1.60000     50.000    50.00
    5.00000     50.000    50.00
    22.00000    150.000    30.00
```

```
>
```

```
<
```

```

1 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 60.00 0 0.00
1 0.0000 0.0000 1.0000 0.0000 0.0000 0.0000 90.00 0 0.00
1 0.0000 1.0000 0.0000 0.0000 0.0000 0.0000 67.00 0 0.00
1 0.0000 -1.0000 0.0000 0.0000 0.0000 0.0000 67.00 0 0.00
1 0.0000 0.0000 1.0000 0.0000 0.0000 0.0000 30.00 2 -13622.00
1 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 20.00 1 0.00
1 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 25.00 3 0.00
1 -0.8660 0.0000 0.5000 0.0000 0.0000 0.0000 -90.00 4 0.00
11 0.0000 0.8191 -0.5735 0.0000 0.0000 0.0000 30.00 0 0.00
11 0.0000 -0.8191 -0.5735 0.0000 0.0000 0.0000 30.00 0 0.00
11 -0.9659 0.0000 -0.2588 0.0000 0.0000 0.0000 30.00 0 0.00
11 0.0000 0.8191 -0.5735 0.0000 0.0000 0.0000 20.00 1 0.00
11 0.0000 -0.8191 -0.5735 0.0000 0.0000 0.0000 20.00 1 0.00
11 -0.9659 0.0000 -0.2588 0.0000 0.0000 0.0000 20.00 1 0.00
11 0.0000 0.8191 -0.5735 0.0000 0.0000 0.0000 25.00 3 0.00
11 0.0000 -0.8191 -0.5735 0.0000 0.0000 0.0000 25.00 3 0.00
11 -0.9659 0.0000 -0.2588 0.0000 0.0000 0.0000 25.00 3 0.00

```

```
>
```

```
;
```

[Sources]

"0106+013" "4C+01.02"

01:08:39 01:35:00

<

1.600000	2.000000	0.000000
5.000000	2.500000	0.000000
22.000000	3.500000	0.000000

>

C. JavaSPAS setup text file format

This appendix describes the format of the text file (Using LEX and YACC), which is used to store global setup parameters. Note that this file (*setup.txt* in the *Resources* directory) is read and written so theoretically there is no need to edit it with a text editor. As a matter of fact user comments included in the file that way might not be preserved when saving it from within JavaSPAS.

C.1 LEX-type lexical specification

```

/* ----- */
/* JavaSPAS database LEX file */
/* ----- */

%{
%}

creturn      [\r]
newline     [\n]
tab         [\t]
space       [ ]
colon       {:}
comma       [,]
slash       [/]
semicolon   [;]
equal       [=]
pound       [#]
quote       ["]
dot         [.]
letter      [a-zA-Z]
digit       [0-9]
sign        ({ "+" } | { "-" })
digits      ((digit)+)
opt_fract   ({dot}{digits})
opt_exp     ({ "E" | "e" } ({sign}?) ({digits}))
number      (({sign}?) {digits}) ({opt_fract}?) ({opt_exp}?)
string      {quote}[^"]*{quote}
comment     {semicolon}{^\\r\\n}*{newline}
separator   [{creturn}|{newline}|{tab}|{space}|{colon}|{slash}|{comma}|{equal}] +
startlist  [<]
endlist     [>]
On          ("Yes") | ("On")
Off         ("No") | ("Off")

OnOff       (On) | (Off)
StartTime   "StartTime"
StopTime    "StopTime"
TimeStep    "TimeStep"
BaseLine    "BaseLine"
GtoS        "GroundToSpace"
GtoG        "GroundToGround"
StoS        "SpaceToSpace"
ObsFreq     "ObservingFrequency"
BandWidth   "BandWidth"
VStation    "VLBIStation"
TStation    "TelemetryStation"
Satellite   "Satellite"
Source      "Source"
CutoffAngle "CutoffAngle"
CutoffMode  "CutoffMode"
LocHoriz    "LocalHorizon"
SatConstr   "SatelliteConstraints"
Precession  "Precession"
Zoom        "Zoom"

%%
(newline)   { /* increase line number */ }
(comment)   { /* increase line number */ }
(separator) { /* do nothing */ }
.           { /* unrecognized character */ }
%%

```


C.2 YACC-type grammar specification

```

/* ----- */
/* JavaSPAS setup YACC file */
/* ----- */

%{
/* capitalized identifiers are tokens from LEX */
%}

%%
setup      :      start_time
              stop_time
              time_step
              obs_frequency
              bandwidth
              baseline
              v_station
              t_station
              cutoff_angle
              cutoff_mode
              satellite
              sat_constr
              source
              precession
              zoom

start_time :      STARTIME date time /* Starting time of observation */
stop_time  :      STOPTIME date time /* Ending time of observation */
time_step  :      TIMESTEP time      /* Size of consecutive simulation time steps
obs_frequency :    OBSFREQ NUMBER    /* Observation frequency [GHz] */
bandwidth  :      BANDWIDTH NUMBER   /* Bandwidth [MHz] */
baseline   :      BASELINE GTOG      /* Use ground to ground baselines */
              |      BASELINE GTOS    /* Use ground to space baselines */
              |      BASELINE STOS    /* Use space to space baselines */
v_station  :      VSTATION STRING     /* VLBI station name from database */
t_station  :      TSTATION STRING     /* telemetry station name from database */
satellite  :      SATELLITE STRING   /* satellite name from database */
source     :      SOURCE STRING      /* source name from database */
cutoff_angle :    CUTOFFANGLE NUMBER /* cutoff angle for stations [deg] */
cutoff_mode :    CUTOFFMODE CUTOFFANGLE /* use cutoff angle value for stations */
              |    CUTOFFMODE LOCHORIZ /* use local horizon value for each station */
sat_constr :      SATCONSTR ONOFF    /* use satellite constraints or not */
precession :      PRECESSION ONOFF   /* use source precession or not */

date       :      NUMBER              /* year */
              |      NUMBER              /* month */
              |      NUMBER              /* day */

time       :      NUMBER              /* hour */
              |      NUMBER              /* minute */
              |      NUMBER              /* second */

```

C.3 A simple example setup file

```
; --- Time ---
StartTime = 1996/12/01 00:00:00
StopTime = 1996/12/02 00:00:00
TimeStep = 00:03:00

; --- Observation ---
ObservingFrequency = 1.6000
BandWidth = 32.0000

; --- Baselines ---
BaseLine = GroundToGround
BaseLine = GroundToSpace
BaseLine = SpaceToSpace

; --- Stations ---
CutoffAngle = 0
CutoffMode = LocalHorizon
VLBIStation = "Effelsberg"
VLBIStation = "GreenBank"
VLBIStation = "Goldstone"
VLBIStation = "Arecibo"
TelemetryStation = "Goldstone"
TelemetryStation = "Usuda"
TelemetryStation = "GreenBank"

; --- Satellites ---
SatelliteConstraints = On
Satellite = "VSOP"

; --- Source ----
Precession = On
Source = "0234+285"

; --- 3DView ---
Zoom = 1.0000
```

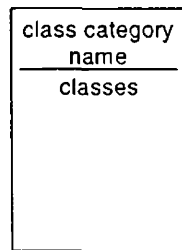
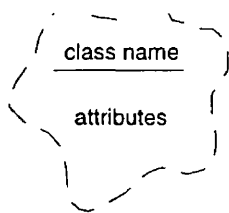
D. The Booch method artifacts

Shown here are the elements of the Booch notation [8] that are used in the JavaSPAS design.

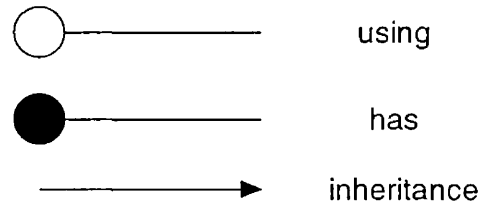
Class diagram

Shows the existence of classes and their relationships in the logical view of a system

Class icons



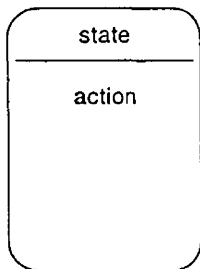
Class relationships



State Transition Diagram

Shows the state of a given context and the events that cause a transition from one state to another

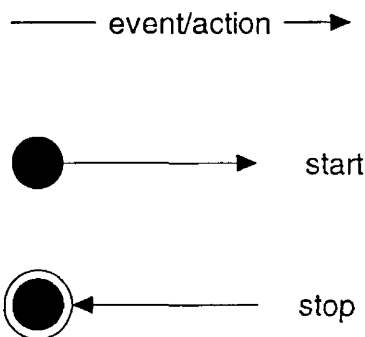
State icon



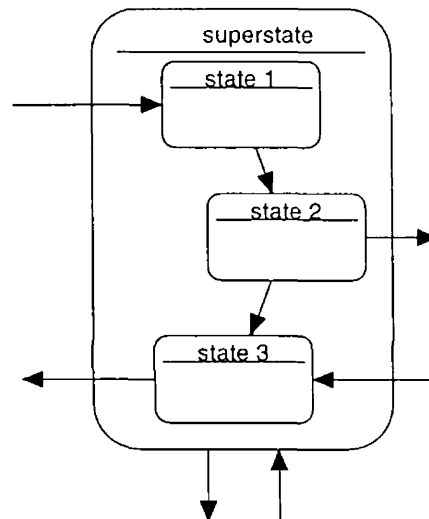
History



State transitions

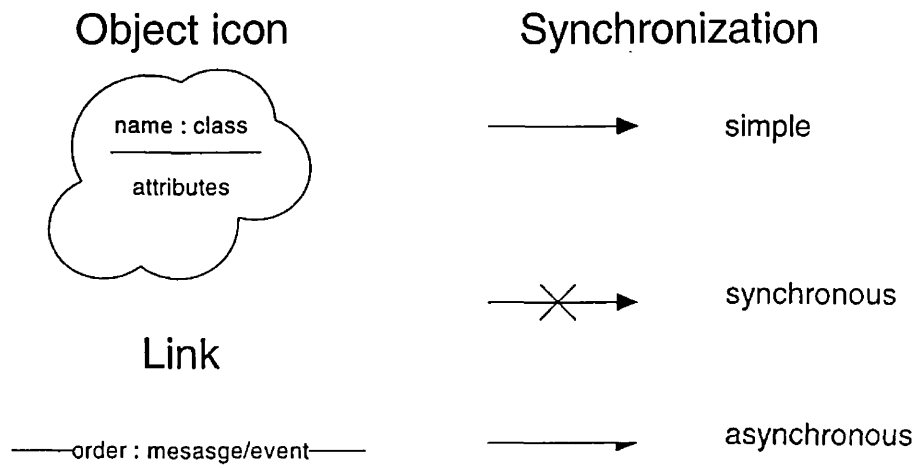


Nesting

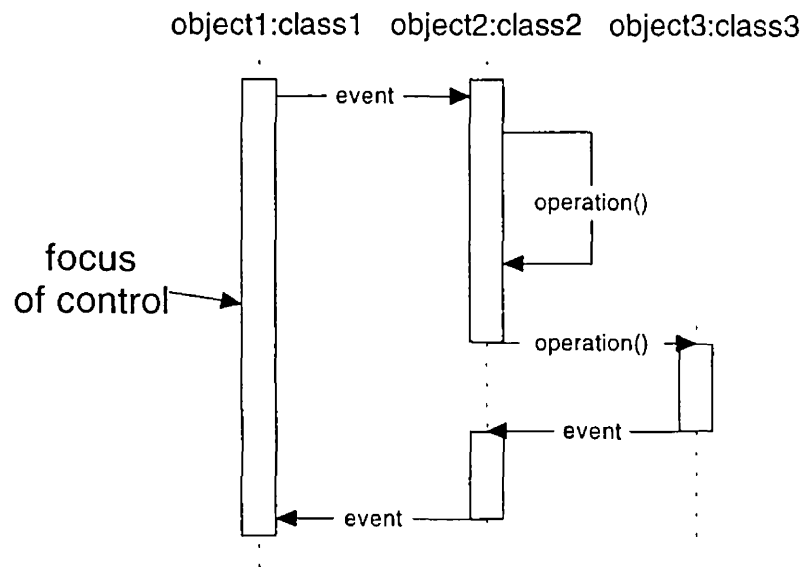


Object Diagram

Shows the existence of objects and their relationships in the logical view of the system

**Interaction Diagram**

Traces the execution of a scenario



E. Explanation of satellite constraints and their encoding

This chapter is based on Appendix C of [5].

CType	RefDir X	RefDir Y	RefDir Z	RefNor X	RefNor Y	RefNor Z	Angle [deg]	Object	Earth Distance [km]
1	0	0	1	0	0	0	+90	Sun	0
3	0	0	-1	0	1	0	-30	Sun	0
3	0	0	-1	0	-1	0	-30	Sun	0
11	0	cos 35°	-sin 35°	0	0	0	+30	Sun	0
11	0	-cos 35°	-sin 35°	0	0	0	+30	Sun	0
11	-cos 15°	0	-sin 15°	0	0	0	+30	Sun	0
11	0	cos 35°	-sin 35°	0	0	0	+20	Moon	0
11	0	-cos 35°	-sin 35°	0	0	0	+20	Moon	0
11	-cos 15°	0	-sin 15°	0	0	0	+20	Moon	0
11	0	cos 35°	-sin 35°	0	0	0	+25	Earth- Edge	0
11	0	-cos 35°	-sin 35°	0	0	0	+25	Earth- Edge	0
11	-cos 15°	0	-sin 15°	0	0	0	+25	Earth- Edge	0
3	0	0	0	0	1	0	-23	Sun	0
3	0	0	0	0	-1	0	-23	Sun	0
3	0	0	-1	1	0	0	-30	Sun	0
3	0	0	-1	-1	0	0	-90	Sun	0
1	-cos 30°	0	sin 30°	0	0	0	-90	Earth- Station	0
3	0	0	0	0	1	0	-30	Sun	0
3	0	0	0	0	-1	0	-30	Sun	0
1	0	0	1	0	0	0	+30	Earth- Center	-13622
1	1	0	0	0	0	0	+60	Sun	0
1	1	0	0	0	0	0	+5	Moon	0
1	1	0	0	0	0	0	+5	Earth- Edge	0
1	1	0	0	0	0	0	+30	Sun	0
1	1	0	0	0	0	0	+20	Moon	0
1	1	0	0	0	0	0	+25	Earth- Edge	0

Table E.1
RadioAstron constraint list

CType	RefDir X	RefDir Y	RefDir Z	RefNor X	RefNor Y	RefNor Z	Angle [deg]	Object	Earth Distance [km]
1	1	0	0	0	0	0	+70	Sun	0
1	1	0	0	0	0	0	+70	Earth- Station	0
4	0	0	0	0	0	0	0	Dummy	0
1	0	1	0	0	0	0	+85	Sun	0
1	0	1	0	0	0	0	-95	Sun	0
1	1	0	0	0	0	0	+1	Earth- Edge	0

Table E.2
VSOP constraint list

Explanation of constraint parameters

Reference direction (RefDirX, RefDirY, RefDirZ):

x , y and z are the coordinates of the unit vector which defines the reference direction in the satellite's coordinate system. The reference direction should be compared with the direction of a celestial object.

The satellite's coordinate system is a right-handed Cartesian system. The X axis coincides with the main axis of the on-board radio telescope. During the observation this axis points to the radio source on the sky. The solar panels lie along the Y and $-Y$ axes. The Z axis points to the "bottom" of the satellite where usually the high-gain communications antenna is mounted.

Reference normal vector (RefNorX, RefNorY, RefNorZ):

x , y and z coordinates are relevant for the *Type 2* constraints only. They define the normal (unit) vector of a plane in the satellite's system. The direction of the celestial object should be projected to this plane.

Angle [°]:

The angular limitation between the reference direction and the direction of the celestial object (or its projection). Note that the sign of Angle is only an indicator: A positive value means that the angle between the two directions should always be more than the Angle. A negative value means that the angle should be less than the Angle to perform successful observation with the satellite.

Object:

There are different celestial objects which the pointing restrictions connected to. These objects are coded in the following way:

0 = Sun (center), 1 = Moon (center), 2 = Earth (center), 3 = Earth (edge), 4 = Earth (tracking station)

Earth distance [km]:

A constraint may be valid only for certain satellite-Earth distances. Non-zero value of this parameter defines this distance limit. A negative value means that the constraint should be considered only for Earth distances less than the distance limit. A positive value means that the constraint is valid only for larger Earth distances than the distance limit. The distance of the satellite from the Earth is measured from the Earth's surface.

Explanation of constraint types:

CType = 1

A reference direction in satellite coordinate system is given by a unit vector (RefDirX, RefDirY, RefDirZ). Direction of a celestial object (Object) is also given.

If Angle < 0 then the angle between the two directions (Object's and RefDir) must not exceed Angle.

If Angle > 0 then the angle between the two directions (Object's and RefDir) must exceed Angle.

CType = 11, 12 and 13

These are CType = 1 constraints but only two of them must be OK at the same time.

CType = 2

A reference direction in satellite coordinate system is given by a unit vector (RefDirX, RefDirY, RefDirZ). Direction of a celestial object (Object) is also given.

A plane in satellite system is given by a normal vector (RefNorX, RefNorY, RefNorZ). The direction of Object must be projected onto this plane.

If Angle < 0 then the angle between the reference and projected directions must not exceed Angle.

If Angle > 0 then the angle between the reference and projected directions must exceed Angle.

CType = 3

A reference plane in satellite system is given by a normal vector (RefNorX, RefNorY, RefNorZ). Direction of a celestial object (Object) is also given.

If $\text{Angle} < 0$ then the angle between the reference plane and the object's direction must not exceed Angle .
 If $\text{Angle} > 0$ then the angle between the reference plane and the object's direction must exceed Angle .
 If one of RefDirX , RefDirY or RefDirZ equal with $+1$ or -1 then the reference plane is only the semi-plane $\pm X$, $\pm Y$ or $\pm Z$ respectively.

***CType = 3* constraints can be translated into *CType = 1* constraints easily or even formally using the following algorithm:**

```

If CType = 3 then
  If Angle < 0 then
    if RefDirX = 1 or RefDirX = -1 then
      create a new Record:
        (1, RefDirX, 0, 0, 0, 0, 0, - $\pi/2$ , Object, EarthDistance)
    (end if)
    if RefDirY = 1 or RefDirY = -1 then
      create a new Record:
        (1, 0, RefDirY, 0, 0, 0, 0, - $\pi/2$ , Object, EarthDistance)
    (end if)
    if RefDirZ = 1 or RefDirZ = -1 then
      create a new Record:
        (1, 0, 0, RefDirZ,, 0, 0, 0, - $\pi/2$ , Object, EarthDistance)
    (end if)
  (end if)

  CType = 1
  RefDirX = RefNorX
  RefDirY = RefNorY
  RefDirZ = RefNorZ
  RefNorX = 0
  RefNorY = 0
  RefNorZ = 0
  Angle = -Sgn(Angle) * ( $\pi/2$  - Abs(Angle))
(end if)
Next i

```

EarthDistance

If $\text{EarthDistance} < 0$ then the constraint is valid only for distances above the Earth surface less than EarthDistance .

If $\text{EarthDistance} > 0$ then the constraint is valid only for distances above the Earth surface more than EarthDistance .

CType = 4

Satellite cannot observe in the shadow of Earth, when no sunlight reaches the solar panels ("eclipse" constraint).

The protocol of RadioAstron constraints contains a lot of redundant information. This is due to the identical or overlapping constraints from different on-board equipment. Avoiding the redundancy we give a complete but reduced list of RadioAstron constraints below. The listed parameters can be used as inputs for the SPAS program.

C-Type	RefDir X	RefDir Y	RefDir Z	RefNor X	RefNor Y	RefNor Z	Angle [deg]	Object	Earth Distance [km]
1	1	0	0	0	0	0	+60	Sun	0
1	1	0	0	0	0	0	+20	Moon	0
1	1	0	0	0	0	0	+25	EarthEdge	0
1	0	1	0	0	0	0	+67	Sun	0
1	0	-1	0	0	0	0	+67	Sun	0
1	0	0	1	0	0	0	+90	Sun	0
1	$-\cos 30^\circ$	0	$\sin 30^\circ$	0	0	0	-90	EarthStation	0
1	0	0	1	0	0	0	+30	EarthCenter	-13622
11	0	$\cos 35^\circ$	$-\sin 35^\circ$	0	0	0	+30	Sun	0
11	0	$-\cos 35^\circ$	$-\sin 35^\circ$	0	0	0	+30	Sun	0
11	$-\cos 15^\circ$	0	$-\sin 15^\circ$	0	0	0	+30	Sun	0
11	0	$\cos 35^\circ$	$-\sin 35^\circ$	0	0	0	+20	Moon	0
11	0	$-\cos 35^\circ$	$-\sin 35^\circ$	0	0	0	+20	Moon	0
11	$-\cos 15^\circ$	0	$-\sin 15^\circ$	0	0	0	+20	Moon	0
11	0	$\cos 35^\circ$	$-\sin 35^\circ$	0	0	0	+25	EarthEdge	0
11	0	$-\cos 35^\circ$	$-\sin 35^\circ$	0	0	0	+25	EarthEdge	0
11	$-\cos 15^\circ$	0	$-\sin 15^\circ$	0	0	0	+25	EarthEdge	0

Table E.3
Simplified RadioAstron constraint list

F. Glossary

This glossary is taken from the SPAS User Manual [6], but only relevant portions are included here.

active baseline: both antennae at the ends of the *baseline* can observe the same radio source.

antenna efficiency: is a proportionality constant on a given frequency, which shows how effective is the telescope at absorbing radiation of this frequency from any particular source direction and making this power available at the output terminals. In ideal case the antenna efficiency is 1. In practice, typical range of the antenna efficiency is 0.4-0.7 depending on the observing frequency.

azimuth: see *horizontal system*

baseline: formed by two VLBI antennae (ground *VLBI stations* and/or satellites)

declination: see *equatorial system*.

ecliptic system: Definition: the primary reference plane is the plane of ecliptic, the secondary is the ecliptic meridian of the vernal equinox. (*Z* axis points to north ecliptic pole, *X* to the vernal equinox and *Y* completes a right-hand system.) A direction on the celestial sphere is defined by the angles *ecliptic longitude* and *ecliptic latitude*. The longitude is measured from 0° to 360° in the ecliptic plane from the vernal equinox to the east. The latitude is measured from -90° to 90°. The sign is positive above the ecliptic.

elevation: see *horizontal system*.

elevation cut-off angle: the minimum *elevation* angle where a source or satellite can be observed from a station.

equatorial system: Definition: the primary reference plane is the plane of equator, the secondary is the equinoctial colure. (*Z* axis points to north equatorial pole, *X* to the vernal equinox and *Y* completes a right-hand system.) A direction on the celestial sphere is defined by the angles *right ascension* and *declination*. The right ascension is measured from 0 h to 24 h in the equatorial plane from the vernal equinox to the east. The declination is measured from -90° to 90°. The sign is positive above the equator.

flux density: is the energy flux per unit frequency interval radiated by the radio source and measured at the receiver site. The radio astronomical unit of the flux density is the Jansky, $1 \text{ Jy} = 10^{-26} \text{ W m}^{-2} \text{ Hz}^{-1}$.

galactic system: The *Z* axis points to galactic pole, *X* to the galactic center (as defined in 1971) and *Y* completes a right-hand system. The coordinates of the galactic pole in the *equatorial system*: 12 h 49 m right ascension, 27° 24' declination. A direction on the celestial sphere is defined by the angles *galactic longitude* and *galactic latitude*. The galactic longitude is measured from 0° to 360° in the plane of galactic equator from *X* axis towards the *Y* axis. The galactic latitude is measured from -90° to 90°. The sign is positive above the galactic equator.

geographic system: The *Z* axis points to north equatorial pole, *X* to the Greenwich meridian and *Y* completes a right-hand system. A point can be given by angles *longitude*, *latitude* and *height* above the reference surface. The reference surface is a rotational ellipsoid. Its semi-major axis is the Earth equatorial radius ($a = 6378.1363 \text{ km}$), the flattening is $f = (a-b)/a = 1/298.257$ (IERS recommended values, *b* is the semi-minor or polar axis). Longitude is measured from 0° to 360° in the equatorial plane from the Greenwich meridian to the east. Latitude is the angle between the equatorial plane and the geodetic normal. It is measured from -90° to 90°. The sign is positive above the equator.

geomagnetic system: Spherical terrestrial coordinate system. The poles are defined as the poles of the Earth's magnetic field considered as a dipole. A direction in the system is defined by the geomagnetic *longitude* and *latitude*.

horizontal system: Definition: the primary reference plane is the plane of horizon, the secondary is the observer's celestial meridian. (*Z* axis points to zenith, *X* to north and *Y* completes a left-hand system.) A direction on the celestial sphere is defined by the angles *azimuth* and *elevation*. The azimuth is measured from 0° to 360° in the horizontal plane; it is 0° northward and 90° eastward. The elevation is measured from -90° to 90°. The sign is positive above the horizon. The system is called *topocentric* if its origin coincides with the observer's location.

ICRF (IERS Celestial Reference Frame): *equatorial system* defined and maintained by the IERS.

ITRF (IERS Terrestrial Reference Frame): *geographic system* defined and maintained by the IERS.

Julian Date (JD): Practical unit of time used in astronomy. The definition: days elapsed from B.C. 4713 January 1, 12 h Universal Time. Its simplified version is the *Modified Julian Date*.

Keplerian orbital elements: Unperturbed motion of a satellite around the Earth, governed by the central gravitation only, has six independent parameters. Usually the following Keplerian orbital elements are used: semi-major axis (a) and numerical eccentricity (e) of orbital ellipse, orbit inclination (i), argument of perigee (ω), longitude of ascending node (Ω) and mean anomaly at epoch (M). These elements determine the position and velocity of satellite at epoch.

latitude: see *ecliptic system, geographic system, geomagnetic system*.

local horizon: the minimal *elevation* angle as a function of *azimuth* where observations can be carried out from a *VLBI station* or *telemetry station*.

longitude: see *ecliptic system, geographic system, geomagnetic system*.

Modified Julian Date (MJD): Simplified version of *Julian Date*.

$MJD = JD - 2\,400\,000.5$

mounting (of radio telescope): It is the mechanical steering of the radio telescope, part of the mechanical structure. Fully steerable telescopes are generally placed on either an equatorial or an alt-azimuth type mounting. Transit telescopes have limited steerability, usually restricted to close of the meridian.

nutiation: see *precession*.

observability: If a source (or satellite) can be observed from VLBI station (or telemetry station) it is called observable. It means that the object is above the *local horizon* of the station (i.e. it is *visible*) and other restrictions don't obstruct the observation.

precession: The precession and *nutiation* mean the motion of the coordinate systems with respect to the stars due to the gravitational action of the extra-terrestrial bodies (Sun, Moon, planets) on the Earth's equatorial bulge. The effect of the Sun and Moon are resolved into two components: the lunisolar precession (moving of the celestial pole around ecliptic pole with amplitude of 23.5° and period 25 800 years) and the astronomic nutiation (relatively short periodic motions of celestial pole superimposed on lunisolar precession). The planetary precession is the effect of planets. The lunisolar and planetary precessions are considered together as general precession.

refraction: Change of direction of electromagnetic wave propagation in an inhomogenous medium like the Earth's atmosphere. From the ground based observer's point of view the *elevation* of a celestial object increases due to the refraction. The refraction angle is a function of elevation, it is 0 in zenith.

right ascension: see *equatorial system*

satellite conflict: Two satellites are in the field of view of the same *telemetry/tracking station* at the same time. A priority decision is needed in this case.

satellite constraints: on-board technical restrictions which determine the pointing of the antenna on the spacecraft.

satellite main axis: The axis of the radio antenna of the space VLBI satellite. This axis is considered as the X axis of satellite reference frame in SPAS.

secular perturbations: perturbations on satellite orbits which are linear functions of time (cf. *Earth gravity field*).

telemetry station: ground station (antenna) used for command uploading to satellites.

topocentric horizontal coordinates: see *horizontal system*.

tracking station: ground station (antenna) used for data downloading from satellites; it often coincides with a *telemetry station*.

UV-plot: is a projection of the *baselines* to a plane perpendicular to the source direction in units of wavelength. u is the east component and v is the north component of the projected baseline vector seen from the source. The third component w is the component of the baseline vector in the direction of the source.

visibility: If a source (or satellite) is above the *local horizon* of the station, the object is called visible. However, the *observability* may be impossible due to technical restrictions.

VLBI station: ground radio telescope used in VLBI measurement.

G. Bibliography

- [1]. ESA Software Engineering standards, ESA PSS-05-0 Issue 2, February, 1991, Paris, France
- [2]. *S. Frey*, SPAS User Requirements Document V2.1, May, 1993, Penc, Hungary
- [3]. *S. Frey*, SPAS Software Requirements Document V1.3, October 11, 1993, Penc Hungary
- [4]. *G. Heitler*, SPAS Architectural Design Document, V3.1, June 20, 1994, Penc, Hungary
- [5]. *G. Heitler*, (ed.) SPAS Detailed Design Document, V3.5, December 11, 1995, Penc, Hungary
- [6]. *S. Frey*, SPAS User Manual, Volume 1, V2.4, March 31, 1996, Penc Hungary
- [7]. *Barry W. Boehm*, A Spiral Model of Software Development and Enhancement, *Computer*, May 1988 pp. 61-72
- [8]. *G. Booch*, Object-Oriented Analysis and Design - With Applications, second edition, The Benjamin/Cummings Publishing Company, Inc., 1994
- [9]. *H Deitel, P. Deitel*, Java How to Program: with an Introduction to Visual J++, Prentice Hall, Inc., 1997
- [10]. *D. Flanagan*, Java in a Nutshell, first edition (Java 1.0), O'Reilly & Associates, Inc., 1996

WWW references:

- [11]. The Java White Papers by Sun Microsystems, Inc., <http://java.sun.com/docs/white/index.html>
- [12]. The Hungarian Notation (C. Simonyi), <http://www.freenet.tlh.fl.us/~joeo/hungarian.html>
- [13]. comp.lang.java.programmer FAQ list, July 15 1997, compiled by Peter van der Linden <http://www.best.com/~pvdl/javafaq.txt>
- [14]. JavaSPAS source (*.java) and object (*.class) code: <http://www.sgo.fomi.hu/~noszti/javaspas/>

Freely available Java packages used in the JavaSPAS implementation:

- [15]. *ImageButton.java* and related classes
Copyright(c) 1997 DTAI, Incorporated (<http://www.dtai.com>)
- [16]. *MultiLineLabel.java* and related classes (from [10])